Оглавление

Предисловие

Когда ко мне обратились с идей написать *«Программируй на Haskell»*, я не был уверен, стоит ли мне это делать. В то время моим главным интересом было написание блога Count Bayesie по теории вероятностей. Хотя у меня уже был опыт преподавания как Haskell, так и вообще функционального программирования, с тех пор прошло время и, откровенно говоря, я немного вышел из формы. Мой активный интерес к анализу данных, теории вероятностей и машинному обучению родился из личного разочарования в Haskell. Конечно, язык был красив и мощен, но с помощью нескольких некрасивых строк в R и линейной алгебры я мог выполнять сложный анализ и строить модели, чтобы предсказывать будущее. В Haskell даже ввод/вывод нетривиален! Едва ли я был подходящим евангелистом для написания книги по Haskell.

Затем я вспомнил цитату Д. Д. Сэлинджера из книги «Симор: Введение», где он описывает такую уловку, для того чтобы начать писать:

> Спроси себя как читателя, какую вещь ты хотел бы прочитать больше всего на свете, если бы тебе предложили выбрать что-то по душе? И мне просто не верится, как жутко и вместе с тем как просто будет тогда сделать шаг, о котором я сейчас тебе напишу. Тебе надо будет сесть и без всякого стеснения самому написать такую вещь.

В тот момент я осознал, что именно поэтому я должен написать *Программируй на Haskell*. Есть много хороших книг по Haskell, но ни одна из них

не утоляла мою жажду изучения Haskell. Я всегда хотел прочитать книгу, которая покажет, как решать практические задачи, что зачастую является настоящей болью на Haskell. Я хотел видеть не столько большие промышленного уровня программы, сколько забавные эксперименты, которые позволяют вам исследовать мир с помощью этого впечатляющего языка программирования. Я также всегда хотел прочитать книгу по Haskell, которая разумно коротка и после прочтения позволит мне комфортно заниматься различными весёлыми проектами на Haskell по выходным. Именно такого воплощения книги о Haskell, которую я хотел прочитать, ещё не существовало, и я решил, что написать *«Программируй на Haskell»* было бы неплохо.

Теперь, когда я закончил писать (и читать) эту книгу, я в восторге от того, сколько удовольствия получил. Haskell — это бесконечно интересный язык, в котором всегда есть что-то ещё, чему можно научить. Это сложный для изучения язык, но в этом часть удовольствия. Почти каждая тема в этой книге не будет похожа на то, что вы видели раньше (только если вы не опытный разработчик на Haskell). Радость при изучении Haskell состоит в том, чтобы открываться богатому обучающему опыту. Если вы слишком поспешите в освоении Haskell, то это может оказаться ужасным времяпрепровождением. Однако если вы потратите время на исследования, снова став новичком, то будете вознаграждены.

Благодарности

Написание книги — это огромная работа, и автор — всего лишь один из многих людей, необходимых для обеспечения успеха проекта. Первые, кого я должен поблагодарить, — это люди, оказывавшие мне эмоциональную и интеллектуальную поддержку во время этого большого приключения. Моя жена Лиза и сын Арчер были очень терпеливы к моим долгим часам работы и бесконечно поддерживали меня на протяжении всего пути. Также я должен поблагодарить моих дорогих друзей Ричарда Келли и Хавьера Бенгочея, которые были постоянным источником обратной связи, поддержки и интеллектуальной стимуляции. Этой книги никогда бы не было, если бы мой научный руководитель, Фред Харрис, не дал мне удивительную возможность преподавать Haskell группе восторженных студентов. Я также хотел бы поблагодарить моих коллег в Quick Sprout: Стива Кокса, Иена Мэйна и Хитена Ша, которые вынесли мою бесконечную болтовню о Haskell в течение прошлого года.

Сложно переоценить вклад невероятной команды Manning в эту книгу, помощь оказало больше людей, чем может здесь поместиться. Эта книга была бы тенью того, чем она является, без поддержки моего редактора, Дэна Махарри. Дэн требовал доводить каждую мою хорошую идею до совершенства. Я также должен поблагодарить Эрин Тухей за то, что она была первым человеком, кому пришла безумная идея, что я должен написать книгу по Haskell. Мой технический редактор, Палак Матур, проделал отличную работу, обеспечив лёгкое следование за техническим наполнением этой книги и его понимание. Я также должен поблагодарить Виталия Бра-

гилевского за предоставление ценной обратной связи по улучшению кода в этой книге и Шерон Уилки за её терпеливое редактирование. Наконец, я хотел бы упомянуть рецензентов, которые потратили своё время на прочтение и комментирование этой книги: Александр Мыльцев, Арно Байли, Карлос Айя, Клаудио Родригез, Герман Гонзалез-Моррис, Хемант Капила, Джеймс Анаипакос, Кай Геллиен, Макран Дешпанде, Миккель Арентофт, Никита Дюмин, Питер Хемптон, Ричард Тобиас, Серхио Мартинез, Виктор Татай, Виталий Брагилевский и Юрий Клайман.

