

# **14 занимательных эссе о языке Haskell и функциональном программировании**



**Душкин Р. В.**

Душкин Р. В.

# **14 занимательных эссе о языке Haskell и функциональном программировании**



Москва, 2011

**УДК 004.4**  
**ББК 32.973.26-018.2**  
**Д86**

**Душкин Р. В.**  
**Д86** 14 занимательных эссе о языке Haskell и функциональном программировании. – М.: ДМК Пресс, 2011. – 140 с., ил.

**ISBN 978-5-94074-691-1**

В книге представлено 14 статей автора, которые в разное время были опубликованы или подготовлены к публикации в научно-популярном журнале для школьников и учителей «Потенциал». Статьи расположены и связаны таким образом, чтобы они представляли собой логически последовательное повествование от начал к более сложным темам. Также в книге сделан упор на практические знания, предлагается решение многих прикладных задач при помощи языка функционального программирования Haskell.

Книга будет интересна всем, кто живо интересуется функциональным программированием, студентам технических ВУЗов, преподавателям информатики.

**УДК 004.4**  
**ББК 32.973.26-018.2**

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-94074-691-1

© Душкин Р. В., 2011  
© Оформление ДМК Пресс, 2011

### **Принимаются благодарности**

Вниманию всех читателей! Данная книга издана в электронном виде и распространяется абсолютно бесплатно. Вы можете свободно использовать её для чтения, копировать её для друзей, размещать в библиотеках на сайтах в сети Интернет, рассыпать по электронной почте и при помощи иных средств передачи информации. Вы можете использовать текст книги частично или полностью в своих работах при условии размещения ссылок на оригинал и должном цитировании.

При этом автор будет нескажанно рад получить читательскую благодарность, которая позволит как улучшить текст данной книги, так и более качественно подойти к подготовке следующих книг. Благодарности принимаются на счёт Яндекс.Деньги, на который можно перечислить малую лепту, и при помощи терминалов:

**4100137733052**

Убедительная просьба; при перечислении благодарности указывать в пояснении к переводу наименование книги или какое-либо иное указание на то, за что именно выражается благодарность.

# Оглавление

<b>От автора</b>	<b>7</b>
<b>Типовой процесс разработки программ на языке Haskell</b>	<b>8</b>
Инструментальные средства . . . . .	8
Описание процесса разработки . . . . .	11
<b>Функциональный подход в программировании</b>	<b>15</b>
Введение . . . . .	15
Общие свойства функций в функциональных языках программирования . . . . .	16
Примеры определения функций . . . . .	17
Заключение . . . . .	21
<b>Алгебраические типы данных в языке Haskell</b>	<b>22</b>
Введение . . . . .	22
Простые перечисления . . . . .	23
Параметризация . . . . .	25
Параметрический полиморфизм . . . . .	27
Заключение . . . . .	28
<b>Объектно-ориентированное и функциональное программирование</b>	<b>30</b>
Введение . . . . .	30
Именованные поля и структуры . . . . .	31
Классы типов . . . . .	34
Экземпляры классов . . . . .	35
Окончательные замечания . . . . .	37
Заключение . . . . .	39
<b>Введение в <math>\lambda</math>-исчисление для начинающих</b>	<b>40</b>
Введение . . . . .	40
Неформальное описание теории . . . . .	41
Некоторые дополнения . . . . .	43
Редукция как стратегия вычислений . . . . .	43
Примеры кодирования данных и функций . . . . .	46
Заключение . . . . .	51
<b>Комбинаторы? — Это просто!</b>	<b>52</b>
Введение . . . . .	52
Формальная теория . . . . .	52
Примеры сложных комбинаторов . . . . .	55
Модуль на языке Haskell для преобразования комбинаторов . . . . .	57

Представление данных и функций . . . . .	59
Булевские значения . . . . .	59
Нумералы Чёрча . . . . .	60
Упорядоченные пары . . . . .	60
Общие замечания . . . . .	61
Заключение . . . . .	61
<b>Ввод и вывод на языке Haskell</b> . . . . .	<b>63</b>
Введение . . . . .	63
Основы функционального ввода/вывода . . . . .	64
Стандартные функции ввода/вывода . . . . .	66
Примеры программ . . . . .	69
Вывод результатов исполнения функции на экран . . . . .	69
Альтернатива: экран или файл . . . . .	70
Копирование файлов . . . . .	71
Заключение . . . . .	72
<b>Простой интерпретатор команд</b> . . . . .	<b>73</b>
Введение . . . . .	73
Постановка задачи . . . . .	73
Основной набор функций . . . . .	74
Вспомогательные типы данных . . . . .	75
Цикл интерпретации . . . . .	76
Функции для исполнения команд . . . . .	78
Заключение . . . . .	80
<b>Теория чисел и язык Haskell</b> . . . . .	<b>81</b>
Введение . . . . .	81
Простейшие задачи . . . . .	82
Такие непростые простые числа . . . . .	83
Числа Мерсенна . . . . .	85
Числа Ферма . . . . .	86
Числа Софи Жермен . . . . .	86
Другие последовательности простых чисел . . . . .	86
Совершенству нет предела . . . . .	87
Заключение . . . . .	89
<b>Магические квадраты и решение переборных задач</b> . . . . .	<b>90</b>
Введение . . . . .	90
Простейший вариант перебора . . . . .	91
Перебор с использованием перестановок . . . . .	94
Перебор с использованием размещений . . . . .	96
Дальнейшая универсализация алгоритма . . . . .	99
Заключение . . . . .	102
<b>Задача о ранце</b> . . . . .	<b>103</b>
Введение . . . . .	103
Классическая задача . . . . .	104
Реализация решения на языке Haskell . . . . .	105
Заключение . . . . .	108

<b>Кривая Дракона</b>	<b>109</b>
Введение . . . . .	109
Что такое Кривая Дракона? . . . . .	111
Алгоритм построения . . . . .	113
Реализация на языке Haskell . . . . .	113
Подготовительные описания геометрических образов . . . . .	114
Построение Кривой Дракона . . . . .	116
Заключение . . . . .	118
<b>Немного о шахматных задачах</b>	<b>119</b>
Введение . . . . .	119
Вспомогательные программные сущности . . . . .	119
Задача о расстановке фигур . . . . .	122
Задача о ходе коня . . . . .	123
<b>Генерация рекурсивных сказок</b>	<b>126</b>
Введение . . . . .	126
Колобок . . . . .	127
Теремок . . . . .	131
Обобщение функций и построение генератора . . . . .	133
Репка . . . . .	136
Заключение . . . . .	138
<b>Литература</b>	<b>139</b>

# От автора

В период с 2006 по 2009 год в образовательном журнале для старшеклассников и учителей «Потенциал» (ISSN 1814-6422) были опубликованы или подготовлены к публикации четырнадцать научно-популярных статей о функциональном программировании и языке Haskell. Многие из данных статей нашли определённый отклик в сердцах и умах читателей, поэтому автором было решено свести их в одну книгу, выстроив в более или менее стройную систему повествования.

Вместе с тем со времени начала публикации научно-популярных статей прошло достаточное количество времени, чтобы и язык Haskell, и функциональное программирование в целом эволюционировали, поэтому необходима и актуализация информации, приведение её в более современный вид. Всё это значит, что в настоящей книге исходные статьи не только не приведены в исходном порядке публикации, но и в некоторых случаях достаточно серьёзно переработаны.

Книга будет полезна школьникам старших классов, живо интересующимся информатикой и смежными областями науки и техники, а также учителям, которые проводят факультативные занятия по информатике и компьютерной грамотности. Кроме того, я надеюсь, что книга также будет полезна студентам технических высших учебных заведений, которые хотели бы овладеть пониманием функционального программирования в целом и языка Haskell в частности. Ну и в любом случае книга станет неплохим источником идей, задач и их решений для всех, кто интересуется функциональным программированием.

Я выражая самую серьёзную благодарность всем моим коллегам, которые в той или иной мере способствовали появлению многих из приведённых в книге статей на свет. Особо стоит отметить Антонюка Д. А. и Астапова Д. Е. (в том числе и в качестве рецензента), чьи бессменные советы и предложения по улучшению всегда приходились кстати. Так же хочу выразить благодарность всему коллективу редакции журнала «Потенциал» за его самоотверженный труд на поприще просвещения и популяризации науки среди подрастающего поколения. Отдельно хотелось бы отметить Ворожцова А. В., редактора отдела «Информатика».

Буду крайне признателен любым конструктивным комментариям, замечаниям и предложениям, которые можно присыпать по адресу электронной почты [roman.dushkin@gmail.com](mailto:roman.dushkin@gmail.com). Также по этому адресу можно присыпать запросы на файлы с исходными кодами примеров, приведённых в книге (указывайте, пожалуйста, наименования эссе, для которых необходимо выслать исходные коды примеров).

В добрый путь.

Душкин Р. В.  
Москва, 2011.

# Типовой процесс разработки программ на языке Haskell

*Статья была подготовлена к публикации в один из номеров журнала «Потенциал» в конце 2009 года. Опубликована не была.*

*В эссе рассказывается о типовом процессе разработки программных средств на функциональном языке программирования Haskell. Описываются некоторые из имеющихся на текущий момент программных средств и инструментов, делающих процесс разработки простым и быстрым. Даётся краткая суммарная информация о том, где и на каких условиях можно получить весь необходимый для работы инструментарий.*

В дальнейшем изложении в настоящей книге приводятся разнообразнейшие задачи из области математики и информатики, а также описываются решения этих задач при помощи языка функционального программирования Haskell. Опыт общения автора с читателями показал, что имеется проблема непосредственно практического характера — многие читатели просто не знают, что делать с приводимыми примерами и исходными кодами: какой инструментарий использовать, как компилировать программы, как загружать дополнительные пакеты и т. д.

Осмысление проблемы привело к пониманию того, что язык Haskell при всех его достоинствах очень сложно входит в русло практического использования как любителями, так и профессиональными программистами. В связи с этим возникает необходимость в использовании новой парадигмы популяризации языка Haskell и функционального подхода в программировании (впервые введена в книге [8]).

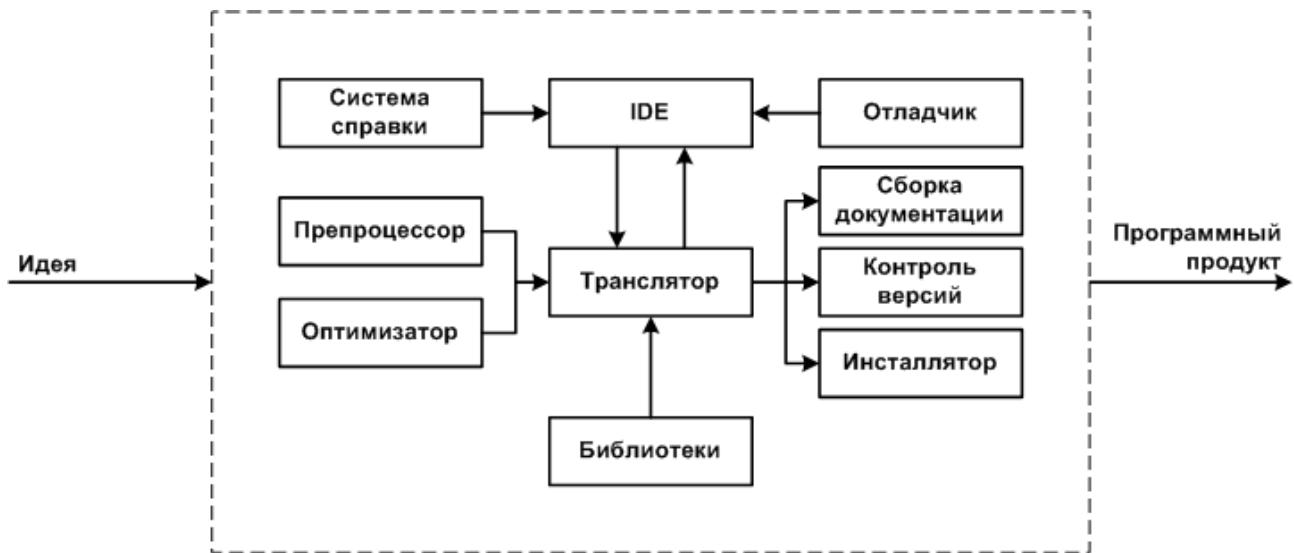
Данная парадигма предполагает необходимость всемерного распространения знаний и информации о практических средствах и методах разработки, а теоретическая часть может быть получена заинтересованными людьми самостоятельно, благо в литературе именно по теоретической части недостатка нет (см. в том числе книги [6, 7, 14, 15, 16]). Этот шаг позволит на первоначальном этапе заинтересовать и через интерес привлечь к функциональному программированию множество неофитов.

Данное эссе описывает типовой процесс разработки программного средства на языке Haskell на примере простейшего приложения «Hello, world!». Сложность разрабатываемого программного средства здесь совершенно не имеет значения, поскольку обрисовываются именно процесс и инструментарий. Читатель волен самостоятельно применять описываемые в разделе средства для решения произвольных задач, хотя бы и для рассмотрения примеров в следующих эссе, входящих в состав книги.

## Инструментальные средства

Весь набор инструментальных средств, используемых в работе над любым проектом, можно разделить на классы. Это относится не только к языку программирования Haskell, но и к любому другому языку

программирования, впрочем, как и к любым иным проектам — от проектирования детской игрушки до постройки города. В принципе, разделить на классы инструментальные средства для работы с языком Haskell можно так, как показано на рис. 1.



**Рис. 1. Кибернетическая схема преобразования идеи в законченное программное средство**

Также инструментарий можно классифицировать по степени полезности и необходимости. Некоторые классы инструментов обычно не используются (препроцессоры для языка Haskell используются редко), а без, к примеру, трансляторов языка вообще невозможно обойтись. Так что ниже в таблице перечислены наиболее часто используемые инструментальные средства, которые подходят для каждого разработчика, для любого проекта.

Инструмент	Описание
1	2
Компилятор GHC	Самый мощный и совершенный компилятор, на сегодняшний день не имеющий аналогов, — GHC. Реализует не только стандарт Haskell-98, но и множество расширений языка, многие из которых уже стали стандартом де-факто. Имеет возможность работы в режиме интерпретации (в состав поставки входит интерпретатор GHCi).  <a href="http://www.haskell.org/ghc/">http://www.haskell.org/ghc/</a>
Тестирование QuickCheck	Для проведения тестирования обычно используются специальные библиотеки вроде QuickCheck. Они позволяют создавать семантические правила, описывающие поведения функций в проекте, после чего запускать эти правила на проверку с различными аргументами. Проверка осуществляется компилятором в процессе сборки программы.  <a href="http://www.cs.chalmers.se/~rjmh/QuickCheck/">http://www.cs.chalmers.se/~rjmh/QuickCheck/</a>

*Продолжение на следующей странице*

1	2
Документирование Haddock	<p>При написании исходных кодов можно сразу документировать программные сущности таким образом, что использование утилиты Haddock позволит собрать справочную систему для разработанного проекта в формате HTML. Это — правильный подход к разработке, поскольку документация позволяет без проблем передавать разработанный проект.</p> <p><a href="http://www.haskell.org/haddock/">http://www.haskell.org/haddock/</a></p>
Контроль версий Darcs	<p>Для хранения файлов с исходными кодами и ведения их версий можно воспользоваться утилитой распределённого хранения Darcs. Она обладает рядом дополнительных возможностей, превышающих стандартную функциональность систем контроля версий.</p> <p><a href="http://www.darcs.net/">http://www.darcs.net/</a></p>
Система сборки Cabal	<p>Утилита Cabal позволяет собрать пакет для распространения среди разработчиков и пользователей. Пакет собирается в стандартном формате, который может быть использован на произвольной платформе, поддерживающей язык Haskell. Кроме того, эта же утилита может установить в систему требуемый пакет, самостоятельно скачав его из сети Интернет.</p> <p><a href="http://www.haskell.org/cabal/">http://www.haskell.org/cabal/</a></p>
Распространение Hackage	<p>Для того чтобы каждый желающий смог воспользоваться разработанным пакетом, его можно положить в общий архив исходных кодов на языке Haskell Hackage. Этот архив является централизованным, и сегодня в нём хранятся сотни различных пакетов для решения практически любых задач.</p> <p><a href="http://hackage.haskell.org/">http://hackage.haskell.org/</a></p>

Перечисленные в приведённой таблице инструментальные средства являются свободно распространяемым программным обеспечением, доступным для получения с указанных адресов в сети Интернет. Читателям настоятельно рекомендуется скачать перечисленные программные средства для возможности реальной работы с языком Haskell, что позволит без проблем работать над примерами, описанными далее в этой книге.

## Описание процесса разработки

После установки на рабочий компьютер перечисленного инструментария можно начинать работу над проектами. Ниже представлена типовая структура проекта на примере простейшего приложения типа «Hello, World!». Типовой проект на языке Haskell будет содержать следующие компоненты:

- 1) `_darcs` — каталог для хранения версий файлов с исходными кодами;
- 2) `Hello.hs` — главный файл проекта (в нём содержится функция `main`);
- 3) `Hello.cabal` — описание проекта для системы Cabal;
- 4) `Setup.hs` — сценарий сборки проекта в системе Cabal;
- 5) `README` — файл с информацией о проекте;
- 6) `LICENSE` — файл с описанием лицензии.

Конечно, каждый разработчик вправе вносить в указанную структуру изменения и дополнения, отражающие суть соответствующего проекта (например, файлы с исходными кодами хорошо бы разместить в подкаталоге `/src`). Здесь приведён именно пример простейшего проекта, состоящего, по сути, из одного файла с исходными кодами. К одному файлу (в примере — `Hello.hs`) добавляются дополнительные файлы и каталоги, необходимые для поддержания инфраструктуры проекта.

Следующая инструкция показывает пошаговое создание проекта «Hello, World!» до получения исполняемого файла и размещения его в архиве проектов.

- 1) Создание каталога проекта и файла с исходным кодом в нём. Содержимое файла может быть следующим:

```
module Main

-- | Функция 'main', выводящая на экран приветствие.
main :: IO ()
main = putStrLn "Hello, World!"
```

- 2) Сохранение информации в системе хранения версий. Это делается при помощи следующих команд:

```
darcs init
darcs add Hello.hs
darcs record
```

При выполнении последней команды утилиты Darcs задаст пользователю ряд вопросов, после чего файл `Hello.hs` будет сохранён в архиве для сохранения первой версии.

- 3) Поскольку файл с исходными кодами готов (само собой разумеется, что в случае сложных проектов необходимо быть уверенным в полной готовности всех файлов проекта), необходимо создать файл с описанием проекта для системы Cabal. Содержимое файла `Hello.cabal` содержит примерно следующие данные:

Name:	hello
Version:	0.0.0.1
Description:	Simplest Haskell Application
License:	GPL3
License-file:	LICENSE
Author:	John Smith

```
Maintainer:      john.smith@example.com
Build-Type:      Simple
Cabal-Version:  >= 1.8
Executable haq
  Main-is:        Hello.hs
  Build-Depends: base >= 3 && < 5
```

Если разрабатываемый проект зависит от каких-либо иных пакетов, то все они должны быть указаны в файле `.cabal` для сборки в поле `Build-Depends`.

- 4) Далее необходимо написать сценарий сборки. Для подавляющего числа проектов сценарий `Setup.hs` одинаков (утилита Cabal позволяет использовать как обычные файлы `.hs`, так и коды в литературном стиле `.lhs`). Его содержимое следующее:

```
import Distribution.Simple
```

```
main = defaultMain
```

- 5) Если есть потребность, можно разработать файлы `README` с общей информацией о проекте и `LICENSE` с информацией о лицензировании проекта. Эти файлы не участвуют в процессе сборки, но предназначаются для более целостного представления проекта потенциальным потребителям.

- 6) Новые файлы также необходимо сохранить в системе хранения версий Darcs. Это делается при помощи выполнения следующих команд (опять же, при их выполнении утилиты может задать дополнительные вопросы, на которые необходимо ответить по существу):

```
darcs add Hello.cabal Setup.hs LICENSE README
darcs record --all
```

- 7) Сборка проекта может осуществляться как при помощи транслятора, так и при помощи утилиты Cabal. Последний способ гарантирует то, что получаемые исполняемые файлы проекта будут готовы для сохранения в централизованном архиве Hackage. Сборка проекта осуществляется при помощи выполнения следующих команд:

```
cabal install --prefix=\$HOME --user
```

Выполнение команды создаст в подкаталоге `/bin` исполняемый файл проекта. Его можно будет запускать на исполнение.

- 8) Теперь можно сгенерировать документацию для проекта. Это также можно осуществить как при помощи утилиты Haddock, так и непосредственно используя систему Cabal, поскольку имеется централизованная интеграция инструментов с этой системой сборки. Документация генерируется следующей командой:

```
cabal haddock
```

В результате исполнения команды будет сгенерирован набор файлов `.html` и некоторых дополнительных к ним служебных файлов и подкаталогов. Данные файлы содержат документацию проекта, созданную на основании комментариев разработчика, написанных в исходных кодах.

- 9) Также в проект можно добавить методы автоматизированного тестирования функций при помощи библиотеки `QuickCheck`. Организацию тестирования можно осуществить при помощи системы хранения версий Darcs при сохранении очередных изменений — утилиты самостоятельно будет

осуществлять тестирование и будет производить уведомление пользователя в случаях, если тестирование завершилось неудачей.

В настоящем примере функция `main` слишком проста для проведения тестирования. В более сложных проектах желательно организовывать автоматизированное тестирование для «отлова» потенциальных логических ошибок. Пусть сценарий тестирования расположен в файле `Test.hs`, в этом случае интеграция процесса тестирования с процессом сохранения версии в системе хранения выполняется при помощи следующей команды:

```
darcs setpref test "runhaskell Tests.hs"
```

Само собой разумеется, что новый файл `Test.hs` также необходимо сохранить в системе хранения версий. Кстати, в последних версиях Cabal тестирование можно проводить средствами этой утилиты.

- 10) После появления стабильной версии проекта (версии, которая не содержит критических ошибок, приводящих к «падению» приложения) её можно пометить специальным образом в системе хранения в целях быстрого доступа к файлам версии. Это делается при помощи следующей команды:

```
darcs tag
```

Утилита Darcs задаст дополнительные вопросы о том, какая метка должна быть присвоена текущим версиям файлов в репозитории.

- 11) Далее при помощи системы Cabal необходимо создать пакет для распространения и помещения в централизованный архив Hackage. Перед этим желательно проверить, всё ли готово для загрузки в централизованный архив, для чего можно выполнить такую команду:

```
cabal check
```

Если всё в порядке, то можно готовить файл для загрузки в Hackage. Это делается просто, при помощи команды:

```
cabal sdist
```

В результате выполнения команды в каталоге проекта будет создан файл `hello-0.0.0.1.tar.gz`, в котором находятся полное описание проекта, исходные коды для него, а сам проект будет полностью готов для размещения в архиве Hackage.

- 12) Окончательным шагом в работе над проектом будет отсылка файла с дистрибутивом (пакетом) в централизованный архив Hackage. Для этого необходимо иметь учётную запись пользователя архива (её можно получить при помощи регистрации на веб-сайте <http://hackage.haskell.org/>). После входа в архив под учётной записью будет доступна форма для загрузки файла. Загрузка файла пакета на веб-сайт приведёт к автоматической проверке и, в случае её успешности, помещению пакета в архив.

Дело сделано. Но для более серьёзного проекта будет хорошим тоном создание на веб-сайте <http://www.haskell.org/> страницы с описанием проекта. Данный сайт — официальный сайт сообщества программистов на языке Haskell. Он использует технологию Wiki, при помощи которой любой желающий посетитель может внести в страницы веб-сайта информацию. Создание новых разделов с описаниями проектов поощряется владельцами сайта, и сегодня на нём содержится описание большинства существующих проектов, созданных как на языке Haskell, так и для работы с ним.

Таков общий процесс разработки проектов на языке Haskell. Автор искренне надеется, что данное краткое эссе станет хорошим подспорьем как для начинающих программистов, так и для умелых разработчиков. Собственно, заниматься со всеми последующими разделами книги можно именно по приведённой схеме.

Также необходимо отметить, что в 2009 году (уже после написания этого эссе) сообществом языка Haskell был подготовлен целостный пакет прикладного программного обеспечения для работы на этом языке Haskell Platform. Данный пакет включает в себя компилятор GHC, систему генерации справки Haddock, систему подготовки дистрибутивов Cabal, пакет тестирования QuickCheck, а также несколько других часто используемых в работе пакетов и утилит.

Актуальное описание того, как разрабатывать программы на языке Haskell, готовить их к распространению и помещать в централизованный архив Hackage, всегда можно найти по адресу [http://www.haskell.org/haskellwiki/How\\_to\\_write\\_a\\_Haskell\\_program](http://www.haskell.org/haskellwiki/How_to_write_a_Haskell_program).

# Функциональный подход в программировании

*Статья опубликована в № 08 (56) журнала «Потенциал» в августе 2009 года. Сама по себе статья является упрощённой переработкой статьи «Функции и функциональный подход», опубликованной в № 01 научно-практического журнала «Практика функционального программирования».*

*В эссе в сжатой форме рассказывается про функциональный подход к описанию вычислительных процессов (и в общем к описанию произвольных процессов в реальном мире), а также про применение этого подхода в информатике в функциональной парадигме программирования. Примеры реализации функций даются на языке программирования Haskell.*

## Введение

Имеется несколько различающихся подходов к организации вычислительных процессов, наиболее известными и в какой-то степени «конкурирующими» являются императивный (процедурный) и функциональный стили. Эти стили вычислений были известны в далёком прошлом, и сейчас уже невозможно узнать, какой подход был разработан первым.

Последовательности шагов вычислений, как особенность процедурного стиля, можно рассматривать в качестве естественного способа выражения человеческой деятельности при её планировании. Это связано с тем, что человеку приходится жить в мире, где неумолимый бег времени и ограниченность ресурсов каждого отдельного индивидуума заставлял людей планировать по шагам свою дальнейшую жизнедеятельность.

Вместе с тем нельзя сказать, что функциональный стиль вычислений был неизвестен человеку, а появился только с возникновением теории вычислений в том или ином виде в конце XIX — начале XX века. Декомпозиция задачи на подзадачи и выражение ещё нерешённых проблем через уже решённые — эти методики также были известны с давних времён, а именно они составляют суть функционального подхода. Именно этот подход и является предметом рассмотрения настоящего раздела, а объясняться его положения будут при помощи функционального языка Haskell (для изучения языка можно воспользоваться книгой [6]).

Несмотря на то что фактически функциональный подход к вычислениям был известен с давних времён, его теоретические основы стали разрабатываться вместе с началом работ над вычислительными машинами — сначала механическими, а потом уже и электронными. С развитием формальной логики и обобщением множества сходных идей под сводом кибернетики появилось понимание того, что функция является прекрасным математическим формализмом для описания реализуемых в физическом мире устройств [2]. Но не всякая функция, а только такая, которая обладает рядом важных свойств, а именно: во-первых, она оперирует исключительно своей внутренней памятью, а во-вторых, она детерминирована (то есть её значение зависит только от входных параметров — это свойство будет подробно рассмотрено далее). Данные ограничения на реализуемость в реальности связаны с физическими законами сохранения,

в первую очередь энергии. Именно такие «чистые» процессы рассматриваются в кибернетике при помощи методологии чёрного ящика — результат работы такого ящика зависит только от значений входных параметров.

Классическая иллюстрация, демонстрирующая эту ситуацию, встречается в большинстве учебников по кибернетике и смежным дисциплинам:

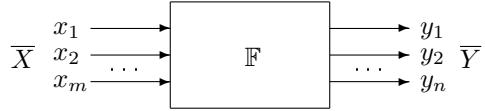


Рис. 2. Чёрный ящик функции  $F$  с вектором входов  $\bar{X}$  и вектором выходов  $\bar{Y}$

Формальные основы теории вычислений были заложены несколькими учёными, одним из ведущих среди которых был Алонзо Чёрч, предложивший в качестве формализма для представления вычислимых функций и процессов  $\lambda$ -исчисление [1]. Само по себе  $\lambda$ -исчисление предлагает нотацию для простейшего языка программирования. Собственно, ядро языка программирования Haskell представляет собой типизированное  $\lambda$ -исчисление.

Также стоит упомянуть про комбинаторную логику [4], которая использует несколько иную нотацию для представления функций, а в качестве базовой операции использует только применение функции к своим аргументам. Базис системы состоит из одного комбинатора, то есть утверждается, что любая функция может быть выражена через этот единственный базисный комбинатор (базисную функцию). Традиционно используются несколько более расширенные базисы (например, базис **S**, **K**, **I**), что позволяет сократить записи в нотации комбинаторной логики. Сама по себе комбинаторная логика изоморфна  $\lambda$ -исчислению, но обладает, по словам некоторых специалистов, большей выразительной силой.

Таким образом, функциональный подход имеет в своём основании серьёзную теоретическую проработку, а потому его изучение может стать серьёзным подспорьем как для начинающих разработчиков программного обеспечения, так и для профессионалов, желающих расширить свой кругозор и получить навыки владения новыми инструментами для решения практических задач.

## Общие свойства функций в функциональных языках программирования

Перед изучением примеров необходимо рассмотреть общие свойства функций, рассматриваемые в функциональном программировании. К таким свойствам наиболее часто относят *чистоту* (то есть отсутствие побочных эффектов и детерминированность), *ленивость* и *каррированность* (наличие у функции особого типа, что влечёт за собой возможность производить частичные вычисления и использовать функции в качестве объекта вычислений — получать их в качестве параметров и возвращать в качестве результатов).

Итак, как уже упоминалось, физически реализуемыми являются такие кибернетические машины, выход которых зависит только от значений входных параметров. Это положение относится и к таким кибернетическим машинам, которые имеют внутренний накопитель — память. Данное положение нашло чёткое отражение в парадигме функционального программирования, поскольку в ней принято, что функции, являясь чистыми математическими объектами, должны обладать свойством чистоты. Это обозначает, что функция может управлять только выделенной для неё памятью, не модифицируя память во вне своей области. Любое изменение сторонней памяти называется побочным эффектом, и функциям в функциональных языках программирования обычно запрещено иметь побочные эффекты.

То же и с детерминированностью. Детерминированной называется функция, выходное значение которой зависит только от значений входных параметров. Если при одинаковых значениях входных параметров в различных вызовах функция может возвращать различные значения, то говорят, что

такая функция является недетерминированной. Соответственно, обычно все функции в функциональной парадигме являются детерминированными.

Конечно, есть некоторые исключения, поскольку, к примеру, систему ввода-вывода невозможно сделать без побочных эффектов и в условиях полной детерминированности. Также и генерация псевдослучайных чисел осуществляется недетерминированной функцией. Само собой разумеется, что эти практические задачи должны решаться универсальным языком программирования, коим является язык Haskell. В языке имеются специальные механизмы для этого, но их рассмотрение выходит за рамки настоящего раздела.

Очень интересным свойством функций является ленивость. Не все функциональные языки предоставляют разработчику возможность определять ленивые функции, но язык Haskell изначально является ленивым, и разработчику необходимо делать специальные пометки для функций, которые должны осуществлять «энергичные» вычисления. Ленивая же стратегия вычислений заключается в том, что функция не производит вычислений до тех пор, пока их результат не будет необходим в работе программы [9]. Так, значения входных параметров никогда не вычисляются, если они не требуются в теле функции. Это позволяет в том числе создавать потенциально бесконечные структуры данных (списки, деревья и т. д.), которые ограничены только физическим размером компьютерной памяти. Такие бесконечные структуры вполне можно обрабатывать ленивым способом, поскольку вычисляются в них только те элементы, которые необходимы для работы. И передача на вход какой-либо функции бесконечного списка не влечёт зацикливания программы, поскольку она не вычисляет весь этот список целиком (что было бы невозможным).

Наконец, уже упоминалось, что у функций есть тип. В функциональных языках программирования принято, что тип функций является каррированным, то есть таким, общий вид которого выглядит следующим образом:

$$A_1 \rightarrow (A_2 \rightarrow \dots (A_n \rightarrow B) \dots), \quad (1)$$

где  $A_1, A_2, \dots, A_n$  — типы входных параметров, а  $B$  — тип результата.

Каррированность функций означает, что такие функции принимают входные параметры поодиночке, а в результате такого одиночного применения получается новая функция. Так, если в функцию указанного выше типа подать первый параметр типа  $A_1$ , то в итоге получится новая функция с типом:

$$A_2 \rightarrow (A_3 \rightarrow \dots (A_n \rightarrow B) \dots). \quad (2)$$

Когда на вход функции подаются все входные параметры, в результате получается значение типа  $B$ .

В свою очередь, это означает не только возможность *частичного применения* (на вход функции передаётся только часть параметров, а не все сразу), но и то, что функции сами по себе могут быть объектами вычислений, то есть передаваться в качестве параметров другим функциям и быть возвращаемыми в качестве результатов. Ведь никто не ограничивает указанные типы  $A_1, A_2, \dots, A_n$  и  $B$  только типами данных, это всё могут быть также и функциональные типы.

Перечисленные свойства функций в функциональных языках программирования открывают дополнительные возможности по использованию функционального подхода, поэтому разработчикам программного обеспечения рекомендуется изучить этот вопрос более подробно.

## Примеры определения функций

В одной из коммерческих компаний при отборе кандидатов на вакантные должности инженеров-программистов пришедшем задавалась простая задача — необходимо написать функцию, которая получает на вход некоторое целое число, а возвращает строку с представлением данного числа в шестнадцатеричном виде. Задача очень простая, но вместе с тем она достаточно легко может проявить методы решения задач у кандидатов, поэтому основной упор на собеседовании делался не на правильность

написания кода, а на подход и канву рассуждений при написании этой функции. Более того, если кандидат затруднялся с алгоритмом, ему он полностью разъяснялся, поскольку интересны были именно ход рассуждений и окончательный способ реализации алгоритма. Соответственно, для решения задачи можно было использовать произвольный язык программирования на выбор кандидата. Можно было даже использовать псевдоязык описания алгоритмов, блок-схемы и прочие подобные вещи.

Сам алгоритм прост. Необходимо делить заданное число на основание (в задаваемой задаче, стало быть, на 16), собирать остатки и продолжать этот процесс до тех пор, пока в результате деления не получится 0. Полученные остатки необходимо перевести в строковый вид посимвольно (учитывая шестнадцатеричные цифры), после чего скепить (при помощи операции конкатенации) все эти символы в результирующую строку в правильном направлении (первый остаток должен быть последним символом в результирующей строке, второй — предпоследним и т. д.).

Каковы были типовые рассуждения большинства приходящих на собеседование? «Получаем входное число — организуем цикл `while` до тех пор, пока параметр цикла не станет равен 0 — в цикле собираем остатки от деления параметра на основание, тут же переводим их в символы и конкатенируем с переменной, которая потом будет возвращена в качестве результата — перед возвращением переменную обращаем». К сожалению, за всё время работы данной организации ни один из кандидатов не предложил решения задачи в функциональном стиле.

Вот как выглядит типовая функция для описанной цели на языке C++:

```
std::string int2hex (int i)
{
    std::string result = "";
    while (i)
    {
        result = hexDigit (i % 16) + result;
        i /= 16;
    }
    return result;
}
```

Здесь функция `hexDigit` возвращает символ, соответствующий шестнадцатеричной цифре.

Как же решить эту задачу при помощи функционального подхода? При размышлении становится ясно, что, взяв первый остаток от деления на 16 и после этого целочисленно разделив само число на 16, задача сводится к той же самой. И такое сведение будет происходить до тех пор, пока число, которое необходимо делить, не станет равным 0. Налицо рекурсия, которая является одним из широко используемых методов функционального программирования. На языке Haskell эта задача может быть решена следующим образом:

```
int2hex :: Integer -> String
int2hex 0 = ""
int2hex i = int2hex (div i 16) ++ hexDigit (mod i 16)
```

Здесь функции `div` и `mod` возвращают соответственно результат целочисленного деления и остаток от такого деления. Функция `(++)` конкатенирует две строки. Все эти функции определены в стандартном модуле `Prelude`. Первая строка определения функции — так называемая сигнатура, которая определяет тип функции (необходимо обратить внимание на эту запись — она полностью соответствует тому, что описано в предыдущем разделе относительно функциональных типов).

Вторая строка определяет результат функции `int2hex` в случае, если значением её единственного входного параметра будет 0 (здесь также необходимо отметить, что определение не очень корректно — оно не будет работать на граничном значении входного параметра 0, поэтому фактически функция определена только для натуральных чисел). Третья строка, соответственно, определяет результат

функции в оставшихся случаях (когда значение входного параметра ненулевое). Здесь применён механизм *сопоставления с образцами*, когда для определения функции записываются несколько выражений (иногда в литературе по функциональному программированию используется термин «*клоз*» от англ. «clause» для обозначения одного такого выражения в определении функции), каждый из которых определяет значение функции в определённых условиях.

Представленный пример уже достаточно показывает отличие двух подходов к представлению вычислений. Тем не менее уже сейчас видно, что имеется широкий простор для усовершенствования кода. В первую очередь это касается основания преобразования, ведь часто при программировании необходимы числа в двоичной и восьмеричной записи. Более того, почему бы не сделать универсальную функцию для преобразования числа в произвольную систему счисления? Эта задача легко решается преобразованием уже написанной функции:

```
convert :: Int -> Int -> String
convert _ 0 = ""
convert r i = convert r (div i r) ++ digit r (mod i r)
```

Здесь в сигнатуру внесены два изменения. Во-первых, тип `Integer` изменён на тип `Int`, что связано с необходимостью ограничения (тип `Integer` представляет неограниченные целые числа, тип `Int` — ограниченные интервалом  $[-2^{29}, 2^{29} - 1]$ ) для оптимизации вычислений. Во-вторых, теперь функция `convert` принимает два параметра — первым параметром она принимает значение основания, в систему счисления по которому необходимо преобразовать второй параметр. Как видно, определение функции стало не намного сложнее. Ну и в-третьих, в первом клозе определения на месте первого параметра стоит так называемая *маска подстановки* (`_`), которая обозначает, что данный параметр не используется в теле функции.

Соответственно, функция `digit`, возвращающая цифру в заданном основании, теперь тоже должна получать и само основание. Но её вид, в отличие от функции `hexDigit`, которая являлась простейшим отображением первых шестнадцати чисел на соответствующие символы шестнадцатеричной системы счисления, теперь должен стать совершенно иным. Например, вот таким:

```
digit r i | r < 37    = if (i < 10)
              then show i
              else [(toEnum (i + 55))::Char]
| otherwise = "(" ++ (show i) ++ ")"
```

Здесь, в определении функции `digit`, имеются несколько интересных особенностей языка Haskell. Во-первых, вместо механизма сопоставления с образцами определение произведено через механизм *охраны* (охраных выражений), которые также позволяют сравнивать входные параметры с некоторыми значениями и осуществлять ветвление вычислений. Вторая особенность — использование выражения `if-then-else` для тех же самых целей в первом варианте. Особой разницы между этими подходами нет, вдумчивому читателю предлагается поэкспериментировать с охранными и условными выражениями (подробности синтаксиса — в специализированной литературе, рекомендуется использовать справочник [7]).

Функции `show` и `toEnum` опять же описаны в стандартном модуле `Prelude`, который подгружается в транслятор всегда. Первая функция преобразует любое значение в строку (её тип — `a -> String`), вторая — преобразует целое число в заданный тип (её тип — `Int -> a`, причём конкретно в данном случае она преобразует целое в код символа `Char`). Таким образом, алгоритм работы функции `digit` прост: если основание системы счисления меньше 37 (это число — первое число, которое на единицу больше суммы количества десятеричных цифр и букв латинского алфавита), то результирующая строка собирается из символов цифр и латинских букв. Если же основание больше или равно 37, то каждая цифра в таких системах счисления записывается как соответствующее число в десятеричной системе, взятое в круглые скобки.

Теперь можно определить несколько дополнительных функций, которые наиболее часто могут использоваться на практике:

```
int2bin = convert 2
int2oct = convert 8
int2hex = convert 16
```

Вот здесь и применяется подход, который называется *частичным применением*. В данных определениях производится частичный вызов уже определённой ранее функции `convert`, которая, как видно, ожидает на вход два параметра. Но здесь ей на вход передаётся всего один параметр, в результате чего получаются новые функции, ожидающие на вход один параметр. Этот подход проще всего понять, представив, что первый параметр функции `convert` просто подставлен во все места, где он встречается в теле функции. Так частичная подстановка `convert 2` превращает определение в:

```
convert :: Int -> Int -> String
convert 2 0 = ""
convert 2 i = convert 2 (i `div` 2) ++ digit 2 (i `mod` 2)
```

Поскольку данное определение можно легко преобразовать в функцию одного параметра (первый же теперь зафиксирован и является константой), современные трансляторы языка Haskell проводят именно такую оптимизацию, создавая дополнительное определение новой функции для частичных применений.

Осталось немного оптимизировать определение функции `convert`, чтобы изучить некоторые элементы функционального программирования, связанные с оптимизацией, улучшением внешнего вида исходного кода и т. д. Вот новое определение функции преобразования числа:

```
convert' :: Int -> Int -> String
convert' r i = convert_a r i ""
where
    convert_a _ 0 result = result
    convert_a r i result = convert_a r (i `div` r)
                           (digit r (i `mod` r) ++ result)
```

Данное определение необходимо разобрать подробно.

Функция `convert'` выполняет абсолютно то же вычисление, что и функция `convert`, однако оно основано на подходе, который называется «*накапливающий параметр*» (или «аккумулятор»). Дело в том, что в изначальном определении функции `convert` используется рекурсия, которая в некоторых случаях может приводить к неоптимальным вычислительным цепочкам. Ну и, собственно, для некоторых рекурсивных функций можно провести преобразование так, что они принимают вид «*хвостовой рекурсии*», которая может выполняться в постоянном объёме памяти.

В функциональном программировании такое преобразование делают при помощи накапливающего параметра. Определение начальной функции заменяют на вызов новой функции с накапливающим параметром, а в данном вызове передают начальное значение этого параметра. Дополнительная же функция производит вычисления как раз в накапливающем параметре, делая рекурсивный вызов самой себя в конце всех вычислений (в этом и заключается смысл хвостовой рекурсии). Соответственно, здесь видно, что локальная функция `convert_a` вызывает саму себя в самом конце вычислений, а приращение цифр в новой системе счисления производится как раз в третьем параметре, который и является накапливающим.

Особо надо обратить внимание на вид функции `convert_a`. Её определение записано непосредственно в теле функции `convert'` после ключевого слова `where`. Это — ещё один из элементов программирования, который заключается в создании локальных определений функций, или «*замыканий*». Замыкание находится в области имён основной функции, поэтому из его тела видны все параметры. Кроме того, замыкания используются для оптимизации вычислений — если в теле основной функции несколько раз

вызывать локальную функцию с одним и тем же набором параметров, то результат будет вычислен один раз.

Дополнительные приёмы программирования, описание ключевых слов, а также описание метода преобразования функции к хвостовой рекурсии можно детально изучить при помощи книги [7]. Здесь же осталось упомянуть то, что полученные функции `convert` и `convert'` можно использовать так, как любые иные.

## Заключение

Обсуждение преимуществ и недостатков тех или иных подходов к программированию — удел идеалистов. Вместе с тем знание обоих методов описания вычислительных процессов позволяет более полноценно взглянуть на проектирование и разработку программных средств. Но, к сожалению, в начальных учебных заведениях редко изучают оба подхода на уроках информатики, а потому у начинающих специалистов и интересующихся имеется известный перекос в сторону процедурного стиля.

Вместе с тем владение функциональным стилем и его основными методиками (декомпозицией и выражением ещё нерешённых задач через уже решённые) позволяет более эффективно решать не только повседневные, но и управленические задачи, поскольку эти приёмы также повсеместно встречаются в области регулирования и управления. Ввиду вышеизложенного автор надеется, что распространение и популяризация парадигмы функционального программирования позволят не только возвращать более серьёзных и вдумчивых специалистов в область информационных и автоматизированных систем, но и решит некоторые проблемы подготовки управленических кадров.

# Алгебраические типы данных в языке Haskell

*Статья была опубликована в № 12 (24) журнала «Потенциал» в декабре 2006 года.*

*Настоящее эссе продолжает собой цикл, направленный на предоставление всем желающим необходимого минимума в овладении парадигмой функционального программирования на примере языка Haskell. Пришло время более детально изучить такой немаловажный аспект программирования, как создание пользовательских типов данных, которые в языке Haskell носят наименование «алгебраических типов данных».*

## Введение

В процессе создания программного обеспечения практически всегда встаёт задача создания собственных типов данных, так как обычно базовый набор первичных типов слишком узок. Зачастую этот набор состоит только из трёх или четырёх типов, отражающих основные множества чисел в математике, а также символы, которые могут быть выведены на экран или записаны в файл.

Однако современное высокоуровневое программирование подразумевает, что сама программа должна обладать высокой степенью абстракции данных, что позволило бы не только грамотно описывать проблемную область решаемой задачи, но и повторно использовать созданные определения. Именно поэтому каждый развитый язык программирования предоставляет программисту обширное множество инструментов для определения новых типов. При этом зачастую сами новые типы можно создавать как на основе уже имеющихся, так и непосредственно, что называется «с нуля».

Не обошёл стороной эту проблему и язык Haskell — современный функциональный язык программирования. В его ядре имеется достаточно мощная и развитая система типизации, которая позволяет не только определять типы используемых в программе объектов автоматически, но и создавать собственные типы любой сложности, в том числе и с использованием *параметрического полиморфизма* (понимание параметрического полиморфизма будет предложено в одном из следующих подразделов этого раздела). Этим язык Haskell отличается от многих императивных языков программирования, в которых полиморфизм в лучшем случае представлен обычной перегрузкой имён объектов (обычно наименований функций или процедур).

