14 занимательных эссе о языке Haskell и функциональном программировании

Душкин Р. В.

ДМК издательство

УДК 004.4
ББК 32.973.26-018.2
Д86

Душкин Р. В.
Д86 14 занимательных эссе о языке Haskell и функциональном программировании. — Изд. 2-ое, исп., 2011. — 284 с., ил.

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 всё равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несёт ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-94074-691-1

© Душкин Р. В., 2011
© Иллюстрации: журнал «Потенциал»
© Оформление: ДМК Пресс, 2011

Содержание

От автора ... 7
Предисловие ко второму изданию ... 8
Эссе 1. Типовой процесс разработки на языке Haskell ... 12
Инструментальные средства ... 14
Описание процесса разработки ... 16
Эссе 2. Функциональный подход в программировании ... 23
Введение ... 24
Общие свойства функций в функциональных языках программирования ... 26
Примеры определения функций ... 29
Заключение ... 37
Эссе 3. Алгебраические типы данных в языке Haskell ... 38
Введение ... 38
Простые перечисления ... 40
Параметризация ... 46
Параметрический полиморфизм ... 49
Заключение ... 53
Эссе 4. Объектно-ориентированное и функциональное программирование ... 55
Введение ... 56

Именованные поля и структуры ... 58
Классы типов ... 63
Экземпляры классов ... 67
Окончательные замечания ... 71
Заключение ... 74
ЭССЕ 5. ВВЕДЕНИЕ В Λ-ИСЧИСЛЕНИЕ ДЛЯ НАЧИНАЮЩИХ ... 76
Введение ... 77
Неформальное описание теории ... 78
Некоторые дополнения ... 81
Редукция как стратегия вычислений ... 83
Примеры кодирования данных и функций ... 88
Заключение ... 97
ЭССЕ 6. КОМБИНАТОРЫ? — ЭТО ПРОСТО! ... 99
Введение ... 99
Формальная теория ... 100
Примеры сложных комбинаторов ... 105
Модуль на языке Haskell для преобразования комбинаторов ... 109
Представление данных и функций ... 112
Булевские значения ... 113
Нумералы Чёрча ... 114
Упорядоченные пары ... 116
Общие замечания ... 117
Заключение ... 117
ЭССЕ 7. ВВОД И ВЫВОД НА ЯЗЫКЕ HASKELL ... 120
Введение ... 120
Основы функционального ввода/вывода ... 123
Стандартные функции ввода/вывода ... 128
Примеры программ ... 133
Вывод результатов исполнения функции на экран ... 134
Альтернатива: экран или файл ... 136
Копирование файлов ... 139
Заключение ... 141
ЭССЕ 8. ПРОСТОЙ ИНТЕРПРЕТАТОР КОМАНД ... 142

Введение 142
Постановка задачи 143
Основной набор функций 146
Вспомогательные типы данных *147*
Цикл интерпретации *149*
Функции для исполнения команд 153
Заключение 158
ЭССЕ 9. ТЕОРИЯ ЧИСЕЛ И ЯЗЫК HASKELL 159
Введение 159
Простейшие задачи 161
Такие непростые простые числа 165
Числа Мерсенна *168*
Числа Ферма *170*
Числа Софи Жермен *171*
Другие последовательности простых чисел *172*
Совершенству нет предела 173
Заключение 176
ЭССЕ 10. МАГИЧЕСКИЕ КВАДРАТЫ И РЕШЕНИЕ ПЕРЕБОРНЫХ ЗАДАЧ 178
Введение 179
Простейший вариант перебора 181
Перебор с использованием перестановок 186
Перебор с использованием размещений 190
Дальнейшая универсализация алгоритма 197
Заключение 201
ЭССЕ 11. ЗАДАЧА О РАНЦЕ 203
Введение 203
Классическая задача 205
Реализация решения на языке Haskell 209
Заключение 215
ЭССЕ 12. КРИВАЯ ДРАКОНА 217
Введение 217
Что такое Кривая Дракона? 221

Алгоритм построения 225
Реализация на языке Haskell 226
Подготовительные описания геометрических образов *226*
Построение Кривой Дракона *231*
Заключение 234

Эссе 13. Немного о шахматных задачах 236
Введение 236
Вспомогательные программные сущности 237
Задача о расстановке фигур 243
Задача о ходе коня 245
Заключение 247

Эссе 14. Генерация рекурсивных сказок 248
Введение 248
Колобок 250
Теремок 258
Обобщение функций и построение генератора 264
Репка 270
Заключение 274

Литература 276

Эссе 1. Типовой процесс разработки 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.