Об этой книге

Цель книги *«Программируй на Haskell»* — в том, чтобы дать достаточно полное введение в программирование на языке Haskell, позволяющее вам после её завершения писать нетривиальные, полезные на практике программы. Многие другие книги сильно фокусируются на академических основаниях Haskell, но зачастую оставляют читателей немного озадаченными, когда дело доходит до решения практических задач, совершенно обыденных в других языках. К концу этой книги у вас должно возникнуть стойкое понимание того, что именно делает Haskell интересным как язык программирования, вы также сможете уверенно создавать не совсем игрушечные приложения, которые работают с вводом-выводом, генерируют случайные числа, используют базы данных и в целом выполняют те же вещи, что и программы на других знакомых вам языках программирования.

Кому следует читать эту книгу

Это книга для всех, у кого есть опыт программирования и кто хочет поднять свои навыки программирования и понимания языков программирования на новый уровень. Вы можете прийти к своему заключению относительно практичности Haskell, но существуют две хорошие и вполне прагматичные причины для его изучения.

В первую очередь, даже если вы больше никогда не притронетесь к Haskell, получение навыков программирования на Haskell сделает вас более сильным программистом в целом. Haskell принуждает вас писать безопасный функциональный код, а также аккуратно моделировать ваши задачи. Обучение работе с Haskell научит вас правильнее рассуждать об абстракциях и предотвращать потенциальные ошибки в любых языках про-

граммирования. Не уверен, что мне удастся повстречать разработчика программного обеспечения, который хорошо разбирается в Haskell, но при этом не является программистом уровня выше среднего.

Второе преимущество изучения Haskell — в том, что оно, по сути, сопровождается ускоренным курсом теории языков программирования. Вы вряд ли сможете изучить Haskell на уровне, достаточном для написания нетривиальных программ, обойдясь без значительного объёма знаний о функциональном программировании, ленивых вычислениях и сложных системах типов. Эти основы теории языков программирования не только полезны из академического любопытства, но и служат вполне прагматичным целям. Элементы Haskell постоянно проникают как в новые языки программирования, так и в уже существующие. Знание Haskell и его особенностей поможет вам понимать, чего можно ожидать на горизонтах программирования на годы вперёд.

Как организована эта книга

Структура *«Программируй на Haskell»* может отличаться от многих книг по программированию, которые вы читали раньше. Вместо длинных глав эта книга поделена на короткие и простые для усвоения уроки. Эти уроки объединены в семь модулей, которые покрывают основной материал. Все модули, кроме последнего, заканчиваются итоговыми проектами. Эти итоговые проекты объединяют всё освоенное в модуле с целью разработки расширенного примера. Все уроки содержат упражнения с возможностью быстрой проверки и несложные вопросы, с помощью которых можно убедиться, что вы всё схватываете. В конце каждого урока мы даём несколько более серьёзных задач (их решения приведены в конце книги). Модули покрывают следующее содержание:

- *модуль 1* — этот модуль закладывает основы функционального программирования в целом, а также представляет большинство уникальных характеристик Haskell — после прочтения этого модуля вы будете достаточно знакомы с основами функционального программирования и сможете изучать любой другой функциональный язык, находя при этом материал знакомым;
- *модуль 2* — здесь вы начнёте рассматривать мощную систему типов языка Haskell — этот модуль покрывает базовые типы вроде `Int`, `Char` и `Boolean` и как с их помощью создавать свои типы данных, вы также начнёте рассматривать систему классов типов Haskell, которая позволяет вам использовать одну и ту же функцию с данными различных типов;