Все определения, приведённые в данном разделе, можно непосредственно проверить в интерпретаторе языка Haskell — HUGS 98, который можно бесплатно получить на официальном сайте изучаемого языка в сети Интернет <http://www.haskell.org/hugs/>. Все представленные определения протестированы в этом интерпретаторе, поэтому их правильность гарантируется. Использование других трансляторов языка Haskell (например, компиляторов GHC или NHC) также возможно, однако для их использования, возможно, придётся вносить в определения функций и типов незначительные изменения.

Для понимания материала, изложенного в разделе, необходимо обладать базовыми познаниями в информатике, понимать, что такое типы данных, а также знать основной синтаксис языка Haskell.

Для изучения последнего можно воспользоваться современными книгами и справочниками на эту тему, например [6, 7, 14].

## Простые перечисления

Для определения пользовательских типов данных в языке Haskell используется служебное слово **data**, которое должно стоять на самом верхнем уровне определений в модуле. Это служебное слово определяет так называемый «*алгебраический тип данных*», выделяя для него определённое наименование, а также набор возможных конструкторов, то есть функций, которые возвращают объект определяемого типа. Все возможные конструкторы должны быть разделены символом `()` — «вертикальная черта». В каждом алгебраическом типе данных должен быть по крайней мере один конструктор.

Например, при помощи алгебраического типа данных в стандартном модуле `Prelude` определён тип для представления булевых значений истинности (можно отметить, что во многих языках программирования булевский тип является одним из примитивных типов, предоставляемых самим языком, — например, в языке Pascal это тип `Boolean`, а в языках C, C++ и многих им подобных — тип `bool`). Это сделано так:

```
data Bool = True
          | False
```

Надо особо отметить, что по соглашению о наименовании объектов в языке Haskell все наименования типов, а также их конструкторы должны начинаться с заглавной буквы. Это можно видеть в представленном определении. Такое соглашение в том числе помогает чётко разделять встречающиеся в исходном коде объекты — первая буква идентификатора свидетельствует о роде объекта.

Можно видеть, что представленное определение булевского типа в языке Haskell полностью совпадает с математическим определением этого типа, а именно:

$$\mathbb{B} = \langle \text{true}, \text{false} \rangle.$$

Здесь можно дополнительно убедиться в необычайной выразительности языка Haskell — его создатели, как обычно, стремились сделать его синтаксис наиболее приближенным к математической нотации.

Представленное определение является собой так называемое *простое перечисление*, то есть примерно то же самое, что в языках С и С++ вводится ключевым словом `enum`. Простое перечисление представляет собой неупорядоченный набор наименований объектов, принадлежащих определённому множеству, составляющему описываемый тип. Другими словами, каждое простое перечисление определяет один или более наименований объектов, которые могут принадлежать типу. При этом надо отметить, что одно из таких наименований может совпадать с наименованием типа (они находятся в разных пространствах имён).

Другим примером, который очень чётко даёт представление о том, что такое простое перечисление, является описание типа для представления различных цветов. При этом каждый цвет можно описать его названием на естественном языке (в данном случае на английском, так как синтаксис языка Haskell пока не разрешает использование символов с кодом выше 127). Это можно сделать примерно так (естественно, надо принимать во внимание, что подобным образом можно описать любые наборы цветов, которые будут отличаться друг от друга как количеством, так и наименованием конкретных элементов):

```
data Color = Black    — Чёрный
            | Blue     — Синий
            | Brown   — Коричневый
            | Cyan    — Голубой
            | Gray    — Серый
            | Green   — Зелёный
```

Magenta	— Розовый
Orange	— Оранжевый
Red	— Красный
White	— Белый
Yellow	— Жёлтый

Всё, что стоит после двух символов «дефис» (–), является комментарием, который длится до конца строки. Также надо отметить, что в языке Haskell во многих местах действует правило двумерного синтаксиса, когда определения объектов можно располагать на нескольких строчках кода, выравнивая их по одной колонке. Здесь виден именно такой способ записи, хотя это и не обязательно, так как можно записывать всё в одну строку (тогда, естественно, комментарии так ставить нельзя).

При помощи простых перечислений можно легко пояснить, что такое *сопоставление с образцом*. Можно вернуться к определению булевского типа и рассмотреть различные операции над объектами этого типа. Из формальной науки известно множество таких операций — отрицание, конъюнкция, дизъюнкция, исключающее «или», эквивалентность, импликация и многие другие. Вот, к примеру, определение функции для вычисления отрицания:

```
not :: Bool -> Bool
not True = False
not False = True
```

В этом определении, состоящем из трёх строк, первая строка представляет собой ограничение, накладываемое на тип функции `not` (иначе называемое *сигнатурой*). Эта запись говорит интерпретатору, что функция `not` принимает на вход один аргумент типа `Bool` и возвращает значение этого же типа. Вторая строка определяет результат функции `not`, если ей на вход подано значение `True`, а третья — для значения `False` соответственно.

Образцы представляют собой наборы входных значений в определениях функций, с которыми сравниваются фактические параметры, поданные на вход функций. Наборы образцов в идеальном случае должны покрывать всё множество возможных входных параметров. Сопоставление с образцом, то есть выбор конкретной строки набора определений, происходит просто — интерпретатор движется сверху вниз по определениям и пытается сопоставить фактический входной параметр с тем, что представлен в образце. Как только такое сопоставление происходит успешно, выбирается эта строка и вычисляется результат. Так, в приведённом выше определении вторая строка выбирается в случае, если на вход функции `not` подано значение `True` и т. д.

Естественно, что образцы могут быть намного сложнее, ибо для бесконечных множеств (типов) сложно перечислить все возможные варианты значений. Поэтому часто используется так называемая маска подстановки, обозначаемая символом `_` «подчёркивание». Этот символ обозначает любое значение любого типа. Например, при помощи него можно определить операцию конъюнкции на булевых значениях:

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_     && _     = False
```

В данном определении показано, что значение конъюнкции равно `True` только в случае, если оба входных аргумента равны `True`, а в противном случае (и это уже не важно, какие конкретно там значения в каких комбинациях) значением конъюнкции будет `False`. Примерно такое же определение можно привести и для дизъюнкции:

```
(||) :: Bool -> Bool -> Bool
False || False = False
_      || _      = True
```

Как видно, всё довольно просто. Читателю предлагается самостоятельно разработать функции для вычисления исключающего «или» (наименование —  $(|+|)$ ), эквивалентности (наименование —  $(==)$ ) и импликации (наименование —  $(==>)$ ). Определения этих функций можно выразить как при помощи уже имеющихся, так и создать на основе механизма сопоставления с образцом (что более предпочтительно ввиду возможности дополнительного закрепления пройденного материала).

Остаётся предложить вдумчивому читателю для изучения ещё один тип данных, представляющий собой отображение значений *троичной логики*. Эта логика используется в тех случаях, когда значением истинности могут выступать не только полностью определённые значения «ИСТИНА» и «ЛОЖЬ», но и третье значение — «НЕ ОПРЕДЕЛЕНО». Определение этого типа может выглядеть так:

```
data Ternary = TTrue
  | TUdefined
  | TFalse
```

В конструкторах этого типа первой поставлен символ Т в целях исключения конфликта с конструкторами типа `Bool`. Опять же читателю предлагается самостоятельно создать операции для работы с этим типом данных. При необходимости надо обратиться к дополнительным источникам для понимания того, как работает троичная логика (например, книги [10, 12] дают отличное понимание троичной и других многозначных логик).

## Параметризация

Однако было бы странным полагать, что все возможности языка Haskell для определения типов ограничивались бы простыми перечислениями. Конечно, они предоставляют программисту хороший инструмент для работы, но в природе существуют более сложные объекты, чем перечислимые множества. Например, для представления точек на двумерной плоскости используются две координаты, представляющие собой действительные числа, а в трёхмерном пространстве — соответственно три действительных числа.

Для решения этой и многих других подобных задач имеется возможность *параметризации* алгебраических типов данных. Такая параметризация делается при помощи перечисления после имён конструкторов типа дополнительных типов, от которых зависит определяемый тип. Например, для представления двумерных координат можно использовать такое определение:

```
data Point2D = Point2D Float Float
```

Это определение обозначает, что для получения объекта типа `Point2D` в его конструктор, который называется также, необходимо передать в качестве параметров два действительных числа (тип `Float`, представляющий действительные числа одинарной точности, является встроенным примитивным типом в языке Haskell).

Точно так же можно определить и тип данных для представления трёхмерных точек:

```
data Point3D = Point3D Float Float Float
```

Вполне естественно, что механизм сопоставления с образцами в данном случае работает абсолютно так же, как и в случае простых перечислений. В качестве образцов необходимо указывать конструкторы алгебраических типов данных и множество возможных значений, которыми параметризуется этот тип. Например, функция для сдвига точки в двумерном пространстве на определённое смещение по обеим осям координат может быть определена так:

```
shift2D :: Point2D -> Float -> Float -> Point2D
shift2D (Point2D x y) dx dy = Point2D (x + dx) (y + dy)
```

Точно так же может быть определена функция и для сдвига в трёхмерном пространстве, однако в этом случае будут две практически одинаковые функции. Ничто не мешает определить единственный

типа для представления точек как на плоскости, так и в трёхмерном пространстве. Это можно сделать, объединив два вышеприведённых определения:

```
data Point = Point2D Float Float
           | Point3D Float Float Float
```

Тогда для подобного типа данных определение функции для сдвига точки будет выглядеть так:

```
shift :: Point -> Point -> Point
shift (Point2D x y)
      (Point2D dx dy) = Point2D (x + dx) (y + dy)
shift (Point3D x y z)
      (Point3D dx dy dz) = Point3D (x + dx) (y + dy) (z + dz)
```

Однако представленное определение типа не так красиво, как кажется на первый взгляд. Оно не универсально, а ограничивается только точками на плоскости и в трёхмерном пространстве. Что делать, если вдруг понадобится решить какую-либо задачу в четырёхмерном пространстве? А в  $n$ -мерном? В самом деле, не перечислять же в списке конструкторов все возможные размерности. В этом случае на помощь приходит список, чья длина заранее не определена. Поэтому сам тип для представления точек можно определить так:

```
data Point = Point [Float]
```

Такое определение обозначает, что тип `Point` параметризуется списком действительных чисел, а длина списка определяет размерность описываемого этим типом пространства. В этом решении, однако, остаётся одна возможность для множества логических ошибок, которые могут прокрасться в программу, — у списков нет заранее указанной длины, поэтому в функциях, которые будут работать с такими точками, необходимо очень внимательно следить за тем, чтобы размерности точек совпадали. Это можно сделать, либо сравнивая длины получаемых на входе списков координат и выдавая сообщение об ошибке при их несовпадении, либо выбирая минимальную длину и отсекая все координаты, лежащие далее такой минимальной длины. По этому пути решено пойти при создании нового определения функции `shift`:

```
shift :: Point -> Point -> Point
shift (Point []) _ = Point []
shift _ (Point []) = Point []
shift (Point (x:xs)) (Point (dx:dxs)) = Point ((x + dx):others)
  where
    Point others = shift (Point xs) (Point dxs)
```

Это определение уже не так очевидно и выразительно, однако работает на точках любой размерности, даже нулевой (можно попробовать осуществить вызов `shift (Point []) (Point [])` — в результате будет выдан результат `Point []`).

Вполне естественно, что ничто не ограничивает программиста использовать в одном определении алгебраического типа данных как конструкторы без параметров (именно такие конструкторы используются в простых перечислениях), так и конструкторы с параметрами. Так, к примеру, можно заново определить тип для представления цветов, добавив дополнительный конструктор, принимающий на вход три целочисленных аргумента и возвращающий цвет в формате RGB (RGB — одна из цветовых систем, используемая для кодирования цвета в компьютерах; представляет собой три целочисленные координаты в цветовом пространстве, соответствующие насыщенности красного `Red`, зелёного `Green` и синего `Blue` цветов). Тогда определение типа `Color` будет выглядеть уже следующим образом:

```
data Color = Black      — Чёрный
            | Blue       — Синий
            | Brown     — Коричневый
```

Cyan	— Голубой
Gray	— Серый
Green	— Зелёный
Magenta	— Розовый
Orange	— Оранжевый
Red	— Красный
White	— Белый
Yellow	— Жёлтый
RGB Int Int Int	— RGB

Остается отметить, что в качестве типов, которыми параметризуются создаваемые алгебраические типы, могут быть как примитивные типы, встроенные в интерпретатор, так и типы, созданные пользователем.

## Параметрический полиморфизм

Но даже такой мощный механизм, как параметризация типов, не может обеспечить всех потребностей в создании новых типов данных. Ведь иногда имеется необходимость создать *контейнерный тип* (контейнерным называется такой тип данных, который содержит внутри себя объекты иных типов, в том числе и другие объекты контейнерных типов, иначе называемые просто «контейнерами»; например, список — это контейнерный тип данных), внутри которого могут содержаться объекты любого типа. Перечислять все такие типы в конструкторах нецелесообразно, так как самих типов может быть огромное число, да и невозможно заранее предусмотреть, объекты каких типов захочет кто-либо положить в контейнер. Здесь на помощь приходит *параметрический полиморфизм*, иногда называемый истинным полиморфизмом данных.

Параметрический полиморфизм в применении к алгебраическим типам данных в языке Haskell заключается в том, что в конструкторах типов могут употребляться так называемые параметрические переменные типов, которые обычно обозначаются строчными буквами из начала латинского алфавита. Такие переменные могут обозначать любой тип, а самих переменных может быть столько, сколько необходимо программисту. Единственная особенность заключается в том, что все используемые параметрические переменные должны быть перечислены после наименования типа.

В качестве примера можно привести тип для представления бинарных деревьев, в вершинах которых могут находиться метки любого типа. Этот тип меток должен быть одинаков для всех вершин дерева, но при помощи параметрического полиморфизма, имея единственный конструктор для бинарного дерева, можно создавать такие деревья с метками различных типов в вершинах. Определение такого типа может выглядеть так:

```
data BTree a = Empty
             | Node (a, BTree a, BTree a)
```

Первый конструктор `Empty` определяет пустое дерево. Второй конструктор `Node` определяет вершину, на которой стоит пометка некоторого типа `a` и которая имеет два дочерних поддерева с метками того же типа. Как уже было сказано, на месте параметрической переменной `a` может стоять любой тип данных. Например, если имеется необходимость в существовании деревьев, метки в вершинах которых представляют собой целые числа, то тип таких деревьев должен быть `BTree Int` и т. п.

Другими словами, параметрический полиморфизм позволяет определять наиболее общие типы данных, в которых конкретные типы не определены, но имеются параметрические переменные типов, которые по соглашению об именовании объектов в языке Haskell должны начинаться со строчной буквы. Это — достаточно мощный механизм, который позволяет достигать наибольшей степени абстракции при определении типов.

Естественно, что параметризованные подобным образом алгебраические типы всё так же могут участвовать в процессе сопоставления с образцами, который используется при вычислении значений функций. Образцы в данном случае ничем не отличаются от рассмотренных ранее, а в типах функций появляются такие же параметрические переменные типов. Для изучения этого аспекта можно рассмотреть несколько функций, работающих с бинарными деревьями.

Первая функция называется `depth` — она вычисляет максимальную глубину дерева, то есть наибольшую длину из всех путей, которые можно проложить от корневой вершины дерева к его конечным листьевым вершинам. Определение этой функции выглядит так:

```
depth :: (Num a, Ord a) => BTREE b -> a
depth Empty = 0
depth (Node (_, left, right)) = 1 + max (depth left) (depth right)
```

Вторая функция подсчитывает общее количество вершин в заданном дереве. Её определение практически такое же, как и у функции `depth` (по крайней мере, оно построено на тех же самых принципах обхода дерева):

```
count :: Num a => BTREE b -> a
count Empty = 0
count (Node (_, left, right)) = 1 + count left + count right
```

Третья функция возвращает список меток в вершинах дерева при обходе этого дерева по схеме «левый — корень — правый». Её определение уже немногим более интересное:

```
flatten :: BTREE a -> [a]
flatten Empty = []
flatten (Node (x, left, right)) = flatten left ++ [x] ++ flatten right
```

Наконец, самой интересной является функция, которая принимает на вход другую функцию и некоторое бинарное дерево, а возвращает другое бинарное дерево, ко всем меткам в вершинах которого применена заданная в качестве первого аргумента функция. Это — аналог функции `map` для списков:

```
mapBTREE :: (a -> b) -> BTREE a -> BTREE b
mapBTREE _ Empty = Empty
mapBTREE f (Node (x, left, right)) = Node (f x,
                                              (mapBTREE f left),
                                              (mapBTREE f right))
```

Как видно, функция `mapBTREE` является *функцией высшего порядка*, которая принимает на вход другую функцию в качестве своего первого аргумента. Это позволяет решать при помощи такой функции достаточно широкий набор задач, связанных с преобразованием деревьев.

Таким образом, параметрический полиморфизм является весьма мощным механизмом, который при использовании в определениях типов данных позволяет решить практически любую задачу по описанию сущностей любых проблемных областей. Понимание того, как работают полиморфные типы, помогает более полноценно использовать все механизмы, которые предоставляет программисту язык Haskell. Поэтому при изучении этого мощного языка необходимо достаточное внимание уделить именно этой теме.

## Заключение

Умение грамотно описывать проблемную область исследуемой задачи — один из главных аспектов технологии программирования, какая бы парадигма и стиль при этом ни использовались бы. Разработчик, который может построить оптимальное определение типов для объектов и связей между ними, то есть

для тех сущностей, которыми оперирует программа, дорогое стоит. Поэтому при изучении того или иного языка программирования необходимо очень внимательно подходить к пониманию способов определения типов.

В следующем разделе будет показано слияние функциональной и объектно-ориентированной парадигм программирования в языке Haskell, а также способы написания объектно-ориентированных программ на этом замечательном языке программирования. Кроме того, будут показаны дополнительные аспекты работы с алгебраическими типами данных в языке Haskell.

# Объектно-ориентированное и функциональное программирование

*Статья была опубликована в № 02 (26) журнала «Потенциал» в феврале 2007 года.*

*Данное эссе рассматривает вопросы слияния двух наиболее интересных парадигм программирования — объектно-ориентированной и функциональной, которые наиболее массово используются в настоящее время в прикладной области. Примеры применения языком объектно-ориентированного программирования приводятся на языке Haskell.*

## Введение

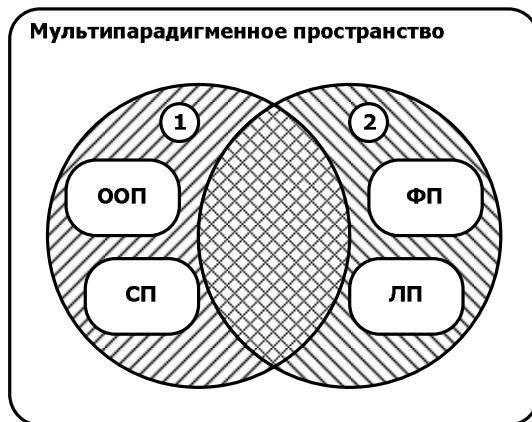
С момента рождения науки о вычислительных процессах в её прикладных аспектах было разработано достаточно широкое множество парадигм программирования, которые использовались и до сих пор используются в различных областях компьютерной науки. Под парадигмой (от греческого слова παράδειγμα — пример, модель, образец) здесь и далее понимается общая концептуальная схема постановки проблем и методов их решения. В более узком смысле под парадигмой программирования будет пониматься не просто стиль написания программ, а способ мышления, который позволяет использовать тот или иной стиль при создании таких программ.

При этом часто так получалось, что новая парадигма разрабатывалась для решения новых классов задач. Но всё разнообразие идей можно разделить на два больших класса — императивное и декларативное программирование. Внутри же этих классов рассматриваются отдельные парадигмы, которые могут вполне и пересекаться друг с другом. Да и сами эти направления также пересекаются — некоторые задачи можно легко решить как с помощью императивного, так и с помощью декларативного стилей.

В принципе, все разработанные парадигмы программирования можно свести к единой схеме, которая будет включать в себя области в так называемом «мультипарадигменном» пространстве, отвечающие за ту или иную конкретную парадигму. Такие области будут пересекаться, и все их можно разделить в соответствии с упомянутым ранее принципом императивности или декларативности. Схематически подобное пространство показано на рис. 3.

На указанной схеме цифрой 1 указано именно императивное программирование, которое характеризуется тем, что написанные на императивном стиле программы как бы предписывают ЭВМ выполнять некоторые алгоритмические действия (от лат. *imperativus* — повелительный). С другой стороны (под цифрой 2) находится декларативное программирование, характеризующееся описательным подходом к построению программ (от лат. *declarativus* — описательный). Декларативные программы описывают способ решения задачи в терминах проблемной области, а ЭВМ сама выбирает способ нахождения решения.

Во всём множестве парадигм выделяются две, которые используются особенно часто. Первая — объектно-ориентированная парадигма программирования, которая нашла самое широкое применение в прикладных областях, связанных с разработкой программного обеспечения любой сложности.



**Рис. 3. Схематическое изображение «мультипарадигменного» пространства**

Вторая — функциональное программирование, ставшее основным инструментом в научных исследованиях по компьютерной науке. Первый стиль оперирует объектами и их классами, при помощи которых описываются проблемная область и все взаимосвязи между сущностями в ней. Второй стиль использует функции для представления вычислительных процессов, так как функции являются естественным способом описания преобразования входных параметров в выходной результат.

В данном разделе описывается слияние объектно-ориентированной и функциональной парадигм на примере функционального языка Haskell, в котором имеются достаточные средства для выражения всех принятых в объектно-ориентированном стиле конструкций и идиом.

## Именованные поля и структуры

В одном из предыдущих разделов «Алгебраические типы данных в языке Haskell» начато рассмотрение различных способов определения пользовательских типов данных. Однако внимательный читатель должен был задаться вопросом: «Имеется ли в языке Haskell механизм для организации некоторых типов, соответствующих записям или структурам?» Действительно, в предыдущем разделе вопрос организации структур (то есть таких типов данных, к полям которых можно обращаться по определённым именам, — такие типы данных определяются в языках C, C++ и им подобных при помощи ключевого слова `struct`) рассмотрен не был, и могло сложиться впечатление, что таких механизмов в языке Haskell просто нет.

Но разработчики языка Haskell решили дать программисту возможность определять структуры для своих нужд. И хотя их определение выглядит весьма традиционным, но их внутренняя реализация и использование достаточно сильно отличаются от структур в императивных языках программирования. Ничего не поделаешь — функциональный стиль накладывает свои ограничения на все аспекты деятельности разработчика программного обеспечения.

Однако стоит рассмотреть пример, чтобы понять принципы и способы использования в языке Haskell алгебраических типов данных с *именованными полями*. В качестве такого примера очень хорошо подойдёт описание типов для работы с различными геометрическими фигурами, которые задаются своими координатами и некоторыми иными параметрами. Заодно такой пример поможет вспомнить основы геометрии.

Для начала необходимо определить тип, который будет представлять собой точку на плоскости. Это базовый тип для любых геометрических фигур («базовый» не в смысле объектно-ориентированного программирования, а такой, на основе которого строятся все остальные фигуры). Точка характеризуется наличием двух координат — *x* и *y*, причём они имеют одинаковый тип, который позволяет производить над координатами арифметические операции и вычислять тригонометрические функции (естественно, для преобразования координат на плоскости). Всё это можно описать на языке Haskell достаточно просто:

```
data Floating a => Point a
= Point
{
  x :: a,
  y :: a
}
```

Как видно, в этой записи определяется алгебраический тип данных с одним конструктором `Point`, который параметризуется типом `a`, на который, в свою очередь, накладывается ограничение — он должен иметь возможность участвовать в арифметических операциях и тригонометрических функциях в качестве типа операндов (ограничение записано как `Floating a =>` — такие конструкции подробно описываются в следующем разделе).

Другими словами, для определения структуры с именованными полями данных используется абсолютно такой же механизм создания алгебраических типов данных. Единственное отличие — определённый синтаксис, который позволяет давать полям требуемые имена. На самом деле запись `Point { x :: a, y :: a }` абсолютно тождественна традиционной: `Point a a`. Дополнительные имена `x` и `y` заставляют транслятор языка Haskell создать две одноимённые функции для доступа к определённым полям. Поэтому такое определение неявно обуславливает наличие двух функций, находящихся на верхнем уровне иерархии объектов в модуле:

```
x :: Fractional a => Point a -> a
x (Point v1 v2) = v1
```

```
y :: Fractional a => Point a -> a
y (Point v1 v2) = v2
```

Но и это ещё не всё. При помощи этих имён можно менять определённые поля в созданной структуре. Конечно, согласно принципам парадигмы функционального программирования, в памяти будет создан новый объект с изменённым значением одного или нескольких полей, но с точки зрения программиста это будет выглядеть примерно так:

```
shiftByX :: Fractional a => Point a -> a -> Point a
shiftByX (Point v _) dX = Point { x = v + dX }
```

```
shiftByY :: Fractional a => Point a -> a -> Point a
shiftByY (Point _ v) dY = Point { y = v + dY }
```

Здесь надо отметить, что при использовании записи новых значений в некоторые поля данных остальные поля получат значение `undefined` («не определено» —  $\perp$ ), поэтому вызов `x (shiftByY (Point 0 0) 1)` выдаст ошибку, а вызов `y (shiftByY (Point 0 0) 1)` вернёт значение 1. Здесь надо быть очень внимательным при программировании. Для того чтобы избежать таких ошибок, достаточно использовать именованные образцы, для которых уже изменять значения определённых полей. В этом случае значения других полей останутся неизменными, а не получат значение `undefined`:

```
shiftByX :: Fractional a => Point a -> a -> Point a
shiftByX p@(Point v _) dX = p { x = v + dX }
```

```
shiftByY :: Fractional a => Point a -> a -> Point a
shiftByY p@(Point _ v) dY = p { y = v + dY }
```

Но при этом остаётся возможность пользоваться обычным способом записи новых значений в поля структуры:

```
shift :: Fractional a => Point a -> a -> a -> Point a
```

```
shift (Point v1 v2) dX dY = Point (v1 + dX) (v2 + dY)
```

Данные функции являются примером того, как можно работать с вновь созданным типом данных, представляющим собой точку. Однако сам тип готов, чтобы на его основе сделать тип данных, представляющий собой любую геометрическую фигуру. Надо заметить, что для алгебраических типов с именованными полями действуют те же самые правила определения типов, поэтому вполне можно сделать что-то типа следующего:

```
data Floating a => Figure a
= Circle
{
    center :: Point a,
    radius :: a
}
| Rectangle
{
    anchor :: Point a,
    width :: a,
    height :: a
}
| Polygon
{
    points :: [Point a]
}
```

Однако такая запись не очень хороша с точки зрения масштабируемости решения. Если потребуется добавить новое описание какой-либо геометрической фигуры, то придётся менять весь тип, что приведёт к необходимости перекомпиляции проекта. Поэтому лучше сделать несколько типов, набор которых можно дополнять по мере необходимости, в том числе и из внешних модулей, описывая в них новые геометрические фигуры. Таким образом, первоначальный набор фигур таков:

```
data Floating a => Circle a
= Circle
{
    center :: Point a,
    radius :: a
}

data Floating a => Rectangle a
= Rectangle
{
    anchor :: Point a,
    width :: a,
    height :: a
}

data Floating a => Polygon a
= Polygon
{
    points :: [Point a]
}
```

Но теперь возникает желание применить к объектам этих типов такую же операцию для сдвига `shift`. Просто так это не получится, так как сигнатура функции `shift` позволяет ей работать только с объектами типа `Point a`. Придётся писать отдельные методы с уникальными именами для каждой геометрической фигуры. Но ведь этого так не хочется делать! На сцену выходят такие сущности языка Haskell, как *классы типов*.

## Классы типов

Необходимо сразу отметить, что в языке Haskell под понятием «класс» понимается нечто иное, нежели в объектно-ориентированном подходе к программированию. Класс в языке Haskell — это более абстрактное понятие, которое относится к системе типов языка программирования. Класс — это не тип данных, это «тип типов». Экземплярами класса могут быть типы данных.

Другая точка зрения на класс заключается в том, что под классом в языке Haskell понимается интерфейс, который специфицирует определённый набор методов для работы с некоторой структурой данных. Такой подход имеет право на существование, так как в действительности при определении классов происходит описание функций, которые необходимо определить для тех типов, которые впоследствии станут экземплярами создаваемого класса.

В этом и заключается ключ к решению тех проблем, которые поставлены в предыдущем разделе. Класс определяет имена функций, которые будут оперировать с объектами определённых типов, причём достаточно все эти типы определить экземплярами класса. Прежде чем перейти к созданию класса для описания действий, которые могут быть предприняты над геометрическими фигурами, стоит рассмотреть пример из стандартного модуля `Prelude` языка Haskell, чтобы более чётко понять то, что такое классы.

В качестве примера можно рассмотреть базовые арифметические операции. В математике для их обозначения традиционно используются символы  $(+)$ ,  $(-)$  и  $(\times)$  (при этом операция деления  $(:)$  не рассматривается, так как она не является замкнутой относительно множества целых чисел, а сейчас важно рассмотреть именно целые числа в качестве обучающего примера). Но применять арифметические операции можно над операндами разных типов — натуральные числа, целые числа, действительные числа и т. д. В «чистой» математике учёный редко задумывается над тем, что эти объекты разных типов, однако программист об этом помнит всегда. Но, естественно, в силу традиции хочется обозначать арифметические операции одинаково для всех типов чисел.

В разных языках программирования для этих целей используются разные механизмы. В большинстве базовые арифметические операции и перегрузка их имён «вшиты» в транслятор языка. Но создатели языка Haskell пошли иным путём. Для этих целей в стандартном модуле `Prelude` описан класс `Num`, представляющий собой класс типов, объекты которых «похожи» на целые числа. Над объектами этих чисел можно производить базовые арифметические операции, целочисленное деление, взятие остатка от деления, а также несколько других функций. Определение этого класса выглядит так:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs, signum    :: a -> a
  fromInteger   :: Integer -> a
  fromInt       :: Int -> a

  x - y      = x + negate y
  negate x   = 0 - x
  fromInt    = fromIntegral
```

Как видно, определение класса начинается с ключевого слова `class`, после которого идёт секция со спецификацией имени класса (о ней позже) до ключевого слова `where`. После этого ключевого слова идёт перечисление методов класса, а также описание взаимосвязей методов друг с другом (при необходимости). Методы описываются просто — после имени стоит символ `(::)` «имеет тип», после которого описывается сигнатура соответствующего метода. Методы с одинаковыми сигнатурами можно перечислять через запятую.

Секция с описанием взаимосвязей записывается при необходимости и возможности выразить один метод класса через другой. Так, в приведённом примере вычитание одного числа из другого можно выразить через сложение и функцию `negate` (отрицание). Естественно, что такие взаимосвязи можно описать не всегда.

Спецификация имени класса состоит из ограничения на переменные типов (необязательная часть), собственно имени класса и переменной, которая обозначает будущие экземпляры описываемого класса (в рассматриваемом примере — `a`). Ограничение на тип `a` задаётся при помощи перечисления классов, экземпляром которых должен быть тип `a`, написанный до символа `(=>)`. Если таких ограничений несколько, то их надо перечислить через запятую и заключить в круглые скобки.

Таким образом, приведённое определение класса `Num` можно читать следующим образом: «Класс `Num` специфицирует для некоторого типа `a`, который обязан являться экземпляром классов `Eq` и `Show`, методы `(+)`, `(-)`, `(*)`, `negate`, `abs`, `signum`, `fromInteger`, `fromInt`, которые имеют заданные сигнатуры. Кроме того, для методов `(-)`, `negate` и `fromInt` определяются выражения через иные методы этого класса и прочие функции». Такой класс можно использовать для определения методов над элементами определённого типа, которые предназначены для сложения, вычитания и т. д. Это могут быть любые элементы — всё зависит только от фантазии разработчика.

Теперь можно возвратиться к примеру с геометрическими фигурами. К любой фигуре, независимо от её фактического типа, могут применяться так называемые афинные преобразования, то есть она может перемещаться, масштабироваться и вращаться относительно какой-то точки. Поэтому можно определить три метода: `shift`, `scale` и `rotate` — в классе `Figure`:

```
class Figure f where
    shift  :: Floating a => f a -> a -> a -> f a
    scale   :: Floating a => f a -> Point a -> a -> f a
    rotate  :: Floating a => f a -> Point a -> a -> f a
```

К сожалению, эти методы нельзя выразить друг через друга, поэтому здесь описаны только сигнатуры этих методов. Сигнатуры могут помочь понять, что должны делать сами методы. Так, к примеру, метод `shift` получает на вход описание фигуры и два коэффициента сдвига, а возвращает новую фигуру. Методы `scale` и `rotate` получают на вход фигуру, точку, относительно которой производится преобразование, и коэффициент преобразования (масштаб и угол соответственно), а возвращают опять-таки новую фигуру.

Теперь остаётся объявить все определённые ранее алгебраические типы данных для представления геометрических фигур экземплярами нового класса, ибо само по себе определение класса `Figure` не создаёт никаких функций.

## Экземпляры классов

Как уже сказано, само по себе создание класса не влечёт ничего, что влияет на исполнение программы. Для создания функций, сигнатуры которых описаны в классе и которые бы работали с определёнными типами данных, необходимо определить такие данные в качестве экземпляров соответствующего класса.

Для определения экземпляра необходимо воспользоваться ключевым словом `instance`. После этого ключевого слова указывается спецификация экземпляра, которая практически тождественна такой же спецификации для классов, за исключением того, что в качестве специфицируемого типа указывается

не абстрактная переменная типов, а тот тип, для которого определяется экземпляр. После ключевого слова `where` перечисляются определения методов класса для конкретного типа. Например, для типа `Point` нужно определить экземпляр класса `Figure`, это делается примерно следующим образом:

```
instance Figure Point where
    shift (Point x y) dX dY = Point (x + dX)
                                         (y + dY)

    scale (Point x y) (Point xa ya) k = Point (xa + k * (x - xa))
                                         (ya + k * (y - ya))

    rotate (Point x y) (Point xa ya) k = Point (xa + r * cos (alpha + k))
                                         (ya + r * sin (alpha + k))

    where
        r      = sqrt ((x - xa)^2 + (y - ya)^2)
        alpha = acos ((x - xa) / r)
```

Как видно, ничего сложного нет. Это определение гласит, что для типа `Point` определены методы класса `Figure`: `shift`, `scale` и `rotate`. После этого можно использовать данные методы на операндах типа `Point`, при этом транслятор языка Haskell самостоятельно выберет необходимый экземпляр, чтобы вызвать метод на исполнение.

Надо отметить, что сами по себе алгебраические типы данных и классы — совершенно не связанные друг с другом единицы исходного кода на языке Haskell. Экземпляры классов — это та сущность, при помощи которой происходит связывание классов и типов. Но и экземпляр — тоже независимая программная сущность, она может существовать, а может и не существовать. Существование экземпляров позволяет говорить транслятору языка Haskell, что функции могут использовать объекты определённых типов, если они соответствуют ограничениям, прописанным в сигнатурах функций.

Для того чтобы работать с описанными ранее геометрическими фигурами, необходимо создать соответствующие экземпляры класса `Figure`. При этом следует учесть, что, имея в наличии экземпляр типа `Point`, остальные экземпляры определяются уже достаточно легко. Надо обратить внимание, что для элементов типа `Point` можно использовать определённые методы `shift`, `scale` и `rotate`. Собственно, дело за малым:

```
instance Figure Circle where
    shift (Circle c r) dX dY
        = Circle (shift c dX dY) r
    scale (Circle c r) pa k
        = Circle (scale c pa k) (k * r)
    rotate (Circle c r) pa k
        = Circle (rotate c pa k) r

instance Figure Rectangle where
    shift (Rectangle a w h) dX dY
        = Rectangle (shift a dX dY) w h
    scale (Rectangle a w h) pa k
        = Rectangle (scale a pa k) (w * k) (h * k)
    rotate (Rectangle a w h) pa k
        = Rectangle (rotate a pa k) w h

instance Figure Polygon where
    shift (Polygon pts) dX dY
```

```

= Polygon (modify pts dx dy)
where
  modify [] _ _ = []
  modify (p:ps) dx dy = (shift p dx dy):(modify ps dx dy)

scale (Polygon pts) pa k
= Polygon (modify pts pa k)
where
  modify [] _ _ = []
  modify (p:ps) pa k = (scale p pa k):(modify ps pa k)

rotate (Polygon pts) pa k
= Polygon (modify pts pa k)
where
  modify [] _ _ = []
  modify (p:ps) pa k = (rotate p pa k):(modify ps pa k)

```

Всё просто, но остаётся несколько проблем больше эстетического характера. Первое, что бросается в глаза, — много практически одинаковых строк кода в определении экземпляра `Figure~Polygon`. Конечно, хотелось бы все подобные повторения свести в единую функцию `modify`, которая была бы *функцией высшего порядка* и принимала бы на вход тот метод, который необходимо применить к набору точек. Но тут не всё так просто. К сожалению, класс `Figure` спроектирован так, что у метода `shift` тип совершенно иной, нежели у остальных методов. Поэтому простая функция не поможет. Решение данной задачи оставляется вдумчивому читателю.

Вторая проблема в этом наборе определений кроется в методе `rotate` для прямоугольника `Rectangle`. Этот тип описан так, что вращать можно только опорную точку, а его стороны всегда остаются параллельны осям координат (хотя это явно и не описано — при таком описании можно предположить, что стороны вообще направлены так, как заблагорассудится, — здесь вопрос в интерпретации типа). Поэтому если есть необходимость во вращении прямоугольника в виде целостной фигуры, необходимо менять сам тип данных. Эта задача также остаётся для самостоятельного решения.

В качестве подсказки для второй задачи можно указать, что проще и лучше всего прямоугольник представлять при помощи двух опорных точек, которые не находятся на одной стороне (противоположны по диагонали). В этом случае, однако, понадобятся функции (глобальные) для вычисления длин сторон такого прямоугольника.

## Окончательные замечания

Хотя классы существуют во многих других языках программирования, понятие класса в языке Haskell несколько отличается. Это было уже показано в предыдущих разделах. Здесь же собраны и структурированы все отличия понятия «класс» в языке Haskell и в прочих объектно-ориентированных языках программирования. Итак:

1. Язык Haskell разделяет определения классов и их методов, в то время как такие языки, как C++ и Java, вместе определяют структуру данных и методы для её обработки. В определении класса в языке Haskell даётся только сигнатура методов, которые должны быть реализованы в экземплярах класса. Вместе с этим возможно написание определений таких методов, используемых по умолчанию, когда в типах-экземплярах класса реализации метода нет. Возможность реализовывать методы по умолчанию предоставляют далеко не все объектно-ориентированные языки программирования.

2. Можно сказать, что определения методов классов в языке Haskell больше всего соответствуют виртуальным функциям языка C++. Каждый конкретный экземпляр класса должен переопределять методы класса для использования их над своей собственной структурой. В случае необходимости и возможности можно воспользоваться определениями методов, определённых по умолчанию.
3. С точки зрения языка Java более всего классы в языке Haskell похожи на интерфейсы. Как и определение интерфейса, классы в языке Haskell предоставляют только протокол использования объекта вместо определения самих объектов. На это же намекает и возможность множественного наследования классов (в языке Java можно реализовать в каком-либо классе столько интерфейсов, сколько потребуется, но наследовать можно только один класс). Кроме того, типы в языке Haskell могут быть экземплярами стольких классов, скольких потребуется, — равно как и в языке Java.
4. Язык Haskell не поддерживает стиль перегрузки функции, используемый в C++, когда функции с одним и тем же именем получают данные различных типов для обработки. Вместо этого применяется технология использования синонимов имён функций, когда разные функции с различными наименованиями для обработки разных данных в определённых ситуациях могут вызываться по одному идентификатору. Это как раз и реализуется при помощи классов в языке Haskell, которые позволяют использовать в этом языке полиморфизм «ad hoc».
5. В классах языка Haskell не существует понятия контроля за доступом — нет публичных и защищённых методов, то есть нет аналогов служебных слов `public`, `private` и т. д. Все методы в классах языка Haskell являются открытыми, более того, они определены в контексте модуля, то есть являются декларациями самого высокого уровня. Однако при этом имеется возможность импортировать из модуля только те программные сущности, которые требуются для решения задачи.

Всё, что было рассмотрено в предыдущих разделах, а также в предыдущих разделах на эту тему, подводит к мысли о том, что при программировании на языке Haskell используются всего пять видов деклараций, пять сущностей: функции, типы, классы типов, экземпляры классов и модули. Все эти сущности являются самостоятельными, только опосредованно связанными друг с другом. Это особенно касается классов и их экземпляров — эти сущности не связаны ни друг с другом, ни с типами. Классы и алгебраические типы данных могут быть описаны независимо друг от друга, а экземпляр класса — это сущность, связывающая класс и тип.

То, что записывается в контексте (`Class a =>`), является всего лишь ограничением вида «должен существовать экземпляр указанного класса `Class` для заданного типа `a`». Это значит, что типы и их реализации для классов существуют раздельно.

В качестве примера можно рассмотреть такую ситуацию. Пусть некто в давние времена создал некий тип данных `D` и некоторую очень полезную функцию `f`, которая в своей сигнатуре имеет ограничение (`C a`), где `a` — тип одного из аргументов. Пусть эти сущности определены в разных модулях, которые никак не связаны друг с другом. Разработчики, создававшие эти исходные сущности, даже не догадывались о взаимном существовании. И вдруг третий разработчик понял, что ему необходимо использовать функцию `f` над объектами типа `D`. Что делать?

Нет никакой необходимости искать программиста, создавшего класс `C`, чтобы он включил в его определение возможность работать с типом `D`. Достаточно в своём модуле реализовать экземпляр этого класса для типа `D`, что позволит использовать значения этого типа в функции `f` автоматически. Исходные модули остаются нетронутыми. Описанное — серьёзное преимущество системы модулей, классов и их экземпляров языка Haskell.

## Заключение

Таким образом в результате получился некоторый модуль с описанием базовых методов и типов для проведения геометрических преобразований. Этот модуль можно использовать в качестве «затравки» для полноценной библиотеки, решающей различные геометрические задачи. При этом надо учесть, что сами определения из этого модуля являются достаточно абстрактными — они не привязаны к какой-либо предметной области. Функциям и методам классов безразлично, что обсчитывать — графические образы на экране монитора (в пикселях) либо аналитические геометрические задачи с координатами произвольной точности — везде созданные методы будут применимы. При особом желании модуль действительно можно развить в нечто большее.

Вдумчивому же читателю рекомендуется подумать и сделать подобный модуль работы с геометрическими фигурами, узловые точки которых задаются в полярных координатах. Для этого можно использовать созданный модуль, а также определение типа:

```
data Floating a => PolarPoint a
= PolarPoint
{
    angle    :: a,
    distance :: a
}
```

В стандартном модуле `Prelude` описана большая иерархия классов, которые подходят для определения типов данных различной природы — от перечислимых множеств до комплексных чисел и более сложных объектов. Можно обратиться к этому модулю для дополнительного изучения того, что представляют собой классы в языке Haskell.

# Введение в $\lambda$ -исчисление для начинающих

*Статья была подготовлена к публикации в журнале «Потенциал», но опубликована не была.*

Данное эссе описывает одно из самых интереснейших направлений дискретной математики, напрямую связанное с информатикой и программированием, — теорию вычислений. Вместе с комбинаторной логикой, которая будет рассмотрена в следующем разделе,  $\lambda$ -исчисление изучает объекты и способы их комбинирования друг с другом. В настоящем разделе рассматривается нотация  $\lambda$ -исчисления для записи комбинаторов, более приближенная к функциональному стилю.

## Введение

В своё время видный логик Берtrand Рассел пошатнул сами основы математики, придумав так называемый парадокс брадобрея, который традиционно формулируется так:

*В одном городе живёт брадобрей. Однажды он вывесил на двери своей цирюльни объявление: «Брею тех и только тех жителей города, кто не бреется сам». Сразу же у некоторых особо прозорливых жителей этого города возник резонный вопрос: «А может ли этот брадобрей побрить сам себя?»*

Попытка ответить на этот простой вопрос привела к неожиданным результатам. Ведь если перевести эту историю на строгий язык математики, то можно увидеть, что эта, казалось бы, невинная история действительно сводит на нет все математические построения. Достаточно лишь под брадобреем понимать некоторое множество, включающее в себя все прочие множества, которые не являются собственными подмножествами, как основной вопрос истории звучит следующим образом: «включает ли это множество в качестве подмножества само себя?»

Действительно, если это множество включает само себя в качестве подмножества (брадобрей бреется сам), то оно по своему определению не может включать само себя в качестве подмножества (ибо брадобрей не бреет тех, кто бреется сам). Но если оно не включает само себя в качестве подмножества (брадобрей не бреется сам), то опять же, по определению, оно должно включать в себя само себя (ведь брадобрей бреет тех, кто не бреется сам).

Это множество является противоречивым по своей сути, а потому и само понятие множества в его традиционном наивном определении является противоречивым. Наивное определение дал в своё время Георг Кантор, назвав множеством «многое, понимаемое как единое». Такое неформальное определение множества стояло в основе многих математических теорий, а потому находка этого парадокса весьма серьёзно ударила по основам математики. Но в рамках наивной теории множеств разрешить этот парадокс невозможно, поэтому после его формулировки многими математиками были сделаны попытки

формализовать теорию множеств, сделав её устойчивой к подобным *антиномиям* (так впоследствии назвали этот и схожие с ним парадоксы).