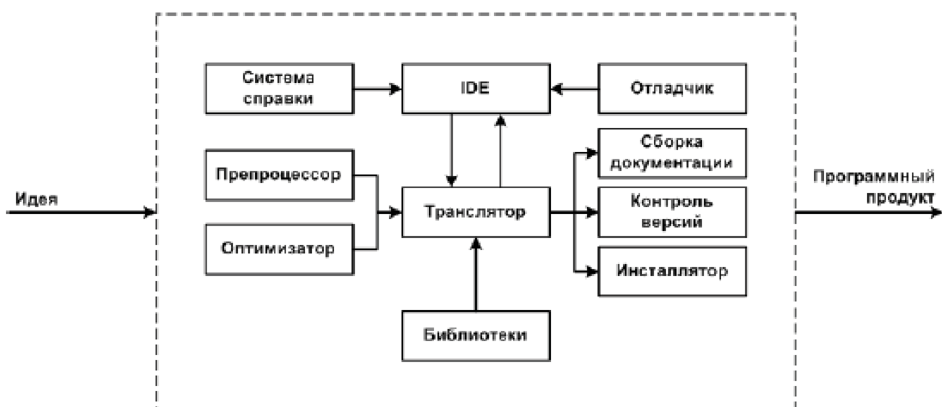

Рисунок 1. Кибернетическая схема преобразования идеи в законченное программное средство

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.

Класс / Инструмент	Описание
Компилятор	Самый мощный и совершенный компилятор, на сегодняшний день не имеющий аналогов, — GHC.

Класс / Инструмент	Описание
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). http://www.haskell.org/ghc/
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. http://www.md.chalmers.se/~rjmh/QuickCheck/
Документирование Haddock	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. http://www.haskell.org/haddock/
Контроль версий Darcs	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. http://www.darcs.net/
Система сборки Cabal	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 установо-

Класс / Инструмент	Описание
	вить в систему требуемый пакет, самостоятельно скачав его из сети Интернет. http://www.haskell.org/cabal/
Распространение Hackage	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. http://hackage.haskell.org/

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:

- `_darcs` — каталог для хранения версий файлов с исходными кодами;
- `Hello.hs` — главный файл проекта (в нём содержится функция `main`);
- `Hello.cabal` — описание проекта для системы Cabal;
- `Setup.hs` — сценарий сборки проекта в системе Cabal;

- `README` — файл с информацией о проекте;
- `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.

- 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 = putStr "Hello, World!"
```

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

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

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

- 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`.

- 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
```

- 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.
- Новые файлы также необходимо сохранить в системе хранения версий Darcs. Это делается при помощи выполнения следующих команд (опять же, при их выполнении утилита может задать до-

полнительные вопросы, на которые необходимо ответить по существу):

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

- 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.

- 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.

- 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 — утилита 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.

- 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.

- 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.

- 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 ком-

пилятор 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.

Эссе 2. 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. Соответственно, 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. Вместе с развитием формальной логики и обобщением множества сходных идей под сводом кибернетики появилось понимание того, что функция является прекрасным мате-

матическим 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:

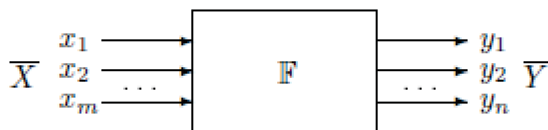

Рисунок 2. Чёрный ящик функции F с вектором входов $\overline{X}$ и вектором выходов $\overline{Y}$

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 λ-исчисление [1]. Само по себе λ-исчисление предлагает нотацию для простейшего языка программирования. Собственно, ядро языка программирования Haskell представляет собой типизированное λ-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 комбина-