- *модуль 3* — когда вы изучите основы типов в Haskell, вы сможете перейти к более абстрактным типам и классам типов, которые делают Haskell таким мощным, вы увидите, как Haskell позволяет комбинировать типы способами, которые невозможны в большинстве других языков программирования, вы изучите классы типов `Monoid` и `Semigroup`, а также увидите, как тип `Maybe` позволяет избавиться от целого класса ошибок в ваших программах;
- *модуль 4* — наконец, вы достаточно изучили Haskell, чтобы обсудить ввод-вывод — этот модуль представляет все основы исполнения ввода-вывода в Haskell и объясняет, что делает его уникальным (и порой сложным), к концу этого модуля вы сможете комфортно писать инструментарий для командной строки, читать и записывать файлы, работать с данными в Юникоде и преобразовывать двоичные данные;
- *модуль 5* — к этому моменту в книге вы уже встречали несколько типов, которые создают *контекст* для других типов. Типы `Maybe` определяют контекст для возможно отсутствующих значений, типы `IO` — это значения, которые имеют контекст использования при вводе-выводе. В этом модуле вы погрузитесь в семейство классов типов, которые необходимы для работы со значениями в контексте: `Functor`, `Applicative` и `Monad`. Хотя у них запугивающие имена, они играют довольно простую роль: применение любой функции в часто используемых контекстах. Несмотря на свою абстрактность эти концепции предлагают вам единый способ работы с типами `Maybe`, `IO` и даже списками;
- *модуль 6* — одна из самых сложных тем позади, время начать думать о написании кода для реального мира. Во-первых, вам надо убедиться, что ваш код правильно организован. Этот модуль начинается с урока по системе модулей в Haskell. Затем в оставшейся части модуля вы будете изучать stack, мощный инструмент для создания и поддержки проектов на Haskell;
- *модуль 7* — мы завершим эту книгу, взглянув на то, чего нам не хватало для работы с Haskell в реальном мире. Этот модуль начинается с обзора обработки ошибок в Haskell, которая отличается от многих других языков. После этого вы взглянете на три практические задачи, реализуемые с помощью Haskell: использование HTTP для создания запросов к REST API, разбор JSON-данных с использованием библиотеки Aeson и разработка приложения, общающегося с базой данных. Вы закончите чтение этой книги задачей, при выборе инструмента для решения которой вряд ли подумали бы о Haskell: эффективные, основанные на массивах алгоритмы с изменяемым состоянием.

Самое сложное в изучении (и преподавании) Haskell — то, что вам нужно пройти довольно большое количество тем, прежде чем вы сможете выполнять даже базовый ввод-вывод. Если ваша цель — понимать и использовать Haskell, то я советую вам читать модули последовательно. Но цель этой книги состоит в том, чтобы вы могли остановиться в нескольких местах, заполучив при этом нечто ценное. Модуль 1 организован так, чтобы предоставить вам крепкий фундамент для изучения любого функционального языка программирования. Будь это Clojure, Scala, F#, Racket или Common Lisp, все они разделяют основные черты, описанные в модуле 1. Если вы уже имеете опыт в функциональном программировании, вы можете пропустить модуль 1, хотя вам всё равно следует обратить внимание на уроки по частичному применению и ленивым вычислениям. К концу модуля 4 вы должны знать Haskell достаточно хорошо, чтобы играться с проектами на выходных. После модуля 5 вы вполне сможете переходить к более сложным темам самостоятельно. Модули 6 и 7 сконцентрированы на использовании Haskell для практических задач.

О коде

В этой книге содержится много примеров кода. Код в этой книге представлен `шрифтом с~фиксированной шириной, таким как этот`, благодаря чему его несложно отличить от обычного текста. Многие фрагменты кода аннотированы цифрами, с помощью которых эти фрагменты потом можно прокомментировать. Более сложные примеры включают в себя стрелки, указывающие на каждый раздел и объясняющие его более детально. При написании кода на Haskell вы будете часто пользоваться REPL для взаимодействия с кодом. Эти секции будут отличаться от остальных, так как они будут иметь текст вида `GHCi>`, что обозначает место, где пользователь вводит код. К сожалению, GHCi по умолчанию отображает строки с кириллическими символами в виде последовательностей кодов, например так:

```
GHCi> "Привет, мир!"
"\1055\1088\1080\1074\1077\1090, \1084\1080\1088!"
```

Во всех таких случаях мы будем оставлять в выводе GHCi кириллицу. При желании вы можете сконфигурировать GHCi так, чтобы кириллица отображалась. Для этого можно установить пакет `unescaping-print`, дальнейшие инструкции приведены в описании пакета на Hackage: `http://hackage.haskell.org/package/unescaping-print`.