Парадокс Рассела стал причиной того, что в середине 30-х годов XX века Алонзо Чёрч начал разработку своей формализации теории множеств с целью разрешить парадокс. В результате его работы на свет появилась нотация, которая была названа им  $\lambda$ -исчисление. К сожалению, она не помогла разрешить парадокс, но, к счастью,  $\lambda$ -исчисление само по себе стало новым направлением в математике, открыв серьёзные перспективы в исследованиях вычислительных процессов.

## Неформальное описание теории

Понять  $\lambda$ -исчисление не так уж и сложно, достаточно внимательно изучить разработанную нотацию. Для этого не надо вдумываться в построенную формальную систему, а всего лишь понять неформальное описание  $\lambda$ -исчисления. Интересующийся же сутью читатель сможет легко найти строгое описание теории  $\lambda$ -исчисления в разных источниках (например, в книгах [1, 6]).

С неформальной же точки зрения  $\lambda$ -исчисление изучает формулы, построенные из  $\lambda$ -термов. Такие формулы можно подвергать трём видам преобразований, которые будут описаны далее. В свою очередь,  $\lambda$ -термы выглядят достаточно просто:

- 1) если  $x$  — некоторая переменная или константа, то  $x$  —  $\lambda$ -терм (базис индуктивного определения);
- 2) если  $M$  и  $N$  — некоторые  $\lambda$ -термы, то  $(MN)$  —  $\lambda$ -терм (терм-аппликация);
- 3) если  $x$  — переменная, а  $M$  —  $\lambda$ -терм, то  $(\lambda x.M)$  —  $\lambda$ -терм (терм-абстракция);
- 4) других  $\lambda$ -термов нет.

Как видно, первый пункт определения задаёт его базис. Второй пункт определяет приложение термов друг к другу (так называемая *аппликация*). Третий пункт определения описывает так называемую *абстракцию*, в которой переменная  $x$  связывается с  $\lambda$ -термом  $M$ .

Примеры простейших  $\lambda$ -термов:

- $x$  — простая переменная;
- $(\lambda x.x)$  —  $\lambda$ -терм тождества;
- $((\lambda x.x)y)$  — аппликация переменной  $y$  к  $\lambda$ -терму тождества

и т. д.

Данное определение весьма напоминает определение *комбинатора*. Действительно, в комбинаторной логике отсутствует абстракция, а всё остальное выглядит абсолютно так же (более подробно комбинаторная логика будет описана в следующем разделе). Но если вспомнить историю создания комбинаторной логики, то можно понять, что Хаскелл Карри при её создании руководствовался принципом удаления из рассмотрения *связанных переменных*, то есть абстракции.

Другими словами,  $\lambda$ -термы представляют собой комбинаторы, только записанные достаточно своеобразным способом. Действительно, если рассмотреть комбинаторный базис **S**, **K**, **I**, то выражение его комбинаторов через  $\lambda$ -нотацию будет выглядеть так:

$$\mathbf{S} \equiv (\lambda x.(\lambda y.(\lambda z.(xz)(yz)))) \quad (\bar{\mathbf{S}})$$

$$\mathbf{K} \equiv (\lambda x.(\lambda y.x)) \quad (\bar{\mathbf{K}})$$

$$\mathbf{I} \equiv (\lambda x.x) \quad (\bar{\mathbf{I}})$$

Честно говоря, налицо весьма сомнительное преимущество, так как воспринимаемость подобных записей далека от идеальной. В первую очередь в глаза бросается достаточно большое количество скобок, однако с ним разобраться проще всего. В  $\lambda$ -исчислении, естественно, принято соглашение о пропуске лишних скобок, которые можно восстановить по определённому правилу. Так, в аппликациях скобки восстанавливаются по ассоциативности вправо, то есть запись  $((MN)Q)$  тождественна простому  $MNQ$ . В абстракциях же, наоборот, скобки можно восстановить по ассоциации влево, а множественные символы ( $\lambda$ ) можно свести в один, тогда запись  $(\lambda x.(\lambda y.x))$  будет тождественна записи  $\lambda xy.x$ . В этом случае базовые комбинаторы в  $\lambda$ -нотации получают свой традиционный вид:

$$\mathbf{S} \equiv \lambda xyz.xz(yz) \quad (\text{S})$$

$$\mathbf{K} \equiv \lambda xy.x \quad (\text{K})$$

$$\mathbf{I} \equiv \lambda x.x \quad (\text{I})$$

Но что с этим делать дальше? Ведь упрощение записи нисколько не сказывается на применимости самой теории. Действительно, если в комбинаторной логике имелась операция *применения*, то и в  $\lambda$ -исчислении должно быть что-то похожее. Так и есть, в  $\lambda$ -исчислении имеется абсолютно такая же единственная операция, которая действует при аппликации  $\lambda$ -термов друг к другу. Если подходить неформально, то можно сказать, что любой  $\lambda$ -терм «ожидает» для применения к нему стольких объектов (других  $\lambda$ -термов), сколько связанных переменных находится в его *сигнатуре*. Так, к примеру,  $\lambda$ -терм **S** ожидает три объекта (для означивания переменных  $x$ ,  $y$  и  $z$ ),  $\lambda$ -терм **K** — два объекта и т. д. Всё, как в комбинаторной логике.

При применении некоторого объекта к  $\lambda$ -терму происходит *подстановка* применяемого объекта вместо самой первой связанной переменной. Другими словами, если к  $\lambda$ -терму **S** применить некоторый объект **O**, то в результате этого применения останется  $\lambda$ -терм:  $\lambda yz.Oz(yz)$ . То есть в случае произвольной абстракции  $(\lambda x.M)$  при применении к ней некоторого объекта **O** в результате получится новый  $\lambda$ -терм, который вычисляется при помощи замены всех вхождений переменной  $x$  в  $\lambda$ -терме  $M$  на объект **O**. Данный факт записывается следующим образом:

$$(\lambda x.M)\mathbf{O} \Rightarrow M[x \leftarrow \mathbf{O}] \quad (3)$$

Эту запись легче всего понять на простейших примерах:

- $(\lambda x.x)5 \Rightarrow x[x \leftarrow 5] \Rightarrow 5$ .
- $((\lambda xy.x + y)2)3 \Rightarrow (x + y)[x \leftarrow 2][y \leftarrow 3] \Rightarrow 2 + 3$ .
- $(\lambda x.xx)(\lambda z.zz) \Rightarrow (xx)[x \leftarrow (\lambda z.zz)] \Rightarrow (\lambda z.zz)(\lambda z.zz)$ .

И т. д.

В этом и заключается сама суть подстановки. И теперь видно некоторое отличие  $\lambda$ -исчисления от комбинаторной логики. В этой нотации ясно видно, сколько operandов имеется у каждого  $\lambda$ -терма, а также какой новый  $\lambda$ -терм и сколько operandов у него останется в результате *частичного применения*.

Другими словами, комбинаторная логика может использоваться как средство для сокращения записи  $\lambda$ -термов, а в остальном эти теории абсолютно равнозначны. То есть комбинаторная логика вводит определённые наименования для некоторых наиболее часто используемых  $\lambda$ -термов.

## Некоторые дополнения

В целях внесения разнообразия в сухую теорию можно дополнить  $\lambda$ -исчисление некоторыми не входящими в базовый алфавит символами. В первую очередь было бы весьма интересно ввести в  $\lambda$ -исчисление различные математические знаки, чтобы иметь возможность записывать в качестве  $\lambda$ -термов разные формулы. Это позволит повысить степень применимости данной теории, дав возможность записывать в виде  $\lambda$ -термов математические функции.

Например, как выразить через  $\lambda$ -нотацию обычное арифметическое сложение чисел? Достаточно просто:

$$(+)\equiv(\lambda xy.x+y) \tag{4}$$

В этом случае очень хорошо видно весьма важное свойство функционального стиля программирования — частичное применение. Ведь в школе, к примеру, изучают, что у операции сложения имеются два операнда, и для получения определённого результата необходимо передать операции сложения именно два числа. В противном случае получается уравнение, которое возможно разрешить только при некоторых условиях. В  $\lambda$ -исчислении даже при применении одного означенного операнда в формуле (4) в результате получается вполне определённая функция, выраженная при помощи  $\lambda$ -терма. Например:

$$(\lambda xy.x+y)7=\lambda y.7+y$$

Таким образом, получен новый  $\lambda$ -терм, функциональность которого заключается в том, что он прибавляет 7 к своему единственному аргументу.

Подобное можно делать с любыми математическими формулами, которые хочется оформить в виде  $\lambda$ -термов, то есть функций с возможностью частичного применения. Более того, абсолютно таким же способом можно внедрять в  $\lambda$ -исчисление разные конструкции языков программирования, чтобы иметь возможность для описания более сложных функций (например, с ветвлением или итеративными процессами). Вот как, к примеру, может выглядеть функция для вычисления абсолютного значения своего аргумента:

$$\mathbf{abs}\equiv\lambda x.\text{if }(x\geqslant 0)\text{ then }x\text{ else }(-x)$$

Необходимо особо подчеркнуть, что в «чистом»  $\lambda$ -исчислении всего этого нет, и введённые в этом разделе дополнения являются некоторой вольностью. В свою очередь, аппарат  $\lambda$ -исчисления предоставляет все возможности для кодирования различных математических объектов (натуральные числа, операции и функции над ними, булевские значения истинности, логические операции и многое, многое другое — некоторые способы кодирования будут рассмотрены в четвёртом подразделе).

## Редукция как стратегия вычислений

В начале первого подраздела упоминались три вида преобразований, которые могут применяться к  $\lambda$ -термам. По сути, один вид преобразований уже был рассмотрен — это подстановка в качестве операнда какого-либо объекта, то есть снятие уровня абстракции  $\lambda$ -терма. Пришло время более детально рассмотреть все преобразования, чтобы тоньше прочувствовать смысл  $\lambda$ -исчисления.

Каждое из рассматриваемых преобразований в рамках  $\lambda$ -исчисления называется *редукцией*. Таким образом, вводятся три типа редукции:

- 1)  $\alpha$ -редукция (иногда называемая  $\alpha$ -конверсией) — замена имён переменных с целью избежать *коллизии имён*, то есть совпадения имён переменных в  $\lambda$ -терме, в который производится подстановка (например, при применении  $(\lambda xy.x+y)y$  возникнет коллизия по переменной  $y$ , так как в результате

получится явно ошибочный  $\lambda$ -терм  $(\lambda y.y + y)$  — для того чтобы этого избежать, необходимо в исходном  $\lambda$ -терме изменить имя переменной  $y$ );

- 2)  $\beta$ -редукция — основное преобразование  $\lambda$ -термов, которое уже неоднократно рассмотрено ранее, заключается в подстановке некоторого объекта в качестве аргумента и снятии тем самым одного уровня абстракции;
- 3)  $\eta$ -редукция — дополнительное преобразование, которое не необходимо и запрещено в некоторых особых типах  $\lambda$ -исчисления. Его суть заключается в предположении о том, что  $\lambda$ -термы  $\lambda x.Mx$  и  $M$  тождественно равны. В некоторых особенных случаях это — довольно сильное предположение, но в обычном  $\lambda$ -исчислении первый  $\lambda$ -терм можно свободно заменять на второй.

Все типы редукции в формулах обозначаются при помощи стрелок с указанием типа:  $(\xrightarrow{\alpha})$ ,  $(\xrightarrow{\beta})$  и  $(\xrightarrow{\eta})$ . В этом ряду для  $\beta$ -редукции сделано исключение: так как это основной тип редукции, то её можно обозначать простой стрелкой  $(\rightarrow)$ . Несколько последовательных редукций обычно обозначаются стрелкой с двумя наконечниками:  $(\rightarrow\rightarrow)$ .

Что же представляет собой редукция? Вдумчивый читатель уже должен был понять, что это — моделирование функциональных вычислений. В  $\lambda$ -термы подставляются определённые значения, после чего производятся сами вычисления. Этот процесс происходит до получения окончательного результата (если это возможно), который представляет собой такой же  $\lambda$ -терм. Его интерпретация уже зависит от конкретной области применения теории.

Редукция вводит между  $\lambda$ -термами отношение редуцируемости, которое обладает свойствами *идемпотентности*, *симметричности* и *транзитивности*, то есть по своей сути является отношением эквивалентности. Другими словами, всё множество возможных  $\lambda$ -термов разбивается на классы эквивалентности, внутри которых  $\lambda$ -термы могут быть преобразованы друг к другу при помощи редукции. При этом во многих таких классах эквивалентности существует один  $\lambda$ -терм, который не может быть подвергнут ни  $\beta$ -, ни  $\eta$ -редукции (любой  $\lambda$ -терм может быть подвергнут  $\alpha$ -конверсии неограниченное число раз). Говорят, что такой терм находится в *нормальной форме*.

В  $\lambda$ -исчислении существует основная теорема (*теорема Чёрча-Россера*), которая утверждает, что если нормальная форма существует, то она единственна. Это — очень важная теорема, которая, в принципе, определила применимость  $\lambda$ -исчисления как теории вычислений. Доказательство этой теоремы не очень сложное, но занимает не одну страницу текста, поэтому его рассмотрение выходит за рамки данной книги.

Одним из важных следствий теоремы Чёрча-Россера является то, что при наличии нормальной формы не важно, каким образом проводились вычисления для её получения. Это предоставляет исследователям возможность разработки различных стратегий редукции, которые заключаются в выборе определённых правил, которым подчиняется процесс редукции. Изначально были разработаны две редукционные стратегии: *нормальная* и *аппликативная*.

Нормальная редукционная стратегия заключается в том, что на каждом шаге редукции выбирается самый левый и самый внешний  $\lambda$ -терм, если воспринимать его буквально и не принимать во внимание скобки. Редукция самого левого  $\lambda$ -терма предполагает, что в  $\lambda$ -выражении  $(MN)$  первым редуцируется  $\lambda$ -терм  $M$ . Редукция самого внешнего  $\lambda$ -терма предполагает, что сначала редуцируется выражение  $(\lambda x.M)N$  перед редукцией  $\lambda$ -термов  $M$  или  $N$ . Кроме того, если в процессе редукции выражения используется optionalная  $\eta$ -редукция, то она проводится после всех  $\beta$ -редукций.

Под аппликативной редукционной стратегией понимается такая стратегия, при которой на каждом шаге редукции выбирается такой  $\lambda$ -терм, внутри которого невозможно провести  $\beta$ -редукцию. Например, в  $\lambda$ -выражении  $(\lambda x.((\lambda z.zz)x))y$  первым редуцируется  $\lambda$ -терм  $(\lambda z.zz)x$ .

Доказано, что нормальная редукционная стратегия гарантирует получение нормальной формы  $\lambda$ -терма, если она существует. С другой стороны, обычно аппликативная редукционная стратегия достигает нормальной формы быстрее (хотя и не всегда), но она не гарантирует получения нормальной

формы в принципе. Например, при попытке редуцировать  $\lambda$ -терм  $(\lambda y.x)\Omega$ , где  $\Omega = (\lambda x.xx)(\lambda x.xx)$ , нормальная редукционная стратегия приведёт к получению нормальной формы, а аппликативная — нет.

В программировании нормальная редукционная стратегия соответствует *вызову по имени*. То есть аргумент выражения не вычисляется до тех пор, пока к нему не возникнет обращения в теле выражения. Аппликативная редукционная стратегия соответствует *вызову по значению*, когда перед передачей фактического параметра в функцию его значение предварительно вычисляется. Дополнительно о стратегиях вычислений можно прочитать в [9, 20].

Остается рассмотреть несколько примеров проведения редукции различных  $\lambda$ -термов (редуцируемый на каждом шаге  $\lambda$ -терм подчёркивается для наглядности)...

**Редукция  $\lambda$ -терма  $(\lambda x.xyx)((\lambda z.z)w)$ :**

1) Нормальная редукционная стратегия:

$$\begin{aligned}
\underline{(\lambda x. xyxx)((\lambda z.z)w)} &\rightarrow \\
&\rightarrow \underline{((\lambda z.z)w)y((\lambda z.z)w)((\lambda z.z)w)} \rightarrow \\
&\rightarrow wy\underline{((\lambda z.z)w)}((\lambda z.z)w) \rightarrow \\
&\rightarrow wyw\underline{((\lambda z.z)w)} \rightarrow \\
&\hspace{10em} \rightarrow wyww.
\end{aligned}$$

## 2) Аппликативная редукционная стратегия:

$$\begin{aligned}
 (\lambda x. xyxx) \underline{((\lambda z.z)w)} &\rightarrow \\
 \rightarrow \underline{(\lambda x. xyxx)w} &\rightarrow \\
 \rightarrow wyw.
 \end{aligned}$$

**Редукция  $\lambda$ -терма**  $(\lambda xyz.xz(yz))(\lambda x.xx)(\lambda x.xx)z$ :

#### 1) Нормальная редукционная стратегия:

$$\begin{aligned}
\underline{(\lambda xyz.xz(yz))(\lambda x.xx)(\lambda x.xx)z} &\rightarrow \\
&\rightarrow \underline{(\lambda yz.(\lambda x.xx)z(yz))(\lambda x.xx)z} \rightarrow \\
&\rightarrow \underline{(\lambda z.(\lambda x.xx)z((\lambda x.xx)z))z} \rightarrow \\
&\rightarrow \underline{(\lambda x.xx)z((\lambda x.xx)z)} \rightarrow \\
&\rightarrow zz(\underline{(\lambda x.xx)z}) \rightarrow \\
&\hspace{10em} \rightarrow zz(zz).
\end{aligned}$$

## 2) Аппликативная редукционная стратегия:

**Редукция  $\lambda$ -терма  $(\lambda y.x)\Omega$ :**

1) *Нормальная редукционная стратегия:*

$$\begin{aligned} (\lambda y.x)\Omega &\rightarrow \\ &\rightarrow (\lambda y.x)[y \leftarrow \Omega] \rightarrow \\ &\quad \rightarrow x. \end{aligned}$$

2) *Аппликативная редукционная стратегия:*

$$\begin{aligned} (\lambda y.x)\Omega &\rightarrow \\ &\rightarrow (\lambda y.x)((\lambda x.xx)(\lambda x.xx)) \rightarrow \\ &\rightarrow (\lambda y.x)((\lambda x.xx)(\lambda x.xx)) \rightarrow \\ &\quad \rightarrow \dots \end{aligned}$$

Последний пример как раз и показывает, что нормальная редукционная стратегия позволяет получить нормальную форму, в то время как аппликативная стратегия «зацикливается» на редукции одного и того же  $\lambda$ -терма.

## Примеры кодирования данных и функций

Как уже говорилось, аппарат  $\lambda$ -исчисления позволяет внутри весьма ограниченных рамок кодировать различные типы данных и операции над ними. В следующем разделе, который посвящён комбинаторной логике, будут приведены расширенные примеры такого кодирования при помощи комбинаторов. Здесь же можно привести лишь некоторые простые примеры кодирования информации посредством  $\lambda$ -термов. Однако из-за того, что в  $\lambda$ -исчислении используются связанные переменные, многое кодируется несколько иным способом. Для этих целей вполне достаточно рассмотреть только *нумералы Чёрча*, так как этот формализм помогает полностью понять то, как при помощи  $\lambda$ -исчисления кодируются различные данные и функции для их обработки.

После того как стало ясно, что  $\lambda$ -исчисление является весьма интересной наукой, Алонзо Чёрч начал изучение различных аспектов этого направления научной мысли. В первую очередь он попытался выразить через  $\lambda$ -термы различные математические объекты. Одним из базовых объектов изучения математики является натуральное число, поэтому была произведена попытка закодировать такие числа.

Необходимо напомнить, что для кодирования чисел или подобных им объектов можно использовать любой способ выражения, главное, чтобы потом на построенных  $\lambda$ -термах можно было бы определить приемлемые операции, тождественные тем, что имеются для чисел. Поэтому, собственно, способов кодирования чисел существует множество. Первым же на этом поприще был А. Чёрч, создав следующие определения:

$$\begin{aligned} \bar{0} &\equiv \lambda f x. x. \\ \bar{1} &\equiv \lambda f x. f x. \\ \bar{2} &\equiv \lambda f x. f(f x). \\ &\dots \\ \bar{n} &\equiv \lambda f x. \underbrace{f(\dots(f}_{n \text{ раз}} x \dots)). \end{aligned}$$

Каждый нумерал Чёрча является итератором, который соответствующее число раз применяет функцию  $f$  к результату предыдущего применения этой функции. Поэтому здесь видно, что такие нумералы сами по себе являются функциями высшего порядка, то есть функционалами, которые работают

с другими функциями, принимая их в качестве своего первого аргумента. Это позволило сделать сами нумералы достаточно абстрактными и не связанными с какой-либо предметной областью. В каждой конкретной области применения можно будет передать таким нумералам определённую функцию, которая в конечном итоге построит само число. Например, в арифметике для построения чисел нумералам могут передаваться в качестве входных параметров функция  $(+1)$  и начальное значение переменной  $x = 0$ .

То есть видно, что нумералы Чёрча — это кодирование натуральных чисел более высокого уровня абстракции, нежели сами натуральные числа. Внутри нумералов заложена функциональность для вычисления объектов, схожих с натуральными числами, которые, в свою очередь, зависят от области применения нумералов.

В качестве базовых операций над нумералами Чёрча вводятся функции для сложения, умножения и возведения в степень. Такие функции могут быть определены непосредственно:

- 1)  $\text{add} \equiv \lambda mnfx.mf(nfx)$
- 2)  $\text{mlt} \equiv \lambda mnfx.m(nfx)x$
- 3)  $\text{exp} \equiv \lambda mnfx.nmf^x$

В этих  $\lambda$ -термах под переменными  $m$  и  $n$  понимаются операнды, на место которых ожидается подстановка двух нумералов Чёрча, участвующих в операции. Собственно, проверка этих определений тривиальна и оставляется для самостоятельной проработки (достаточно лишь убрать два уровня абстракции, подставив вместо переменных  $m$  и  $n$  два нумерала  $\bar{m}$  и  $\bar{n}$  и заодно вспомнив их определения в виде  $\lambda$ -термов). При этом надо помнить, что данные функции определены только на нумералах Чёрча. Для любых произвольных  $\lambda$ -термов  $M$  и  $N$  эти функции будут работать некорректно.

Однако этих функций обычно не хватает для работы с натуральными числами. Для более успешного использования нумералов Чёрча необходимо определить дополнительные функции, которые позволяли бы выполнять более сложные действия, например вычитание (это действие осложнено тем, что может вывести за рамки натуральных чисел, равно как и операции деления, извлечения корня и взятия логарифма).

Для того чтобы подготовиться к созданию подобных «сложных» функций, необходимо разработать вспомогательные  $\lambda$ -термы, чтобы была возможность осуществлять разного рода проверки. В первую очередь необходимы функции для вычисления следующего нумерала относительно заданного, а также для проверки заданного нумерала на равенство нулевому нумералу  $\bar{0}$ . Определения этих функций выглядят так<sup>1</sup>:

- 1)  $\text{nxt} \equiv \lambda nfx.f(nfx)$
- 2)  $\text{iszero} \equiv \lambda n.n(\lambda x.\text{false})\text{true}$

Опять же, доказательство этих выражений представляется делом достаточно простым, поэтому в этой книге приводиться не будет. Здесь же главным является то, что для любых нумералов Чёрча безусловно выполняются следующие редукционные цепочки (опять следует напомнить, что на других  $\lambda$ -термах, кроме нумералов Чёрча, эти функции могут вести себя некорректно):

$$\text{nxt } \bar{n} \rightarrow \overline{\bar{n} + 1}$$

---

<sup>1</sup> Используемые в данных выражениях комбинаторы **true** и **false** определяются стандартным образом:

$$\begin{aligned}\text{true} &\equiv \lambda xy.x \\ \text{false} &\equiv \lambda xy.y\end{aligned}$$

Хотя, с другой стороны, эти определения совершенно не релевантны к рассмотрению функции **iszero**, так как её задача заключается в том, чтобы возвратить истинное значение, если переданный ей на вход нумерал равен  $\bar{0}$ , и ложное значение в противном случае. Поэтому способ кодирования булевых значений истинности роли не играет.

$$\text{iszzero } \bar{0} \rightarrow \text{true}$$

$$\text{iszzero } \bar{n+1} \rightarrow \text{false}$$

Но на этом простые определения заканчиваются. Как только появляется задача получения определения функции, которая для заданного нумерала Чёрча находит предыдущий нумерал, начинаются достаточно серьёзные сложности. А тем более это касается операции вычитания. Ведь, как уже было сказано, вычитание может вывести за рамки натуральных чисел, в случае если из меньшего нумерала вычесть больший. А отрицательные нумералы не определены, так как невозможно провести итеративное применение функции отрицательное число раз. Поэтому при определении функции для вычитания необходимо вводить ограничения.

Сперва же надо разобраться с получением предыдущего нумерала для заданного. Для решения этой задачи над нумералами Чёрча достаточно сначала представить себе задачу менее сложную, а именно получение предыдущего натурального числа с использованием только операции сложения. Злые языки поговаривают, что А. Чёрч не смог справиться с этой задачей. Но на самом деле она не так сложна, как это может показаться на первый взгляд. Вот как может выглядеть подобная функция на языке Haskell:

```
decrement :: Num a => a -> a
decrement n = dcr n 0
where
    dcr 0 _ = 0
    dcr n k = if ((k + 1) == n)
                then k
                else dcr n (k + 1)
```

Другими словами, для получения предыдущего натурального числа для некоторого заданного необходимо организовать цикл, в котором постепенно прибавлять единицу и сравнивать результат с исходным числом. Задача действительно не так сложна, как это может показаться. Обычно новичков пугает её формулировка.

В случае с нумералами Чёрча дело обстоит несколько сложнее, ведь нумерал — это более абстрактное понятие, чем просто натуральное число. Получение предыдущего нумерала необходимо выполнять, по сути, для итератора, то есть необходимо из итератора  $\bar{n+1}$  получить итератор  $\bar{n}$ . Это можно сделать при помощи некоторой вспомогательной функции  $g$ , которая связывается с исходными  $f$  и  $x$ , прилагаемыми к нумералу  $\bar{n}$ , следующим соотношением:

$$\bar{n+1} g y \rightarrow \bar{n} f x \tag{5}$$

Первое, что приходит в голову при проектировании такой функции  $g$ , — это работа с парой аргументов, при этом на первой позиции в такой паре стоит нумерал, который на единицу больше второго элемента пары:

$$\bar{n+1} g (x, x) \Rightarrow (\bar{n+1} f x, \bar{n} f x) \tag{6}$$

Подобной редукции можно достичь, если взять в качестве функции  $g$  следующее выражение:

$$g(x, x) = (f x, x) \tag{7}$$

В этом случае для получения предыдущего нумерала для заданного достаточно взять второй элемент пары, которая получена при помощи применения функции  $g$ . Другими словами, функция  $\text{pre}$  для получения предыдущего нумерала Чёрча выглядит так:

$$\mathbf{pre} \overline{\mathbf{n} + 1} = \mathbf{snd} (\overline{\mathbf{n} + 1} g(\mathbf{pair} \mathbf{x} \mathbf{x}))$$

Эта запись не очень похожа на уже изученные  $\lambda$ -термы. Её действительно необходимо преобразовать в обычный  $\lambda$ -терм. Это можно сделать следующим образом, введя связанные переменные для нумерала, функции  $g$  и переменной  $x$ :

$$\mathbf{pre} \equiv \lambda n g x. \mathbf{snd} (n g (\mathbf{pair} x x)) \quad (8)$$

В этой формуле, как видно, решена проблема, обозначенная в формуле (6). Действительно, пока не принимая в расчёт  $\lambda$ -термы **pair** и **snd**, которые нигде не введены, можно увидеть, что функция **pre** действительно возвращает предыдущий нумерал. Более того, формула (8) работает и для нумерала  $\bar{0}$ , в чём можно убедиться непосредственно, подставив этот нумерал на место первого операнда. Но остаётся одна проблема — функция  $g$  до сих пор не выражена через функцию  $f$ , как это должно быть на самом деле, поэтому эта функция остаётся неизвестным членом формулы.

Однако эта проблема решается просто, так как в формуле (7) уже дано выражение одной функции через другую. Осталось записать эту формулу в виде  $\lambda$ -терма. Это можно сделать следующим образом:

$$g \equiv \lambda f p. \mathbf{pair} (f (\mathbf{fst} p)) (\mathbf{fst} p) \quad (9)$$

Таким образом, осталось записать окончательное решение для функции **pre**:

$$\mathbf{pre} \equiv \lambda n f x. \mathbf{snd} (n (g f) (\mathbf{pair} x x)) \quad (10)$$

Собственно, проверка этого выражения является делом достаточно интересным, так как позволяет понять смысл всего вышеизложенного. Читателю рекомендуется самостоятельно проверить функцию **pre** для нумералов  $\bar{0}$ ,  $\bar{1}$  и  $\bar{2}$ , после чего рассмотреть нумерал  $\overline{\mathbf{n} + 1}$  в приложении к этой функции.

Что интересно, полученную функцию **pre** можно легко представить в виде функции на языке Haskell, так как в этом языке есть средства для прямого отображения  $\lambda$ -термов. Это, к примеру, можно сделать так:

```
pre = \n f x -> snd (n (g f) (x, x))
where
  g = \f p -> (f (fst p), (fst p))
```

Однако это не будет функцией, которая вычисляет декремент натуральных чисел, как этого можно было бы ожидать. Она действительно будет работать с нумералами Чёрча, ожидая первым аргументом именно такие объекты, которые, как надо помнить, являются функциями высшего порядка. Если пытаться преобразовать эту функцию к такой, что работает с натуральными числами, то необходимо многое упростить. Если вспомнить, то при переходе к арифметическим операциям необходимо функцию  $f$  преобразовать в функцию  $(+ 1)$ , а переменную  $x$  сделать равной 0. Поэтому первым шагом к преобразованию будет запись указанных величин непосредственно в тело функции **pre**:

```
pre = \n -> snd (n (g (+ 1)) (0, 0))
where
  g = \f p -> (f (fst p), (fst p))
```

Дальше можно увидеть, что локальная функция  $g$  используется только в одном месте, поэтому её выделение в качестве локального определения бессмысленно, а её тело можно подставить непосредственно в функцию **pre**:

```
pre = \n -> snd (n (\p -> ((+ 1) (fst p),
                               (fst p)))
                  (0, 0))
```

К сожалению, это не прибавило ясности в определение. Но определённый шаг к упрощению сделан. Теперь осталось сделать так, чтобы функция `pre` принимала на вход натуральное число, а не нумерал Чёрча. Для этого надо заметить, что в данном случае аргумент `n` просто является итератором, который применяет `n` раз следующую функцию к аргументу. Для схожих целей в стандартном модуле `Prelude` определена функция `until`, которая циклически применяет функцию к результату предыдущего применения, пока не будет достигнуто условие, определяемое некоторым предикатом. В данном случае необходимо проверять на равенство входному параметру функции `pre`, который уже должен стать натуральным числом. Поэтому практически окончательное определение функции `pre` будет выглядеть следующим образом:

```
pre n = snd (until ((== n) . fst)
                  (((\p -> ((+ 1) (fst p),
                               (fst p))))
                   (0, 0)))
```

Осталось немного «причесать» определение, убрав из него не очень приятный на вид  $\lambda$ -терм и заменив его некоторой локальной функцией, которая использует мощный механизм *сопоставления с образцом*. Кроме того, можно убрать лишние скобки, применив так называемый *бесточечный стиль* определения функций. В итоге получится следующее определение:

```
pre :: Num a => a -> a
pre n = snd $ until ((== n) . fst) dcr (0, 0)
where
  dcr (n, _) = (n + 1, n)
```

Как видно, тип этой функции уже полностью удовлетворяет первоначальным целям — получение декремента натурального числа. И она действительно вычисляет предыдущее число для заданного натурального, причём для числа 0 возвращается результат 0, поэтому функция не выходит за пределы множества натуральных чисел.

Остаётся рассказать несколько слов о комбинаторах `pair`, `fst` и `snd`, а также об одноимённых функциях `fst` и `snd` из стандартного модуля `Prelude`. Что касается комбинаторов, то, как уже говорилось, кодировка этих функций при помощи  $\lambda$ -термов совершенно не важна, главное, чтобы выполнялось условие редуцируемости следующих цепочек:

$$\text{fst} (\text{pair } x y) \rightarrow x$$

$$\text{snd} (\text{pair } x y) \rightarrow y$$

Это условие проще всего выполнить при помощи следующих определений (при этом комбинаторы `true` и `false` определяются так же, как и в прошлый раз):

- 1)  $\text{pair} \equiv \lambda xyf.fxy$
- 2)  $\text{fst} \equiv \lambda p.p \text{ true}$
- 3)  $\text{snd} \equiv \lambda p.p \text{ false}$

В стандартном же модуле `Prelude` определены функции для работы с парами, которые представляют собой кортежи из двух элементов. Эти функции и были использованы при построении функции `pre`. Здесь также является главным то, что выполняются следующие условия:

```
fst (x, y) = x
snd (x, y) = y
```

Эти функции именно так и определяются в стандартном модуле `Prelude`.

Теперь всё готово для того, чтобы определить функцию для вычисления разности двух нумералов Чёрча, причём таким образом, что если вычисляется разность большего нумерала из меньшего, то должен возвратиться нумерал  $\overline{0}$ . Какова суть операции вычитания? В записи  $(m - n)$  говорится, что необходимо из числа  $m$  вычесть единицу  $n$  раз. Собственно, это и есть ключ к определению операции вычитания через  $\lambda$ -терм:

$$\mathbf{sub} \equiv \lambda m n. n \mathbf{pre} m \quad (11)$$

Путь намечен. Полнотью осознав новое знание, можно двигаться далее. Формализм  $\lambda$ -исчисления позволяет выразить практически всё. Изучая нумералы Чёрча, можно попытаться выразить операции для целочисленного деления и взятия остатка при делении одного числа на другое. Таким же образом можно закодировать все функции, работающие с натуральными числами. А затем можно расширить границы рассмотрения и перейти к другим численным множествам. Более того,  $\lambda$ -исчисление может выразить в своих терминах многие направления дискретной математики — исчисления высказываний и предикатов первого порядка, теорию множеств и, конечно же, *функциональное программирование*. Можно больше сказать, что многие (если не все) функциональные языки программирования являются надстройками над  $\lambda$ -исчислением, «обёртывая» его в удобный синтаксис и добавляя некоторые возможности. Это утверждение относится и к языку Haskell, ядром которого является типизированное  $\lambda$ -исчисление. Поэтому понимание сути  $\lambda$ -исчисления позволит глубже освоить само функциональное программирование и любой функциональный язык.

## Заключение

А что же с парадоксом Рассела?

Алонзо Чёрч начал разрабатывать  $\lambda$ -исчисление для изменения нотации классической теории множеств, полагая, что таким способом он сможет найти разрешение этого парадокса. Так, любая характеристическая функция из теории множеств была закодирована при помощи некоторого  $\lambda$ -терма, которые обозначаются заглавными буквами:  $M$ ,  $N$  и т. д. Факт того, что одно множество является подмножеством другого, то есть  $M \subseteq N$ , кодируется при помощи аппликации  $MN$ . Наконец, абстракция кодирует множество всех элементов  $x$ , удовлетворяющих некоторому предикату  $P$ , то есть  $\{x \mid P\}$  кодируется как  $\lambda x. P$ .

Эта нотация позволяет закодировать парадокс Рассела следующим образом:

$$R = \lambda x. \mathbf{not} (xx) \quad (12)$$

В этом случае основной вопрос парадокса кодируется как  $RR$ , то есть  $(\lambda x. \mathbf{not} (xx))(\lambda x. \mathbf{not} (xx))$ . Если попытаться редуцировать этот  $\lambda$ -терм, то станет понятно, что у него нет нормальной формы. Редукция будет длиться до бесконечности. Другими словами, математическая нотация  $\lambda$ -исчисления повторила рассуждения на естественном языке, которые приведены во введении к разделу. Парадокс разрешить не удалось.

Но, как было показано в этом разделе,  $\lambda$ -исчисление серьёзно повлияло на дальнейшее развитие компьютерной науки в частности и дискретной математики в целом.

# Комбинаторы? — Это просто!

*Статья была опубликована в № 07 (19) журнала «Потенциал» в июле 2006 года.*

*Данное эссе описывает основы комбинаторной логики — математической науки о вычислениях, в которой любые объекты изучения выражаются при помощи двух базовых объектов **S** и **K** (или даже одного базового объекта **X**), называемых комбинаторами, и одной операции над ними — операции применения одного комбинатора к другому. Большинство примеров из данного эссе были рассмотрены в предыдущем эссе, здесь же приводятся в качестве дополнения в целях показать сходства и различия между двумя «родственными» формализмами.*

## Введение

Комбинаторная логика (от слова «комбинатор», а не «комбинаторика») — это направление в математической логике, разработанное в первой половине XX века американскими логиками Мозесом Шёнфинкелем и Хаскеллом Карри в качестве науки о вычислительных процессах [21, 17]. Хотя первоначально этот вид логики претендовал только на то, чтобы удалить из логических высказываний переменные, через некоторое время в *компьютерной науке* (computer science) были получены прикладные результаты, которые показали, что комбинаторную логику можно использовать для проведения вычислений.

На комбинаторную логику можно смотреть как на некоторое упрощение  $\lambda$ -исчисления<sup>2</sup>, в котором нет символа  $(\lambda)$ , а все функциональные абстракции представлены ограниченным набором символов, называемых «комбинаторами». Такие комбинаторы не содержат переменных, являются *функциями высшего порядка*, то есть в качестве аргументов могут принимать на вход другие функции (такие же комбинаторы), а также описывают определённые правила преобразования объектов, поданных им на вход в качестве аргументов.

Другими словами, комбинаторная логика — это формализм для представления функциональных зависимостей между входными и выходными параметрами, обладающий весьма ограниченным алфавитом для кодирования любых данных и операций над ними. В последующих подразделах этот формализм будет подробно рассмотрен.

## Формальная теория

Для того чтобы не быть голословным, необходимо чётко описать базовые объекты комбинаторной логики, которые участвуют в формальной системе, определяющей саму комбинаторную логику. Согласно математической практики, необходимо определить следующие элементы формальной системы:

- 1) алфавит;

---

<sup>2</sup>  $\lambda$ -исчисление — это наука о вычислениях, первоначально разработанная Алонзо Чёрчем для разрешения антиномии Б. Рассела о множестве всех множеств, не включающих самих себя в качестве подмножества. Рассмотрение этого направления логики произведено в предыдущем эссе.

- 2) утверждения (множество правильно построенных формул);
- 3) аксиомы;
- 4) правила вывода.

Под алфавитом понимается набор символов, допустимых в нотации записей термов в комбинаторной логике. В общем виде при помощи нотации комбинаторной логики можно записывать объекты нескольких типов — константы, переменные, термы и специальные знаки.

Константы обозначаются малыми буквами латинского алфавита, возможно с индексами. Обычно для обозначения констант берутся символы с начала алфавита. Переменные обозначаются также малыми буквами, возможно с индексами, но при этом они обычно выбираются с конца алфавита:  $c_1, c_2$  — константы;  $x, y$  — переменные.

Однако для выделения константных объектов иногда будет использоваться иной способ записи. Такой способ заключается в выделении наименований констант полужирным начертанием — эта запись будет использоваться, если наименование константы состоит более чем из одного символа: **if**, **true**, **pair** и т. д.

Комбинаторные термы (или просто «выражения») будут обозначаться заглавными буквами латинского алфавита, возможно также с индексами:  $M, N$  и т. д. Комбинаторы, являющиеся термами, также обозначаются заглавными буквами латинского алфавита, при этом они записываются в прямом полужирном начертании.

Из специальных символов в комбинаторной логике используются всего три: это скобки «(» и «)», а также знак равенства (=). Последний обозначает отношение конвертируемости термов друг в друга, то есть возможность преобразования таких термов из одного в другой. С точки зрения вычислительных процессов, отношение конвертируемости определяет сам процесс вычисления значения функции, представленной комбинаторным термом.

Множество правильно построенных формул в комбинаторной логике выглядит очень просто. Его составляют выражения вида  $M = N$ , где  $M$  и  $N$  — комбинаторные термы. При этом сами комбинаторные термы строятся по индукции:

- 1) если  $c$  — константа, то  $c$  — комбинаторный терм;
- 2) если  $x$  — переменная, то  $x$  — комбинаторный терм;
- 3) если  $M$  и  $N$  — комбинаторные термы, то  $(MN)$  — комбинаторный терм;
- 4) других комбинаторных термов нет.

В этом индуктивном определении выражение  $(MN)$  обозначает операцию аппликации, единственную операцию комбинаторной логики. Аппликация описывает применение функции (в данном примере — терм  $M$ ) к её аргументу или аргументам (в данном примере — терм  $N$ ).

Как видно, создание комбинаторных термов повлечёт за собой лавинообразное внедрение в запись таких термов скобок. Для того чтобы уменьшить количество скобок в записях комбинаторных термов, вводятся соглашения о скобках, которые заключаются в том, что пропущенные скобки восстанавливаются по ассоциативности влево:

$$XY = (XY)$$

$$XYZ = ((XY)Z)$$

В качестве аксиом, входящих в множество правильно построенных формул, обычно выделяются следующие:

$$\mathbf{I}x = x \tag{I}$$

$$\mathbf{K}xy = x \quad (\mathbf{K})$$

$$\mathbf{S}xyz = xz(yz) \quad (\mathbf{S})$$

Данные аксиомы не надо доказывать, их наличие просто постулируется. Эти аксиомы определяют три базовых комбинатора, использующихся для вывода (вычислений) новых комбинаторов. Традиционно базис **S**, **K**, **I** является тем набором первоначальных комбинаторов, через который выражаются все прочие комбинаторы. Однако это не минимальный базис, ибо наличие комбинатора **I** не обязательно в нём, так как его можно выразить через комбинаторы **S** и **K**<sup>3</sup>:

$$\mathbf{SKK}x \stackrel{\mathbf{S}}{=} \mathbf{K}x(\mathbf{K}x) \stackrel{\mathbf{K}}{=} x \equiv \mathbf{Ix}$$

Другими словами, комбинаторный базис — это набор комбинаторов, через которые при помощи отношения конвертируемости ( $\equiv$ ) можно выразить все остальные комбинаторы. Двухкомбинаторным базисом является набор из комбинаторов **S** и **K**, но обычно его дополняют комбинатором **I** для упрощения записи выражаемых комбинаторных термов. Однако разнообразных базисов может быть бесчисленное множество, и разные исследователи вводили разные базисы для своих целей. В следующем разделе будут рассмотрены некоторые примеры таких базисов.

Остаётся рассмотреть правила вывода, которые используются в рамках комбинаторной логики для преобразования одних термов в другие и описывают отношение конвертируемости ( $\equiv$ ). Данные правила вывода, по своей сути, описывают это отношение, задавая его характеристики.

$$a = a \quad (\rho)$$

$$(a = b) \Rightarrow (b = a) \quad (\sigma)$$

$$(a = b) \wedge (b = c) \Rightarrow (a = c) \quad (\tau)$$

$$(a = b) \Rightarrow (ca = cb) \quad (\mu)$$

$$(a = b) \Rightarrow (ac = bc) \quad (\nu)$$

Как видно, первые три правила являются описанием свойств рефлексивности, симметричности и транзитивности. Набор этих свойств является собой характеристики отношения эквивалентности, то есть отношение конвертируемости ( $\equiv$ ) делит всё множество комбинаторных термов на некоторые классы эквивалентности, относительно которых можно сказать, что находящиеся в них термы конвертируемы друг в друга (эквивалентны друг другу).

Дополнительно надо отметить, что далее под именем комбинатора будет пониматься символ, его обозначающий, под *сигнатурой* комбинатора будет пониматься левая часть правила, а под *характеристикой* — правая часть правила, описывающая сам комбинатор.

<sup>3</sup> Более того, сами комбинаторы **S** и **K** можно выразить через один комбинатор **X** ( $\mathbf{K} \equiv \mathbf{XX}$ ,  $\mathbf{S} \equiv \mathbf{XK} \equiv \mathbf{X}(\mathbf{XX})$ ), так что минимальный базис в комбинаторной логике состоит из одного комбинатора. Подробно об этом можно почитать в [18].

## Примеры сложных комбинаторов

Вполне естественно, что создатели комбинаторной логики не ограничились использованием двух базовых комбинаторов — **S** и **K**. Хотя они и составляют минимальный базис, но запись функций только через них увеличивает громоздкость определений с ростом сложности функций. Поэтому для упрощения записи определений более сложных комбинаторов и комбинаторных термов были разработаны дополнительные комбинаторы, из которых также можно составлять базисы.

К таким комбинаторам относятся следующие:

- 1)  $Bxyz = x(yz)$ .
- 2)  $Cxyz = xzy$ .
- 3)  $Wxy = xyy$ .

Данные комбинаторы весьма существенно сокращают некоторые преобразования<sup>4</sup>. Ради этого многие комбинаторы даже получили свои собственные наименования. Так, комбинатор **I** называется «комбинатором тождества», комбинатор **K** — «канцелятор», комбинатор **S** — «коннектор», комбинатор **B** — «композитор», комбинатор **C** — «пермутатор» и комбинатор **W** — «дубликатор».

Однако вполне понятно, что все перечисленные комбинаторы можно выразить друг через друга. Следующие формулы показывают выражение новых комбинаторов через уже известные:

$$B \equiv S(KS)K$$

$$C \equiv S((S(KS)K)(S(KS)K)S)(KK)$$

$$W \equiv SS(K(SK))$$

Вдумчивому читателю предлагается самостоятельно проверить данные формулы. Для этого достаточно воспользоваться аксиомами для комбинаторов **S** и **K** и подставить в формулы переменные в заданном количестве (согласно определений).

Однако возникает вопрос: на основании чего получены перечисленные выражения комбинаторов **B**, **C** и **W** через базисные, как создать такие выражения самостоятельно? Можно попытаться ответить на этот вопрос, самостоятельно найдя выражение комбинатора **I** как самого простого. Это можно проделать на основании следующих рассуждений...

Выражение комбинатора **I** может начинаться либо с комбинатора **S**, либо с комбинатора **K**. При помощи цифр в следующих записях будут обозначаться недостающие объекты, которые ещё необходимо найти. В этом случае два возможных пути поиска выражения для комбинатора **I** выглядят так:

- 1)  $K1x \stackrel{K}{=} 1 \neq x$ .
- 2)  $S12x \stackrel{S}{=} 1x(2x) = \dots$

Здесь видно, что первая альтернатива не подходит автоматически, так как канцелятор **K** отменяет переменную  $x$ , которая должна появиться в самом конце. Соответственно, далее надо рассматривать только вторую альтернативу и искать способ выражения неизвестных объектов 1 и 2. Они также могут быть либо комбинатором **S**, либо комбинатором **K**. Соответственно, подставляя эти комбинаторы вместо неизвестного объекта 1, можно получить:

---

<sup>4</sup> Кстати, первоначально Х. Карри ввёл в своей работе именно комбинаторы **B** и **C**. В дальнейшем первоначальный базис был упрощён.

$$1) \mathbf{K}x(2x) \stackrel{\mathbf{K}}{=} x.$$

$$2) \mathbf{S}x(2x) = ?.$$

Как видно, первая альтернатива уже дала необходимый результат, а во второй альтернативе не хватает операндов у комбинатора **S**. Поэтому вторую альтернативу можно не рассматривать, а вернуться только к первой. В ней остаётся неизвестный объект 2, который не участвует в цепочке вывода, так как уничтожается канцелятором. Поэтому вместо него может стоять любой комбинатор, например всё тот же **K**. Так и получается, что выражение комбинатора тождества таково:

$$\mathbf{I} \equiv \mathbf{SKK}$$

Этот пример также показал, что способов разложения объектов в комбинаторном базисе существует бесконечное множество, поэтому всегда имеет смысл говорить о «минимальном» разложении, то есть таком, в записи которого используется минимальное число комбинаторов и скобок.

Но описанный процесс не так прост, как может показаться на самом деле. Достаточно попробовать разложить подобным образом относительно простой комбинатор **B**, чтобы понять, что в процессе такого рассмотрения альтернатив дерево решений растёт подобно снежному кому. Поэтому ради облегчения процесса выражения объектов с заданными комбинаторными характеристиками были созданы специальные правила. Данные правила относятся к выражению в базисе **S**, **K**, **I**:

- 1)  $\mathbf{T}[v] \Rightarrow v$ , где  $v$  — переменная.
- 2)  $\mathbf{T}[(E_1 E_2)] \Rightarrow (\mathbf{T}[E_1] \mathbf{T}[E_2]).$
- 3)  $\mathbf{T}[\lambda x.x] \Rightarrow \mathbf{I}.$
- 4)  $\mathbf{T}[\lambda x.E] \Rightarrow \mathbf{K} \mathbf{T}[E]$ , если  $x$  несвободна в  $E$ .
- 5)  $\mathbf{T}[\lambda x.\lambda y.E] \Rightarrow \mathbf{T}[\lambda x.\mathbf{T}[\lambda y.E]]$ , если  $x$  свободна в  $E$ .
- 6)  $\mathbf{T}[\lambda x.(E_1 E_2)] \Rightarrow \mathbf{S} \mathbf{T}[\lambda x.E_1] \mathbf{T}[\lambda x.E_2].$

В этих правилах используется  $\lambda$ -нотация, которая принята в  $\lambda$ -исчислении. Привести комбинатор с его сигнатурой и характеристикой к  $\lambda$ -нотации довольно просто — надо вместо символа, обозначающего сам комбинатор, использовать символ  $(\lambda)$ , а вместо символа  $(=)$  использовать точку  $(.)$ .

В следующем разделе приведены определения типов данных и функций на языке Haskell, которые осуществляют перевод заданного комбинатора в базис **S**, **K**, **I**. Однако для закрепления материала читателю предлагается самостоятельно выразить в базисе **S**, **K**, **I** следующие комбинаторы с такими характеристиками<sup>5</sup>:

- 1)  $\Phi abcd = a(bc)(bd).$
- 2)  $\mathbf{C}^{[2]}abcd = acdb.$
- 3)  $\mathbf{C}_{[2]}abcd = adbc.$
- 4)  $\mathbf{B}^2abcd = a(bcd).$
- 5)  $\mathbf{C}^{[3]}abcde = acdeb.$
- 6)  $\mathbf{C}_{[3]}abcde = aebcd.$
- 7)  $\mathbf{B}^3abcde = a(bcde).$
- 8)  $\Phi abcd = a(bd)(cd).$

<sup>5</sup> Для решения этих задач может потребоваться дополнительная литература, в которой описывается так называемая *иерархическая комбинаторная логика*, предназначенная для выражения исчисления высказываний первого порядка через комбинаторы — например, [3].

## Модуль на языке Haskell для преобразования комбинаторов

В этом разделе приводится текст программы на языке Haskell, которая преобразует заданный комбинатор в базис **S**, **K**, **I**. Для того чтобы понимать приведённые определения, необходимо быть знакомым с синтаксисом этого языка на уровне, достаточном для определения типов и функций. Рассмотрение этих тем выходит за рамки этой книги, поэтому заинтересованного читателя можно отослать к специализированной литературе [6, 14].

Для преобразования комбинаторов необходимо для начала определить тип данных для их представления. В соответствии с определением комбинаторного терма определение такого типа выглядит так:

```
data Combinator = Var String
    | App Combinator Combinator
    | Lam String Combinator
deriving Eq
```

Это определение полностью соответствует математическому, которое гласит, что комбинаторный терм это либо переменная (`Var` — от английского слова *variable*), либо абстракция (`Lam` — от английского слова *lambda*), либо приложение одного комбинаторного терма к другому (`App` — от английского слова *application*).

Для того чтобы иметь возможность просматривать полученные результаты преобразования комбинаторных термов, необходимо определить тип `Combinator` экземпляром класса `Show`, который является классом величин, которые могут быть отображены. Конечно, интерпретатор языка Haskell может самостоятельно определить такой экземпляр, но он сделает это не так красиво, как можно сделать вручную. Поэтому вводится следующее определение:

```
instance Show Combinator where
    show (Var x)      = x

    show (App x y)   = case y of
        App _ _ -> showLam x ++
                        "(" ++
                        show y ++
                        ")"
        _           -> showLam x ++
                        showLam y

    where
        showLam l@(Lam _ _) = "(" ++
                                show l ++
                                ")"
        showLam x           = show x

    show (Lam x e)   = "\\" ++
                        x ++
                        "."
                        ++
                        show e
```

Здесь видно, что переменная отображается просто своим именем, которое представляется строкой символов. Применение комбинаторных термов друг к другу просто записывается перечислением комбинаторных термов друг за другом со взятием в скобки, если второй терм сложный. А абстракция записывается при помощи символа (\) с указанием сигнатуры и характеристики.

Базис **S**, **K**, **I** определяется при помощи константных функций, возвращающих комбинаторные термы в виде переменных с известными именами:

```
i = Var "I"
k = Var "K"
s = Var "S"
```

Дополнительно, хотя это и необязательно, можно реализовать предикат, который проверяет, является ли заданная переменная свободной в исследуемом комбинаторном терме. Этот предикат определяется так:

```
free :: String -> Combinator -> Bool
free x (Var y)      = x == y
free x (App e1 e2) = free x e1 || free x e2
free x (Lam y e)   = free x e
```

Всё в полном соответствии с математическим определением.

Наконец, сама функция `transform`, которая осуществляет перевод комбинаторного терма в базис **S**, **K**, **I**. Её определение также выглядит в полном соответствии с перечисленными в предыдущем разделе правилами трансформации. Здесь можно видеть всю силу выразительности языка Haskell, которая позволяет записывать определения функций практически в математической нотации.

```
transform :: Combinator -> Combinator
transform (Var x) = Var x
transform (App x y) = App (transform x)
                           (transform y)
transform (Lam x (Var y)) | x == y
                           = i transform (Lam x e)
                           | (not . free x) e
                           = App k (transform e) transform (Lam x l@((Lam y e)))
                           | free x e
                           = transform (Lam x (transform l))
transform (Lam x (App e1 e2))
                           = App (App s (transform (Lam x e1)))
                           (transform (Lam x e2))
```

Видно, что определение этой функции полностью совпадает с описанием правил трансформации комбинаторных термов. Однако данная функция не позволяет получить минимальное выражение комбинаторного терма в базисе **S**, **K**, **I**. Она возвращает один из возможных вариантов разложения терма. Например, комбинатор **B** кодируется при помощи  $\lambda$ -нотации следующим образом:

```
b = Lam "x"
      (Lam "y"
            (Lam "z"
                  (App (Var "x")
                        (App (Var "y")
                              (Var "z")))))
```

Если передать это определение функции `transform`, то на выходе будет такая конструкция:

```
S(S(KS)(S(KK)(S(KS)(S(KK)I))))(K(S(S(KS)(S(KK)I))(KI)))
```

## Представление данных и функций

Было бы странно, если бы комбинаторная логика не могла быть тем инструментом, при помощи которого можно было бы выражать различные объекты. Действительно, формализм комбинаторной логики, несмотря на весьма ограниченный алфавит, является вполне достаточным для выражения таких базовых понятий математики, как булевские значения истины, упорядоченные пары некоторых значений, натуральные числа, списки. Это, в свою очередь, позволяет практически полностью смоделировать теорию функционального программирования в рамках простой комбинаторной логики.

Сам по себе способ кодирования данных в рамках комбинаторной логики не является достаточно выразительным, более того, иной раз кажется, что такое кодирование вычурно и надуманно. С точки зрения эффективности вычислений также имеются проблемы — оптимизация в прикладных трансляторах языков программирования позволяет кодировать и проводить вычисления над закодированными данными более эффективно. Однако этот способ кодирования данных является довольно интересным с точки зрения математики, так как позволяет понять в том числе и то, что данные могут нести внутри себя и способы их обработки.

### Булевские значения

Для кодирования булевых величин необходимо иметь способ представления в комбинаторной логике значений **true**, **false**, а также служебную структуру **if**. Обычно для этих целей используются следующие правила кодирования:

- 1) **true**  $\equiv K$ ;
- 2) **false**  $\equiv KI$ ;
- 3) **if**  $\equiv I$ .

Действительно, предложенный способ кодирования можно легко проверить:

$$\text{if true } x \ y \equiv I \ K \ x \ y \stackrel{I}{=} K \ x \ y \stackrel{K}{=} x$$

$$\text{if false } x \ y \equiv I \ K \ I \ x \ y \stackrel{I}{=} K \ I \ x \ y \stackrel{K}{=} I \ y \stackrel{I}{=} y$$

Как можно видеть, кодирование служебного слова **if** вообще не является необходимым — можно вполне обойтись и без него. Сами по себе значения истинности являются условными выражениями, так как возвращают первый или второй операнд в зависимости от своей природы. Поэтому комбинация **if** является тождеством для трёх operandов, первый из которых должен быть значением истинности.

Вполне естественно, что над представленными значениями истинности должны иметься функции для выполнения базовых операций булевой логики. Такие базовые операции могут быть выражены через условные выражения. Способы кодирования трёх базисных булевых операций (отрицание, конъюнкция и дизъюнкция) выглядят следующим образом:

- 1) **not**  $\equiv C(C \text{ if false}) \text{ true}$ ;
- 2) **and**  $\equiv B(CC \text{ false}) \text{ if}$ ;
- 3) **or**  $\equiv C \text{ if true}$ .

Читателю предлагается самостоятельно проверить данные тождества на предмет их верности. Для этого необходимо рассмотреть таблицы истинности для перечисленных логических операций и сравнить их с традиционными таблицами истинности.

## Нумералы Чёрча

Вполне понятно, что для кодирования чисел или подобных им объектов можно использовать любой способ выражения, главное, чтобы потом на таком способе можно было бы построить приемлемые операции, тождественные тем, что определены для чисел. Поэтому, собственно, способов кодирования чисел существует множество. Но самым первым способом был тот, что предложен А. Чёрчем и теперь носит наименование «нумералы Чёрча».

Нумералом Чёрча порядка  $n$  называется такой объект  $\bar{n}$  ( $n$  — натуральное число из расширенного множества натуральных чисел  $\mathbb{N}^+$ ), который выражается через базовые комбинаторы следующим образом:

$$\bar{n} = (\mathbf{SB})^n(\mathbf{KI}),$$

где под записью  $(\mathbf{SB})^n$  понимается  $n$ -кратное приложение объекта  $(\mathbf{SB})$  к самому себе:

$$(\mathbf{SB})^0(\mathbf{KI}) = \mathbf{KI},$$

$$(\mathbf{SB})^1(\mathbf{KI}) = \mathbf{SB}(\mathbf{KI}),$$

$$(\mathbf{SB})^2(\mathbf{KI}) = \mathbf{SB}(\mathbf{SB}(\mathbf{KI})),$$

$$(\mathbf{SB})^3(\mathbf{KI}) = \mathbf{SB}(\mathbf{SB}(\mathbf{SB}(\mathbf{KI})))$$

и т. д. То есть по индукции эти объекты можно определить как:

$$1) \quad \bar{0} = (\mathbf{SB})^0(\mathbf{KI}) = \mathbf{KI};$$

$$2) \quad \bar{n} = (\mathbf{SB})^n(\mathbf{KI}) = \mathbf{SB}(\mathbf{SB}^{n-1}(\mathbf{KI})), \quad n > 0.$$

По своей сути, эти комбинаторы создают итеративное применение заданной функции к некоторому аргументу, причём количество итераций равно определяемому нумералом числу:

$$\bar{0} f x = x$$

$$\bar{1} f x = f x$$

$$\bar{n} f x = \underbrace{f(f(\dots(f}_{n \text{ раз}} x)\dots))$$

Для этих объектов довольно простым способом можно определить функции для сложения, умножения и возведения в степень. Это делается следующим образом:

$$1) \quad \mathbf{add} \equiv \mathbf{CI}(\mathbf{SB});$$

$$2) \quad \mathbf{mlt} \equiv \mathbf{B};$$

$$3) \quad \mathbf{exp} \equiv \mathbf{CI}.$$

Доказательство данных тождеств легко проводится по индукции опять же и предлагается для самостоятельной проработки.

## Упорядоченные пары

Ещё одним достаточно важным объектом, имеющим большое практическое значение в функциональном программировании, является упорядоченная пара, состоящая из двух значений. Из пар создаются списки и списочные структуры, которые, в свою очередь, являются одним из основных объектов обработки в функциональных языках<sup>6</sup>.

<sup>6</sup> Например, первый функциональный язык LISP назван так из-за своего назначения — «LISt Processing» — «обработка списков».

Для кодирования пары при помощи комбинаторных термов необходимо создать функцию, которая является конструктором такой пары. Это можно сделать следующим образом:

$$\mathbf{pair} \equiv \mathbf{BC}(\mathbf{CI})$$

Данный комбинатор составляет пару из двух заданных объектов любой природы. Для того чтобы достать эти объекты из пары, необходимы так называемые «селекторы», то есть функции для доступа к элементам пары. Эти селекторы можно определить так:

$$\mathbf{head} \equiv \mathbf{CI} \text{ true}$$

$$\mathbf{tail} \equiv \mathbf{CI} \text{ false}$$

Эти комбинаторы «вынимают» первое или второе значение из переданной им на вход пары. Например, можно доказать, что выражение:

$$\mathbf{head} \mathbf{pair} xy = x$$

для любого выражения  $x$ . То же самое можно сказать и о селекторе  $\mathbf{tail}$ . Читателю опять же рекомендуется провести исследования этого выражения — это позволит закрепить изученный материал и более тонко понять смысл и назначение комбинаторной логики.

## Общие замечания

Надо отметить, что предложенные способы кодирования данных и методов для их обработки при помощи инструментов комбинаторной логики являются скорее не заданной догмой, но шаблонами, по которым можно производить вычисления над закодированными данными. Если рассмотреть такие способы кодирования более подробно, то видно, что и нумералы Чёрча, и упорядоченные пары могут принимать на вход не только сами значения для кодирования, но и функции для их обработки.

Так, определение пары является достаточно универсальным — оно не ограничивает понятие упорядоченной пары какими-то специальными рамками, а оставляет разработчику выбирать способ упаковки объектов в пару. Поэтому любая пара в качестве функции ожидает на вход некоторую функцию двух аргументов, которая после применения возвращает к изначальным объектам и определяет саму пару. Это значит, что и операции для распаковки пары (селекторы) должны быть различными в каждом конкретном случае. Приведённые выше определения являются шаблонами. Однако и эти шаблоны сами по себе также работают.

Все эти рассуждения касаются и остальных примеров (рассмотренных в данном подразделе и всех других). Это значит, что комбинаторная логика предоставляет учёным и разработчикам универсальный инструмент для абстрактного проектирования методов для решения широкого класса задач.

## Заключение

Рассмотренное в этом разделе введение в основы комбинаторной логики не претендует на целостность изложения, да и невозможно в малой научно-популярной книге изложить сложную науку о вычислениях, на которой основана реализация многих функциональных языков программирования. Поэтому всех заинтересовавшихся читателей можно отослать к изучению комбинаторной логики и  $\lambda$ -исчисления по учебникам, полноценно описывающим эти интереснейшие направления логики. В качестве направлений для дальнейшего изучения можно посоветовать рассмотрение следующих вопросов:

- 1) Синтез нового объекта с заданными комбинаторными характеристиками.

- 2) Использование редукции комбинаторов при помощи графов, что позволяет проводить ленивые вычисления, в том числе и потенциально бесконечных структур данных.
- 3) Преобразование  $n$ -местных операторных функций в каррированные, позволяющие производить частичные вычисления.
- 4) Типизация комбинаторов, которая позволяет разбить всё множество комбинаторов на некие классы эквивалентности по их типам (сортам).
- 5) Оболочка Каруби — специальная категория в рамках комбинаторной логики, при помощи которой кодируются все объекты операторных вычислений, в том числе и их типы.
- 6) Выражение при помощи комбинаторов различных систем программирования, в том числе выражение языков Lisp, Haskell и прочих.
- 7) Изучение суперкомбинаторов — объектов для ленивого вычисления значений некоторых выражений.
- 8) Оптимизация вычислений путём комбинирования параметров — шаг к построению систем суперкомпиляции.
- 9) Техники проведения синтаксического анализа в свете применения оного для интерпретации текстов на функциональных языках программирования (непосредственные вычисления значений, кодирование по де Брейну, категориальная абстрактная машина и т. д.).

Подробно про комбинаторную логику можно почитать в источниках [1, 3, 6] либо на специализированных научных интернет-ресурсах.

# Ввод и вывод на языке Haskell

*Статья была опубликована в № 12 (36) журнала «Потенциал» в декабре 2007 года.*

*В настоящем эссе применительно к функциональному языку Haskell рассматривается такой немаловажный аспект любого языка программирования, как система ввода/вывода. Приводятся основные определения, способы использования и примеры построения функций, реализующих ввод/вывод и взаимодействие с внешним миром (окружением программы).*

## Введение

Язык Haskell является чистым функциональным языком программирования, а это значит, что любая функция, которая определена при разработке программ, должна быть *детерминированной*, а также не должна использовать в своей работе так называемые *побочные эффекты*.

Под детерминированностью функций понимается такое свойство, что результат, вычисляемый функцией, зависит только от значений входных параметров. Это значит, что для любого заданного набора значений входных параметров результат детерминирован (определен) функцией. Два вызова функции с одним и тем же набором значений входных параметров всегда возвратят один и тот же результат.

Отсутствие побочных эффектов означает, в свою очередь, что функция в процессе своей работы обращается и изменяет только ту область памяти, которая выделена для её работы. В функциональном программировании в этой области памяти расположены значения выходных параметров, которые доступны только для чтения, а также результат выполнения функции, который доступен для чтения и записи. Кроме того, в области памяти, выделенной для работы функции, могут быть созданы так называемые *замыкания*, то есть локальные переменные, область видимости (контекст) которых ограничена функцией.

Всё вышеперечисленное налагает на чистый язык программирования определённые ограничения. Например, невозможно программировать в терминах изменения некоторых глобальных переменных, поскольку их использование сделает некоторые функции недетерминированными (функция, к примеру, может возвращать значение некоторой глобальной переменной, которая изменяется из другой функции, а потому в зависимости от времени вызова первой функции её результат может быть различным) и использующими сторонние эффекты (изменение глобальных переменных и есть сторонний эффект).

Другая проблема — система ввода/вывода языка, то есть общение с внешним миром (или, как говорят, с *окружением программы*). В общем виде ввод/вывод можно понимать как чтение информации с какого-либо внешнего устройства (например, клавиатуры), а также запись данных опять-таки на внешнее устройство (например, в видеопамять терминала). Для понимания сущности проблемы организации системы ввода/вывода в чистом языке программирования необходимо детально рассмотреть оба случая.

*Ввод*, то есть чтение данных с внешнего устройства — клавиатуры, мыши, из файла, из сети и т. д. Например, пусть имеется некоторая функция `read`, которая считывает с клавиатуры один символ и возвращает его. Каков должен быть тип этой функции? Совершенно ясно, что принимать на вход

ей ничего не нужно. А возвращает такая функция символ, то есть значение типа `Char`. Таким образом, её *сигнатура* (описание типа) выглядит следующим образом:

```
read :: Char
```

Однако ясно, что эта функция недетерминирована, поскольку её результат будет зависеть от того, какой символ на клавиатуре нажал пользователь. Если пользователь нажмёт кнопку «A», функция вернёт символ с кодом 65. Нажмёт другую кнопку — функция возвратит другой код. Полная недетерминированность в поведении.

*Вывод*, то есть запись данных на некоторое внешнее устройство — на экран, в файл, в сеть и т. д. Также пусть имеется некоторая функция `put`, которая выводит заданный символ на экран. Какой тип у этой функции? Ясно, что она должна принимать на вход один параметр — код символа, который должен быть выведен. Так что тип первого (единственного) параметра — `Char`. А что должна возвращать эта функция? Какое значение? Совершенно не ясно. Скорее всего, она не должна возвращать ничего. В языках программирования типа С для этих целей используется тип `void`. В языке Haskell используется тип «*пустой кортеж*», то есть последовательность значений, состоящая из нуля элементов, — `()`. Так что сигнатура функции `put` выглядит так:

```
put :: Char -> ()
```

Эта функция использует сторонние эффекты, поскольку модифицирует внешнюю память — видеопамять экрана, на которой производится вывод. Если бы функция записывала информацию в файл, она модифицировала бы память на носителе информации, где записан файл.

Видно, что и ввод, и вывод являются недопустимыми с точки зрения чистого языка программирования. Однако является совершеннейшим абсурдом отсутствие системы ввода/вывода в языке программирования общего назначения. Для чего нужен такой язык? Как организовывать взаимодействие с пользователем? Как работать с файлами? Как организовывать многопотоковые программы и распределённые вычисления с передачей информации между потоками? Как взаимодействовать с внешними программами по сети? Ведь без перечисленных технологий в современном мире делать нечего. И если в языке программирования не существует средств для работы с ними, этому языку просто нет места в арсенале современного программиста.

Естественно, что в языке Haskell имеются все необходимые средства для работы с перечисленными технологиями (и даже больше), поскольку этот язык является языком общего назначения. Но возникает резонный вопрос: как может ужиться чистота языка с заведомо «нечистыми» способами вычислений? Этот вопрос на примере системы ввода/вывода и будет детально исследован в этом разделе далее.

## Основы функционального ввода/вывода

Все предыдущие разделы, посвящённые вопросам использования языка Haskell, обходились без ввода или вывода информации. Рассматривались общие вопросы, определялись типы и взаимосвязанные функции, а результаты их работы исследовались при помощи интерпретатора (например, HUGS 98). Интерпретатор всегда выводит на экран результат, возвращаемый функцией, а потому использование ввода/вывода при выполнении расчётов задач смысла особого не имело.

Однако разработка современного программного обеспечения, как уже указано во введении, не может обойтись без взаимодействия с окружением программы, в первую очередь без взаимодействия с пользователем. Интерактивные программы уже настолько плотно вошли в нашу жизнь, что работа с интерпретаторами для получения результатов выглядит глубоко архаичной. Так что для умения создавать активно взаимодействующие программы необходимо изучить основы системы ввода/вывода в языке Haskell.

Но как быть с недетерминированностью и наличием побочных эффектов? Если принять их как есть, можно распрощаться с чистотой языка. Необходимо отметить, что некоторые функциональные языки

пошли в своей эволюции именно по этому пути — их разработчики пожертвовали чистотой, но внедрили в язык обычную системы ввода/вывода. Но разработчики языка Haskell пошли иным путём.

В языке Haskell создан некоторый достаточно узкий «мирок», где разрешены недетерминированность и побочные эффекты. Этот «мирок» ограничен в случае системы ввода/вывода единственным типом данных, с которым можно работать, — `IO`. Любая функция, которая осуществляет ввод или вывод, должна возвращать (а иногда и принимать на вход) значения типа `IO`. И более того, нет возможности выхода из этого маленького «мирка» — как только некоторый вычислительный процесс «попал в лапы» к системе ввода/вывода, он обречён остаться в ней до конца.

Итак, рассмотренные во введении гипотетические функции `read` и `put` в этом случае должны иметь следующие сигнатуры:

```
read :: IO Char
put :: Char -> IO ()
```

Как видно из этих описаний, тип `IO` представляет собой *контейнерный тип* — внутри себя он может содержать значения произвольного типа (про контейнерные типы дополнительно можно прочитать в эссе «Алгебраические типы данных в языке Haskell» в настоящем сборнике). Другим контейнерным типом, к примеру, является список `[]`. Да и более того, любой *алгебраический тип данных*, если только это не *перечисление*, является контейнерным типом. Так что в этом нет ничего особенного.

Тип данных `IO` как бы оборачивает собой недетерминированные операции и операции, связанные с побочными эффектами. Поэтому в рамках системы ввода/вывода такие операции и могут происходить только внутри типа `IO`. Это сделано для того, чтобы оставить сам язык Haskell чистым.

Но это ещё не всё. Для выстраивания последовательности действий ввода/вывода в синтаксисе языка Haskell имеется специальное ключевое слово `do`, которое позволяет последовательно выполнить несколько операций ввода/вывода — одну за другой. Например, при помощи исследованных ранее гипотетических функций `read` и `put` можно написать несложную программу — считать символ с клавиатуры и вывести его на экран. На языке Haskell это будет выглядеть следующим образом:

```
do c <- read
   put c
```

Ключевое слово `do` поддерживает так называемый *двумерный синтаксис*, когда последовательность операций ввода/вывода можно не отделять друг от друга точкой с запятой (`;`), как того требует строгий синтаксис языка, а записывать в столбик друг под другом. Главное, чтобы начало каждой операции находилось на том же знакоместе на новой строке, что и весь столбец (заодно необходимо отметить, что двумерный синтаксис повсеместно используется в языке Haskell и для записи других языковых конструкций).

В синтаксисе ключевого слова `do` имеются два момента. Первый — использование символа `(<-)`. Этот символ вводит *образец*, который сопоставляется со значением, которое возвращается функцией после символа `(<-)`. Если сопоставление прошло успешно, то ниже этот образец можно использовать с фактическим значением. Кроме того, символ `(<-)` «разворачивает» тип `IO`, поэтому тип значений в образце не имеет контейнерной оболочки. Так что выполнение первой операции ввода/вывода в примере выше запишет в образец `c` код символа, который пользователь ввёл при помощи клавиатуры.

Второй момент — операции ввода/вывода без символа `(<-)`. Эти операции просто выполняются, так что в примере выше во второй строке на экран будет выведен символ, код которого записан в образце `c`. Как видно, функция `put` принимает аргумент типа `Char`, и именно такой тип имеет значение в образце `c`, несмотря на то что функция `read` возвращает значение типа `IO Char`.

Кроме указанных двух типов операций ввода/вывода, в конструкции `do` могут присутствовать определения замыканий посредством ключевого слова `let`, при этом использование ключевого слова `in` не нужно. Это — синтаксическое послабление, которое ведёт к тому, что следующие два примера тождественны с точки зрения транслятора языка Haskell.

```
let p = someFunction a1 a2 a3
in do outputProcess p

do let p = someFunction a1 a2 a3
    outputProcess p
```

Ещё один аспект — вложение выражений **do** друг в друга. Можно неограниченно вкладывать списки операций ввода/вывода друг в друга, только необходимо помнить, что выравнивание операций в строках соответствует тому или иному уровню выражения **do**. Нарушение принципов двумерного синтаксиса может привести не только к синтаксическим ошибкам, но и к логическим, которые не проявляются на этапе компиляции, но самым беспощадным образом нарушают работоспособность программы во время её исполнения. Кроме того, такие ошибки очень сложно отлавливать.

Наконец, необходимо упомянуть, что любое выражение **do** имеет тип (впрочем, как и любое выражение в функциональной парадигме). Тип выражения **do** равен типу последней операции ввода/вывода в списке. Поэтому в списке операций ввода/вывода на последнем месте обязательно должна стоять операция, которая имеет некоторый тип, а не возвращает его в образец (то есть в последней строке выражения **do** не должно использоваться символа (`<-`), равно как и локального определения **let**).

Поскольку выражение **do** является обычным выражением с точки зрения транслятора языка Haskell, оно может использоваться во всех тех местах, где имеет смысл использование выражения соответствующего типа. Поэтому, к примеру, можно создавать следующие объекты (значения):

```
ioList = [do {c1 <- read; c2 <- read; put '>'},
          put 'a',
          do {put 'b'; put 'c'}]
```

Что интересно, определённая таким образом функция не приведёт к выполнению каких-либо операций ввода/вывода. При её вызове произойдёт простой возврат списка, состоящего из трёх операций ввода/вывода, каждая из которых имеет тип `Io()`. Так что тип функции `ioList` равен `[Io()]`. Это — замечательное свойство системы ввода/вывода в языке Haskell, которое делает саму систему полностью функциональной.

## Стандартные функции ввода/вывода

В стандартном модуле `Prelude` определён базовый набор функций, работающих с системой ввода/вывода в языке Haskell. Этих базовых функций вполне достаточно для решения большинства прикладных задач. В стандартной поставке любого транслятора языка имеется также дополнительный модуль `IO`, в котором определены расширенные возможности, позволяющие работать с каналами, исключениями (хотя и в модуле `Prelude` имеются средства для этого, но весьма ограниченные) и предоставляющие множество вспомогательных функций для работы с файлами, консолью и клавиатурой. Модуль `IO` является достаточно большим, его описание выходит за рамки этой книги. Поэтому ниже описываются только некоторые функции из стандартного модуля `Prelude`, которые позволят в следующем подразделе создать несколько интересных прикладных программ.

Самым первым объектом, который определён в системе ввода/вывода Haskell, является синоним типа для представления путей к файлам. Этот синоним определён исключительно для удобочитаемости сигнатур функций, работающих с файлами. Его определение выглядит следующим образом:

```
type FilePath = String
```

Второй важный тип, который используется при работе с файлами, представляет собой перечисление, описывающее режимы открытия файлов — открытие для чтения, записи, дозаписывания; а также для чтения и записи одновременно. Это стандартные константы, которые поддерживаются большинством современных операционных систем:

```
data IOMode
= ReadMode
| WriteMode
| AppendMode
| ReadWriteMode
deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)
```

Очень важными функциями являются функции для работы с дескрипторами файлов. Все они определены в виде *примитивов*, то есть объектов, запрятанных глубоко в недрах трансляторов и имеющих только отображение в стандартном модуле `Prelude`. Это отображение представляет собой сигнатуру, которая разрешает использование соответствующего идентификатора. Ниже перечисляются все такие примитивы:

- 1) `openFile :: FilePath -> IOMode -> IO Handle` — открывает файл по заданному имени и режиму открытия, возвращает дескриптор файла, который впоследствии может использоваться для работы с открытым файлом;
- 2) `hClose :: Handle -> IO ()` — закрывает указанный при помощи дескриптора файл;
- 3) `hGetContents :: Handle -> IO String` — читает всё содержимое заданного файла в одну строку;
- 4) `hGetChar :: Handle -> IO Char` — читает один символ с текущей позиции в заданном файле;
- 5) `hGetLine :: Handle -> IO String` — читает одну строку с текущей позиции в заданном файле, при этом строка заканчивается символом перевода строки;
- 6) `hPutChar :: Handle -> Char -> IO ()` — записывает один заданный символ на текущую позицию в заданном файле;
- 7) `hPutStr :: Handle -> String -> IO ()` — записывает строку на текущую позицию в заданном файле.

Тип `Handle` является синонимом типа `Int` (для текущей реализации интерпретатора HUGS 98), но может иметь иную реализацию в других трансляторах языка Haskell, поэтому при программировании необходимо использовать только этот синоним.

Кроме всего прочего, в виде примитивов определены три *константные функции*, которые возвращают стандартные потоки — `stdin` (стандартный поток ввода с клавиатуры), `stdout` (стандартный поток вывода в обычную консоль) и `stderr` (стандартный поток вывода в консоль ошибок). Все эти константные функции возвращают значение типа `Handle`, а сами возвращаемые значения отличаются от файлов тем, что их не надо ни открывать, ни закрывать — они всегда доступны для работы.

В принципе, перечисленных функций достаточно для выполнения произвольных операций ввода/вывода. Однако для удобной работы с клавиатурой и консолью определены дополнительные функции (все они выражены через перечисленные выше семь функций):

- 1) `putChar :: Char -> IO ()` — выводит заданный символ в стандартную консоль вывода `stdout`;
- 2) `putStr :: String -> IO ()` — выводит заданную строку в стандартную консоль вывода `stdout`;
- 3) `putStrLn :: String -> IO ()` — вариант функции `putStr`, добавляющий после выведенной в стандартную консоль вывода `stdout` строки символ перевода строки;
- 4) `print :: Show a => a -> IO ()` — выводит в стандартную консоль вывода `stdout` некоторое значение, которое может быть преобразовано в строку (тип этого значения должен иметь экземпляр класса `Show`);

- 5) `getChar :: IO Char` — читает из стандартного потока ввода `stdin` один символ;
- 6) `getContents :: IO String` — считывает всё содержимое стандартного потока ввода `stdin` в одну строку;
- 7) `getLine :: IO String` — считывает из стандартного потока ввода `stdin` одну строку, заканчивающуюся символом перевода строки;
- 8) `readLn :: Read a => IO a` — считывает из стандартного потока ввода `stdin` некоторое значение, которое может быть получено при помощи синтаксического разбора строки (тип этого значения должен иметь экземпляр класса `Read`), само значение должно занимать всю строку, заканчивающуюся символом перевода строки.

Как уже сказано, все эти функции выражены через примитивы. Вдумчивый читатель может самостоятельно открыть стандартный модуль `Prelude` для того, чтобы изучить способ такого выражения. Это позволит заодно изучить неплохие примеры разработки функциональных операций ввода/вывода.

Следующий набор функций предоставляет удобные инструменты для работы с файлами. Все они также выражены через перечисленные ранее семь примитивных функций:

- 1) `writeFile :: FilePath -> String -> IO ()` — создаёт файл с заданным именем, содержимым которого является заданная строка, сам файл после записи закрывается;
- 2) `appendFile :: FilePath -> String -> IO ()` — дозаписывает в заданный по имени файл заданную строку, сам файл после записи закрывается;
- 3) `readFile :: FilePath -> IO String` — открывает файл по заданному имени и считывает всё его содержимое в одну строку, которая возвращается в качестве результата, файл остаётся открытym;
- 4) `interact :: (String -> String) -> IO ()` — интересная функция, которая считывает всё содержимое стандартного потока ввода `stdin` в строку, применяет к этой строке заданную функцию, а результат работы этой функции выводит в стандартный поток вывода `stdout`.

Наконец, последней функцией для работы с системой ввода/вывода, которая описана в стандартном модуле `Prelude`, является функция `catch`. Эта функция предназначена для отлова *исключений*. Любая операция ввода/вывода в процессе своей работы может генерировать исключение некоторого типа. Это может произойти по многим причинам — заданный по имени файл отсутствует, нет прав доступа для чтения файла и т. д. Для того чтобы программа не произвела аварийного останова с выходом в операционную систему, такие исключения необходимо ловить и обрабатывать. Для этого как раз и используется функция `catch`, которая имеет следующую сигнатуру:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

Эта функция принимает на вход два параметра — операцию ввода/вывода, а также обработчик некоторого исключения (типы исключений описываются перечислением `IOError`). Если в процессе выполнения операции ввода/вывода исключений не произошло, функция `catch` возвращает результат операции. Если же произошло исключение, то оно передаётся на вход функции-обработчику, которая задана вторым аргументом функции `catch`. Эта функция может обработать исключение, тогда результат её работы будет и результатом вызова функции `catch`, а может и не обрабатывать. В последнем случае исключение будет генерировано повторно и передано в обработчик более высокого уровня. Этим достигается иерархичность обработки исключений. Самым верхним обработчиком исключений является системная функция, которая останавливает программу и выводит стандартное сообщение об ошибке. Этот обработчик вызывается всегда, если исключение не обработано ни одним из обработчиков, определённых программистом.

Таким образом, в стандартном модуле `Prelude` определены достаточные средства для решения произвольных прикладных задач, связанных с использованием ввода/вывода. В следующем разделе будут приведены некоторые примеры использования этих функций.

## Примеры программ

В языке Haskell имеется одно соглашение, сходное с соглашением об имени главной функции в таких языках программирования, как С или С++. Если программа готовится к компиляции, то в ней должна быть определена функция `main`, при этом она должна быть определена в модуле `Main` (или главном модуле без наименования — по умолчанию наименование модуля берётся как раз `Main`), а её тип должен быть:

```
main :: IO ()
```

Если подобные программы использовать в интерпретаторах (например, в интерпретаторе HUGS 98), то функция `main` будет запущена на исполнение при нажатии на кнопку «Запуск» на панели инструментов интерпретатора. Название главной функции должно учитываться при программировании прикладных задач.

### Вывод результатов исполнения функции на экран

До этого во всех примерах, приводимых в разделах книги, которые были посвящены языку Haskell, использовались функции, результаты которых выводились автоматически в интерпретаторе. Для этого было необходимо просто написать в строке приглашения интерпретатора наименование функции и список её фактических параметров, чтобы интерпретатор проинтерпретировал этот вызов функции и вывел результат на экран.

В этом моменте имеется одна очень важная вещь. Интерпретатор может вывести на экран только такое значение, тип которого имеет экземпляр класса `Show`. Например, функции с типом `a -> b` не могут быть выведены на экран в качестве результата вычисления, поскольку для этого типа нет экземпляра класса `Show`. Обычно такая ошибка происходит, если вызвать какую-либо функцию, передав ей на одно фактическое значение меньше, чем того требует сигнатура. В этом случае произойдёт частичное применение, результатом которого будет функция одного аргумента, и интерпретатор выведет на экран примерно следующее:

```
ERROR – Cannot find "show" function for:
*** Expression : flip (+) 1
*** Of type     : Integer -> Integer
```

Это происходит потому, что для типа любого значения, которое должен вывести на экран интерпретатор, он ищет экземпляр класса `Show`, чтобы преобразовать это значение в строку при помощи метода `show`, после чего строка выводится на экран. Весь этот процесс делается при помощи стандартной функции `print`.

То же самое необходимо делать и при выводе результатов работы функции в откомпилированной программе. Однако весь этот процесс необходимо делать уже вручную. В качестве примера можно рассмотреть получение списка из пяти первых совершенных чисел. Быстро написав функции для получения списка совершенных чисел:

```
divisors :: Integer -> [ Integer ]
divisors x = [ y | y <- [ 1 .. ( div x 2 ) ] ,
                    mod x y == 0 ]

perfects :: [ Integer ]
perfects = [ x | x <- [ 2 , 4 .. ] ,
                    sum ( divisors x ) == x ]
```

необходимо задуматься, а как вывести первые пять чисел на экран? Для этого необходимо определить функцию `main`, в которой произойдёт вычисление первых пяти совершенных чисел, после чего результат будет выведен на экран при помощи операции ввода/вывода. Достаточно просто:

```
main :: IO ()
main = do let p = take 5 perfects
          print p
```

В принципе, для таких простых примеров можно обойтись и без конструкции `do`, поскольку одна операция ввода/вывода может быть вызвана без этого ключевого слова, а само слово `do` необходимо только для связывания последовательности операций. Так что функция `main` может выглядеть так:

```
main :: IO ()
main = print $ take 5 perfects
```

## Альтернатива: экран или файл

А что, если для каких-то целей необходимо иметь возможность вывести результаты не только на экран, но и в некоторый файл, при этом предварительно имя файла необходимо запросить у пользователя? Для решения этой задачи такой простой функцией `main`, какая была в предыдущем подразделе, уже не обойтись. Пусть теперь необходимо вычислять список простых чисел, а для некоторого усложнения задачи количество чисел, которое необходимо вычислить, также будет запрашиваться у пользователя.

Функция для вычисления списка простых чисел несложная:

```
primes :: [Integer]
primes = sieve [2..]
where
    sieve (x:xs) = x:sieve (filter ((/= 0).(`mod` x)) xs)
```

А вот функция `main` будет уже посложнее:

```
main :: IO ()
main = do putStrLn "Привет. Сколько простых чисел вычислить: "
          npr <- readLn :: IO Int
          let prs = take npr primes
          putStrLn "Куда вывести результаты (f - файл): "
          dst <- getLine
          if (dst /= "f")
              then print prs
          else do putStrLn "Пожалуйста, введите имя файла: "
                  fn <- getLine
                  writeFile fn (show prs)
                  putStrLn ("Файл " ++ fn ++ " записан.")
```

Эту функцию необходимо рассмотреть более подробно. Ниже подробно исследуется каждая операция ввода/вывода, записанная в списке выражения `do` этой функции.

Первая операция выводит на экран при помощи стандартной функции `putStrLn` приветствие и вопрос о том, сколько простых чисел необходимо подсчитать. Вторая строка как раз предназначена для считывания с клавиатуры целого числа, которое введёт пользователь. Это число будет положено в образец `npr`. Поскольку количество простых чисел, которое необходимо подсчитать, уже известно, это можно сделать — в третьей строке создаётся замыкание `prs`, которое содержит список простых чисел, состоящий из введённого количества элементов.

Четвёртая операция опять является запросом относительно того, куда вывести результаты. Соответственно, пятая операция считывает символ с клавиатуры. В шестой строке имеется условное выражение, оформленное конструкцией **if-then-else**. В принципе, любое выражение, которое имеет тип, обрамлённый контейнером **I0**, может присутствовать в списке **do**. Поэтому в списке **do** могут присутствовать выражения **if** или **case** — главное, чтобы они возвращали соответствующее значение. Таким образом, введённый символ проверяется на неравенство символу "f" (на самом деле сравниваются строки, поскольку с клавиатуры всегда вводятся строки символов, даже если символ введён один). Если введённый символ не является символом "f", то стандартным образом на экран выводится подсчитанный список простых чисел.

В части **else**, однако, имеется новый список операций ввода/вывода, оформленный ключевым словом **do**. Это вполне возможно по причинам, описанным выше. Так что здесь опять происходят запрос у пользователя имени файла, запись в файл результата при помощи стандартной функции **writeFile**, а также вывод уведомления о том, что файл записан. Результат работы функции **main** может выглядеть, к примеру, следующим образом:

```
Привет. Сколько простых чисел вычислить: 1000
Куда вывести результаты (f – файл): f
Пожалуйста, введите имя файла: primes.txt
Файл "primes.txt" записан.
```

К сожалению, эта функция **main** имеет ряд существенных недостатков. Во-первых, она аварийно завершается, если пользователь введёт на запрос количества простых чисел какую-нибудь строку, которую невозможно преобразовать в целое число. Во-вторых, она также аварийно завершается, если файл, имя которого введено пользователем, невозможно создать или перезаписать (например, нет прав доступа или у файла имеется атрибут «только чтение»). Ну и, наконец, в-третьих, функция выводит результат на экран в любом случае, если пользователь ввёл не символ "f". Хорошо было бы, если на экран результаты выводились только по нажатии какого-нибудь определённого символа.

Было бы намного интересней, если бы на некорректный ввод во всех трёх случаях программа сообщала об этом и повторно запрашивала информацию. Вдумчивому читателю предлагается в целях тренировки самостоятельно реализовать такую функцию **main**. Это не так сложно, как может показаться на первый взгляд.

## Копирование файлов

Последний пример связан с копированием файлов. Небольшая утилита, которая запрашивает у пользователя имя существующего файла, а также имя нового файла, в который необходимо скопировать содержимое старого файла. Если оба имени корректны, утилита производит копирование. Если что-то неправильно, происходит какое-либо исключение, то утилита просит пользователя заново ввести все данные.