Также в тексте будут встречаться отсылки к командной строке, в которых используется символ `$` как обозначение места, где пользователь вводит свои команды.

Об авторе

Уилл Курт работает аналитиком в Bombora. Он имеет формальное образование в компьютерных науках (магистр) и английской литературе (бакалавр), а потому заинтересован в объяснении сложных технических тем настолько просто и доступно, насколько возможно. Он читал раздел курса, посвящённый Haskell, в Университете Невады (Рено) и вёл практические семинары по функциональному программированию. Также он ведёт блог о теории вероятностей на `CountBayesie.com`.

1

Начало работы с Haskell

После прочтения урока 1 вы:

- сможете установить инструменты для разработки на Haskell;
- начнёте использовать GHC и GHCi;
- научитесь пользоваться подсказками по написанию программ.

1.1. Добро пожаловать в мир Haskell

Прежде чем погрузиться в изучение Haskell, вам потребуется познакомиться с базовым инструментарием, который вы будете использовать на протяжении обучения. Этот урок поможет вам приступить к работе с Haskell, начнётся он с загрузки необходимого программного обеспечения для написания, компиляции и запуска программ на Haskell. Затем вы сможете взглянуть на примеры кода и вникнуть в суть программирования на Haskell. После всего этого вы будете готовы к полному погружению!

1.1.1. Haskell Platform

Худшая часть изучения нового языка программирования — это первоначальная установка окружения для разработки. К счастью, и к удивлению, это не проблема для Haskell. Haskell-сообщество собрало цельный и простой для установки пакет полезных инструментов, называемый *Haskell Platform*. Haskell Platform — способ распространения языка программирования «из коробки».

Haskell Platform включает в себя:

- компилятор языка Haskell (GHC);
- интерактивный интерпретатор (GHCi);
- утилиту stack для управления проектами на Haskell;
- набор полезных пакетов.

Дистрибутив Haskell Platform можно скачать по следующему адресу: `www.haskell.org/downloads#platform`. После этого следуйте инструкции по установке для той операционной системы, которую вы предпочитаете. Эта книга написана с расчётом на использование GHC версии 8.0.1 и выше.

1.1.2. Текстовые редакторы

Теперь, когда вы установили Haskell Platform, вы, возможно, полюбопытствуете по поводу того, какой редактор следует использовать. Известно, что Haskell — это язык, который настоятельно призывает *думать перед написанием кода*. В результате программы на Haskell обычно получаются очень лаконичными. Так что, помимо контроля отступов и подсветки синтаксиса, редактор здесь не помощник. Многие разработчики используют Emacs с `haskell-mode`. Но если вы незнакомы с Emacs (или не любите работать с ним), определённо не стоит заниматься изучением Emacs в дополнение к Haskell. Мы рекомендуем найти Haskell-плагин для того редактора, который вы используете чаще всего. Такие минималистичные редакторы, как Pico или Notepad++, прекрасно подойдут для наших целей, да и для большинства развитых IDE существуют нужные плагины.

1.2. Компилятор GHC языка Haskell

Haskell — компилируемый язык, а GHC — причина, по которой Haskell является настолько мощным языком. Основная цель компилятора заключается в переводе исходного кода, доступного людям для понимания, в бинарные инструкции, которые может принять компьютер. В Ruby, например, иной подход, там другая программа читает исходный код и интерпретирует его на лету (это делается с помощью *интерпретатора*). Главное преимущество компилятора перед интерпретатором в том, что компилятор может изменить код до запуска, что позволяет выполнить анализ и оптимизацию написанного вами кода. А благодаря другой конструктивной

особенности Haskell, а именно мощной системе типов, возникла поговорка: «Если это компилируется, то это работает». Несмотря на то что вы будете использовать GHC достаточно часто, не воспринимайте компилятор как должное. Этот удивительный образчик программного обеспечения сам по себе достоин отдельной книги.

Для вызова GHC откройте консоль и наберите `ghc`:

```
$ ghc
```

В этой книге если вы встречаете символ `$`, то это означает, что вам нужно напечатать что-то в командной строке. Конечно, без файлов для компиляции GHC будет возмущаться. Для начала вам нужно сделать простой файл hello.hs. В редакторе, который вы предпочитаете, создайте новый файл hello.hs и введите следующие строки:

Листинг 1.1 Пример первой программы hello.hs

```
--hello.hs мой первый Haskell-файл!
main = do
  putStrLn "Привет, мир!"
```

Закомментированная строка с названием вашего файла

Начало функции `main`

В функции `main` выводится строка `"Hello World!"`

На этой стадии не волнуйтесь по поводу того, что происходит в коде этого листинга. Сейчас ваша главная цель — изучить инструменты, которые вам потребуются, чтобы они не стояли на вашем пути при освоении языка.