Для начала необходимо определить функцию, которая скопирует содержимое файла в другой файл, а также сообщит об этом. Применяя знания, полученные в подразделе, посвящённом стандартным функциям из модуля **Prelude**, определить такую функцию не составляет труда:

```
copyFile :: String -> String -> I0 ()
copyFile src dst = do body <- readFile src
                      writeFile dst body
                      putStrLn ("Файл \"" ++ src ++
                                "\" скопирован в \"" ++ dst ++ "\".)
```

Однако эта функция всё также не лишена недостатка — она совершил аварийный останов в случае, если с файлами существует какой-то непорядок: нет прав доступа, невозможно прочитать или записать

файл. Обработчик исключений должен охватывать эту функцию и «сторожить» её поведение. Поэтому вызов функции `catch` должен происходить в функции `main`:

```
main :: IO ()
main = do putStrLn "Пожалуйста, введите имя файла для копирования: "
          src <- getLine
          putStrLn "Введите новое имя файла: "
          dst <- getLine
          catch (copyFile src dst)
                (\e -> do putStrLn "Ошибка. Пожалуйста, повторите."
                           main)
```

Первые четыре строки в выражении `do` — это запросы имён файлов. Последняя строка наиболее интересна. Здесь и происходит обработка исключений посредством вызова стандартной функции `catch`. Первый operand этой функции — вызов функции `copyFile` с заданными именами файлов. А второй operand — обработчик ошибки. В данном случае обработчик написан достаточно просто. Он игнорирует тип ошибки, просто выводит на экран сообщение о том, что произошла какая-то ошибка, и запускает процесс сначала. Рекурсивный вызов функции `main` не должен настораживать — это обычное дело.

И опять же, имеется много возможностей для улучшения этой функции. Можно распознавать тип произошедшей ошибки и выводить на экран соответствующее сообщение. Можно не заставлять вводить пользователя оба имени снова, а просить вводить только то, с которым произошла ошибка. Можно придумать много чего ещё. Все эти новые возможности опять оставляются для самостоятельной проработки. Кроме того, вдумчивый читатель увидит здесь одну логическую ошибку, которая нарочно оставлена в целях обучения. Эта ошибка не приводит к генерации исключения, но может быть причиной непредсказуемого поведения программы. Кто первый из читателей сообщит о ней и о возможном способе её преодоления автору, получит интересный подарок.

## Заключение

Данный раздел кратко описывает систему ввода/вывода в чистом функциональном языке Haskell. В ней не затрагивается важнейший аспект этой системы — её реализация при помощи *монады*, поскольку эта теоретическая тема достаточно сложна и выходит за рамки простой научно-популярной книги. Вдумчивый читатель, желающий двигаться дальше, может самостоятельно изучать материалы, посвящённые монадам, поскольку монады являются одной из важнейших частей языка Haskell.

В следующем разделе тема системы ввода/вывода в языке Haskell будет раскрыта дополнительно.

# Простой интерпретатор команд

*Статья была опубликована в № 08 (44) журнала «Потенциал» в августе 2008 года.*

*В эссе рассматривается одна из интереснейших практических задач, заключающаяся в организации цикла интерпретации команд, вводимых пользователем с клавиатуры. Данная задача повсеместно возникает в интерактивных приложениях, от простых вопрос-ответных вычислительных программ до операционных систем. На примере функционального языка программирования Haskell показывается один способ реализации простого интерпретатора команд.*

## Введение

Практически в любой прикладной задаче имеется необходимость организации интерактивного взаимодействия с пользователем. Другое дело, что в различных типах приложений такое взаимодействие осуществляется разными способами. Например, в графических приложениях, управляемых операционной системой типа Windows, интерактивное взаимодействие с пользователем осуществляется при помощи графических управляющих элементов.

Особым случаем являются так называемые *консольные приложения*, в которых взаимодействие осуществляется при помощи ввода команд с параметрами и получения реакции системы на них. Во времена до появления графических операционных систем это был естественный способ взаимодействия. Да и сегодня многие прикладные программы всё так же используют консольный ввод/вывод для общения с пользователем.

Вышесказанное показывает, что создание системы функций, которая позволяет организовывать цикл интерпретации и взаимодействовать с пользователем в интерактивном режиме, является типовой задачей, для решения которой можно подготовить так называемый *повторно используемый компонент* в виде некоторого шаблона. Шаблон можно организовать в виде нескольких модулей, часть из которых может быть заменяема в зависимости от назначения интерпретатора. Таким образом, ниже в разделе предлагается один из вариантов решения этой задачи и создаётся простой интерпретатор команд.

## Постановка задачи

Разработка любого программного обеспечения начинается с постановки задачи, написания *технического задания*. Предварительно написание краткого, но проработанного технического задания даже для небольшой задачи является хорошей практикой и способствует быстрому осознанию целей и задач будущей программы, а также не менее быстрой реализации. Поэтому в этом разделе будет кратко сформулирована и formalизована задача, которую необходимо решить при разработке простого интерпретатора команд.

Итак, простой интерпретатор команд (далее — ПИК) должен:

- 1) при запуске выводить на экран приветственное сообщение, которое может состоять из названия программы и её версии;

- 2) постоянно находится в ожидании команды пользователя, которая вводится при помощи клавиатуры и завершается нажатием клавиши «Ввод»;
- 3) режим ожидания ПИК характеризуется тем, что на экране присутствует надпись «Введите команду:», а сам ПИК готов к вводу последовательности символов;
- 4) любая команда должна состоять из мнемонического имени и произвольного числа её аргументов, отделённых от наименования команды пробелом. Все аргументы также отделяются друг от друга пробелами;
- 5) команды могут иметь синонимы, которые полностью заменяют саму команду. Синоним не отличим от команды, нет никакой разницы — использовать команду или её синоним;
- 6) при вводе неизвестного набора символов, ПИК должен сообщать пользователю, что введённая последовательность не является допустимой командой;
- 7) не должно иметь значения, в каком регистре символов вводятся команды. Последовательности «St0rE» и «store» должны быть неразличимы для ПИК;
- 8) распознавание команды должно производиться при помощи сравнения с началом последовательности символов, составляющих наименование команды. Если введённая пользователем последовательность является началом какой-либо команды, ПИК выполняет эту команду. Другими словами, последовательность «sh» выполнит команду «show»;
- 9) каждая команда должна поддерживать специальный информационный режим, при использовании которого команда не выполняет своё обычное действие, но выводит на экран справочную информацию о самой себе. Параметр, который переводит команду в описанный режим, — «-?»;
- 10) команда «exit» должна позволять выйти из ПИК, при этом на экране должно быть выведено прощание. Возможный синоним команды — «quit»;
- 11) команда «help» должна выводить на экран краткую справочную информацию. Возможные синонимы команды — «?», «manual»;
- 12) команда «version» должна выводить на экран наименование и версию программы;
- 13) команда «store» должна записывать в таблицу строк заданную после неё строку символов. Пользователю должно разрешаться записывать одинаковые строки в таблицу;
- 14) команда «show» должна выводить на экран таблицу хранимых строк.

Как видно, эти четырнадцать несложных правил неплохо описывают функциональность простого интерпретатора команд. Набор команд состоит из пяти несложных действий, на примере которых можно легко понять принципы работы подобных интерпретаторов. Особый интерес представляют две последние команды — «store» и «show», поскольку они работают с некой таблицей символов, то есть, по сути, осуществляют некоторые действия, отличные от служебных — выхода из программы, вывода справочной информации и номера версии программы. Так что в дальнейшем рассмотрении этим двум командам необходимо будет уделить особенное внимание.

## Основной набор функций

Что такое простой интерпретатор команд? Его алгоритм работы на верхнем уровне достаточно прост: вывод запроса пользователю на ввод команды, обработка введённого, выполнение команды, вывод результатов её работы (и выполнение иного эффекта) и возврат в начало этого алгоритма. Получается,

что в *цикле интерпретации* необходимо получить команду от пользователя, выполнить её и вывести на экран результаты её выполнения; после чего всё начинается сначала.

Поэтому для реализации цикла интерпретации необходимо просто описать этот алгоритм на языке программирования. К тому же сам алгоритм весьма декларативен, поэтому его реализация на таком языке, как Haskell, будет простой. Однако перед определением функции, которая реализует описанный алгоритм, необходимо создать некоторые вспомогательные определения.

## Вспомогательные типы данных

Для начала необходимо определить типы данных, с которыми будут работать функции, отвечающие за организацию цикла интерпретации. Прежде всего это — тип *окружения* интерпретатора, в котором хранится информация о записываемых пользователем строках. Поскольку, кроме строк, ничего в окружении храниться не будет, его тип можно определить в виде синонима:

```
type Environment = [String]
```

Одно значение этого типа будет создаваться в самом начале запуска программы и передаваться из функции в функцию для обеспечения того, чтобы в любой момент времени произвольная функция знала состав окружения (даже в случае, если в самой функции это окружение не используется). Данная технология является обычной в функциональном программировании, поскольку глобальные переменные запрещены парадигмой (сторонние эффекты недопустимы, а каждая функция должна быть детерминированной). Само собой, что некоторые функции могут изменять окружение, некоторые могут им вообще не пользоваться, но в качестве параметра такое окружение будет передаваться везде, где оно было бы необходимо. Ниже при реализации суть этого будет ясна.

Второй тип данных, необходимый для работы интерпретатора, — код результата выполнения команд. Этот код необходим для того, чтобы в функции цикла интерпретации обрабатывать результат выполнения команды должным образом. Обычно в качестве множества значений такого кода используются два элемента — код выхода из программы и код успешного выполнения операции. Иногда добавляется код неуспешного выполнения операции, чтобы в цикле интерпретации выводить сообщение об этом в стандартный поток вывода сообщений об ошибках `cerr`. Также часто в это множество добавляют два кода — код необходимости записи окружения в файл и код необходимости чтения окружения из файла. Это делается для того, чтобы избавиться от необходимости использования системы ввода/вывода внутри функций, выполняющих команды, а сам ввод/вывод производился бы только на верхнем уровне в функции организации цикла интерпретации.

Итак, в случае простого интерпретатора команд такой тип может выглядеть следующим образом:

```
data ICode
  = IC_Exit
  | IC_Error
  | IC_HelpMessage
  | IC_Success
deriving Eq
```

Дополнительный конструктор `IC_HelpMessage` добавлен для различия того, что в качестве результата команда вернула справочную информацию. Возможно, что это может пригодиться при развитии интерпретатора, хотя в создаваемой версии этот конструктор использоваться не будет.

Наконец, весьма важным типом является тип значения, который возвращается каждой функцией, которая обрабатывает одну команду. В целях единообразия это будет один тип (да и было бы весьма непросто сделать обработку различных типов, отдельных для каждой команды, хотя и по такому пути пойти, в принципе, возможно). Этот тип должен объединять в себе код выполнения операции `ICode`, строку с сообщением о результате выполнения (если команда что-то рассчитывает, то результат расчётов

можно привести в строковый вид), а также новое значение окружения, которое могло поменяться в результате действия команды. Итого:

```
data IResult
= IResult
{
    code      :: ICode ,
    message   :: String ,
    environment :: Environment
}
```

Всё готово для реализации функции, организующей цикл интерпретации.

## Цикл интерпретации

Итак, как указано в самом начале этого раздела, цикл интерпретации заключается в выполнении одной команды, которая, возможно, модифицирует окружение. Это значит, что функция, реализующая цикл интерпретации, должна получать на вход значение типа `Environment`. Вроде бы ничего дополнительного ей для работы не требуется. Возвращает же эта функция пустое значение `I0 ()`. Собственно, реализация самой функции достаточно проста:

```
runICycle :: Environment -> I0 ()
runICycle env
= do putStrLn "Введите команду: "
     cmd <- getLine
     result <- interprete cmd env
     case (code result) of
         IC_Exit -> putStrLn "Всего доброго."
         _           -> do putStrLn (message result ++ "\n")
                           runICycle $ environment result
```

Как обычно, перевод декларативного алгоритма на некоторый функциональный язык программирования производится чуть ли не слово в слово. В списке операций ввода/вывода выражения `do` (первого уровня) имеются ровно четыре операции, ровно как в описанном в начале алгоритме. Первая строка с функцией `putStrLn` выводит на экран приглашение к вводу команды. Вторая строка с функцией `getLine` считывает с клавиатуры строку символов. Четвёртая строка с оператором `case` проверяет код результата выполнения команды, и если этот код сигнализирует о необходимости выхода из цикла интерпретации и программы (первая альтернатива), то происходит вывод на экран прощания. В противном случае, если код результата какой-то иной, происходит вывод на экран сообщения, которое вернула выполненная команда, а также запуск той же самой функции `runICycle` с новым значением окружения.

Наиболее интересной строкой в этом списке операций ввода/вывода является третья строка, в которой получается результат выполнения команды `cmd` при текущем значении окружения `env`. Функция `interprete`, которую ещё предстоит реализовать, возвращает в виде образца `result`, который имеет тип `IResult`, как раз результат выполнения команды. Как видно из её применения, она получает на вход два параметра типов `String` и `Environment`, а возвращает результат типа `I0 IResult`. Тело функции опять же не представляет ничего сложного:

```
interprete :: String -> Environment -> I0 IResult
interprete cmd env
= if (cmd == "") then return $
```

```

    IResult IC_Success "Пожалуйста, введите команду."
    env
else let (c:args) = words cmd
      command = findCommand c commands
in case command of
  Nothing -> return $
    IResult IC_Error "Ошибка. Нелепая команда."
    env
  Just fnc -> return $ fnc (unwords args) env

```

Прежде всего необходимо проверить, не является ли введённая пользователем команда пустой строкой. Если это так, то пользователю необходимо сообщить, что он должен ввести команду. Это делается при помощи операции в части `then` условия. Метод `return` «оборачивает» переданное ему значение в тип-контейнер `IO`. Во второй альтернативе, если введённая команда не является пустой строкой, она разбивается на «слова» (по пробельным символам) при помощи стандартной функции `words`. Эта функция получает на вход строку, а возвращает список строк. Таким образом первым элементом этого списка является наименование команды, а все последующие — её аргументы. *Замыкание* `command` создаётся при помощи функции `findCommand`, которая ещё будет реализована. Эта функция возвращает значение, «обёрнутое» в тип `Maybe`, — этот тип всегда используется в тех случаях, когда в после выполнения функции может не оказаться какого-либо результата. В данном случае получение значения `Nothing` свидетельствует о том, что функция для выполнения введённой команды не была найдена, то есть команда является неизвестной простому интерпретатору.

Соответственно, если функция `findCommand` находит требуемую команду и возвращает её в конструкторе `Just`, то эту команду необходимо выполнить, передав ей на вход список аргументов (в виде строки) и текущее значение окружения (ведь команда может работать с окружением). Но среди чего функция `findCommand` ищет функции, которые реализуют действия команд? Как видно из её применения, вторым аргументом ей на вход передаётся некоторая функция `commands` (на то, что это внешняя функция, указывает то, что она нигде не определена в функции `interpret` в виде замыкания или образца). Эта константная функция будет написана ниже — она просто возвращает список соответствий имён команд (строк) функциям. Её реализация будет показана в следующем разделе. Ну а функция `findCommand` выглядит не сложнее других, определённых ранее:

```

findCommand :: String -> [(String, Command)] -> Maybe Command
findCommand _ []           = Nothing
findCommand cmd ((n, c):cs) = if ((map toLower cmd) `isPrefixOf` n)
                                then Just c
                                else findCommand cmd cs

```

При определении этой функции используются две функции, входящие в модули стандартной поставки библиотек языка Haskell. Функция `toLower` переводит заданный символ в нижний регистр, а функция `isPrefixOf` возвращает значение `True`, если первый заданный список является началом второго (или они совпадают). Поэтому применение условия `((map toLower cmd) `isPrefixOf` n)` как раз и удовлетворяет требованиям к простому интерпретатору команд — команда должна распознаваться в любом регистре, а само распознавание должно вестись по начальным символам. Для того чтобы использовать эти функции, необходимо произвести подключение модулей:

```

import Char (toLower)
import List (isPrefixOf)

```

Теперь всё готово для реализации функции `main`. Она состоит из двух строк:

```
main :: IO ()
```

```
main = do putStrLn greetings
         runICycle []
```

Функция `greetings` является константной и содержит строку приветствия. Эта строка вынесена в константную функцию в целях повторного использования (она ещё пригодится при реализации функций, обеспечивающих выполнение команд). Её определение тривиально:

```
greetings :: String
greetings = "Простой интерпретатор команд. Версия 2007."
```

## Функции для исполнения команд

Теперь пришло время реализовать все функции, которые выполняют заявленные в требованиях команды. Но перед этим необходимо записать список таких функций и их отображение на различные команды в специальной константной функции `commands`, которая вызывается из функции `findCommand`. Из сигнатуры этой функции также ясно, что типом константной функции `commands` должен быть список пар `[(String, Command)]`. Остаётся неясным, что такое за тип `Command`. Это просто синоним функционального типа, созданный для удобства:

```
type Command = String -> Environment -> IResult
```

Этот тип должна иметь каждая функция, которая исполняет команду, введённую пользователем. Как видно, каждая такая функция принимает на вход строку аргументов и окружение, а возвращает обычный результат выполнения команды, описываемый типом `IResult`. Так что теперь можно создать список команд:

```
commands :: [(String, Command)]
commands = [ ("?",      doHelp),
            ("exit",    doExit),
            ("help",    doHelp),
            ("manual",  doHelp),
            ("quit",    doExit),
            ("show",    doShow),
            ("store",   doStore),
            ("version", doVersion)]
```

Как видно, этот список позволяет одновременно задавать мнемонические имена для команд и их синонимы. В таком списке можно держать произвольное количество синонимов команд, равно как и задавать новые команды. Поскольку функциональные языки программирования трактуют функции как обычновенные значения, нет ничего удивительного в том, что сами функции могут выступать в качестве элементов структур данных — кортежей и списков. Так что теперь остаётся реализовать пять функций: `doExit`, `doHelp`, `doShow`, `doStore` и `doVersion`.

Имеет смысл начать рассмотрение с функции `doHelp`. Она распечатывает общую справочную информацию, а также информацию о заданной команде. Так что внутри неё имеет смысл вызывать все остальные функции со специальным аргументом «`-?`», как это заявлено требованиями к простому интерпретатору команд.

Итак, функция `doHelp` выглядит примерно следующим образом:

```
doHelp :: Command
doHelp args env
= case args of
    ""    -> IResult
```

```

    IC_HelpMessage (greetings ++
                      "\nПожалуйста, введите \"help <cmd>\" \\
                       для получения информации \\
                       о команде \"cmd\".") env

"-?" -> IResult
    IC_HelpMessage "Выводит справочную информацию \\
                     о заданной команде." env

- -> let cmd = findCommand args commands
      in case cmd of
          Nothing -> IResult
              IC_Error ("Ошибка. Невозможно \\
                         найти справочную \\
                         информацию о команде \"\" \\
                         args ++ ".") env
          Just fnc -> fnc "-?" env

```

Как видно, если эта команда введена без аргументов, она выводит общую информацию о программе, в том числе и первоначальное приветствие с версией программы. Если передать этой функции в качестве аргумента заявленную строку «-?», то произойдёт вывод справочной информации о самой команде `help`. Если же ввести какой-либо иной аргумент, то произойдёт поиск команды опять всё в том же списке команд `commands`, и в случае если команда будет найдена, для получения справки она будет вызвана с аргументом «-?». Весьма изящно. Что будет, если в строке ввода простого интерпретатора команд ввести команду `help help?`

Наиболее простыми являются команды `doExit` и `doVersion`. Их реализация тривиальна:

```

doExit :: Command
doExit "-?" env = IResult
    IC_HelpMessage "Выходит из простого \\
                     интерпретатора команд." env

doExit args env = IResult
    IC_Exit "" env

doVersion :: Command
doVersion "-?" env = IResult
    IC_HelpMessage "Выводит информацию о \\
                     текущей версии программы." env

doVersion args env = IResult
    IC_Success greetings env

```

А вот функции `doShow` и `doStore` являются наиболее интересными, поскольку работают с окружением `env`, которое передаётся сквозь все функции простого интерпретатора. Первая команда не модифицирует окружение, но использует его для вывода сообщения на экран. Она немногим более сложная, чем остальные, рассмотренные до этого:

```

doShow :: Command
doShow "-?" env = IResult IC_HelpMessage "Выводит текущее содержимое \\
                                         таблицы строк." env

doShow args env = IResult IC_Success (showTable env) env
where

```

```

showTable []      = ""
showTable (s:[]) = s
showTable (s:ss) = s ++ "\n" ++ showTable ss

```

Наконец, функция `doStore` модифицирует окружение. Но и она сама по себе не так сложна:

```

doStore :: Command
doStore "-?" env = IResult
    IC_HelpMessage "Записывает заданную строку \\
                    в таблицу строк." env
doStore args env = IResult
    IC_Success ("Строка \"" ++ args ++
                "\" успешно сохранена.") (args:env)

```

Как видно, новая строка просто добавляется в голову имеющегося списка строк, представленного окружением. Это новое окружение возвращается в стандартном результате выполнения команды. Нет ничего проще.

Таким образом, описанным несложным способом можно реализовать произвольные наборы команд. Единственное, на что необходимо обращать внимание, — операции ввода/вывода недопустимы в командах при такой реализации, поскольку тип функций `Command` не имеет внутри себя типа `IO`. В случае, если необходимо сохранять окружение в файл и считывать его из файла, а также совершать иные операции ввода/вывода, необходимо или каким-либо образом указывать на этот факт более высоким функциям (например, функции `runICycle`, в которой обрабатываются результаты команд), либо менять тип `Command`, включая в него возможность работы с системой ввода/вывода. Последний способ целесообразен, когда большая часть команд работает с вводом/выводом.

## Заключение

В данном разделе, кроме непосредственного изучения системы ввода/вывода языка Haskell и реализации простого интерпретатора команд, также рассматривается интереснейший вопрос передачи некоторого окружения по всему набору функций, использующихся в цикле интерпретации. В рассмотренном примере эта передача повсюду осуществлялась вручную. Как сказано, это сделано нарочно, чтобы не загружать читателя более серьёзными и сложными технологиями.

Дело в том, что такая «сквозная» передача параметра из одной функции в другую является типовой задачей, которая часто встречается в различных программах. Само собой, что для её решения уже имеется готовая технология, основанная на *монаде*, так же как и сама система ввода/вывода языка Haskell. Данная монада называется `State` — «Состояние». Читатели, заинтересовавшиеся вопросом, могут самостоятельно изучить принципы применения монады `State`.

Дополнительно ознакомиться с пониманием монад в функциональном программировании и в языке Haskell в частности можно при помощи следующих источников: [11, 19, 22].

# Теория чисел и язык Haskell

*Статья была опубликована в № 05 (17) журнала «Потенциал» в мае 2006 года (первая статья из цикла статей о языке Haskell).*

Это эссе рассматривает функциональный язык программирования Haskell на примере некоторых интересных задач из теории чисел, и предлагаются численные способы их решения. Методология представления функций на языке Haskell основана на том, чтобы сделать определения таких функций наиболее похожими на математические формулы.

## Введение

Теория чисел — это одно из направлений математики, которое иногда называют «высшей арифметикой». Данная наука изучает натуральные числа и некоторые сходные с ними объекты, рассматривает различные свойства (делимость, разложимость, взаимосвязи и т. д.), алгоритмы поиска чисел, а также определяет ряд достаточно интересных наборов натуральных чисел.

Так, к примеру, в рамках теории чисел рассматриваются вопросы делимости целых чисел друг на друга, алгоритм Евклида для поиска наибольшего общего делителя, поиск наименьшего общего кратного, малая и большая теоремы Ферма. В качестве самых известных рядов натуральных чисел можно привести ряд Фибоначчи, простые числа, совершенные и дружественные числа, степени и суперстепени натуральных чисел.

С другой стороны, в рамках функционального программирования существуют различные методы для вычисления значений сложных формул. А если рассмотреть мощь и выразительность современных функциональных языков программирования, то становится очевидным, что изучать на практике различные аспекты теории чисел можно при помощи программирования формул на каком-нибудь функциональном языке.

В этом разделе в качестве функционального языка программирования, при помощи которого можно рассматривать теорию чисел, предлагается использовать язык Haskell как уже зарекомендовавший себя язык для использования в науке и прикладных технологиях. Более того, язык Haskell используется в качестве первого языка программирования в некоторых университетах мира, поэтому его рассмотрение для решения задач из теории чисел имеет ещё и практическую цель — научить читателя обращаться с этим языком естественным и ловким образом.

Для работы с функциями, приведёнными в этом разделе, необходимо использовать интерпретатор языка Haskell HUGS 98, бесплатную версию которого можно получить в сети Интернет по адресу <http://www.haskell.org/hugs/>. Все приводимые ниже функции протестированы в этом интерпретаторе, поэтому правильность их определения гарантируется автором. Использование других трансляторов языка Haskell (например, компилятора GHC) также возможно, однако для их использования, вероятно, придётся вносить в определения функций незначительные изменения.

Предполагается, что читатель знаком с базовым синтаксисом языка Haskell, поэтому далее объяснение синтаксиса языка будет проводиться минимальным образом. В случае, если у читателя возникнут

затруднения с пониманием синтаксиса языка или смысла приводимых определений, можно посоветовать обратиться к изданным книгам [6, 7, 8].

## Простейшие задачи

Перед тем как начать рассмотрение каких-то сложных вопросов теории чисел, необходимо провести подготовительную работу в виде разработки некоторых вспомогательных функций, требуемых для вычисления более сложных формул. К таким функциям относятся в первую очередь функции для нахождения наибольшего общего делителя (НОД) и наименьшего общего кратного (НОК).

НОД двух целых чисел  $m$  и  $n$  — это такой общий делитель  $d$  (то есть:  $d \mid m$  и  $d \mid n$ ), который делится на любой другой общий делитель исходных чисел. НОД определён, если хотя бы одно из чисел  $m$  или  $n$  не ноль. Обозначение —  $(m, n)$ . Для вычисления этого числа можно воспользоваться функцией `gcd` (от английского наименования *greatest common divisor*):

```
gcd :: Integral a => a -> a -> a
gcd 0 0 = error "НОД от 0 и 0 не определён."
gcd m n = gcd' (abs m) (abs n)
where
  gcd' m 0 = n
  gcd' m n = gcd' n (rem m n)
```

В этом определении использована функция `abs`, вычисляющая модуль заданного целого числа, а также функция `rem`, которая возвращает остаток от целочисленного деления первого аргумента на второй. Данная функция реализует алгоритм Евклида, который был разработан знаменитым философом для нахождения НОД ещё во времена Древней Греции.

Необходимо напомнить, что строка с символами `(::)` является определением типа функции, тело которой определяется в следующей строке. То есть такая строка определяет сигнатуру. В ней используются два специальных символа: `(=>)` и `(->)`. Первый задаёт контекст использования переменных типа в дальнейшей записи (в указанном примере — переменная `a`). В функции `gcd` аргументы могут быть любого типа `a`, являющегося экземпляром класса `Integral`, то есть классом чисел, для которых определены операции целочисленного деления и взятия остатка от деления.

Стрелка `(->)` используется для определения типа функций. Так, к примеру, запись типа функции `Integer -> Bool` гласит, что функция принимает на вход один параметр типа `Integer`, а возвращает результат типа `Bool`. В свою очередь, запись типа `Integer -> Char -> Bool` говорит, что у функции есть два аргумента: первый типа `Integer`, а второй — типа `Char`. Возвращает функция значение типа `Bool`.

Подробно о типизации функций, классах типов и параметрических переменных типов написано в упомянутых уже книгах по функциональному программированию. Детальное рассмотрение этих аспектов функционального программирования выходит за рамки рассмотрения данного раздела.

НОК двух целых чисел  $m$  и  $n$  — это такое наименьшее целое число, которое делится на  $m$  и  $n$  без остатка. Обозначение —  $[m, n]$ . Для вычисления этого числа можно воспользоваться функцией `lcm` (от английского наименования *least common multiple*):

```
lcm :: Integral a => a -> a -> a
lcm _ 0 = 0
lcm 0 _ = 0
lcm m n = abs ((quot m (gcd m n)) * n)
```

Здесь также встречается уже рассмотренная функция `abs`, а так же функция `quot`, возвращающая значение целочисленного деления первого аргумента на второй. Как видно, НОК вычисляется достаточно

просто — необходимо разделить первый аргумент на НОД двух чисел, а потом результат деления умножить на второй аргумент.

Написанные функции можно использовать для построения бесконечного списка взаимно простых чисел. Два целых числа называются взаимно простыми, если их НОД равен 1. Для вычисления такого списка чисел можно воспользоваться следующей функцией:

```
reciprocals :: Integral a => [(a, a)]
reciprocals = [(m, n) | m <- [1..], 
                      n <- [1..m], 
                      gcd m n == 1]
```

Как видно, это определение полностью соответствует математической формуле, по которой можно было бы вычислить множество пар взаимно простых чисел:

$$\mathbb{N}_{reciprocals} = \{(m, n) \mid m \in \mathbb{N}, n \in \mathbb{N}, (m, n) = 1\} \quad (13)$$

Данная функция не очень интересна с точки зрения её реализации. Всё довольно типично — определитель списка с двумя генераторами (выражения со знаком ( $<-$ )) и одним охраняющим выражением (условное выражение `gcd m n == 1`). Здесь интересно другое. Ведь приведённое определение функции, хоть и является таким простым, на самом деле содержит два скрытых вложенных цикла и одну проверку. Циклы соответствуют генераторам, при этом первый цикл выполняется от единицы до бесконечности, а второй — от единицы до значения переменной первого цикла. Это важно, так как если бы в определении функции стояли бы два одинаковых генератора: `m <- [1..]` и `n <- [1..]`, то результаты работы были бы не так интересны — первый элемент любой пары в полученном бесконечном списке всегда был бы равен единице. Читателю предлагается самостоятельно подумать, почему это так.

Необходимо отметить, что функции `gcd` и `lcm` определены в стандартном модуле `Prelude`, поэтому их определение здесь приведено исключительно в познавательных целях. При разработке собственных программ эти функции можно использовать непосредственно без дополнительного определения.

Одним из самых ключевых понятий теории чисел является понятие делителя. Очень многие целочисленные последовательности, в том числе и те, которые будут рассмотрены далее, определяются через делители. Поэтому было бы интересно иметь функцию для получения списка делителей заданного числа. Пусть такая функция называется `divisors`:

```
divisors :: Integral a => a -> [a]
divisors n = [x | x <- [1..(n - 1)], 
                  rem n x == 0]
```

Необходимо отметить, что данная функция возвращает список так называемых собственных делителей числа  $n$ , то есть таких, которые строго меньше самого числа  $n$ . Таким образом, в результат этой функции не входит само число  $n$  — это свойство будет использоваться в некоторых случаях в последующих определениях функций.

## Такие непростые простые числа

Очень широкую известность в рамках теории чисел имеют простые числа, то есть такие, в списке собственных делителей которых находится только один делитель — 1. Такие числа нашли самое широкое применение во многих прикладных областях, в том числе и в современных методах и алгоритмах шифрования информации. Кроме того, простые числа успешно используются в хеш-таблицах и для генерации псевдослучайных чисел.

К сожалению, в математике не придумано простой формулы для нахождения заданного по порядку простого числа, поэтому построение списка простых чисел делается перебором с применением

всевозможных эвристических правил проверки на простоту. К множеству таких правил относится, к примеру, решето Эратосфена — алгоритм нахождения при помощи перебора всех простых чисел до некоторого заданного  $n$ .

Проще всего находить простые числа перебором:

```
primes = [n | n <- [1..],
             isPrime n]
where
  isPrime x = (divisors x == [1])
```

Однако данный алгоритм весьма несовершенен. Незачем перебирать все числа, тем более что из чётных чисел только число 2 является простым. Однако далее получается, что и все числа, кратные трём, не являются простыми, а там и кратные пяти тоже и т. д. Так и появилось решето Эратосфена — хорошо бы было его внедрить для построения бесконечного списка простых чисел. Читателю предлагается самостоятельно подумать над этой проблемой.

Описанная функция `primes` вполне подходит для решения многих задач в рамках теории чисел. Хотя она работает долго, но зато вполне надёжно, вычисляя действительно бесконечный список простых чисел. Чтобы получить ограниченный список простых чисел, не превышающих заданного  $n$ , необходимо воспользоваться стандартной функцией `take`, которая возвращает заданное количество элементов с начала списка (например, `take 1000 primes` — вернёт список из тысячи первых простых чисел). А для того чтобы получить определённое простое число, можно воспользоваться функцией `(!!)`, возвращающей заданный элемент списка: `primes !! 1000` вернёт тысячное простое число.

Всем вышеперечисленным можно воспользоваться для того, чтобы написать функцию, возвращающую разложение заданного натурального числа на простые делители. По основной теореме арифметики такое разложение существует, и оно единственное (с точностью до порядка следования простых делителей). Такое представление натурального числа в виде произведения простых называется факторизацией. На настоящий момент неизвестно каких-либо алгоритмов факторизации чисел с полиномиальной сложностью, хотя и не доказано, что таковых алгоритмов нет. На гипотезе о том, что факторизовать произвольное число не так просто, основан алгоритм шифрования с открытым ключом RSA.

Таким образом, функция для факторизации заданного числа хотя и будет весьма неоптимизированной, но, тем не менее, вполне будет работать. Особенно для несложных составных чисел, которые равны произведению преимущественно маленьких простых чисел. Её определение выглядит так:

```
expansion :: Integer -> [Integer]
expansion 1 = []
expansion n = x:expansion (quot n x)
where
  primesBN = takeWhile (=< n) primes
  x = head [y | y <- primesBN,
              mod n y == 0]
```

Данная функция будет работать медленнее для чисел, которые раскладываются на большие простые числа. Так, к примеру, число 1 000 000 (миллион) раскладывается на простые множители за доли секунды ([2, 2, 2, 2, 2, 2, 5, 5, 5, 5, 5]), а вот следующее за ним число 1 000 001 (миллион один) факторизуется примерно за полминуты ([101, 9^901]). Читателю предлагается самостоятельно изучить зависимость времени исполнения приведённого алгоритма факторизации от величины аргумента.

Доказано, что ряд простых чисел бесконечен. Однако среди всех таких чисел имеются так называемые числа-близнецы, то есть такие, которые отличаются друг от друга на 2. Неизвестно, сколько таких пар и бесконечно ли их количество вообще. Простейшее описание функции, вычисляющей такие числа, выглядит следующим образом:

```
twins :: [(Integer, Integer)]
```

```
twins = [(p, p + 2) | p <- primes,
           isPrime (p + 2)]
```

Кроме чисел-близнецов, можно вводить ряды пар простых чисел, отличающихся друг от друга на 4, на 6 и т. д. Такие ряды имеют обобщённое наименование родственных простых чисел. Для поиска пар родственных чисел можно написать функцию, параметризованную разницей, которая должна быть между родственными числами. Определение такой функции может выглядеть так:

```
kins :: Integer -> [(Integer, Integer)]
kins n = [(p, p + n) | p <- primes,
             isPrime (p + n)]
```

В этом случае определение функции для поиска простых чисел-близнецов выглядит очень просто:

```
twins :: [(Integer, Integer)]
twins = kins 2
```

Однако, несмотря на всю кажущуюся простоту простых чисел, математики очень любят их. И в связи с этим постоянно ищут различные свойства, которые характеризуют простые числа. Более того, для некоторых целей выделяются особые простые числа, которые получают свои собственные имена, часто по имени своего первооткрывателя. У всех таких подмножеств простых чисел имеются собственные области применения.

## Числа Мерсенна

Так, к примеру, в XVII веке французский математик М. Мерсенн определил последовательность чисел вида:

$$M_n = 2^n - 1 \quad (14)$$

Эта последовательность получила наименование «чисел Мерсенна». Сама по себе она не так интересна, но в ней существуют так называемые простые числа Мерсенна, которые получили свою известность в связи с эффективным критерием простоты Люка — Лемера, благодаря которому простые числа Мерсенна давно удерживают лидерство как самые большие известные простые числа. На данный момент самым большим известным простым числом является число Мерсенна  $M_{30402457} = 2^{30402457} - 1$ , найденное в декабре 2005 года. Оно содержит 9 152 052 десятичные цифры.

Эффективный тест простоты (тест Люка — Лемера) для чисел Мерсенна был предложен в 1878 году и базируется на том наблюдении, что простота числа Мерсенна  $M_p = 2^p - 1$  влечёт простоту его индекса  $p$ , а также на следующем утверждении: «для простого  $p$  число  $M_p$  является простым тогда и только тогда, когда оно делит число  $L_{p-1}$ , где числа  $L_k$  определяются рекуррентным соотношением —  $L_1 = 4$ ,  $L_{k+1} = L_k^2 - 2$ ».

Для установления простоты  $M_p$  последовательность чисел  $L_1, L_2, \dots, L_{p-1}$  достаточно вычислять по модулю числа  $M_p$  (то есть вычислять не сами числа  $L_k$ , длина которых растёт экспоненциально, а остатки от деления  $L_k$  на  $M_p$ , длина которых ограничена  $p$  битами). Последнее число в этой последовательности  $L_{p-1} \bmod M_p$  называется вычетом Люка — Лемера. Таким образом, число Мерсенна является простым тогда и только тогда, когда число  $p$  — простое и вычет Люка — Лемера равен нулю.

Для вычисления последовательности простых чисел Мерсенна можно воспользоваться следующими несложными функциями (необходимо в очередной раз заметить, что данные определения весьма далеки от оптимизированного варианта — они лишь показывают, насколько определения функций на языке Haskell похожи на математические формулы):

```
lucas :: (Num a, Num b) => b -> a
lucas 1 = 4
lucas n = (lucas (n - 1))^2 - 2
```

```

mersenne :: [Integer]
mersenne = [m p | p <- primes,
                  rem (lucas (p - 1))
                  (m p) == 0]
where
  m p = 2^p - 1

```

При помощи функции `mersenne` во время написания данного подраздела вычислены первые семь простых чисел Мерсенна: [7, 31, 127, 8 191, 131 071, 524 287, 2 147 483 647].

## Числа Ферма

Эксцентричный учёный П. Ферма, придумавший малую и большую<sup>7</sup> теоремы своего имени, которые долгое время мучили математиков (а большая и вовсе до последнего времени не была доказана), также интересовался простыми числами, в связи с чем пытался разработать формулу, при помощи которой можно было бы оные числа вычислять. После некоторых усилий он получил такое соотношение:

$$F_n = 2^{2^n} + 1 \quad (15)$$

Сам П. Ферма смог проверить простоту чисел из данной последовательности только до  $n = 4$ . Далее он просто предположил, что остальные числа из этого ряда тоже простые. Однако в 1732 году Л. Эйлер нашёл разложение числа  $F_5$ . На сегодняшний день известно, что все числа Ферма для  $5 \leq n \leq 32$  являются составными. Большие числа из этого ряда на простоту пока не проверены.

Для проверки простоты чисел Ферма используется тест Пепина, являющийся полиномиальным. Данный тест утверждает, что число Ферма  $F_n$  является простым тогда и только тогда, когда  $3^{\frac{F_n-1}{2}} \equiv -1 \pmod{F_n}$ .

Читателю рекомендуется самостоятельно реализовать функцию или набор функций для осуществления теста Пепина для простых чисел Ферма.

## Числа Софи Жермен

Софи Жермен доказала большую теорему Ферма для показателей  $n$ , являющихся простыми числами  $p$ , такими, что числа  $2p + 1$  также простые. Тем самым подмножество таких простых чисел получило наименование чисел Софи Жермен. Неизвестно, является ли эта последовательность бесконечной, хотя предполагается, что это так.

Для вычисления пар чисел Софи Жермен можно воспользоваться следующей функцией:

```

germain :: [(Integer, Integer)]
germain = [(p, 2 * p + 1) | p <- primes,
                           isPrime (2 * p + 1)]

```

## Другие последовательности простых чисел

Существует ещё большое количество различных подмножеств простых чисел, которые используются как для развлечения, так и для некоторых прикладных аспектов науки и техники. Так, к примеру, выведены формулы для простых чисел имени Вильсона (на сегодняшний день известны три таких числа) и имени Вольстенхольма (известны два таких числа). В англоязычных математических справочниках вводится до шестидесяти различных типов простых чисел, многие из которых используются в доказательствах тех или иных теорем.

<sup>7</sup>Данная теорема утверждает, что для любого целого  $n > 2$  уравнение  $a^n + b^n = c^n$  не имеет положительных целых решений  $a, b$  и  $c$ . Сам П. Ферма доказал теорему для  $n = 4$ , затем Л. Эйлер доказал её для  $n = 3$ , а позже И. Дирихле привёл доказательство для  $n = 5$ . Окончательно доказали теорему только в 1994 году для любого  $n$ .

Особый интерес у учёных, занимающихся теорией чисел, вызывают так называемые факториальные простые числа. Факториальным называется такое простое число, которое отличается на единицу в ту или иную сторону от факториала некоторого натурального числа:  $n! \pm 1$ . Эти числа интересны тем, что сигнализируют своим присутствием о начале или конце длинной последовательности составных чисел. Для получения бесконечного списка таких простых чисел можно воспользоваться следующими функциями:

```
fact :: (Num a, Enum a) => a -> a
fact n = product [1..n]

fp :: [Integer]
fp = [p | n <- test,
         p <- n,
         isPrime p]

where
  test = [[x-1, x+1] | x <- map fact [1..]]
```

Функция `product` из стандартного модуля `Prelude` возвращает произведение элементов переданного ей в качестве аргумента списка.

С другой стороны, простые числа можно использовать и для развлечения. Например, в ряду простых чисел можно искать такие, которые читаются одинаково в обе стороны (в десятичной системе счисления) — числа-палиндромы. Такие палиндромы можно также искать и среди простых чисел в различных системах счисления. Читателю предлагается самостоятельно разработать функцию для проверки того, что заданное число является палиндромом, и реализовать бесконечный список простых чисел-палиндромов.

## Совершенству нет предела

Другим широким классом чисел, любимых в теории чисел, являются так называемые совершенные числа, которые равны сумме всех своих собственных делителей. Совершенных чисел очень мало. В натуральном ряду до одного миллиона встречаются только четыре таких числа, а до триллиона — всего шесть. Проще всего искать такие числа при помощи следующей функции:

```
perfects :: [Integer]
perfects = [n | n <- [1..],
               sum (divisors n) == n]
```

Функция `sum` из стандартного модуля `Prelude` возвращает сумму элементов переданного ей в качестве аргумента списка. Однако данное определение весьма несовершенно. Если обратиться к теории чисел, то в ней можно найти доказательство того, что чётные совершенные числа и числа Мерсенна, рассмотренные в предыдущем подразделе, связаны друг с другом простым соотношением:

$$P_p = 2^{p-1} * (2^p - 1), \quad (16)$$

где число  $2^p - 1$  является простым (числом Мерсенна). Таким образом, каждому чётному совершенному числу соответствует число Мерсенна, и наоборот.

Это соотношение нашёл ещё древнегреческий математик Евклид, а строго доказал Л. Эйлер. Однако не доказано, существуют ли нечётные совершенные числа, потому данное соотношение необходимо использовать с осторожностью. Известно лишь, что если нечётное совершенное число существует, то оно должно превышать  $10^{300}$ .

Используя указанное соотношение, можно реализовать функцию, вычисляющую совершенные числа не медленным перебором, а достаточно быстро:

```

perfects :: [Integer]
perfects = [p n | n <- primes,
                  isPrime (m n)]
where
  p n = 2^(n - 1) * (m n)
  m n = 2^n - 1

```

Дополнительно проверить получаемые числа можно при помощи предиката `isPerfect`:

```

isPerfect :: Integral a => a -> Bool
isPerfect n = sum (divisors n) == n

```

Однако математики не остановились на такой формулировке. В обиход было введено понятие дружественных чисел, которые связаны друг с другом таким соотношением, при котором первое число в паре равно сумме собственных делителей второго числа, а второе — сумме собственных делителей первого соответственно. Таким образом, совершенные числа являются дружественными по отношению сами к себе. Список пар дружественных чисел можно получить при помощи следующей функции:

```

friends :: [(Integer, Integer)]
friends = [(m, n) | m <- [1..],
                   n <- [1..(m - 1)],
                   sum (divisors m) == n,
                   sum (divisors n) == m]

```

Как уже упоминалось, совершенные числа очень редки. Для того чтобы поле для исследований в этом направлении было немного шире, математики ввели некоторые дополнительные определения целочисленных последовательностей: недостаточное число и избыточное число. К недостаточным относятся такие натуральные числа, сумма собственных делителей которых меньше самого числа. Соответственно, к избыточным относятся числа, сумма собственных делителей которых больше самого числа. Таким образом, весь класс натуральных чисел может быть раздёлен на три непересекающихся подмножества — недостаточных, совершенных и избыточных чисел. А каждое натуральное число, в свою очередь, находится в одном из этих трёх подмножеств.

Но и этого адептам математики показалось мало. Были введены слегка недостаточные и слегка избыточные числа. Такие числа отличаются от совершенных в ту или иную сторону ровно на единицу. Однако при определении подобных множеств математиков ждало некоторое разочарование. Если использовать следующие функции для поиска таких чисел:

```

imperfects :: [Integer]
imperfects = [n | n <- [1..],
                  sum (divisors n) == n - 1]

excesses :: [Integer]
excesses = [n | n <- [1..],
                  sum (divisors n) == n + 1]

```

то будет ясно, что первая функция возвращает список степеней числа 2, а вторая функция не выдаёт вообще никакого результата. Так и получилось — в математике до сих пор неизвестно, существуют ли иные слегка недостаточные числа, кроме степеней двойки, а также существуют ли в принципе слегка избыточные числа.

Читателю рекомендуется самостоятельно разработать функции для генерации бесконечных списков недостаточных и избыточных чисел, а также разработать функцию высшего порядка, при помощи которой можно получить все пять перечисленных классов натуральных чисел — совершенные, избыточные и слегка избыточные, недостаточные и слегка недостаточные числа.

## Заключение

Теория чисел — занимательная наука. На ней основаны многие интересные аспекты прикладных технологий в самых различных областях науки и техники. Знание основ теории чисел помогает разрабатывать более оптимизированные алгоритмы в вычислительных задачах, а также успешно применять численные методы при решении различных задач при помощи вычислительной техники. Более того, теория чисел позволяет быть более внимательным к различным числовым последовательностям, развивает умение находить скрытые взаимосвязи в казалось бы хаотических множествах чисел. Всё это, в свою очередь, самым благотворным образом сказывается на развитии интеллектуальных способностей человека.

В сети Интернет по адресу <http://www.research.att.com/~njas/sequences/> находится энциклопедия целочисленных последовательностей (к сожалению, на английском языке), в которой представлена информация более чем о ста тысячах различных конечных и бесконечных последовательностей, состоящих из целых чисел. Данную энциклопедию в том числе можно использовать и для самостоятельного создания новых задач в рамках теории чисел для последующего решения методами функционального программирования.

# Магические квадраты и решение переборных задач

Статья была опубликована в № 06 (30) журнала «Потенциал» в июне 2007 года.

В данном эссе рассматриваются типовые способы решения задач на перебор, которые часто попадаются в рамках функционального программирования и программирования вообще. В качестве примера для рассмотрения предлагаемых методик даётся задача получения списка магических квадратов для заданного размера квадрата. Рассмотрение, как обычно, производится на функциональном языке программирования Haskell.

## Введение

Большая советская энциклопедия даёт сухое и математически чёткое определение магического квадрата:

*Магический квадрат — квадрат, разделённый на равное число  $n$  столбцов и строк, со вписанными в полученные клетки первыми  $n^2$  натуральными числами, которые дают в сумме по каждому столбцу, каждой строке и двум большим диагоналям одно и то же число (равное, как легко доказать,  $\frac{n(n^2+1)}{2}$ ). Доказано, что магический квадрат можно построить для любого  $n$ , начиная с  $n = 3$ . Существуют магические квадраты, удовлетворяющие ряду дополнительных условий, например магический квадрат с 64 клетками, который можно разбить на 4 меньших, содержащих по 16 клеток квадрата, причём в каждом из них сумма чисел любой строки, столбца или большой диагонали одна и та же (130). В Индии и некоторых других странах магические квадраты употребляли в качестве талисманов. Составление магических квадратов — классический образец математических развлечений и головоломок.*

Но строгая наука, как обычно, суха. На самом же деле магические квадраты с древнейших времён используются для различных целей, связанных с волшебством и колдовством, — иначе откуда у них такое название? И действительно, магические квадраты — это символы числовой гармонии, которые выражают космический принцип миропорядка. Каждый магический квадрат иллюстрирует законы онтологической симметрии, тем самым через него постигается присутствие рационального начала в мироздании.

В магической практике использовались квадраты, предназначенные для обретения сверхъестественных способностей — полётов в облике птиц, понимания языка животных, обретения невидимого для глаз состояния, нахождения кладов, раскрытия секретов прошлого, проникновения в будущее и т. п. Имелись квадраты-заклинания, направленные против врагов. Даже переписывание их в тетрадь представлялось весьма опасным, и маг в таких случаях воспроизводил их с умышленной ошибкой.

Разные магические квадраты приписывались разным силам. Так, к примеру, числовой квадрат с трёхклеточными сторонами соответствует планете Сатурн. Он демонстрирует симметрию между чётными и нечётными числами. Квадрат с четырёхклеточными сторонами относится к Юпитеру. Квадрат с пятиклеточными сторонами является архетипом Марса, считалось, что заклинания над ним развивают воинственность. Квадрат с шестиклеточными сторонами символизирует Солнце. Квадрат с семиклеточными сторонами соотносится с Венерой, так же как трёхклеточный, демонстрировал симметрию чётных и нечётных цифр.

Как видно, магические квадраты в древние времена имели чрезвычайно серьёзное практическое значение. А посему можно предположить, что знание о магических квадратах весьма важно, поэтому их изучение может способствовать развитию магической силы. Ну а людям, настроенным на абсолютное рациональное мышление, понравится решать задачу на поиск всех магических квадратов заданного размера в автоматизированном (или даже автоматическом) режиме, так как со времён древних философов и мистиков мы получили в своё распоряжение такие прекрасные инструменты для этих целей, как электронные вычислительные машины и функциональное программирование.

Принимая во внимание оба указанных аспекта, далее в этом разделе рассматриваются переборные задачи сложного характера с большим объёмом перебора на примере поиска магических квадратов. Рассмотрение ведётся при помощи функционального языка программирования Haskell, поэтому для понимания читатель должен быть знаком с синтаксисом этого языка в объёме опубликованных в книгах [6] и [14].

## Простейший вариант перебора

Для того чтобы начать двигаться в каком-то направлении при решении поставленной задачи, необходимо рассмотреть какие-нибудь простейшие частные случаи, чтобы понять, что должно происходить при поиске магических квадратов. Для этих целей подойдёт магический квадрат размера  $3 \times 3$ , так как такой квадрат один (с точностью до поворотов и отражений), а потому осуществить его поиск будет легко. Кроме того, количество вариантов для перебора не так велико.

Что есть магический квадрат размера  $3 \times 3$  с математической точки зрения? Это матрица указанного размера, в которой размещены числа от 1 до 9 таким образом, что суммы чисел в горизонтальных рядах, вертикальных столбцах и двух диагоналях равны между собой. Это наталкивает на мысль о том, как должна выглядеть функция для перебора в первом приближении:

```
(//) :: Eq a => [a] -> a -> [a]
[] // y      = []
(x:xs) // y = if (x == y)
              then xs
              else x:(xs // y)

(///) :: Eq a => [a] -> [a] -> [a]
s /// [] = s
s /// (x:xs) = (s // x) /// xs

ms_3 :: [[Integer]]
ms_3 = [[x1, x2, x3,
          y1, y2, y3,
          z1, z2, z3] |
        x1 <- [1..9],
        x2 <- [1..9] /// [x1],
        x3 <- [1..9] /// [x1, x2],
        y1 <- [1..9] /// [x1, x2, x3],
```

```

y2 <- [ 1..9 ]  /// [ x1 , x2 , x3 ,
                      y1 ] ,
y3 <- [ 1..9 ]  /// [ x1 , x2 , x3 ,
                      y1 , y2 ] ,
z1 <- [ 1..9 ]  /// [ x1 , x2 , x3 ,
                      y1 , y2 , y3 ] ,
z2 <- [ 1..9 ]  /// [ x1 , x2 , x3 ,
                      y1 , y2 , y3 ,
                      z1 ] ,
z3 <- [ 1..9 ]  /// [ x1 , x2 , x3 ,
                      y1 , y2 , y3 ,
                      z1 , z2 ] ,
x1 + x2 + x3 == y1 + y2 + y3 ,
x1 + x2 + x3 == z1 + z2 + z3 ,
x1 + x2 + x3 == x1 + y1 + z1 ,
x1 + x2 + x3 == x2 + y2 + z2 ,
x1 + x2 + x3 == x3 + y3 + z3 ,
x1 + x2 + x3 == x1 + y2 + z3 ,
x1 + x2 + x3 == x3 + y2 + z1 ]

```

Вспомогательные операции `(//)` и `(///)` используются для получения списка с исключённым из него элементом и подсписком соответственно.

Как видно, эта функция достаточно громоздка. Однако она полностью воспроизводит математическое определение, данное магическому квадрату размера  $3 \times 3$ . Она использует представление матрицы в виде списка из девяти элементов (что не так принципиально), а сами элементы выбираются из набора чисел от 1 до 9, при этом на каждом следующем шаге из этого набора удаляются предыдущие выбранные элементы. После чего производится проверка.

Запуск этой функции на исполнение в интерпретаторе HUGS 98 даст неутешительные результаты. Весь поиск займёт около минуты времени, но при этом будет произведено 150 миллионов *редукций*, будет задействовано 267 миллионов ячеек памяти, а сборщик мусора запустится 295 раз. И это всё только ради того, чтобы получить один-единственный магический квадрат (в результате, конечно, будет выведено 8 изоморфных друг другу квадратов) следующего вида:

2	7	6
9	5	1
4	3	8

Это совершенно неприемлемый результат, так как это не позволит вычислить ни одного магического квадрата размером  $4 \times 4$  и выше за доступное время. Поэтому необходима какая-то оптимизация. Первое, что приходит в голову после изучения функции генерации, — это использование магической суммы для сравнения в *выражениях охраны*. Ведь формула её вычисления известна. Ну а второе — использование иного порядка перебора и поиска, который позволит отсечь заведомо тупиковые ветви дерева перебора. Например, новая, более оптимизированная функция может выглядеть следующим образом:

```

magicSum :: Fractional a => a -> a
magicSum n = n * (n^2 + 1) / 2

ms_3' :: [[Double]]
ms_3' = [[x1 , x2 , x3 ,
           y1 , y2 , y3 ,
           z1 , z2 , z3] |

```

```

x1 <- [1..9] ,
x2 <- [1..9] /// [x1] ,
x3 <- [1..9] /// [x1, x2] ,
x1 + x2 + x3 == ms ,
y1 <- [1..9] /// [x1, x2, x3] ,
z1 <- [1..9] /// [x1, x2, x3,
               y1] ,
x1 + y1 + z1 == ms ,
y2 <- [1..9] /// [x1, x2, x3,
               y1, z1] ,
x3 + y2 + z1 == ms ,
y3 <- [1..9] /// [x1, x2, x3,
               y1, z1, y2] ,
y1 + y2 + y3 == ms ,
z2 <- [1..9] /// [x1, x2, x3,
               y1, z1, y2,
               y3] ,
x2 + y2 + z2 == ms ,
z3 <- [1..9] /// [x1, x2, x3,
               y1, z1, y2,
               y3, z2] ,
z1 + z2 + z3 == ms ,
x3 + y3 + z3 == ms ,
x1 + y2 + z3 == ms]

```

where `ms = magicSum 3`

Выполнение функции `ms_3'` происходит уже практически мгновенно. При этом требуется 208 тысяч редукций и 365 тысяч ячеек памяти. Это выигрыш в эффективности почти в 723 раза по сравнению с предыдущим определением. Можно ещё поиграться с дальнейшей оптимизацией алгоритма перебора, например удаляя из набора `[1..9]` не подсписки, а одиночные элементы, полученные на предыдущем шаге перебора, определяя локальные переменные для этих целей, но смысла в этом особенного нет, так как в подобных определениях функций имеются две проблемы: одна — не очень серьёзная, а другая — весьма серьёзная.

Первая заключается в том, что при использовании функции `magicSum`, которая вычисляет сумму магического квадрата, происходит выход за пределы множества натуральных чисел. Это нехорошо, ибо использование операций над числами с плавающей точкой, когда такие числа даже не используются, необоснованно. Эта проблема решается просто заменой определения функции (самостоятельно можно доказать, что это определение тождественно по результату предыдущему, то есть имеет место *экстенсиональное тождество* определений функций):

```

magicSum :: Integral a => a -> a
magicSum n = sum [1..(n ^ 2)] `div` n

```

Использование такого определения ещё больше снижает затраты вычислительных ресурсов. Вместо вычисления суммы элементов списка можно также пользоваться оператором целочисленного деления `div` в предыдущем определении. Эффективность примерно одинаковая.

Однако новое определение функции `magicSum` не устраниет другую проблему, более серьёзную. Проблема сия связана с весьма частным видом функций `ms_3` и `ms_3'` — они предназначены только для вычисления магических квадратов  $3 \times 3$ . А что, если потребуется вычислить магические квадраты произвольного размера  $n \times n$ ? Эти функции, естественно, не подойдут. Само собой разумеется, что

необходимо создать универсальное определение функции для любого заданного  $n$ . Но подойти к этой задаче не так просто...

## Перебор с использованием перестановок

При рассмотрении задачи для произвольного размера магического квадрата  $n \times n$  возникает проблема представления такого квадрата в памяти. Можно отметить, что предложенный в предыдущем разделе способ представления в виде списка имеет свою привлекательность как наиболее простой способ. Им можно воспользоваться и в общем случае.

Однако всё ещё стоит вопрос о способах получения различных комбинаций, которые будут рассматриваться в качестве магических квадратов. В этом деле поможет такая отрасль математики, как комбинаторика, так как она имеет достаточные механизмы для описания различных способов перебора. К ней и необходимо обратиться.

Для решения задачи о построении списка магических квадратов размерности  $n \times n$  необходимо получать всевозможные комбинации чисел от 1 до  $n^2$ , выстроенные в список, после чего каждую комбинацию проверять на «магичность». Это — простейшее понимание обобщённого переборного алгоритма. А для получения списка всех возможных комбинаций заданных чисел как раз и можно воспользоваться одним из понятий комбинаторики. Это понятие — *перестановка* (необходимо отметить, что сегодня функции для получения перестановок и других комбинаторных размещений имеются в стандартном модуле `Data.List`).

Перестановками из  $n$  элементов называются комбинации таких элементов, каждое из которых содержит все  $n$  элементов, отличающихся поэтому друг от друга только порядком расположения элементов. Например, из 3 элементов  $(a, b, c)$  можно образовать следующие перестановки:  $abc, bac, cab, acb, bca, cba$ . Число всех возможных перестановок, которые можно образовать из  $n$  элементов, обозначается символом  $P_n$  и вычисляется по формуле:

$$P_n = n! \quad (17)$$

Таким образом, необходимо создать функции для генерации всех возможных перестановок для элементов списка от 1 до  $n^2$  и последующей проверки сгенерированных комбинаций. Первую функцию написать достаточно просто:

```
permutations :: Eq a => [a] -> [[a]]
permutations [] = [[]]
permutations l = [x:ps | x <- l,
                      ps <- (permutations (l // x))]
```

Список всех перестановок для заданного списка элементов вычисляется Первый *клоуз* в этом определении является выходом из рекурсии, определяя результат функции на пустом списке. Второй клоуз определения — самый главный. Именно он определяет способ вычисления списка перестановок элементов от 1 до  $n^2$ . Смысл можно понять, прочитав определение этого клоуза примерно так:

*Перестановки элементов списка  $l$  вычисляются как список списков, где голова каждого списка выбирается из списка  $l$ , а хвост — из списка перестановок, полученных для исходного списка  $l$  без выбранного на предыдущем шаге элемента  $x$ .*

Имея в своём арсенале эту функцию, можно получить список всех возможных перестановок чисел от 1 до  $n^2$ . Остается проверить такие комбинации на «магичность». А для этого необходима функция (или набор функций), которая возвратит булевское значение в зависимости от того, является ли переданная ей на вход комбинация магическим квадратом. При представлении магических квадратов в виде списка такая функция может выглядеть следующим образом:

```

isMagic :: Int -> [Int] -> Bool
isMagic n ms = if (n^2 /= length ms)
    then False
    else (testH s n ms) &&
        (testV s n ms) &&
        (testD s n ms)
where s = magicSum n

```

Эта функция для заданного размера  $n$  и заданной комбинации  $ms$  возвращает значение `True`, если комбинация  $ms$  является магическим квадратом  $n \times n$ , и значение `False` в противном случае. В её определении используются вспомогательные функции `testH`, `testV` и `testD`, которые применяются для проверки суммы чисел горизонталей, вертикалей и диагоналей магического квадрата  $ms$  на равенство магической сумме. Эти функции определить не так сложно, принимая во внимание способ представления магического квадрата.

```

testH :: Int -> Int -> [Int] -> Bool
testH _ _ [] = True
testH s n ms = (sum (take n ms) == s) &&
    (testH s n (drop n ms))

testV :: Int -> Int -> [Int] -> Bool
testV s n ms = testV' s n n ms
where
    testV' _ 0 _ _ = True
    testV' s a n ms =
        (sum [ms !! ((a - 1) + (j * n)) | j <- [0..(n - 1)]] == s) &&
        (testV' s (a - 1) n ms)

testD :: Int -> Int -> [Int] -> Bool
testD s n ms =
    (sum [ms !! (i * (n + 1)) | i <- [0..(n - 1)]] == s) &&
    (sum [ms !! ((i + 1) * (n - 1)) | i <- [0..(n - 1)]] == s)

```

Функция `testH` устроена просто. Она последовательно сверяет сумму каждого  $n$  элементов переданного ей для проверки списка с магическим числом, которое также передаётся в качестве входного параметра (это сделано в качестве оптимизации). Это делается выборкой первых  $n$  элементов (стандартная функция `take`), проверкой их суммы и передачей оставшихся элементов списка (стандартная функция `drop`) в эту же самую функцию `testH`.

Функция `testV` немного сложнее. Главный смысл её работы заключается в получении списка элементов исходного списка, расположенных в одном столбце. Для этого используется вспомогательная функция `testV'`, у которой во втором клозе первым операндом конъюнкции (`&&`) как раз и находится формула для вычисления позиций элементов в списке, которые в квадрате находятся в одном столбце (для нумерации столбца используется параметр `a`). Читателю рекомендуется самостоятельно проверить эту формулу.

Наконец, функция `testD` просто проверяет две диагонали квадрата. Опять же, вдумчивый читатель найдёт проверку формул получения элементов из диагоналей квадрата, который представляется списком, интересной задачей.

Остаётся реализовать главную функцию для получения списка магических квадратов размера  $n \times n$ . После проведённых подготовительных работ это не представляется сложным делом. Её определение выглядит просто:

```
getMagicSquares :: Int -> [[Int]]
```

```
getMagicSquares n = [ms | ms <- permutations [1..(n^2)],
                        isMagic n ms]
```

Как видно, здесь устроен перебор всех возможных перестановок с последующей проверкой их на «магичность». В очередной раз можно удивиться выразительности языка Haskell. Единственное, что необходимо объяснить, — это использование ограничения на тип функции: `Int -> [[Int]]`. Использование типа `Int` (ограниченные целые числа) вместо `Integer` (целые числа неограниченного размера) позволяет весьма серьёзно снизить ресурсные затраты на проведение вычислений.

Замеры производительности работы этой функции имеет смысл делать для  $n = 3$ , так как для больших  $n$  функция работает слишком медленно. Её запуск с параметром 3 даёт такие результаты: 140 миллионов редукций, 205 миллионов ячеек памяти и 230 запусков процесса сборки мусора. Как видно, эти значения не сильно отличаются от значений функции `ms_3`, поэтому можно предположить, что для  $n > 3$  функция будет работать очень медленно. Поэтому имеет смысл задуматься об оптимизации, ибо, несмотря на то что функция `getMagicSquares` стала универсальной с точки зрения размера магических квадратов, её производительность всё же оставляет желать лучшего.

## Перебор с использованием размещений

При рассмотрении предыдущего алгоритма можно было увидеть одну довольно интересную его особенность, которая и сводит на нет все возможности построения за реальное время магических квадратов размера  $4 \times 4$ . Особенность сия заключается в том, что во время осуществления перебора совершается полная выборка комбинации, которая только после этого проверяется на «магичность», хотя эту проверку можно сделать до получения полного набора чисел (ср. как это сделано в функции `ms_3`).

Например, в случае размера  $4 \times 4$  нет никакой надобности рассматривать всю комбинацию, если сумма первых четырёх чисел не равна 34. Поэтому для магического квадрата размера  $n \times n$  необходимо ограничивать перебор при помощи проверки суммы каждого  $n$  чисел в получаемой комбинации. Это поможет сделать понятие *размещения*, которое также определяется в комбинаторике.

Размещениями из  $n$  элементов по  $k$  называются комбинации, которые можно образовать из заданных  $n$  элементов, собирая в каждую комбинацию по  $k$  элементов, при этом сами комбинации могут отличаться друг от друга как самими элементами, так и порядком их взаимного расположения. Например, из 3 элементов  $(a, b, c)$  по 2 можно образовать следующие размещения:  $ab, ac, ba, bc, ca, cb$ . Число всех возможных размещений, которые можно образовать из  $n$  элементов по  $k$ , обозначается символом  $A_n^k$  и вычисляется по формуле:

$$A_n^k = \frac{n!}{(n-k)!} \quad (18)$$

Итак, в случае магического квадрата размера  $n \times n$  необходимо генерировать  $n$  размещений из  $n^2$  по  $n$  и каждое из них проверять на соответствие магической сумме. Для этого нужна функция генерации всех размещений элементов из заданного списка. Определить её несложно:

```
arrangements :: (Num a, Eq b) => a -> [b] -> [[b]]
arrangements 0 _ = [[]]
arrangements k l = [x:as | x <- l,
                           as <- (arrangements (k-1) (l // x))]
```

Смысл её вполне понятен — первый аргумент определяет количество элементов в размещении, а второй — список элементов. В результате будет выдан список списков, каждым элементом которого является список, являющийся одним из размещений по  $k$  элементов из списка  $l$ . Однако, как видно, эта функция не генерирует магические квадраты заданного размера, так как она должна лишь использоватьсь для генерации таких квадратов. Поэтому необходима дополнительная функция, которая возвращала бы список списков, каждым элементом которого было бы представление одного варианта

«кандидата» в магические квадраты, как это делает функция `permutations`. Реализовать такую функцию можно по аналогии:

```
constructSquares :: Integral a => a -> [a] -> [[a]]
constructSquares [] = []
constructSquares n 1 = [a ++ as | a <- (arrangements n 1),
                                sum a == s,
                                as <- (constructSquares n (1 //| a))]
where s = magicSum n
```

Как видно, определение этой функции практически полностью повторяет определение функции `permutations` (конечно, в рамках используемых понятий), за исключением нового условия охраны, которое отсекает перебор в случаях, если очередной набор из  $n$  чисел не проходит первоначальный тест на «магичность». В этом выражении охраны использовано локальное определение `s` для того, чтобы вычислить магическую сумму для заданного `n` только один раз, а не для каждого `a` в отдельности (один из способов оптимизации вычислений — читатель самостоятельно может сравнить показатели эффективности в случаях использования *локальной переменной* и в случае её отсутствия).

Остается создать новое определение функции для получения списка магических квадратов заданного размера. Это уже не представляет и вовсе никакой сложности:

```
getMagicSquares' :: Int -> [[Int]]
getMagicSquares' n = [ms | ms <- constructSquares n [1 .. (n ^ 2)],
                        isMagic n ms]
```

Однако как оценить возможности нового определения? Сможет ли оно помочь получить список всех магических квадратов с размером хотя бы  $4 \times 4$ ? Для получения ответа на этот вопрос необходимо провести небольшое исследование, которое поможет оценить время выполнения функции перед её непосредственным запуском. Для этого необходимо собрать фактические данные, которые обозримы за малое время. Это — замеры вычислительных параметров для функций `getMagicSquares` и `getMagicSquares'` на аргументах 1, 2, и 3. Все эти замеры были предварительно сделаны и сведены в одну таблицу:

$n$	<code>getMagicSquares</code>	<code>getMagicSquares'</code>		
1	801	1 352	975	1 488
2	13 742	19 850	8 983	13 152
3	$2.42 \cdot 10^8$	$3.44 \cdot 10^8$	$4.78 \cdot 10^6$	$6.69 \cdot 10^6$
4	$4.36 \cdot 10^{15}$	$7.02 \cdot 10^{15}$	$1.47 \cdot 10^{11}$	$1.96 \cdot 10^{11}$

В первой строке для каждой функции показано количество проведённых во время вычислений редукций, вторая — количество использованных ячеек памяти соответственно. Четвёртая строка таблицы показывает вычисленные (прогнозируемые) значения. Как были получены эти значения? При помощи несложных вычислений.

Первоначально было замечено, что *порядок* (степень числа 10) перечисленных значений изменяется в некоторой последовательности. Это можно видеть в представленной таблице — для первой функции порядок изменяется как: 2, 4, 8 (очень похоже на степени числа 2); для второй функции последовательность такая: 2, 4, 6 (вроде просто чётные числа, но это может быть и неверным предположением). Соответственно, из предположения о том, что отношение порядков значений не изменяется при увеличении  $n$ , сделано вычисление этих параметров для  $n = 4$ . Например, значение параметра в первой ячейке четвёртой строки вычислено по следующей формуле:

$$R_4 = R_3 \cdot \frac{\left(\frac{R_3}{R_2}\right)^2}{\left(\frac{R_2}{R_1}\right)} = \frac{R_3^3 \cdot R_1}{R_2^3} \quad (19)$$

Аналогично вычисляются и значения других ячеек четвёртой строки (при вычислении по таблице значения могут немного отличаться от тех, что представлены, так как при вычислении во время эксперимента использовались более точные значения, чем те, которые внесены в таблицу).

Для проверки полученных результатов можно построить график (см. рис. 4), при помощи которого можно оценить правильность спрогнозированных результатов. Конечно, из-за гигантских значений необходимо использовать *логарифмическую шкалу* по оси  $Y$ , которая характеризуется тем, что равные расстояния по ней соответствуют равным отношениям величин. Использование такой шкалы следует ещё и из того, что именно порядок величин применяется при прогнозировании.

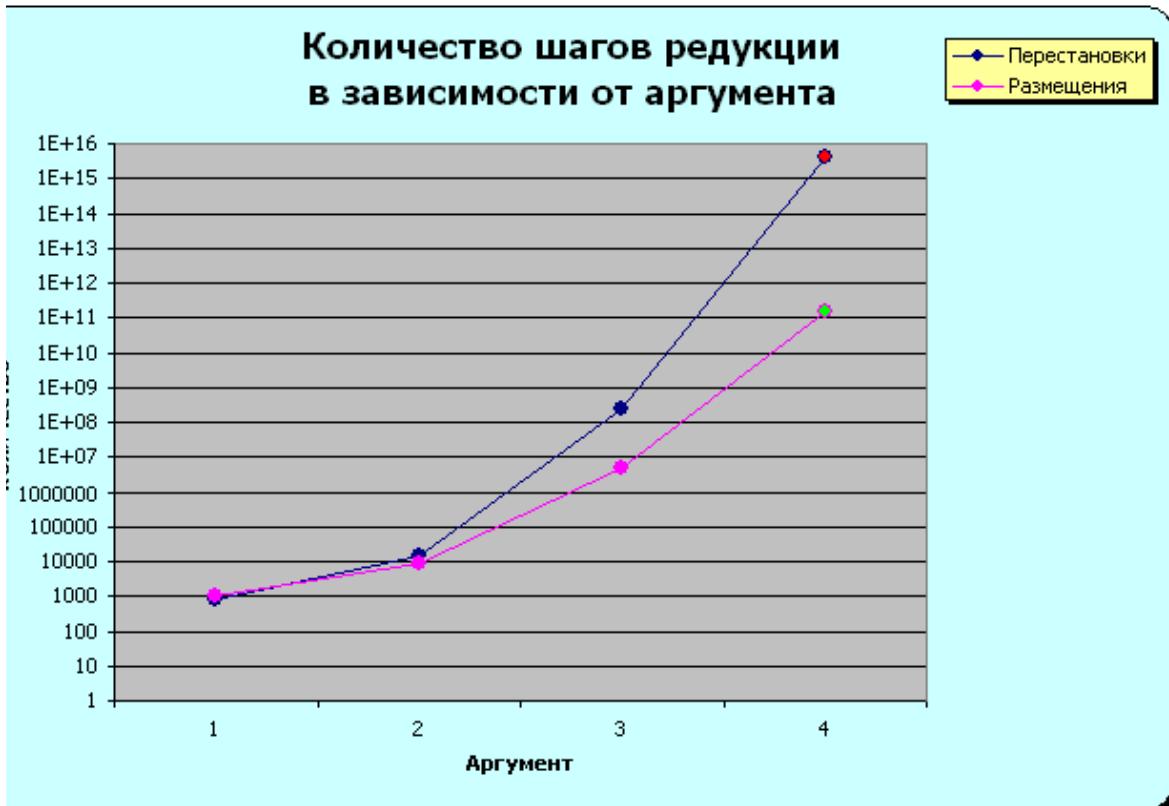


Рис. 4. График изменения количества редукций для функций `getMagicSquares` и `getMagicSquares'`

Как видно на графике, кривые достаточно плавные, а потому новые точки (спрогнозированные) вполне вписываются в изучаемую канву. Следовательно, на основании этих значений можно спрогнозировать время, которое будет затрачено на перебор и поиск всех магических квадратов размера  $4 \times 4$ . Время можно вычислить по следующей формуле (учитывая, что время вычислений зависит линейно от количества редукций):

$$T_4 = T_3 \cdot \frac{R_4}{R_3} \quad (20)$$

Для вычисления всех магических квадратов размера  $3 \times 3$  при помощи первой функции было затрачено 28.98 секунды, при помощи второй — 1.13 секунды (эти времена замерены уже после компиляции функций в компиляторе GHC в целях ускорения процесса, поэтому они отличаются от временных значений, упомянутых ранее). Соответственно, для функции `getMagicSquares` время выполнения для  $n = 4$  будет равно 522 миллиона секунд (16 с половиной лет), а для функции `getMagicSquares'` — 34.7 тысячи секунд (9 с половиной часов).

После получения такого оптимистического результата было решено проверить теоретические выкладки на практике. Для этого на ночь была запущена функция `getMagicSquares'` с параметром 4. И результат

был получен: через 10 часов было получено 7040 магических квадратов размера  $4 \times 4$ , первый и последний из которых представлены ниже:

1	2	15	16
12	14	3	5
13	7	10	4
8	11	6	9

16	15	2	1
5	3	14	12
4	10	7	13
9	6	11	8

Как видно, эти два квадрата являются зеркальным отражением друг друга. Действительно, алгоритм получает все магические квадраты, в том числе отражённые и повёрнутые. Количество неизоморфных друг другу магических квадратов, таким образом, необходимо получать, деля общее их количество на 8. Для размера  $4 \times 4$  это количество будет равняться 880.

И действительно, в 1930 году немецкий математик Ф. Фиттинг теоретически рассчитал количество магических квадратов размером  $4 \times 4$ , которое по его расчётом равно как раз 880. А ещё ранее, в XVII веке французский математик В. Френникль построил (вручную) все возможные магические квадраты размера  $4 \times 4$ .

## Дальнейшая универсализация алгоритма

Полученные функцией `getMagicSquares'` результаты впечатляют. Однако у неё всё ещё остаются недоработки, которые желательно устраниить. Недоработки эти касаются уже больше интерфейсного плана, нежели оптимизации переборного процесса. Например, вывод магических квадратов на экран оставляет желать лучшего — они представляются в виде списка, а потому так и выводятся. Интерпретировать результаты очень сложно, читать полученные записи неудобно. С другой стороны, представленный алгоритм работает только с числовыми магическими **квадратами**, хотя в целях развлечения можно придумать и нечисловые квадраты, а также прочие магические фигуры. Из нечисловых магических квадратов, к примеру, можно привести такой, который до сих пор используется разными людьми в «магических» целях:

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
R	O	T	A	S

Вот и хотелось бы сделать некоторый интерфейсный *класс*, который описывал бы методы для генерации различных комбинаций и дальнейшей проверки их на «магичность». Это позволит определять дополнительные типы данных и реализовывать для них методы генерации магических фигур.

Описать сам интерфейсный класс достаточно просто (необходимость введения изоморфного списка типа `MagicList` для представления списка магических фигур будет объяснена позже):

```
newtype MagicList f = ML [f]

class Magic f where
    getMagicFigures :: Int -> MagicList f
    isMagic          :: f -> Bool
```

Как видно, этот класс описывает два метода: для генерации списка магических фигур (`getMagicFigures`) и для проверки некой комбинации на «магичность» (`isMagic`). Теперь остаётся определить новые *алгебраические типы данных* (АТД), при помощи которых представлять в памяти сами магические фигуры. Список в этом деле также может помочь, но можно сделать и более интересные способы представления, которые помогут в том числе оптимизировать вычислительные процессы.

Собственно, изученные уже числовые магические квадраты можно представлять списком, но добавить к нему дополнительное поле в виде размера квадрата, чтобы не заниматься постоянной передачей этого параметра из функции в функцию. Это делается при помощи такого определения:

```
data Square a =
    Square
    {
        dimension :: Int,
        values     :: [a]
    }
```

Тут видно, что данный АТД можно использовать для представления любых квадратов. Чтобы его использовать, необходимо описать экземпляр класса `Magic` для этого АТД. Так как пока исследуются только числовые магические квадраты, да и невозможно сделать общее правило для проверки «магичности» числовых и нечисловых квадратов, экземпляр класса будет несколько суженным:

```
instance Magic (Square Int) where
    getMagicFigures d =
        ML [sq | sq <- constructSquares d [1 .. (d ^ 2)], isMagic sq]

    isMagic (Square d v) =
        if (d ^ 2 /= length v)
            then False
        else (testH ms d v) &&
            (testV ms d v) &&
            (testD ms d v)
    where ms = magicSum d
```

Как видно, здесь использованы старые вспомогательные функции для генерации комбинаций, а также для проверки «магичности» по горизонталям, вертикалям и диагоналям. В этом нет ничего зазорного.

Но больше всего вызывает интерес определение для этого АТД экземпляра стандартного класса `Show`, методы которого используются для вывода значений типов на экран (и в файлы). Хотелось бы, чтобы на экране квадраты выглядели именно квадратами, а не невзрачными списками чисел. Да и список квадратов тоже надо бы приукрасить, а именно выводить квадраты один за другим с отделением друг от друга пустой строкой. Для этого и необходим изоморфный тип `MagicList` — его также необходимо сделать экземпляром класса `Show` (в случае, если бы не было определения такого типа, невозможно было бы переопределить способ вывода списка магических фигур на экран). Такие экземпляры определяются следующим образом:

```
instance Show a =>
    Show (Square a) where
        show (Square d v) =
            if (null v)
                then "+" ++ (showLine d)
            else "+" ++ (showLine d) ++
                "\n|" ++ showRow (take d v) ++
```

```

    "\n" ++ show (Square d (drop d v))
where showRow []      = ""
      showRow (x:xs) = (showCell x) ++
                        (showRow xs)

showCell x = " " ++ (replicate
                      (nLength d2 -
                       nLength x)
                      ',') ++
            (show x) ++ " |"

showLine 0 = ""
showLine i = "-" ++ (replicate
                      (nLength d2)
                      ',') ++
            "-+" ++
            (showLine (i - 1))

nLength x = length $ show x

d2 = d^2

instance Show a =>
    Show (MagicList (Square a)) where
  show (ML [])      = ""
  show (ML (x:xs)) = show x ++
                      "\n\n" ++ show (ML xs)

```

Не стоит пугаться столь непростых на первый взгляд определений. Все эти функции направлены лишь на то, чтобы выводить числовые магические квадраты в читабельном виде. Например, один из магических квадратов размера  $5 \times 5$ , полученный в процессе работы над разделом, был выведен этими функциями в таком виде:

3	16	9	22	15	
20	8	21	14	2	
7	25	13	1	19	
24	12	5	18	6	
11	4	17	10	23	

Остаётся отметить, что вызывать новую функцию `getMagicFigures` так просто уже не получится, так как транслятор языка Haskell не сможет автоматически определить тип возвращаемого результата, чтобы выбрать требуемый экземпляр класса `Magic`. Поэтому тип необходимо в данном случае указывать явно:

```
main :: Int -> IO ()
```

```
main d = print (getMagicFigures d :: MagicList (Square Int))
```

Это — небольшая плата за большую универсальность в представлении.

## Заключение

Дальнейшая работа над усовершенствованием переборного алгоритма может принести ощутимые результаты. Ведь в этом небольшом разделе намечен только путь, по которому можно двигаться. Здесь показаны некоторые из принципов оптимизации программ на функциональных языках программирования. Но в случае необходимости далее надо двигаться самостоятельно.

Например, можно ещё больше поработать над представлением магических квадратов, чтобы сделать функции `test*` более простыми. Этого можно добиться при помощи явного хранения позиции числа в самом магическом квадрате. Это увеличит количество занимаемой памяти, но достаточно сильно ускорит процессы проверки на «магичность» горизонталей, вертикалей и диагоналей, так как отпадёт необходимость перебирать списки при помощи оператора `(!!)`.

Также необходимо поработать над самим алгоритмом перебора, так как он ещё весьма далёк от совершенства (например, оценка времени, необходимого для вычисления всех магических квадратов размера  $5 \times 5$ , показала, что вычисление их при помощи функции `getMagicSquares` займёт почти 17 тысячелетий, а при помощи функции `getMagicSquares'` — 24 дня). В этом деле поможет также внедрение в определения функций возможности осуществлять поиск магических квадратов, начиная с определённой комбинации. Это позволит распределить усилия по поиску магических фигур как во времени, так и в пространстве (задействовав, к примеру, несколько компьютеров, которые будут работать в ночное время).

Так что возможностей для самостоятельного творчества в этом направлении — тьма. Дерзайте!

# Задача о ранце

*Статья была опубликована в № 09 (57) журнала «Потенциал» в сентябре 2009 года.*

*В эссе предлагается рассмотрение одной из классических оптимизационных задач — задача о ранце. Даются математическая формулировка задачи и её решения, предлагается реализация этого решения на функциональном языке программирования Haskell, а также рассматривается пример использования разработанной обобщённой функции для поиска решения конкретной проблемы.*

## Введение

Научно-технический прогресс и развитие методов прикладной математики дали начало новой дисциплине, которая занимается разработкой и применением методов нахождения оптимальных решений на основе математического и статистического моделирования и различных эвристических техник в различных областях человеческой деятельности. Данная дисциплина получила название «исследование операций» (в российских высших учебных заведениях может преподаваться под названием «методы оптимизации» или схожими).

Особый толчок к развитию методов оптимизации дала Вторая мировая война. Во время неё исследование операций стало широко применяться для планирования боевых действий. Например, для Военно-воздушных сил союзников учёными были выработаны рекомендации, которые позволили увеличить эффективность бомбометания в четыре раза.

После войны группы специалистов по исследованию операций продолжили свою работу в вооружённых силах США и Великобритании. Публикация ряда результатов в открытой печати вызвала всплеск общественного интереса к этому научному направлению. Возникла тенденция к применению методов исследования операций в коммерческой деятельности, в целях реорганизации производства, перевода промышленности на мирные рельсы. В СССР исследование операций также получило широкое распространение (вплоть до того, что советский математик и экономист Л. В. Канторович получил в 1975 году Нобелевскую премию по экономике за вклад в теорию оптимального распределения ресурсов).

Характерными особенностями исследования операций являются системный подход к поставленной проблеме и анализ. Системный подход является главным методологическим принципом исследования операций. Любая задача, которая решается, должна рассматриваться с точки зрения влияния функционирования системы в целом. Для исследования операций характерно то, что при решении каждой проблемы могут возникать новые задачи. Важной особенностью исследования операций есть стремление найти оптимальное решение поставленной задачи (принцип «оптимальности»). Однако на практике такое решение часто бывает невозможно найти по следующим причинам:

- 1) отсутствие методов, дающих возможность найти глобальное оптимальное решение задачи;
- 2) ограниченность существующих ресурсов (к примеру, ограниченность машинного времени ЭВМ), что делает невозможным реализацию точных методов оптимизации.

В таких случаях обычно ограничиваются поиском не оптимальных, а достаточно хороших с точки зрения практики решений. Приходится искать компромисс между эффективностью решений и затратами на их поиск. Исследование операций даёт инструмент для поиска таких компромиссов.

Исследование операций тесно связано с теорией управления, системным анализом, математическим программированием, теорией игр, теорией оптимальных решений, эвристическими подходами и методами искусственного интеллекта. В качестве канонических задач, которые решаются методами оптимизации, можно отметить задачу коммивояжёра, транспортную задачу, задачу об упаковке в контейнеры, задачу о ранце и др. О последней как раз и пойдёт речь в настоящем разделе.

## Классическая задача

Задача о ранце (бывает, что говорят «задача о рюкзаке») является классической задачей исследования операций и комбинаторной оптимизации. Задача получила своё название от максимизационной проблемы укладки как можно большего числа *нужных* вещей в рюкзак (ранец) при условии, что общий объём (или вес) всех предметов *ограничен*. Подобные задачи часто возникают в прикладной математике, экономике, криптографии.

В общем виде задачу можно сформулировать так: из неограниченного (или ограниченного) множества предметов со свойствами «стоимость» и «вес» требуется отобрать некое число предметов таким образом, чтобы получить максимальную суммарную стоимость при одновременном соблюдении ограничения на суммарный вес.

Само собой разумеется, что к данной классической формулировке задачи могут сводиться многие иные задачи разных размерностей. В качестве стоимости и веса могут использоваться совершенно различные характеристики и даже их комбинации. В этом вопросе необходимо лишь наличие некоторого преобразования (функции), которая позволит свести требуемую задачу к классической.

В принципе, существуют две формулировки классической задачи о ранце:

- 1) каждый предмет из множества можно выбирать неограниченное количество раз (пока есть возможность удовлетворять ограничение на вес);
- 2) каждый предмет можно использовать только один раз.

Сама по себе задача о ранце является NP-полной задачей, то есть такой, время работы алгоритма для решения которой существенно зависит от размера входных данных, при этом если предоставить алгоритму некоторые дополнительные сведения, то он сможет за время, не превосходящее некоторого многочлена от размера входных данных, решить задачу. Дополнительные сведения в данном случае называются «свидетелем решения».

В свою очередь, это обозначает, что задачу можно решить при помощи методов динамического программирования. Данный раздел исследования операций позволяет решать задачи с оптимальной структурой и перекрывающимися подзадачами, при этом метод решения основан на постепенном решении задачи посредством вывода решения через уже решённые подзадачи. Слово «программирование» в наименовании дисциплины не имеет ничего общего с написанием кода, поскольку обозначает оптимальную последовательность действий для поиска решения задачи (можно сравнить с такими терминами, как «линейное программирование», «математическое программирование»).

Итак, формулировка задачи о ранце с возможностью неограниченного выбора звучит так. По заданному набору из  $n$  предметов со стоимостями  $v_1, v_2, \dots, v_n$  и весами  $w_1, w_2, \dots, w_n$  необходимо найти такой поднабор (с учётом того, что можно брать любой предмет неограниченное число раз), что его стоимость будет максимальна среди всех поднаборов с общим весом не более  $W$ .

Решение такой задачи несложно. Пусть  $K_w$  — максимальная стоимость, которую можно набрать при весе не более  $w$ . Следующее рекуррентное соотношение позволяет найти решение:

- 1)  $K_0 = 0$  (при весе не более 0 максимальная стоимость равна 0) — базис рекурсии;
- 2)  $K_w = \max\{K_{w-w_i} + v_i\}_{i=1}^n, w_i \leq w$ , где  $n$  — размер набора — шаг рекурсии.

С другой стороны, задача о ранце с возможностью использовать любой предмет из набора не более одного раза формулируется следующим образом. Пусть  $K_{w,i}$  — максимальная стоимость, которую можно набрать при весе не более  $w$  среди  $i$  первых предметов. Следующее рекуррентное соотношение находит решение:

- 1)  $K_{0,i} = 0, 0 \leq i \leq n$ ;
- 2)  $K_{w,0} = 0, 0 \leq w \leq W$ ;
- 3)  $K_{w,i} = \max\{K_{w,i-1}, K_{w-w_i,i-1} + v_i\}, 0 \leq w \leq W, w_i \leq w$ ;
- 4)  $K_{w,i} = K_{w,i-1}$ , если  $w_i > w$  (невозможно добавить элемент этого веса).

Собственно, на следующем рисунке идеографически проиллюстрирована такая задача.

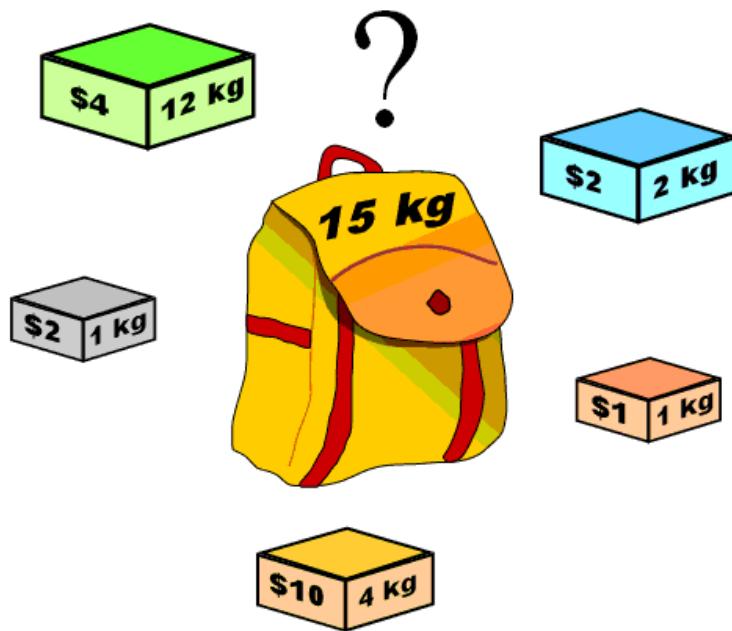


Рис. 5. Иллюстрация задачи о ранце — исходные данные

## Реализация решения на языке Haskell

Классическая математическая задача так и будет оставаться «интересной абстракцией», если не реализовать алгоритм для решения, что называется, в коде. Как обычно, для иллюстрации методов решения описанной задачи можно воспользоваться функциональным подходом и реализовать возможный алгоритм на языке программирования Haskell.

Реализацию алгоритма необходимо начать с небольшого проектирования тех программных сущностей, которые должны быть определены в программе. Дело в том, что математическое описание задачи говорит о произвольных объектах, которые могут фигурировать в процессе упаковки в ранец. Для таких объектов необходимо иметь две операции — получение стоимости и получение веса. На основании этих операций можно произвести вычисления и найти решение.

Итак, язык Haskell имеет механизм для абстракции подобных требований — это классы, описывающие необходимые методы (подробно о классах в языке Haskell можно ознакомиться в разделе «Объектно-ориентированное и функциональное программирование», стр. 30). Так что описать класс для объектов, которые могут быть упакованы в ранец, можно так:

```
class KnapsackItem a where
    cost   :: a -> Integer
    weight :: a -> Integer
```

Любой тип данных, имеющий экземпляра класса `KnapsackItem`, можно будет использовать в функциях, которые производят поиск решения в задаче о ранце. Конечно, и соответствующие функции необходимо реализовать с учётом этого требования. Но это будет сделать уже совсем несложно.

В определении указанного класса имеется одна тонкость. Функции `cost` («стоимость») и `weight` («вес») возвращают целое число неограниченного размера. В классической формулировке задачи ничего не говорится о типе свойств «стоимость» и «вес», а потому их типами могут быть целые числа, действительные числа, да и вообще произвольные типы, для которых определены некоторые операции: для стоимости — сложение, для веса — сравнение (это следует из формул решения задачи, представленных в предыдущем разделе).

К сожалению, не так просто реализовать данную возможность на языке Haskell, поскольку стандарт языка Haskell-98 не позволяет сделать это удобным способом. Тем не менее некоторые расширения языка дают возможность написать не менее эффективное определение. Например:

```
class KnapsackItem a where
    cost   :: Num b => a -> b
    weight :: Ord b => a -> b
```

Или:

```
class Num b => KnapsackItem a b where
    cost   :: a -> b
    weight :: a -> b
```

В первом из этих случаев в сигнатуре методов вводятся ограничения на типы возвращаемых значений (при этом методы могут вообще возвращать значения различных типов — одинаковое обозначение `b` не должно смущать). Метод `cost` должен возвращать значение, над которым можно производить арифметические операции (класс `Num`). Соответственно, метод `weight` возвращает значение, которое можно сравнивать (класс `Ord`). Но стандарт Haskell-98 не разрешает вводить ограничения на типы в сигнатуре методов, ограничение может быть только на параметрическую переменную класса.

Во втором случае класс `KnapsackItem` параметризует два типа, причём на второй наложено ограничение — он должен иметь экземпляр класса `Num` (что автоматически влечёт наличие экземпляра класса `Ord`). Но опять же стандарт Haskell-98 не разрешает использовать многопараметрические классы. Это возможно только в расширениях компилятора GHC.

Использование любого из указанных подходов влечёт необходимость не только подключения того или иного расширения языка Haskell, но и дополнительного удовлетворения требований при написании функций. Таким образом, изначальное определение класса подчиняется стандарту, но вместе тем позволяет написать простые и понятные функции. Также это определение класса `KnapsackItem` вполне достаточно и не сказывается на общности рассуждений при решении задач.

Теперь можно реализовать непосредственную функцию, которая будет производить поиск решения задачи о ранце. Ради удобства такая функция будет выбирать по одному предмету из заданного списка элементов (неограниченный выбор элементов можно сделать из той же функции, убрав вызов функции удаления элемента из списка `delete`). Собственно, такая функция выглядит следующим образом:

```
knapsack 0 _ = 0
```

```

knapsack _ [] = 0
knapsack w vs | null l      = 0
                | otherwise = maximum l
where
  l = [knapsack (w - weight v) (delete v vs) + cost v |
        v <- vs,
        weight v <= w]

```

Как видно, два первых клоза определяют базис рекурсии, при этом отслеживать пустой список элементов также надо (математическое описание задачи это подразумевает). Третий клоз наиболее интересен, поскольку определяет шаг рекурсии, и в нём производится динамический перебор и поиск решения. Два выражения охраны определяют результат функции в случае, если список возможных значений, полученных при рекурсивном вызове, пуст и непуст. В первом случае результатом всё так же должен быть 0, во втором необходимо выбрать максимальное значение, как то предписывает математическое рекуррентное соотношение.

Выражение 1, которое вынесено в локальное определение в оптимизационных целях (в теле функции оно встречается два раза), как раз и запускает процесс рекурсивного вызова функции `knapsack` для всех элементов входного списка, чьи веса не больше заданного веса. Вычисление основано на генераторе списка (подробно о генерации списков можно ознакомиться в разделе «Магические квадраты и решение переборных задач», стр. 90).

Ту же самую функцию можно написать и с префиксным определением локального выражения 1. В методических целях будет интересно рассмотреть такой вариант:

```

knapsack 0 _ = 0
knapsack _ [] = 0
knapsack w vs = let l = [knapsack (w - weight v)
                           (delete v vs) + cost v |
                           v <- vs,
                           weight v <= w]
                  in if (null l)
                     then 0
                     else maximum l

```

Понятно, что этот вариант ничем не отличается от предыдущего, кроме своей формы.

Всё хорошо, только определённая любым из представленных вариантов функция `knapsack` не имеет высокого практического значения. Это связано с тем, что она возвращает лишь максимально возможную стоимость упакованных в ограниченный ранец элементов. Но в прикладных задачах обычно необходимо знать не только такую максимальную стоимость, но и тот набор элементов, который приводит к такой стоимости. Для решения этой задачи функцию придётся слегка преобразовать — она будет возвращать пару, первым значением которой является максимальная стоимость, а вторым — набор элементов.

```

knapsack' 0 vs = (0, [])
knapsack' _ [] = (0, [])
knapsack' w vs = getMaxCost [((fst prevKS) + cost v, v:(snd prevKS)) |
                               v <- vs,
                               let prevKS = knapsack' (w - weight v)
                                   (delete v vs),
                               weight v <= w]
where
  getMaxCost []          = (0, [])
  getMaxCost ((c, v):vs) = let (c', v') = getMaxCost vs

```

```
in  if (c > c')
      then (c, v)
      else (c', v')
```

Как видно, из-за того, что результатом выполнения функции стала пара, определение сделалось сложнее. Всё по причине того, что пару необходимо собирать непосредственно в генераторе списка (а для этого её необходимо разбирать — есть стандартные функции `fst` и `snd`), а над каждым элементом пары надо производить свои операции. Над первым — сложение весов, над вторым — добавление элемента в список. Это также влечёт необходимость реализации собственной функции поиска максимума, поскольку стандартная `maximum` работает только со списками сравниваемых элементов. Локальная функция `getMaxCost` ищет максимальную стоимость в списке пар, в которых стоимость стоит на первом месте.

Ну и, собственно, при помощи этих функций теперь можно решать задачи о ранце. Например, та задача, которая показана на рис. 5, кодируется следующими программными сущностями:

```
data KI = KI Integer Integer
deriving (Eq, Show)

instance KnapsackItem KI where
  cost (KI c w) = c
  weight (KI c w) = w

example = [KI 4 12, KI 2 2, KI 2 1, KI 1 1, KI 10 4]
```

Здесь тип `KI` содержит в себе два поля: одно — для стоимости, второе — для веса. Для соответствия требованиям на типы значений, использующихся в функциях `knapsack` и `knapsack'`, необходимо определить экземпляр класса `KnapsackItem`. Функция `example` просто определяет список элементов, которые изображены на рис. 5.

Теперь можно запустить функции на проверку:

```
> knapsack 15 example
15

>knapsack' 15 example
(15, [KI 10 4, KI 1 1, KI 2 1, KI 2 2])
```

## Заключение

Как видно, язык функционального программирования Haskell в очередной раз показал свою близость к математике и простоту в использовании для решения задач. Полученные функции могут быть использованы для различных целей — достаточно лишь закодировать исходные данные в соответствии с требованиями. Так, к примеру, автор использовал функцию `knapsack'` для оптимального распределения музыкальных альбомов в формате MP3 на нескольких компакт-дисках (задача о ранце применялась рекурсивно) для неутомительного прослушивания, при этом критерием стоимости выступала достаточно сложная формула, принимающая во внимание размер альбома, его жанр, субъективную оценку слушателя и некоторые иные факторы.

Таким образом, язык Haskell вполне успешно можно использовать в качестве инструмента быстрой разработки небольших утилит, предназначенных для решения какой-то одной несложной задачи.

# Кривая Дракона

*Статья была опубликована в № 09 (45) журнала «Потенциал» в сентябре 2008 года.*

*В данном эссе в очередной раз рассматриваются практические аспекты применения языка Haskell при решении расчётных задач. В качестве примера изучается один из видов фракталов, имеющий поэтическое наименование — «Кривая Дракона». Приводятся теоретические аспекты, рассматривается построение алгоритма на математическом уровне, после чего алгоритм реализуется на языке Haskell.*

## Введение

В математике имеется одна интереснейшая отрасль, которая занимается изучением самоподобных объектов, каждая мельчайшая часть которых подобна всему объекту. Под подобием здесь понимается похожесть до некоторой степени. Сама степень подобия в данном случае может являться фактором, при помощи которого можно классифицировать изучаемые объекты. Ну а сами такие объекты называются *фракталами*.

Одним из наиболее известных фракталов является *множество Мандельброта*, которое строится при помощи достаточно простой процедуры (подробности см., например, в [13]). Для произвольного комплексного числа  $z_0$  строится отображение по формуле:

$$z_{i+1} = z_i^2 + c, \quad (21)$$

где  $c$  — некоторая константа. В этой процедуре индекс  $i$  последовательно пробегает все натуральные значения от 0 и далее. Так итеративно строится последовательность  $z_0, z_1, z_2, \dots$ . Если зафиксировать  $z_0$ , то получаемые на очередном шаге значения будут полностью зависеть от выбора константы  $c$ , которая на любом шаге должна быть одинаковой.

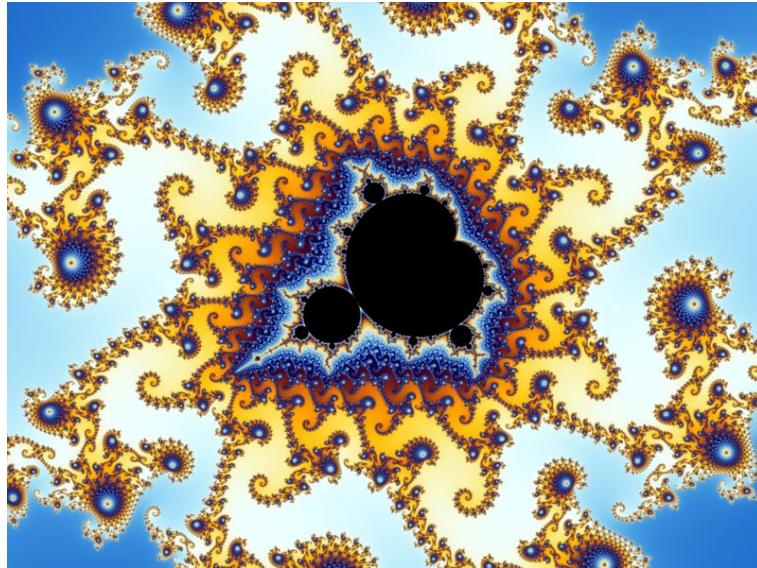
Например, если  $z_0 = 0$ , то последовательность будет выглядеть так:

- $z_0 = 0;$
- $z_1 = c;$
- $z_2 = c^2 + c;$
- $z_3 = c^4 + 2c^3 + c^2 + c;$
- $z_4 = c^8 + 4c^7 + 6c^6 + 6c^5 + 5c^4 + 2c^3 + c^2 + c;$
- ...

И вот оказывается, что для некоторых значений  $c$  при фиксированном  $z_0$  описанная процедура создаёт бесконечную последовательность, в которой любое число ограничено некоторой небольшой областью около начала координат — комплексного числа  $0 + 0i$ . Например, тривиальным значением константы  $c$

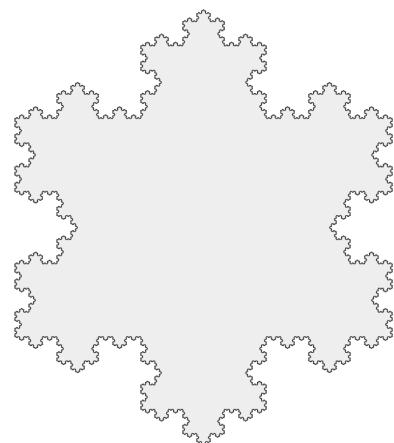
является 0, поскольку в данном случае любой член последовательности также равен 0. Другой пример:  $c = -1$ .

Множество Мандельброта есть множество всех значений константы  $c$ , для которых получаемая по описанной процедуре последовательность является ограниченной. На рис. 6 показано множество Мандельброта для  $z_0 = 0$ . Как видно, множество Мандельброта впечатляет своей сложностью, особенно учитывая, как это часто бывает в математике, удивительную простоту его определения.



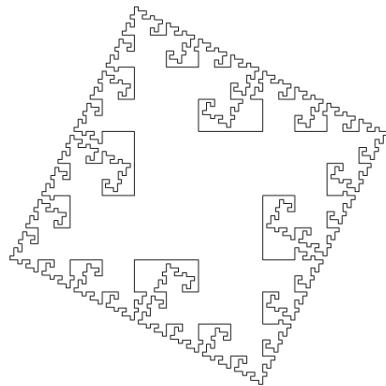
**Рис. 6. Общий вид множества Мандельброта для  $z_0 = 0$**

Другой пример фрактала приводит в своей книге [5] замечательный американский популяризатор науки Мартин Гарднер. Он строит снежинку, начиная от простого равностороннего треугольника, заканчивая фигурой с бесконечным периметром, но ограниченной площадью. На каждой итерации построения периметр фигуры становится всё сложнее и замысловатее, повышение точности его измерения влечёт неограниченный рост его длины, а площадь остаётся ограниченной очень небольшой областью. Как пишет сам М. Гарднер, такая фигура может быть изображена на обычной почтовой марке, а сумма длин её сторон будет сравнима с диаметром нашей галактики (естественно, при должной точности нанесения изображения и последующего измерения). Примерный вид фрактала «Снежинка» изображён на рис. 7.



**Рис. 7. Примерный вид фрактала «Снежинка» для небольшого числа итераций**

Ещё один фрактал в своё время разработан автором по мотивам прочтённого описания фрактала «Снежинка». Для его построения используются уже не равносторонние треугольники, а квадраты. Его периметр также стремится к бесконечности, хотя площадь всегда ограничена. Примерный вид этого фрактала показан на рис. 8.



**Рис. 8. Примерный вид квадратного фрактала для небольшого числа итераций**

Все эти примеры сложных фигур объединяет одно свойство — они самоподобны. Если рассмотреть какую-либо часть фигуры произвольного размера на границе, то при должном приближении и достаточной точности будет видно, что малая часть очень похожа на всю фигуру — подобна ей.

Фракталы имеют множество областей применения как в чистой науке, так и на практике. Например, некоторые современные техники шифрования информации основаны на результатах, полученных при изучении фракталов. Но, кроме научного и практического применения, у этих фигур имеется ещё одно — эстетическое. Они необычайно красивы.

Далее в настоящем разделе рассматривается ещё один вид фрактала, имеющий название «Кривая Дракона».

## Что такое Кривая Дракона?

Как и рассмотренные во введении фракталы, Кривая Дракона строится весьма простым способом. Любой может построить начальные итерации при помощи бумажной ленты. Любознательному читателю можно посоветовать следующий способ построения Кривой Дракона.

Необходимо взять бумажную ленту, скажем, размера  $100 \times 1$  см. Чем тоньше будет бумага, тем лучше, поскольку такую ленту придётся многократно сгибать, а при сгибании толщина бумаги будет многократно складываться, внося погрешности в создаваемую фигуру. Лучше всего подойдёт тонкая калька — достаточно тонкая, но в то же время прочная бумага.

Итак, имеется лента длиной в 100 сантиметров. Один конец этой ленты необходимо закрепить (или держать в уме, что этот конец закреплённый; в действительности закреплять kleem или чем-то подобным не следует, поскольку впоследствии ленту придётся разворачивать). Второй конец ленты является свободным. Каждая итерация по построению Кривой Дракона заключается в простой процедуре. Лента складывается пополам — свободный конец совмещается с закреплённым. На первом шаге получается сложенная вдвое лента размера  $50 \times 1$  см. Первоначальный свободный конец оказался совмещённым с закреплённым концом. Этот совмещённый конец ленты остаётся закреплённым, а свободным теперь считается место сгиба ленты.

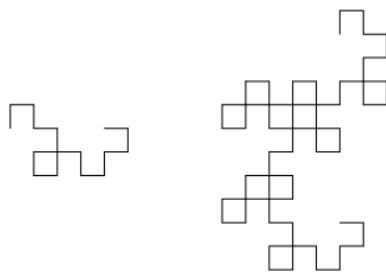
Вторая итерация повторяет первую — свободный конец совмещается с закреплённым. Получается сложенная вчетверо лента размера  $25 \times 1$  см. Места сложения необходимо тщательно проглаживать, чтобы они были как можно более плоскими (именно тут нарастает неточность при дальнейших сложениях,

поскольку на сгиб необходим материал, и на самом деле сложенная вчетверо лента имеет меньший размер, но пока эта неточность незаметна).

Описанную итерацию необходимо повторять столько, сколько это возможно. В идеале для достаточно тонкой бумаги можно получить сложенную в шестьдесят четыре раза ленту размера  $1.5625 \times 1$  см. Но до этого вряд ли дойдёт — помешает именно толщина бумаги. Пусть получится сложенная в шестнадцать раз лента размера  $6.25 \times 1$  см. После этих сложений лента будет уже напоминать бруск. Но необходимо помнить, что это всего лишь грубая физическая модель математического процесса. Математическая лента не имеет толщины, её можно складывать до бесконечности.

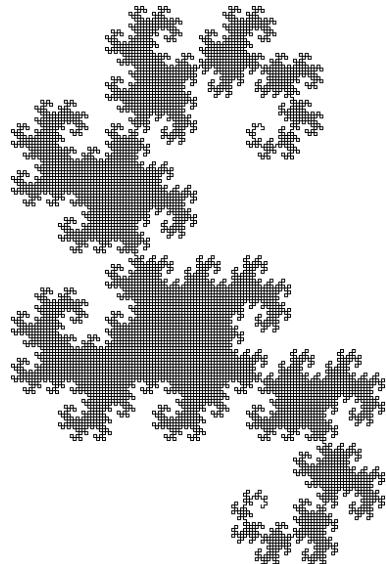
Наконец, можно приступить к самому построению Кривой Дракона. Для этого необходимо развернуть получившийся бруск бумаги, озабочившись тем, чтобы каждый угол на сгибе равнялся  $90^\circ$ . Разворачивать, естественно, необходимо, поставив согнутую ленту «на попа», а не просто положив её. Это сделать не так просто, но при должном терпении лента развернётся в Кривую Дракона для определённого шага построения (в идеале сгибание ленты нулевой толщины производится бесконечное количество раз).

На рис. 9 показана Кривая Дракона при сгибании ленты в шестнадцать (часть а) и в шестьдесят четыре раза (часть б) соответственно.



**Рис. 9. Кривая Дракона при сгибании ленты в шестнадцать и шестьдесят четыре раза**

Интерес представляет то факт, что, сколько бы ни сгибали ленту, при разворачивании её указанным образом она никогда не пересечётся сама с собой. Лента поразительным образом выстраивается в замысловатую кривую, несколько напоминающую рассмотренное ранее множество Мандельброта. Если рассмотреть математическое описание этой кривой, то её внешняя граница опять получится самоподобной — каждая мельчайшая часть напоминает всю кривую. Общий вид математической Кривой Дракона представлен на рис. 10.



**Рис. 10. Математически точная Кривая Дракона**

Другим интересным фактом относительно этой кривой является то, что саму кривую можно построить из отрезка любой длины. Длина отрезка полностью определяет площадь кривой. Собственно, длина отрезка — это и есть длина кривой, уложенной замысловатым образом. Чем длиннее кривая, тем большую площадь она захватит. Однако идеальный математический объект этой замысловатой формы может быть построен из отрезка любой длины. Обычно используется единичный отрезок.

## Алгоритм построения

Рассмотренный в предыдущем подразделе способ построения Кривой Дракона позволяет сформулировать алгоритм её построения для некоторого языка программирования. Собственно, при разворачивании ленты этот алгоритм должен был появиться в голове вдумчивого экспериментатора, поскольку каждый очередной разворот показывает, как именно строится кривая на произвольном шаге.

Алгоритм достаточно прост. Однако в целях упрощения далее будет предполагаться, что на каждом очередном шаге длина кривой увеличивается в два раза, а не остаётся постоянной, как в случае сгибаия ленты. Это непринципиально, поскольку на общность рассуждений никак не влияет. Построенная для какого-то шага кривая будет иметь определённую длину, и эту длину можно считать длиной первоначальной ленты.

Итак, алгоритм построения Кривой Дракона идёт не от сгибаия ленты, а от разворачивания единичного отрезка в кривую. Он состоит всего из двух шагов:

- 1) на нулевом шаге строится единичный отрезок с концами в координатах  $(0, 0)$  и  $(1, 0)$  соответственно. Этот отрезок называется Кривой Дракона нулевой итерации;
- 2) полученная на предыдущем шаге Кривая Дракона  $n$ -ой итерации дублируется, и дубликат поворачивается вокруг начала координат  $(0, 0)$  против часовой стрелки на  $90^\circ$ , после чего сдвигается к концу оригинала кривой своим свободным концом. Свободный конец — это тот конец, который лежит вне начала координат. Он свободен потому, что свободно вращается. Первая точка Кривой Дракона лежит в начале координат, а потому при вращении не изменяет своего положения, поэтому она и называется закреплённой. Оригинал и дубликат шиваются в одну кривую. Получается Кривая Дракона  $(n + 1)$ -ой итерации. Данный процесс повторяется до тех пор, пока не достигнута требуемая точность (в идеале — до бесконечности).

Применение этого алгоритма для четырёх итераций показано на рис. 11.

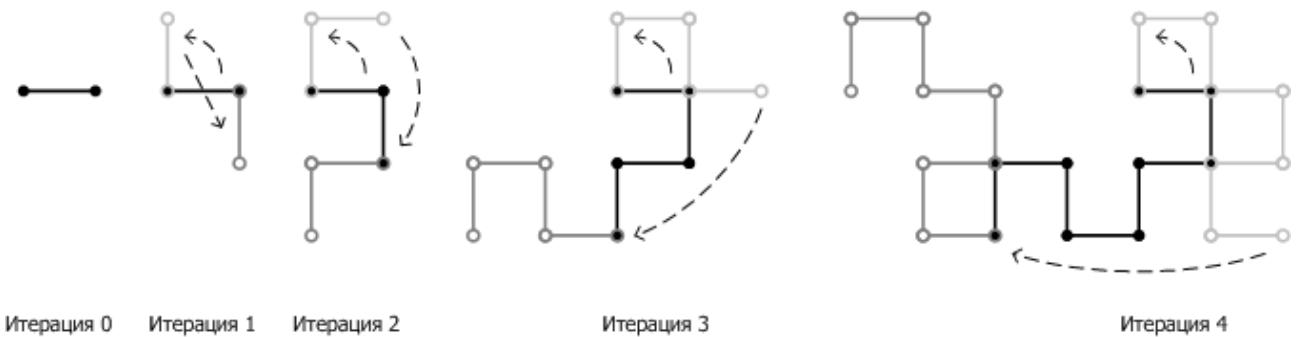


Рис. 11. Последовательное построение Кривой Дракона при помощи описанного алгоритма

## Реализация на языке Haskell

Описанный в предыдущем подразделе алгоритм на первый взгляд выглядит достаточно императивно. Шаг за шагом строится кривая, последовательно наращивая свою сложность. Однако на самом деле этот

алгоритм достаточно легко преобразуется в рекурсивный декларативный алгоритм, причём практически без каких-либо дополнений и изменений. Ниже в последующих подразделах будет написана небольшая программа на функциональном языке Haskell, которая вычисляет координаты последовательных точек Кривой Дракона.

## Подготовительные описания геометрических образов

Прежде чем начать реализацию функции, возвращающей последовательность координат точек Кривой Дракона, необходимо провести некоторые подготовительные работы. Для построения кривой нужно определить специальные типы данных, которые будут использоваться для описания примитивных элементов, из которых будет состоять кривая, а также для описания самой кривой. Резонно все такие описания вынести в отдельный модуль, ответственный за геометрические описания:

```
module Geometry where
```

Первым делом необходимо подумать, что имеется общего во всех тех объектах, которые будут использоваться в алгоритме. Сам алгоритм подсказывает — все объекты можно поворачивать на  $90^\circ$  против часовой стрелки относительно начала координат и сдвигать на заданное расстояние. Сам алгоритм ничего не говорит о типе координат — целочисленные ли они или произвольные. Для описания таких ситуаций в языке Haskell имеется отличный формализм — классы с параметризацией типов. Поэтому для описания интерфейса к объектам, которые можно вращать и двигать, можно определить следующий класс:

```
class Figure f where
    shift      :: Num a => a -> a -> f a -> f a
    rotate90 :: Num a => f a -> f a
```

Здесь, в этом определении, сделано небольшое допущение. Предполагается, что произвольная фигура типа *f* параметризует собой некоторый специальный тип *a*, со значениями которого можно совершать арифметические операции (тем читателям, которым сложно понимать вышеупомянутые определения, необходимо обратиться к предыдущим разделам данного сборника либо к источникам [6, 7]). Параметризация типа *a* сделана именно для того, чтобы не ограничивать координаты объектов каким-либо специализированным типом, поскольку в изначальном алгоритме о типе ничего не сказано. То, что над координатами необходимо совершать арифметические операции, следует из того, что их необходимо сдвигать, то есть к координатам необходимо прибавлять значения сдвигов по двум осям.

Следующий шаг — определение типа данных для представления точки на плоскости. Для этих целей можно было бы воспользоваться обычной парой, однако этот путь не совсем правильный, поскольку не использует возможности языка Haskell по абстракции данных. И, несмотря на то что в описываемом достаточно простом примере возможности по абстракции будут использованы слабо, хороший тон в программировании на языке Haskell требует применения грамотных идиом всегда, когда это возможно. Итак, алгебраический тип данных для представления одной точки можно определить так:

```
data Point a
= Point
{
    x :: a,
    y :: a
}
```

Это определение означает, что тип для представления точки *Point* параметризует некоторый тип *a*, который используется для представления типов двух координат — *x* и *y*.

Для того чтобы значениями только что созданного типа `Point` можно было оперировать в соответствии с описанным ранее алгоритмом, для этого типа необходимо определить экземпляр класса `Figure`. Это делается достаточно просто:

```
instance Figure Point where
    shift dx dy (Point x y) = Point (x + dx) (y + dy)
    rotate90 (Point x y)     = Point (-y) x
```

Теперь в этом определении видно, что метод `shift` используется для сдвига объекта на заданные значения по осям абсцисс и ординат, а метод `rotate90` всего лишь поворачивает созданный объект на  $90^\circ$  против часовой стрелки. Для реализации описанного алгоритма нет необходимости создавать метод для вращения на произвольное количество градусов, поскольку это не потребуется. Если бы это было необходимо, то было бы невозможно использовать для представления координат произвольные типы, над значениями которых можно производить только арифметические операции, поскольку для вращения на произвольное число градусов необходимо использовать функции `sin` и `cos`, которые в языке Haskell определены над действительными значениями.

Дополнительно желательно определить для типа `Point` экземпляр класса `Show`, чтобы было возможно выводить результаты работы с точками на координатной плоскости на экран. Реализация экземпляра по умолчанию преобразует значения не очень приятным способом, поэтому в нижеследующем определении используется обычная математическая нотация для записи точек на плоскости.

```
instance Show a => Show (Point a) where
    show (Point x y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

Также необходимо определить утилитарную функцию, которая возвращала бы расстояние по двум осям между двумя точками. Эта функция потребуется при определении того, на какое расстояние сдвигать один из экземпляров кривой на очередном шаге построения. Функция `distance` выглядит следующим образом:

```
distance :: Num a => Point a -> Point a -> (a, a)
distance (Point x1 y1) (Point x2 y2) = (x1 - x2, y1 - y2)
```

Эта функция возвращает пару расстояний. Первое значение в паре символизирует расстояние между точками по оси абсцисс, второе — по оси ординат соответственно. Такое представление наиболее просто, а так как будет использоваться только в одном месте, создавать новый тип для представления такой пары не требуется.

Сама кривая состоит из последовательного набора точек. Можно было бы использовать обычный список значений типа `Point`, однако не для этого вначале был определён класс `Figure`. Простой список использовать нельзя, поскольку по правилам языка Haskell его нельзя будет сделать экземпляром класса. Необходимо определить новый тип, который имел абсолютно такое же поведение, как и имеющийся тип `[]`. Для этих целей используется ключевое слово `newtype`:

```
newtype Curve a = Curve [Point a]
deriving Show
```

Данное определение говорит транслятору языка Haskell, что необходимо создать тип, изоморфный списку значений типа `Point a`. Изоморфность обозначает в данном случае полную тождественность. Разница заключается лишь в том, что для нового типа `Curve` можно определить экземпляр класса `Figure`, что позволит оперировать значениями этого типа абсолютно так же, как простыми точками, — вращать и сдвигать их. Делается это намного проще, чем для точек типа `Point`:

```
instance Figure Curve where
    shift dx dy (Curve ps) = Curve (map (shift dx dy) ps)
    rotate90 (Curve ps)    = Curve (map rotate90 ps)
```

Наконец, было бы очень неплохо определить ещё одну утилитарную функцию, которая возвращала бы список точек из значения типа `Curve`. Эта функция просто «разворачивает» тип `Curve`, возвращая его содержимое:

```
points :: Curve a -> [Point a]
points (Curve ps) = ps
```

Таким образом, весь модуль `Geometry` выглядит следующим образом:

```
module Geometry where

class Figure f where
    shift      :: Num a => a -> a -> f a -> f a
    rotate90  :: Num a => f a -> f a

data Point a
= Point
{
    x :: a,
    y :: a
}

instance Figure Point where
    shift dx dy (Point x y) = Point (x + dx) (y + dy)
    rotate90 (Point x y)     = Point (-y) x

instance Show a => Show (Point a) where
    show (Point x y) = "(" ++ show x ++ "," ++ show y ++ ")"

distance :: Num a => Point a -> Point a -> (a, a)
distance (Point x1 y1) (Point x2 y2) = (x1 - x2, y1 - y2)

newtype Curve a = Curve [Point a]
    deriving Show

instance Figure Curve where
    shift dx dy (Curve ps) = Curve (map (shift dx dy) ps)
    rotate90 (Curve ps)   = Curve (map rotate90 ps)

points :: Curve a -> [Point a]
points (Curve ps) = ps
```

## Построение Кривой Дракона

Теперь всё готово, чтобы реализовать описанный в предыдущем подразделе алгоритм построения Кривой Дракона. Для её построения будет использоваться одна функция, полностью реализующая алгоритм так, как он написан:

```
makeDragonCurve :: Int -> Curve Int
makeDragonCurve 0 = Curve [Point 0 0, Point 1 0]
makeDragonCurve n = Curve (points c ++ (reverse $
```

```

    init $  

    points (shift dx dy rc)))  

where  

  c          = makeDragonCurve (n - 1)  

  rc         = rotate90 c  

  (dx, dy) = distance (last $ points c) (last $ points rc)

```

Первая строка определения описывает тип значения, возвращаемого функцией `makeDragonCurve`. В этом описании имеется ограничение на тип координат, используемых для представления точек на координатной плоскости. В предложенной реализации функции этот тип — `Int`, то есть ограниченные целые числа. Однако, как говорилось ранее, в качестве типа координат может использоваться произвольный тип, для которого имеется экземпляр класса `Num`.

Вторая строка один в один соответствует пункту 1 алгоритма построения Кривой Дракона. Для нулевой итерации в построении просто возвращается единичный отрезок  $(0, 0) - (1, 0)$ .

Соответственно, третья строка определения функции `makeDragonCurve` соответствует пункту 2 алгоритма. Как видно, эта строка более сложная, равно как и второй пункт алгоритма занимает намного больше строк в описании, чем первый пункт. Эта строка гласит, что Кривая Дракона на  $n$ -ом шаге построения состоит из точек (функция `points`) некоторой кривой `c`, к которым присоединены обращённое задом наперёд (функция `reverse` из стандартного модуля `Prelude`) начало (функция `init` из стандартного модуля `Prelude`) списка точек смещённой (функция `shift`) на значения `dx` и `dy` по осям абсцисс и ординат соответственно некоторой кривой `rc`.

Это определение необходимо рассмотреть подробнее. Некоторая кривая `c` является просто-напросто Кривой Дракона, построенной на предыдущем,  $(n - 1)$ -ом шаге. Кривая `rc` является дубликатом кривой `c`, который повёрнут на  $90^\circ$  против часовой стрелки вокруг начала координат. Значения для сдвига `dx` и `dy` являются расстоянием между последними точками кривых `c` и `rc`. Последние точки в описании кривой при помощи типа `Curve` являются как раз «свободными» концами, поскольку именно они врачаются вокруг начала координат. Пока всё идёт в точности по алгоритму. Все эти дополнительные объекты получены в локальных определениях, перечисленных после ключевого слова `where`.

Стандартная функция `init` используется для того, чтобы отсечь последнюю точку в дубликате кривой, полученной на предыдущем шаге. Эта точка при помощи сдвига подсоединяется к последней точке оригинала кривой. Поскольку точки совмещаются, в описании кривой на  $n$ -ом шаге нет надобности дублировать запись о точках. Именно для этого производится отсечение.

Наконец, обращение повёрнутой и сдвинутой кривой при помощи стандартной функции `reverse` производится для того, чтобы перевернуть последовательность точек. Ведь именно переворот происходит в процедуре построения, описанной в подразделе, посвящённом математическому описанию кривой, когда свободный конец ленты совмещается с закреплённым. А из этой процедуры непосредственно получен алгоритм построения.

Для тех читателей, кто подзабыл, остаётся отметить, что стандартный оператор `(\$)` определён как обычное применение функций, но с самым низким приоритетом исполнения:

```

infixr 0 $  

(\$) :: (a -> b) -> a -> b  

f \$ x = f x

```

Такое определение позволяет записывать последовательное применение функций к результатам выполнения других функций без лишних скобок.

В конечном итоге получается достаточно миловидный модуль:

```
module Dragon where
```

```
import Geometry
```

```

makeDragonCurve :: Int -> Curve Int
makeDragonCurve 0 = Curve [Point 0 0, Point 1 0]
makeDragonCurve n = Curve (points c ++ (reverse $ init $
                                         points (shift dx dy rc)))
where
  c          = makeDragonCurve (n - 1)
  rc         = rotate90 c
  (dx, dy)   = distance (last $ points c) (last $ points rc)

```

Запуск функции `makeDragonCurve` с некоторым числом, определяющим номер итерации в построении Кривой Дракона, вернёт список точек этой кривой на координатной плоскости для заданного шага. Эта функция является ленивой, поэтому в интерпретаторе языка Haskell она начнёт выводить промежуточные результаты сразу же после запуска.

## Заключение

На этом примере в очередной раз показана мощь языка Haskell в частности и функционального программирования в общем в решении расчётных математических задач. К сожалению, рамки простой научно-популярной книги не позволяют провести дальнейшие исследования и показать возможности по графической визуализации полученных результатов. Для этого было бы необходимо предварительно описать графические библиотеки, что полностью заняло бы несколько разделов подряд. Однако вдумчивому читателю можно порекомендовать создать для типа `Curve` экземпляр класса `Show`, который выводил бы заданную Кривую Дракона в виде псевдографики. Это не такая сложная задача.

# Немного о шахматных задачах

*Статья была подготовлена к публикации в журнале «Потенциал» осенью 2009 года. Опубликована не была.*

Эссе в занимательной форме рассказывает о применении парадигмы функционального программирования и языка Haskell для решения так называемых «шахматных задач» (на примере задачи о размещении фигур и задачи о ходе коня).

## Введение

Шахматы. История этой замечательной игры насчитывает не одно столетие. Умение играть в шахматы всегда ценилось и иной раз почиталось за достоинство дееспособного члена общества. Шахматы использовались в качестве средства испытаний и инициаций, для простого развлечения и для планирования стратегических действий на поле боя. Русские князья и цари, европейские короли, азиатские ханы и падишихи учились играть в шахматы с малого детства.

Сама игра является неистощимым источником для досуга. И это относится не только к игре с противником, но и к решению всевозможнейших шахматных задач. И шахматные задачи бывают двоякие. С одной стороны, решение относится к поиску какой-либо целевой позиции из заданной за определённое количество шагов (например, поставить мат в два хода, найти вилку или связку, взять ферзя противника и т. д.). С другой стороны, можно решать шахматные головоломки, которые основаны на разнообразии шахмат как системы фигур на доске определённого размера. Классическими головоломками в этом отношении являются задача о расстановке фигур и задача об обходе доски фигурай (в частности, нетривиальной из всех остальных является задача об обходе всех клеток доски конём так, чтобы в каждой клетке конь побывал лишь единожды, — задача о ходе коня).

Данный раздел рассказывает о том, как решить обе упомянутые задачи при помощи языка функционального программирования Haskell.

## Вспомогательные программные сущности

Для решения задач необходимо подготовить определения программных сущностей, которые необходимы для описания тех или иных шахматных понятий. Так, например, одна клетка шахматной доски может быть описана парой — координатой по вертикали (обычно обозначаемой строчной латинской буквой) и координатой по горизонтали (обозначается числом). Это понятие может быть выражено на языке Haskell следующим образом:

```
data Cell = Cell Int Int  
deriving Eq
```

Новый алгебраический тип данных, а не, скажем, пара, был выбран потому, что для этого типа будет необходимо определить экземпляры нескольких классов, в частности классов `Show` (класс типов,

значения которых могут быть преобразованы в строку) и `Enum` (класс типов, значения которых могут быть пересчитаны по порядку).

Экземпляр для класса `Show` определяется следующим образом:

```
instance Show Cell where
    show (Cell f r) = (chr (f + 96)):(show r)
```

Такое определение позволяет выводить клетку в строку в стандартном виде. Например, клетка с координатами (1,1) преобразуется в строку «a1», клетка с координатами (8,8) преобразуется в строку «h8» и т. д. Здесь функция `chr` из модуля `Data.Char` позволяет получить символ типа `Char` по его коду. Символ «a» имеет десятичный код ASCII 97, поэтому для получения этого символа и необходимо прибавить к заданному числу дельту 96.

Дополнительно желательно реализовать две утилитарные функции для получения отдельных компонентов типа `Cell`. Функция `file` «достаёт» вертикаль, а функция `rank` — горизонталь соответственно. Это можно было бы абсолютно так же сделать при помощи определения структуры с именованными полями.

```
-- Вертикаль
file :: Cell -> Int
file (Cell v h) = v

-- Горизонталь
rank :: Cell -> Int
rank (Cell v h) = h
```

Следующим типом, который необходимо определить, является перечисление для обозначения типов шахматных фигур. Это достаточно просто:

```
data Figure = Pawn    -- Пешка
            | Knight   -- Конь
            | Bishop   -- Слон
            | Rock     -- Ладья
            | Queen   -- Ферзь
            | King     -- Король
```

Для этого типа опять же необходимо определить экземпляр класса `Show` для представления фигур в соответствии с правилами шахматной алгебраической нотации. Это также несложно:

```
instance Show Figure where
    show Pawn    = "p"
    show Knight  = "N"
    show Bishop  = "B"
    show Rock    = "R"
    show Queen   = "Q"
    show King    = "K"
```

Две следующие константные функции необходимы для облегчения работы. Первая, `boardSize` возвращает размер шахматной доски (предполагается, что сама доска квадратная), что позволяет сделать генерализованные функции решения задач для любого размера доски. Пока это будет константная функция, негативным моментом чего является необходимость перекомпиляции исходного кода в случае изменения условий. В «хорошем» решении размер шахматной доски можно было бы считывать из файла или из стандартного потока ввода, когда этот параметр вводится пользователем (соответственно, он должен передаваться из функции в функцию для следования правильному стилю функционального программирования).

```
boardSize :: Int
boardSize = 8
```

Вторая функция — `initialBoard` — возвращает полный список ячеек пустой доски заданного размера. Эта константа будет необходима для начального запуска функций решения задач.

```
initialBoard :: [Cell]
initialBoard = [Cell f r | f <- [1..boardSize],
                        r <- [1..boardSize]]
```

Наконец, необходима функция, которая для заданной фигуры, клетки, на которой та располагается, и текущего состояния доски возвращает список клеток, на которых фигура может походить. Эту функцию написать уже не так просто, поскольку она должна описывать правила хождения фигур. Для каждой фигуры свои правила, поэтому целесообразно рассмотреть определение этой функции клоз за клозом.

Первый клоз описывает движение пешки. Пешка движется на один шаг вперёд, и лишь на первом ходе, когда она стоит на второй горизонтали, пешка может походить на одну или на две клетки. Этот факт записывается следующим образом:

```
legalMoves Pawn c@(Cell f r) cs
= if (r == 2)
  then intersect [Cell r (f + 1), Cell r (f + 2)] cs
  else intersect [Cell f (r + 1)] cs
```

Функция `intersect` из модуля `Data.List` возвращает пересечение двух списков, то есть список таких и только таких элементов, которые имеются в обоих списках. Эта функция будет использоваться во всех клозах функции `legalMoves` для отсечения тех возможных ходов, которые не входят в перечень клеток текущего состояния доски (это свойство будет использоваться для того, чтобы не ставить фигуры на те клетки, которые не подходят по условиям задачи, — такие клетки просто будут исключаться из текущего состояния доски).

Конь ходит буквой «Г», то есть на две клетки в одном направлении и на одну в направлении, перпендикулярном исходному. С клетки в середине доски конь может походить на восемь других клеток, а из угловой клетки конь может походить на две клетки. Свойство, которое можно использовать для определения нужных клеток, заключается в том, что сумма абсолютных значений разниц координат исходной и целевой клеток всегда равна трём. В связи с этим определение функции может быть таким:

```
legalMoves Knight c@(Cell f r) cs
= intersect [Cell f' r' | f' <- [f - 2, f - 1, f + 1, f + 2],
              r' <- [r - 2, r - 1, r + 1, r + 2],
              abs (f' - f) + abs (r' - r) == 3] cs
```

Слон ходит по диагонали. Общее свойство всех клеток, на которые может походить слон с исходной, заключается в том, что либо сумма, либо разность координат этих клеток равна соответственно сумме или разности координат исходной клетки. Сумма вычисляет клетки основной диагонали, разность — побочной соответственно. Определение функции выглядит так:

```
legalMoves Bishop c@(Cell f r) cs
= intersect [Cell f' r' | f' <- [1..boardSize],
              r' <- [1..boardSize],
              ((f' + r') == (f + r)) ||
              ((f' - r') == (f - r)),
              (f' /= f) && (r' /= r)] cs
```

Ладья ходит по горизонтали или вертикали. Само собой разумеется, что правило для неё выглядит достаточно просто: все клетки с одинаковой координатой по горизонтали или по вертикали. Определение

несложно (функция `union` из модуля `Data.List` возвращает объединение двух списков, то есть список, состоящий из элементов обоих списков):

```
legalMoves Rock c@(Cell f r) cs
= intersect (union [Cell f' r | f' <- [1..boardSize], f' /= f]
                [Cell f r' | r' <- [1..boardSize], r' /= r]) cs
```

Функцию для ферзя написать ещё проще — он ходит как слон и ладья одновременно:

```
legalMoves Queen c@(Cell f r) cs
= union (legalMoves Bishop c cs) (legalMoves Rock c cs)
```

Остался один король. Король ходит на одну клетку в любом направлении. Для описания этого проще всего перечислить все возможные восемь ходов:

```
legalMoves King c@(Cell f r) cs
= intersect [Cell (f - 1) (r - 1),
            Cell f (r - 1),
            Cell (f + 1) (r - 1),
            Cell (f - 1) r,
            Cell (f + 1) r,
            Cell (f - 1) (r + 1),
            Cell f (r + 1),
            Cell (f + 1) (r + 1)] cs
```

Здесь (и в определении функции для пешки и коня) нет нужды беспокоиться, что в некоторых случаях в список могут попасть клетки с отрицательными значениями или значениями, выходящими за границы доски. Функция `intersect` отсечёт их.

Имея все приведённые в настоящем разделе определения программных сущностей, можно приступить к реализации основных функций для решения поставленных задач.

## Задача о расстановке фигур

Задача о расстановке шахматных фигур формулируется достаточно просто. Для доски размера  $n$  необходимо расставить максимальное число экземпляров заданной шахматной фигуры так, чтобы ни один из экземпляров не был никакой другой. Обычно задача формулируется для старших фигур — ферзя, ладьи, но иногда задачу можно сформулировать и для слона и коня. Также обычно требуется найти такое максимальное число фигур, которое можно уместить на доске.

Так, например, доказано, что на доске  $8 \times 8$  можно расставить максимум 8 ферзей так, чтобы ни один из них не был никакого другого (математическим путём найдено, что таких расстановок имеется ровно 92). То же самое касается и ладей, их можно расставить 8 штук. При этом задача для ладей изоморфна задаче о поиске так называемых «латинских квадратов».

Функция `arrangement` должна возвращать все возможные расстановки экземпляров заданной шахматной фигуры на доске так, чтобы ни один из них не был любой другой. Для решения этой задачи можно воспользоваться технологией генерации списков, которая подробно описана в разделе «Магические квадраты и решение переборных задач» (стр. 90). Её определение таково:

```
arrangement :: Figure -> [[Cell]]
arrangement f = arrangement' f initialBoard
where
    arrangement' _ [] = []
    arrangement' f cs
        = [cell:cells | cell <- cs,
```

```
cells <- arrangement' f (cs \\ 
    (cell : legalMoves f cell cs))]
```

Здесь локальное определение `arrangement'` осуществляет перебор и генерацию вариантов методом «грубой силы». На каждом шаге из перечня ячеек текущей доски выбирается очередная ячейка, для которой решается та же самая задача, но состояние доски уже подправлено — из перечня свободных ячеек исключена текущая ячейка, а также все те, которые находятся под боем заданной фигуры. Исключение ячеек осуществляется операция `(\\)` из модуля `Data.List`.

В принципе, ничего сложного. Другое дело, что эта функция возвращает все возможные расстановки, в том числе и такие, которые могут не соответствовать дополнительным требованиям. Например, для ферзя дополнительным условием является наличие количества ферзей, равного размеру доски. А функция `arrangement` возвратит все расстановки ферзей, даже такие, количество ферзей в которых меньше. Для отсечения (фильтрации) ненужных результатов можно пользоваться стандартными функциями языка Haskell (в частности, функцией `filter`, которая описана в стандартном модуле `Prelude`), но для этого также необходимо реализовать требуемые предикаты, определяющие дополнительные условия задачи.

Например, предикат, проверяющий количество ферзей в расстановке, выглядит тривиально:

```
validQueenArrangement :: [Cell] -> Bool
validQueenArrangement cs = (length cs) == boardSize
```

Ещё можно реализовать функцию, которая будет выводить расстановку фигур в удобочитаемом виде. Её реализация не будет приводиться в книге, поскольку работа над ней является интересным упражнением, которое рекомендуется заинтересованным читателям для самостоятельного решения. В частности, результатом работы подобной функции для первого элемента списка возможных расстановок восьми ферзей на доске размера  $8 \times 8$  является следующая строка (как раз для определения такой функции и пригодится экземпляр класса `Show` для типа `Figure`):

```
..Q.....
.....Q..
...Q.....
.Q.......
.....Q
....Q...
.....Q.
Q.....
```

## Задача о ходе коня

Задача о ходе коня была распространена в Средние века в качестве достаточно сложной головоломки. Условие простое: необходимо обойти все клетки шахматной доски при помощи коня так, чтобы конь побывал в каждой клетке доски ровно один раз. Начинать обход доски можно с любой клетки. Особым шиком считалось решение задачи так, чтобы из конечной клетки маршрута коня можно было опять же при помощи последнего шага попасть на начальную клетку (таким образом, маршрут становился замкнутым).

С точки зрения теории графов, маршрут коня на доске является гамильтоновым путём (либо гамильтоновым циклом, если маршрут замкнут). Задача поиска гамильтонова пути в цикле является NP-полной задачей, то есть такой, время работы алгоритма для решения которой существенно зависит от размера входных данных (в данном случае от размера шахматной доски). Для небольших размеров

входных данных задачу опять же можно несложно решить методом «грубой силы» (простым перебором с отсечением явно неправильных решений).

Функция, которая реализует перебор для поиска маршрута коня, может быть реализована на основе функции для поиска расстановок фигур. У них есть общая внутренняя структура, собственно, и сами задачи являются в какой-то мере схожими. Так что определение функции `knightMoves` выглядит так:

```

knightMoves :: Cell -> [[Cell]]
knightMoves c = knightMoves' c initialBoard
  where
    knightMoves' _ [] = []
    knightMoves' c b
      = if (null lm)
        then [[c]]
        else [c : cells | cell <- lm,
                         cells <- (knightMoves' cell (delete c b))]
  where
    lm = legalMoves Knight c b

```

Функция `delete` опять же взята из модуля `Data.List` (в нём, кстати, определено очень много полезных функций для работы со списками произвольной природы). Эта функция удаляет из списка заданный элемент (его первое вхождение).

Здесь опять, как и в случае функции `arrangement`, функция возвращает все возможные пути коня по шахматной доске, даже те, длина которых меньше количества ячеек на доске. Для отражения дополнительных условий, как и в первом случае, необходимо реализовать соответствующие предикаты. Предикат для поиска и фильтрации путей, которые покрывают все ячейки доски, реализовать просто:

```

validKnightMoves :: [Cell] -> Bool
validKnightMoves cs = (length cs) == (boardSize ^ 2)

```

Наконец, заинтересованному читателю также рекомендуется реализовать функцию, которая будет выводить маршрут коня на доске в удобочитаемом виде. Например, таком (функция не должна напрямую зависеть от размера доски, его необходимо брать из уже определённой константы):

```

+---+---+---+---+---+---+---+
| 20| 15| 22| 45| 58| 47| 38| 43|
+---+---+---+---+---+---+---+
| 17| 04| 19| 48| 37| 44| 59| 56|
+---+---+---+---+---+---+---+
| 14| 21| 16| 23| 46| 57| 42| 39|
+---+---+---+---+---+---+---+
| 03| 18| 05| 36| 49| 40| 55| 60|
+---+---+---+---+---+---+---+
| 06| 13| 08| 29| 24| 35| 50| 41|
+---+---+---+---+---+---+---+
| 09| 02| 11| 32| 27| 52| 61| 54|
+---+---+---+---+---+---+---+
| 12| 07| 28| 25| 30| 63| 34| 51|
+---+---+---+---+---+---+---+
| 01| 10| 31| 64| 33| 26| 53| 62|
+---+---+---+---+---+---+---+

```

## Заключение

Таким образом, в разделе написаны две обобщённые функции для решения интересных и часто встречающихся на практике шахматных задач. Читатели могут использовать их для поиска решений подобных шахматных задач для произвольных фигур. Как обычно, кодирование сущностей проблемной области на языке Haskell показывает эффективность и приспособленность этого языка для выражения задач.

Например, интересной задачей будет поиск для заданной позиции хода, ставящего мат королю противника (мат в один ход). Используя функции, описанные в данном разделе, а также техники перебора (см. раздел «Магические квадраты и решение переборных задач», стр. 90), можно достаточно тривиально найти такое решение. Следующим шагом будет решение задачи о мате в два и более ходов. Так что есть простор для творчества.

# Генерация рекурсивных сказок

*Статья была опубликована в № 11 (47) и 12 (48) журнала «Потенциал» в ноябре и декабре 2008 года.*

*В настоящем эссе описывается одна из интереснейших прикладных задач — генерация естественноязыковых текстов на примере так называемых рекурсивных сказок из числа русских народных. Приводятся программы на функциональном языке программирования Haskell, которые производят генерацию текстов сказок «Колобок», «Теремок» и «Репка» на основе списка действующих лиц.*

## Введение

За сотни и даже тысячи лет своего существования русский народ создал широчайший пласт фольклорного творчества — русские народные сказки. В свою очередь, они подразделяются на несколько классов — волшебные, бытовые, сказки о животных и др. Ещё одним из таких классов являются так называемые рекурсивные сказки, которые обычно рассказывают самим маленьким детям, — постоянное повторение одного и того же сюжета с постепенным наращиванием позволяет детям заучивать слова; размежеванный ритм сказки помогает уснуть. Рекурсивные сказки тоже разделяются на несколько видов, некоторые из которых не так интересны (например, сказка про белого бычка, которая никогда не заканчивается).

Более интересны сказки с «раскручивающимся» сюжетом, в которых основное действие повторяется с каждым новым действующим лицом, пока не наступает кульминация. Таковы всем известные сказки про колобка, про теремок (в некоторых вариантах про потерянную мужиком в лесу рукавичку), про репку. И кто в детстве не пытался проникнуть в суть таких сказок, ещё не понимая принципа *рекурсивных вычислений*, когда по собственному разумению добавлял в перечень действующих лиц новых и новых персонажей?

Исследователи народного творчества давно заметили такую особенность рекурсивных сказок, как «раскручивание» сюжета до кульминации. Более того, во всех сказках выделяется одна структура, один «паттерн». Если описывать кратко, то такой паттерн заключается в том, что на обычную структуру повествования прозаического произведения (пролог — действие — эпилог) накладывается дополнительный рекурсивный перебор действующих лиц, иногда с заглядыванием вперёд. Первое действующее лицо осуществляет «зачин», все последующие делают своё дело, а последний персонаж приводит всё действие к своей кульминации.

На основе результатов таких исследований можно попробовать написать небольшие компьютерные программы, которые, будучи сами небольшими по размеру, позволяли бы генерировать сказки на основе входных списков действующих лиц неограниченной длины. Само собой разумеется, что для таких целей в силу рекурсивности мотива сказок целесообразно применять функциональную парадигму программирования. Ниже в настоящем разделе приводятся примеры генераторов сказок на языке программирования Haskell, а также делается попытка создания обобщённого генератора в виде *функции высшего порядка*.

Все предлагаемые к изучению примеры приводятся в классическом варианте. Заинтересованный читатель может самостоятельно генерировать новые варианты соответствующих сказок, произвольно меняя список действующих в них лиц.

## Колобок

Колобок был испечён бабкой по наказу деда из малой горстки муки. Не дождавшись того, что дед приступит к трапезе, колобок соскочил на тропинку около избушки и покатился в лес. В лесу он постоянно встречал различных персонажей, каждого из которых пел песенку, упоминая при этом всех предыдущих встреченных лиц (*рекурсия*), чем вводил их в смятение, и ускользал от хищной пасти. Наконец, колобок встретил наиболее хитрого зверя (лису), который его обманул и съел (кульминация).

Как указано во введении, вся канва сказки укладывается в простую схему «пролог — действие — эпилог». Этот факт записать на языке Haskell проще всего. С этого можно и начать:

```
kolobok :: String
kolobok = prologue ++
          act ++
          epilogue
```

Естественно, что никаких таких функций `prologue`, `act` и `epilogue` ещё нет, их необходимо только создать. Первая и последняя реализуются очень просто:

```
prologue :: String
prologue = "Жили-были дед и баба. Вот однажды дед и говорит бабе:\n" ++
           "-- Испеки-ка мне, старая, колобок.\n" ++
           "-- Да из чего ж я его испеку-то? Муки ведь нет.\n" ++
           "-- А ты по амбарам помети, по сусекам поскреби, " ++
           "авось мука и наберётся.\n" ++
           "Ну, баба по амбарам помела, по сусекам поскребла, " ++
           "набрала горсти две муки. " ++
           "Замесила тесто на сметане, слепила колобок, " ++
           "изжарила его в масле и поставила на окно простинуть.\n" ++
           "Колобок лежал-лежал, скучно ему стало, " ++
           "вот он и прыгнул с окна на завалинку, " ++
           "с завалинки на лавку, с лавки на дорожку " ++
           "и покатился по ней в лес.\n"
```

и

```
epilogue :: String
epilogue = ""
```

Эпилог у этой сказки не реализован, потому как, когда лиса съедает колобка, ничего больше не происходит. А вот в функции `prologue` скрыта одна неприятная логическая ошибка — в ней в виде констант упоминаются такие действующие персонажи, как «дед» и «баба», что может привести к неприятным последствиям, если иметь желание внести их в список действующих лиц. Но пока проводится исследование внутренней структуры сказки, пусть будет так — список персонажей начнётся с зайца.

Самое интересное начинается при попытке реализовать функцию `act`, которая должна перебирать перечень действующих лиц. Для её работы прежде всего необходим как раз этот перечень, а для его создания в виде списка необходим тип, описывающий одно действующее лицо. Такой тип проще всего определить в виде структуры с именованными полями, которых будет четыре — три для наименования

персонажа в именительном, родительном и винительном падежах (все эти формы встречаются в тексте сказки), а также для указания пола персонажа, чтобы правильно генерировать форму глагола в прошедшем времени. Таким образом тип `Actor` выглядит следующим образом:

```
data Actor
= Actor
{
    name :: String,
    name_g :: String,
    name_a :: String,
    gender :: Gender
}
deriving Eq
```

Ну и вспомогательный тип `Gender` является простым перечислением:

```
data Gender
= Male
| Female
| It
deriving Eq
```

Поскольку пол используется для генерации окончания глагола в прошедшем времени, целесообразно сразу реализовать соответствующие функции. Их будет две, поскольку в русском языке имеется ряд нерегулярных глаголов, у которых нет канонического окончания «-л» для мужского рода: «смог», «пренебрёг», «нёс» и т. д., — это глаголы с окончаниями на «-чъ» и «-ти» в неопределенной форме. Итак, функции:

```
ending :: Gender -> String
ending Male = ""
ending Female = "а"
ending It = "о"

ending' :: Gender -> String
ending' Male = ""
ending' Female = "ла"
ending' It = "ло"
```

Функция `ending'` будет использоваться как раз для нерегулярных глаголов.

Теперь можно определить перечень действующих лиц (ещё раз необходимо напомнить, что используется традиционное представление этой сказки):

```
actors :: [Actor]
actors = [Actor "Заяц" "Зайца" "Зайцу" Male,
         Actor "Волк" "Волка" "Волку" Male,
         Actor "Медведь" "Медведя" "Медведю" Male,
         Actor "Лиса" "Лисы" "Лисе" Female]
```

Вроде бы всё готово, чтобы реализовать функцию `act`, для которой была произведена подготовительная работа. Однако, если вспомнить её тип, который является простой строкой `String`, становится понятно, что эта функция должна состоять из единственного клоза, который вызывает дополнительную (и возможно, даже локальную) функцию `act'`, в которой и осуществляется рекурсивный перебор уже имеющегося списка `actors`. Другими словами, реализация функции `act` использует технологию *накапливающего параметра*, передавая в дополнительную функцию список оставшихся действующих лиц, список уже перечисленных персонажей, а также начальное значение результата:

```

act :: String
act = act' actors [] ""
where
  act' :: [Actor] -> [Actor] -> String -> String
  act' []      _ result = result
  act' [a]     i result = result ++ (finalAct a i)
  act' (a:as)  i result = act' as (i ++ [a])
                                (result ++ (interimAct a i))

```

Функции `interimAct` и `finalAct` ещё должны быть определены. Эти функции возвращают, соответственно, строки, описывающие промежуточное действие и окончательное действие последнего действующего лица (кульминацию). Как видно, обе эти функции принимают на вход текущего персонажа, а также список уже пройденных действующих лиц (обозначен параметром `i`).

Сама функция `act'` просто собирает результат рекурсивного перебора списка персонажей в параметре `result`, значение которого возвращается функцией при достижении пустого списка. В этом и состоит суть технологии использования накапливающего параметра (иначе называемого *аккумулятором*). Если посмотреть на последний клоуз функции `act'`, то видно, что производится непосредственно рекурсивный вызов той же функции, а все вычисления производятся при передаче новых значений параметров. Современные компиляторы языка Haskell производят оптимизацию подобных вычислений, производя их в постоянном объёме памяти (при помощи итераций).

Просто реализовать функцию `interimAct`. В ней будет использоваться ещё одна дополнительная функция `meet`, которая возвращает стандартное для всех персонажей сказки описание встречи колобка с ними. Поскольку для последнего персонажа (лисы) описание встречи абсолютно такое же, целесообразно вынести одинаковые строки в одну строку. При этом надо вспомнить, что для первого действующего лица всё-таки имеется небольшое отличие, которое можно обработать при помощи банального условия `if`:

```

interimAct :: Actor -> [Actor] -> String
interimAct actor i = meet actor i ++
  "Прыгнул колобок, только " ++ an ++
  " его и видел" ++ ae ++ ".\n"
where
  an = name actor
  ae = ending $ gender actor

meet :: Actor -> [Actor] -> String
meet actor i = "Катится колобок " ++ (if (i == [])
  then "по тропинке"
  else "далъше") ++
  ", а навстречу ему " ++ an ++ ". " ++
  "Увидел" ++ ae ++ " " ++ an ++
  " колобка и говорит ему:\n" ++
  "-- Колобок, колобок, я тебя съем.\n" ++
  "А колобок отвечает:\n" ++
  "-- Не ешь меня, " ++ an ++
  ", лучше послушай, какую я тебе песенку спою.\n" ++
  "И запел:\n" ++
  song actor i
where
  an = name actor
  ae = ending $ gender actor

```

Самая интересная функция `song` опять выделена отдельно, но теперь уже больше для эстетических целей (она используется только в функции `meet`. В этой функции как раз и происходит рекурсивный перебор уже встреченных ранее персонажей — колобок поёт свою песенку:

```
song :: Actor -> [Actor] -> String
song actor i = " Я — колобок, колобок.\n" ++
    " Колобок — румяный бок.\n" ++
    " По амбарам метён.\n" ++
    " По сусекам скребён.\n" ++
    " На сметане мешён.\n" ++
    " В жаркой печке печён.\n" ++
    " На окошке стужён.\n" ++
    " Я от бабушки ушёл.\n" ++
    " Я от дедушки ушёл.\n" ++
    song' actor i ""

where
song' actor []      result
= result ++ " А от тебя, " ++ (name actor) ++
    ", и подавно уйду!\n"
song' actor (i:is) result
= song' actor is (result ++ " Я от " ++
    (name_g i) ++ " ушёл.\n")
```

Теперь осталось написать код функции `finalAct`, который, впрочем, будет состоять из вызова всей той же функции `meet` и строк с описанием кульминации:

```
finalAct :: Actor -> [Actor] -> String
finalAct actor i = meet actor i ++
    "А " ++ an ++ " и говорит ему:\n" ++
    "- Ax, какая хорошая песенка. Только вот слаб" ++
    ae ++
    " я на уши стал" ++ ae ++ ". " ++
    "Будь любезен, прыгни ко мне на нос" ++
    "и спой свою песенку ещё раз.\n" ++
    "А колобок и рад, что его песенку похвалили. " ++
    "Прыгнул он " ++ (name_a actor) ++
    " на нос и только хотел снова запеть, " ++
    "а " ++ an ++ " его \"Цап!\" и съел" ++ ae ++ ".\n"

where
an = name actor
ae = ending $ gender actor
```

Генератор сказки «Колобок» готов. Конечно, можно задаться вопросом — в чём прелесть? А суть в том, что этот генератор будет создавать новые сказки вновь и вновь, как только меняется список персонажей. Главное — описывать их с необходимой точностью, задавая три падежа их имён и пол для генерации правильного окончания глаголов в прошедшем времени. И этот код на полторы страницы может генерировать сказки на многие и многие страницы. Конечно, данный код не без огрехов — к примеру, в трёх функциях встречаются абсолютно идентичные локальные определения. Но в целом он представляет собой достаточно простой и эффективный пример генератора естественно-языкового текста специального вида.

## Теремок

В лесу организовалось некоторое место для жилья — теремок (то ли гнилой пень, то ли оброненный проходящим мужчиной чугунок или потерянная рукавичка). К этому теремку начали подбегать различные жители леса, которые спрашивали, кто проживает в таком знатном домике. Все, кто уже поселился в нём, выходили наружу и представлялись (*рекурсия*), после чего призывали вновь прибывшего присоединиться к их весёлой компании. Наконец к теремку подходит огромный зверь (медведь), который никак не мог втиснуться и в без того уже тесное пространство, а потому полез на крышу. Это действие пагубно отражается на строении, которое ломается, а жильцы внезапно оказываются на улице (кульминация).

Реализацию генератора для этой сказки можно основать на уже реализованных программных сущностях для сказки «Колобок». К таковым, без сомнений, относятся тип данных `Gender` и функции `ending` и `ending'`. Более того, функция `act`, которая осуществляет запуск рекурсивного перебора действующих лиц, будет также абсолютно идентична (и это не очень удивительно). Само собой, что все эти функции необходимо помещать в новый модуль, поскольку для единообразия остальные функции будут названы так же, но в них, естественно, будет новое наполнение, соответствующее рассматриваемой сказке.

Дополнительно к служебным функциям необходимо реализовать функцию, которая для заданного пола персонажа возвращает соответствующее ему местоимение третьего рода единственного числа:

```
third :: Gender -> String
third Male    = "он"
third Female = "она"
third It      = "оно"
```

Эта функция пригодится при реализации описания кульминации.

Начальная же функция выглядит стандартно, в полном соответствии с уже упомянутой структурой сказки:

```
attic :: String
attic = prologue ++
        act ++
        epilogue
```

Реализация функций `prologue` и `epilogue` для этой сказки достаточно проста:

```
prologue :: String
prologue = "Стоит в лесу теремок, он не низок, не высок.\n"

epilogue :: String
epilogue = "И разбежались звери кто куда."
```

Теперь необходимо подумать над структурой типа `Actor`, описывающего одного персонажа. В этой сказке уже не надо предоставлять программе имена действующих лиц в родительном и винительном падежах, но зато необходимо указать, каким именно образом соответствующий персонаж сказки перемещался по лесу, пока не наткнулся на теремок. Так что определение типа `Actor` будет таковым:

```
data Actor
= Actor
{
    name   :: String,
    action :: String,
    gender :: Gender
}
```

Ну и, собственно, перечень действующих лиц для рассматриваемой сказки в одном из её классических вариантов (таковых несколько, и списки персонажей различаются) выглядит следующим образом:

```

actors :: [ Actor ]
actors = [ Actor "Мышка-норушка"           "Бежала"   Female ,
          Actor "Лягушка-квакушка"         "Прыгала"  Female ,
          Actor "Зайчик-побегайчик"       "Скакал"    Male ,
          Actor "Лисичка-сестричка"        "Бежала"   Female ,
          Actor "Волчок - серый бочок"     "Рыскал"    Male ,
          Actor "Медведь"                  "Шёл"      Male ]

```

Набор специфичных функций для генерации отдельного акта в череде рекурсивной последовательности для сказки «Теремок» более сложен, чем для предыдущей сказки. Здесь надо осуществить перебор вселившихся жильцов, которые представляются вновь прибывшему претенденту один за другим. Поэтому функция `interimAct` определяется так:

```
interimAct :: Actor -> [Actor] -> String
interimAct actor i = ask actor i ++
                     "Вот и стал" ++ ae ++ " " ++ an ++
                     "\n" ++ " в теремке жить.\n"
```

where

an = name actor  
ae = ending \$ gender actor

Функция `ask` возвращает строку, в которой текущее действующее лицо спрашивает о том, кто живёт в теремке. Она выглядит следующим образом:

```
ask :: Actor -> [Actor] -> String
ask actor i = (action actor) ++ " " ++ an ++ ". " ++
    "Увидел" ++ ae ++ " теремок и спрашивает:\n" ++
    "- Кто в теремочке живёт? Кто в невысоком живёт?" ++
    "\n" ++ salute actor i
```

where

an = name actor  
ae = ending \$ gender actor

В конце определения этой функции производится вызов функции `salute`, которая как раз и генерирует перечисление всех уже вселившихся жильцов:

```

salute :: Actor -> [Actor] -> String
salute _ [] = "Никто не отзыается."
salute actor [i] = "Выглянул" ++ (ending $ gender i) ++
                  " " ++ (name i) ++ " и говорит:\n" ++
                  "- Я - " ++ (name i) ++ ". А ты кто?\n" ++
                  answer actor True
salute actor is = "Выглянули жильцы и говорят:" ++
                  salute' is "" ++
                  answer actor False

```

where

Локальное определение `salute'` является переборной рекурсивной функцией, которая опять использует технологию накапливающего параметра. Во всех значимых клозах функции `salute` также производится вызов функции `answer`, которая выделена опять-таки больше в эстетических целях. Она генерирует ответ нового претендента на жительство в теремке:

```
answer :: Actor -> Bool -> String
answer actor one = "- А я - " ++ (name actor) ++ ". Пусти" ++
  (if one then "" else "те") ++
  " меня к себе жить.\n" ++
  "- Да ты не влезешь.\n" ++
  "- Ну я как-нибудь.\n" ++
  "- Ну иди.\n"
```

Второй параметр этой функции является булевским и определяет, сколько жильцов живёт в теремке — один или много. Эта информация используется для правильной генерации окончания императивной формы глагола «пустить». Как видно, все три функции `ask`, `salute` и `answer` можно было вообще объединить в одном определении, которое вызывалось бы из функции `interimAct`. Тем не менее для структуризации кода рекомендуется таким образом выделять логически обоснованные части программы.

Наконец, заключительная функция `finalAct` определяется следующим образом:

```
finalAct :: Actor -> [Actor] -> String
finalAct actor i = ask actor i ++
  "Полез" ++ ae' ++ " " ++ an ++
  " внутрь, да не смог" ++ ae' ++
  " туда поместиться. " ++
  "Тогда влез" ++ ae' ++ " " ++ (third actor) ++
  " на теремок и раздавил" ++ ae ++ " его.\n"

where
  an = name actor
  ae = ending $ gender actor
  ae' = ending' $ gender actor
```

Опять же в построенном генераторе встречаются те же огехи, которые были обнаружены в функции для вывода текста сказки «Колобок». Тем не менее уже на примере этого генератора становится понятным, что объявленная во введении задача имеет типовое решение.

## Обобщение функций и построение генератора

Реализовав два генератора, уже можно увидеть общие принципы, на основе которых можно попытаться создать обобщённый генератор, осуществляющий рекурсивный перебор унифицированных описаний действующих лиц и генерирующий естественно-языковой текст при помощи переданных в качестве параметров специальных функций.

Для создания типовых функций высшего порядка для генерации рекурсивных сказок прежде всего необходимо озабочиться определением обобщённых типов данных для использования в специфических генераторах. Речь прежде всего ведётся о типе для представления описания персонажа сказки, а также о вспомогательных типах для представления пола (грамматического рода), типа окончания глагола и т. д. Последние два типа определяются просто:

```
data VerbPastType
  = RegularVerb
  | IrregularVerb
```

```
deriving (Show, Eq)  
  

data Gender  
= Male  
| Female  
| It  
deriving (Show, Eq)
```

Сразу можно определить специальную функцию для получения окончания глагола в прошедшем времени, поскольку такая функция будет с большой уверенностью использоваться во всех специфических генераторах. Реализацию этой функции необходимо немного изменить по сравнению с функциями **ending** и **ending'**, которые введены в подразделе про сказку «Колобок». Пусть окончание глагола нерегулярного типа в мужском роде будет пустым, а для регулярных глаголов таким окончанием будет «-л». Для женского и среднего родов окончания будут «-ла» и «-ло» соответственно. Тогда для конкатенации с результатом вызова этой функции необходимо применять «основу» глагола без суффикса: «помог», «убежа», «прилете» и т. д.

```
verbPastEnding :: VerbPastType -> Gender -> String  

verbPastEnding RegularVerb Male = "л"  

verbPastEnding IrregularVerb Male = ""  

verbPastEnding _ Female = "ла"  

verbPastEnding _ It = "ло"
```

Что касается обобщённого типа **Actor**, то здесь необходимо подумать. Можно придумать специализированный *класс*, при помощи которого можно было бы генерализовать любой тип (использование *полиморфизма*). Но в данном случае проще обойтись конкретным типом, который позволит решать широкий класс задач по представлению персонажей. Можно отметить, что в двух исследованных сказках каждый персонаж имел наименование (имя) и пол, а остальная информация в том или ином виде передавалась в виде набора строк, при этом интерпретация этой передаваемой информации производится в конкретизированных функциях **interimAct** и **finalAct**. Этим и можно воспользоваться:

```
data Actor  
= Actor  
{  
    name :: String,  
    gender :: Gender,  
    info :: [String]  
}  
deriving (Show, Eq)
```

Начинается самое интересное — необходимо реализовать обобщённую функцию **act** и функцию для генерации всего текста рекурсивной сказки. Прежде всего необходимо ещё раз внимательно изучить функцию **act**, которая используется в генераторах сказок «Колобок» и «Теремок». Она содержит несколько не очень хороших моментов, от которых надо бы избавиться при реализации обобщённой функции.

Первый неприятный момент заключается в том, что она использует две конкретизированные функции для генерации отдельных действий в ряду рекурсивных вызовов — **interimAct** и **finalAct**, но при этом в генераторах обеих сказок в функции **interimAct** производится разделение на действие первого элемента перечня персонажей и на действие остальных. Это желательно отразить непосредственно в функции **act**, осуществив разделение на уровне *сопоставления с образцами* в её клозах. Для этих целей необходимо выделить ещё один клоз.

Второй момент заключается в том, что в функции `interimAct` и `finalAct` передавалось текущее действующее лицо и перечень тех, кто уже произвёл действия. Это не совсем корректно, поскольку в некоторых сказках текущий персонаж может обращаться к будущему времени, называя следующих за ним персонажей сказки. Поэтому в указанные функции и функцию `initialAct`, которая будет приниматься на основании решения предыдущего момента, необходимо передавать два списка действующих лиц.

Наконец, третий момент заключается больше в техническом аспекте. Если рассмотреть последний клоз функции `act'` в определении для сказки «Колобок», то будет видно, что при рекурсивном вызове перенос текущего действующего лица в список уже сделавших своё дело персонажей производится при помощи конкатенации `(++)`, что не совсем оптимально и может негативно сказаться на скорости исполнения для больших списков. Поэтому нового персонажа проще добавлять в голову списка при помощи простого конструирования `(:)`.

Так что теперь, учтя все эти моменты, можно просто записать определение обобщённой функции `act`:

```
act :: [Actor]
-> ([Actor] -> [Actor] -> String)
-> ([Actor] -> [Actor] -> String)
-> ([Actor] -> [Actor] -> String)
-> String

act actors
  initialAct
  interimAct
  finalAct = act' actors [] "" initialAct interimAct finalAct
where
  act' :: [Actor]
  -> [Actor]
  -> String
  -> ([Actor] -> [Actor] -> String)
  -> ([Actor] -> [Actor] -> String)
  -> ([Actor] -> [Actor] -> String)
  -> String

  act' []
  -> result -
  = result

  act' [a]           ops result initialAct interimAct finalAct
  = result ++ (finalAct [a] ops)

  act' actors@(a:as) []
  -> result initialAct interimAct finalAct
  = act' as [a]      (result ++ (initialAct actors []))

initialAct interimAct finalAct

  act' actors@(a:as) ops result initialAct interimAct finalAct
  = act' as (a:ops) (result ++ (interimAct actors ops))
  -> initialAct interimAct finalAct
```

Выглядит впечатляюще и немножко ужасающе, но если разобраться, то становится понятно, что это всё то же использование технологии накапливающего параметра в функции `act'`, причём функция `act` используется только как «обёртка». В свою очередь, функция `act'` просто возвращает значение

накапливающего параметра `result`, когда список текущих действующих лиц становится пустым. Второй клоз вызывает конкретизированную функцию `finalAct` и добавляет её значение к общему результату в случае, если текущий список действующих лиц состоит из одного элемента. Третий клоз, соответственно, ограничивает перечень уже просмотренных персонажей пустым списком и вызывает для этого случая функцию `initialAct`, перенося текущее действующее лицо в список тех, кто уже сделал своё дело. Наконец, четвёртый клоз полностью соответствует третьему клозу изначальной функции `act'` для сказки «Колобок» с учётом третьего негативного момента.

Всё готово для реализации обобщённой функции для генерации рекурсивной сказки. Чтобы подвести черту под этим, необходимо подумать, какие объекты она должна получать на вход в качестве параметров. Во-первых, перечень действующих лиц, это без сомнений. Во-вторых, конкретизированные функции для генерации пролога и эпилога, причём они также должны принимать перечень действующих лиц в качестве параметра, поскольку в тексте пролога или эпилога кто-либо из персонажей может упоминаться. Ну и, в-третьих, три функции `initialAct`, `interimAct` и `finalAct` также должны передаваться на вход. В итоге получается такое определение:

```
generateRT :: [Actor]
    -> ([Actor] -> String)
    -> ([Actor] -> [Actor] -> String)
    -> ([Actor] -> [Actor] -> String)
    -> ([Actor] -> [Actor] -> String)
    -> ([Actor] -> String)
    -> String

generateRT actors
    prologue
    initialAct
    interimAct
    finalAct
    epilogue = (prologue actors) ++
        (act actors initialAct interimAct finalAct) ++
        (epilogue actors)
```

## Репка

Один престарелый мужчина посадил исконный овош — репу. В силу определённых природных явлений, произошедших в том приснопамятном году, репа уродилась крайне урожайная, огромных размеров. Дед один репу выдернуть не мог, а потому звал на помощь своих домочадцев — одного за другим (*рекурсия*). Когда все домашние были позваны, гигантский овош был с успехом вынут из земли и сварен на радость всем (кульминация).

Имея в наличии подготовленный инструмент в виде обобщённых функций, теперь достаточно лишь подготовить генераторы конкретного текста и передать их на вход функции `generateRT` вместе со списком действующих лиц. С него и необходимо начать определение генератора сказки «Репка»:

```
actors :: [Actor]
actors = [Actor "дед" Male ["деда"],
          Actor "бабка" Female ["бабку"],
          Actor "внучка" Female ["внучку"],
          Actor "жучка" Female ["жучку"],
          Actor "кошка" Female ["кошку"],
```

```
Actor "мышка" Female ["мышку"]]
```

Как видно, в третьем поле в списке используется единственное значение — родительный падеж имени персонажа. Эта форма применяется в тексте сказки, поскольку каждый персонаж зовёт следующего (тут, кстати, и пригодится то, что в функции `initialAct`, `interimAct` `finalAct` передаётся список всех грядущих персонажей, в голове которого стоит текущий). Будет удобным создать специальную функцию для получения формы родительного падежа по описанию персонажа:

```
actorGenitiveCase :: Actor -> String
actorGenitiveCase = head . info
```

Функции `prologue` и `epilogue` также реализуются без проблем, и тут опять помогает то, что в них передаётся перечень действующих лиц, поскольку в прологе используется первый персонаж списка — он сажает репку:

```
prologue :: [Actor] -> String
prologue (a:as) = "Посади" ++ ae ++ " " ++ an ++
                  " репку. Выросла репка большая—пребольшая.\n"
where
  an = name a
  ae = verbPastEnding RegularVerb $ gender a

epilogue :: [Actor] -> String
epilogue _ = "Наварили они из репки каши, наелись до отвала."
```

При реализации функций `initialAct` и `interimAct` станет ясно, что в них очень много одинакового кода, поэтому всё же целесообразней их реализовать в виде одной функции с передачей дополнительного первого параметра булевского типа для отделения способа вызова, а в дальнейшем передача в функцию `generateRT` будет производиться с использованием технологии *частичного применения*. Тем не менее выделение отдельных функций `initialAct` и `interimAct` в обобщённых функциях сделано правильно.

```
act :: Bool -> [Actor] -> [Actor] -> String
act isInitial actors ops
= (if (isInitial)
    then "Ста" ++ ae ++ " " ++ an ++ " репку тянуть. " ++
        "Тянет—потянет, вытянуть не может.\n"
    else (capitalize $ queue (actor:ops)) ++
        " Тянут—потянут, вытянуть не могут.\n" ++
        "Позва" ++ ae ++ " " ++ an ++ " " ++ next ++ ". "
where
  actor = head actors
  an = name actor
  ae = verbPastEnding RegularVerb $ gender actor
  next = actorGenitiveCase $ head $ tail actors
```

В теле этой функции используются две вспомогательные функции — `capitalize` и `queue`. Первая просто преобразует первый символ в верхний регистр, поскольку имена персонажей сказки иногда начинают предложение (эту функции вообще резонно вынести в модуль с обобщёнными функциями). Она определяется просто:

```
capitalize :: String -> String
capitalize [] = []
capitalize (c:cs) = (toUpper c):cs
```

Стандартная функция `toUpperCase` определена в модуле `Data.Char`.

Функция `queue` организует рекурсивный перебор действующих лиц:

```
queue :: [Actor] -> String
queue []      = ""
queue [o]     = (name o) ++ " за репку."
queue (o:os) = (name o) ++ " за " ++
               (actorGenitiveCase $ head os) ++ ", " ++ queue os
```

А вот функцию `finalAct`, как это обычно бывает, желательно реализовать отдельно. Она также несложная:

```
finalAct :: [Actor] -> [Actor] -> String
finalAct actors ops = (capitalize $ queue (actor:ops)) ++
                      " Тянут—потянут, вытянули репку.\n"
where
```

```
    actor = head actors
```

Наконец, основная функция для генерации всего текста сказки «Репка» теперь выглядит крайне просто:

```
turnip :: String
turnip = generateRT actors
        prologue
        (act True)
        (act False)
        finalAct
        epilogue
```

## Заключение

Для закрепления полученных навыков заинтересованному читателю предлагается самостоятельно реализовать функции для генерации рекурсивных сказок на основе созданной обобщённой функции высшего порядка `generateRT` на примере других рекурсивных русских народных сказок: «Заюшкина избушка», «Бычок — смоляной бычок», «Зимовые зверей» и др. Результаты работ автор с радостью получит на свой электронный адрес и скрупулёзно изучит.

Также необходимо ещё раз отметить, что рекурсивные сказки являются всего лишь небольшим классом всего фольклорного наследия нашего народа. Более интересной, но и несколько сложной задачей является генерация произвольных сказок, причём не существующих текстов (это было бы тривиально), а текстов сказок на основе сюжетов и мотивов с имеющимися действующими лицами и «NPC» (Иван-царевич, Царевна-лягушка, Серый волк, Чудо-юдо, Баба Яга, Кошечка Бессмертный и т. д.), артефактами (клубочек, ковёр-самолёт, скатерть-самобранка, сапоги-скороходы, меч-кладенец и т. д.), локациями (тридевятое царство, река Смородина, дремучий лес, за седьмой водой и т. д.). Здесь неспроста употреблены жargonные термины из лексикона игроков в стратегические игры и квесты — «NPC» (от англ. *non playing character* — неиграющее действующее лицо), «артефакт», «локация», — поскольку сказки часто становились основой таких игр, да и структура сказок, по мнению многих исследователей-фольклористов, подобна структуре таких игр.

Более того, некоторые действующие лица в сказках могут рассказывать в канве сказки свои рассказы и сказки — налицо рекурсивный вызов. В этом аспекте сказки сборника «Тысяча и одна ночь» очень показательны — в некоторых из них насчитывается до семи уровней вложенности (Шахерезада была отличным генератором сказок). Так что задача создания (сложения) новых сказок и сложна, и интересна одновременно. Над её решением также рекомендуется подумать и реализовать свой генератор.

# Литература

- [1] **Барендрегт Х.** Лямбда-исчисление. Его синтаксис и семантика / пер. с англ. — М.: Мир, 1985. — 606 стр.
- [2] **Винер Н.** Кибернетика, или Управление и связь в животном и машине / пер. с англ. — М.: Советское радио, 1958. — 216 стр.
- [3] **Вольфенгаген В. Э.** Методы и средства вычислений с объектами. Аппликативные вычислительные системы. — М.: ЮРИнформ, 2004. — 787 стр. — ISBN 5-89158-100-0.
- [4] **Вольфенгаген В. Э.** Комбинаторная логика в программировании. Вычисления с объектами в примерах и задачах. — М.: МИФИ, 1994. — 204 стр. — ISBN 5-89158-101-9.
- [5] **Гарднер М.** А ну-ка, догадайся! / пер. с англ. Ю. А. Данилова. — М.: Мир, 1984. — 212 стр., ил.
- [6] **Душкин Р. В.** Функциональное программирование на языке Haskell+ CD. — М.: ДМК Пресс, 2007. — 608 стр., ил. — ISBN 5-94074-335-8.
- [7] **Душкин Р. В.** Справочник по языку Haskell. — М.: ДМК Пресс, 2008. — 544 стр., ил. — ISBN 5-94074-410-9.
- [8] **Душкин Р. В.** Практика работы на языке Haskell+ CD. — М.: ДМК Пресс, 2009. — 288 стр., ил. — ISBN 978-5-94074-588-4.
- [9] **Зефиров С. А.** Лень бояться // Практика функционального программирования. — 2009. — Июль. — № 1. — с. 9-16.
- [10] **Карпенко А. С.** Многозначные логики // Логика и компьютер. Вып. № 4. — М.: Наука, 1997.
- [11] **Кирпичёв Е. Р.** Монады // RSDN Magazine. — 2008. — № 3.
- [12] **Лукасевич Я.** Аристотелевская силлогистика с точки зрения современной формальной логики. — М.: Иностранная литература, 1959.
- [13] **Пенроуз Р.** Новый ум короля: о компьютерах, мышлении и законах физики / пер. с англ., общ. ред. В. О. Малышенко, предисл. Г. Г. Малинецкого. — 2-е изд., испр. — М.: Едиториал УРСС, 2005. — 400 стр. (Синергетика: от прошлого к будущему.) — ISBN 5-354-00993-6.
- [14] **Роганова Н. А.** Функциональное программирование: учеб. пособие для студентов выс. учеб. заведений. — М.: ГИНФО, 2002. — 260 стр.
- [15] **Филд А., Харрисон П.** Функциональное программирование / пер. с англ. — М.: Мир, 1993. — 637 стр., ил. — ISBN 5-03-001870-0.
- [16] **Хендерсон П.** Функциональное программирование. Применение и реализация // Математическое обеспечение ЭВМ / пер. с англ. (Петрова Л. Т.) — М.: Мир, 1983. — 349 стр.

- [17] **Curry H. B.** Grundlagen der kombinatorischen Logik // American Journal of Mathematics 52. — 1930. — P. 509–536, 789–834.
- [18] **Fokker J.** The Systematic Construction of a One-combinator Basis for Lambda-Terms // Formal Aspects of Computing 4. — 1992. — P. 776–780.
- [19] **Newbern J.** All About Monads.
- [20] **Pierce B. C.** Types and Programming Languages. — The MIT Press. Massachusetts Institute of Technology. — Cambridge, Massachusetts 02142. — <http://mitpress.mit.edu/>. — ISBN 0-262-16209-1. (в сети Интернет по адресу <http://newstar.rinet.ru/goga/tapl/> опубликован перевод книги на русский язык.)
- [21] **Schönfinkel M.** Über die Baustein der mathematischen Logik. — Math. Annalen, vol. 92, 1924. — P. 305–316.
- [22] **Wadler P.** Monads for functional programming // J. Jeuring and E. Meijer, editors, Advanced Functional Programming, Proceedings of the Båstad Spring School, May. — 1995. — Springer Verlag Lecture Notes in Computer Science 925.

### Принимаются благодарности

Вниманию всех читателей! Данная книга издана в электронном виде и распространяется абсолютно бесплатно. Вы можете свободно использовать её для чтения, копировать её для друзей, размещать в библиотеках на сайтах в сети Интернет, рассылать по электронной почте и при помощи иных средств передачи информации. Вы можете использовать текст книги частично или полностью в своих работах при условии размещения ссылок на оригинал и должном цитировании.

При этом автор будет нескажанно рад получить читательскую благодарность, которая позволит как улучшить текст данной книги, так и более качественно подойти к подготовке следующих книг. Благодарности принимаются на счёт Яндекс.Деньги, на который можно перечислить малую лепту, и при помощи терминалов:

**4100137733052**

Убедительная просьба; при перечислении благодарности указывать в пояснении к переводу наименование книги или какое-либо иное указание на то, за что именно выражается благодарность.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@aliants-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.aliants-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**; электронный адрес **books@aliants-kniga.ru**.

Душкин Роман Викторович

**14 занимательных эссе о языке Haskell  
и функциональном программировании**

Главный редактор *Мовчан Д. А.*  
*dm@dmk-press.ru*

Корректор *Синяева Г. И.*  
Верстка *Душкин Р. В.*  
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 29.01.2011. Формат 70Х100<sup>1</sup>/<sub>16</sub>.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 27. Тираж 1000 экз.

Издательство ДМК Пресс.  
Web-сайт издательства: [www.dmk-press.ru](http://www.dmk-press.ru)  
Internet-магазин: [www.aliants-kniga.ru](http://www.aliants-kniga.ru)

# 14 занимательных эссе о языке Haskell и функциональном программировании

В книге представлено 14 статей автора, которые в разное время были опубликованы или подготовлены к публикации в научно-популярном журнале для школьников и учителей «Потенциал». Статьи расположены и связаны таким образом, чтобы они представляли собой логически последовательное повествование от начал к более сложным темам. Также в книге сделан упор на практические знания, предлагается решение многих прикладных задач при помощи языка функционального программирования Haskell.

Книга будет интересна всем, кто живо интересуется функциональным программированием, студентам технических ВУЗов, преподавателям информатики. Кроме того, книга будет полезна всем желающим овладеть пониманием функционального программирования в целом и языка Haskell в частности. И в любом случае книга станет хорошим источником идей, задач и их решений для всех, кто интересуется функциональным программированием.

**Роман Викторович Душкин** является автором первой книги на русском языке о функциональном программировании на языке Haskell, а также множества научных публикаций по темам нечеткой математики, искусственного интеллекта и функционального программирования в российских и зарубежных научных изданиях. Состоит в Российской Ассоциации Искусственного Интеллекта и участвовал во множестве национальных и международных научных конференциях, проводимых под ее эгидой. С 2001 года читал лекции по функциональному программированию в Московском инженерно-физическом институте (МИФИ). В настоящее время работает в области автоматизации промышленности и государственного управления, на практике используя все методы создания программных средств, применяющиеся в составе парадигм функционального и объектно-ориентированного программирования, а также искусственного интеллекта.

**Internet-магазин:**

[www.alians-kniga.ru](http://www.alians-kniga.ru)

**Книга - по почтой:**

Россия, 123242, Москва, а/я 20

e-mail: [orders@aliants-kniga.ru](mailto:orders@aliants-kniga.ru)

**Оптовая продажа:**

«Альянс-книга»

(495)258-9194, 258-9195

e-mail: [books@aliants-kniga.ru](mailto:books@aliants-kniga.ru)

ISBN 978-5-94074-691-1



9 785940 746911 >