Сейчас, когда у вас есть этот тестовый файл, запустите GHC снова, на этот раз указав hello.hs как аргумент:

```
$ ghc hello.hs
[1 of 1] Compiling Main
Linking hello ...
```

Если компиляция завершилась успешно, GHC создаст три файла:

- hello (для Windows hello.exe);
- hello.hi;
- hello.o.

Самый важный файл здесь — hello, являющийся исполняемым. А раз он исполняемый, значит, вы можете его запустить:

```
$ ./hello
Привет, мир!
```

Обратите внимание, что стандартное поведение скомпилированной программы — исполнение действий в `main`. Обычно Haskell-программы, которые вы будете компилировать, требуют наличия функции `main`, играющей роль, аналогичную роли метода `Main` в Java/C#, функций `main` в C/C++ или `__main__` в Python.

Как и большинство инструментов командной строки, GHC поддерживает великое множество необязательных флагов. Например, если вы хотите скомпилировать hello.hs в исполняемый файл helloworld, вы можете указать ключ `-o`:

```
$ ghc hello.hs -o helloworld
[1 of 1] Compiling Main
Linking hello ...
```

Для просмотра более полного списка ключей компиляции вызовите `ghc --help` (в этом случае аргумент с названием входного файла не требуется).

Проверка 1.1. Скопируйте код из hello.hs и скомпилируйте свой собственный исполняемый файл testprogram.

1.3. Взаимодействие с Haskell — GHCi

Одним из наиболее полезных инструментов для написания программ на Haskell является GHCi, интерактивный интерпретатор. Как и GHC, его можно запустить, введя простую команду: `ghci`. После запуска GHCi вы будете встречены новой командной строкой:

```
$ ghci
GHCi>
```

Ответ 1.1. Просто скопируйте код в файл и затем, находясь в том же каталоге, в котором был создан файл, введите следующее:

```
ghc hello.hs -o testprogram
```

В этой книге использование GHCi обозначается с помощью `GHCi>` в начале строк, которые вводите вы, и пустого префикса для строк, которые печатает GHCi. Первая вещь, которой следует научиться при использовании любой программы, которую вы запускаете из консоли, — это то, как из неё выйти! В GHCi для выхода достаточно ввести `:q` в командной строке:

```
$ ghci
GHCi> :q
Leaving GHCi.
```

Работа с GHCi очень похожа на работу с интерпретаторами большинства других интерпретируемых языков программирования, например таких, как Python и Ruby. Его можно использовать как обычный калькулятор:

```
ghci> 1+1
2
```

Вы также можете писать код в GHCi на лету:

```
ghci> x=2+2
ghci> x
4
```

До выхода восьмой версии компилятора в GHCi определения функций и переменных требовалось начинать с ключевого слова `let`. Теперь это необязательно, но во многих примерах кода на Haskell, которые можно найти в Интернете и старых книгах, это ключевое слово ещё встречается:

```
ghci> let f x = x+x
ghci> f 2
4
```

Наиболее значимый способ использования GHCi — взаимодействие с программами, которые вы пишете. Существует два способа загрузки файла в GHCi. Первый — передать имя файла в качестве аргумента `ghci`:

```
$ ghci hello.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
```

Другой — использовать команду `:l` (краткая версия `:load`) во время сеанса работы с GHCi:

```
$ ghci
GHCi> :l hello.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
```

В обоих случаях вы сможете вызвать реализованные функции:

```
ghci> :l hello.hs
ghci> main
Привет, мир!
```

В отличие от компиляции файлов с помощью GHC, вашим файлам не требуется наличие `main` для загрузки в GHCi. Каждый раз, загружая файл, вы переписываете существующие определения функций и переменных. Вы можете постоянно загружать ваш файл, пока работаете с ним и вносите изменения. Haskell является, пожалуй, уникальным языком из-за хорошей поддержки компилятора, а также естественной и простой в использовании интерактивной среды. Если вы пришли к Haskell после Python, Ruby или JavaScript, то будете чувствовать себя как дома, используя GHCi. А если знакомы с компилируемыми языками, такими как Java, C# или C++, то, скорее всего, при написании программ на Haskell будете поражаться, что работаете с компилируемым языком.

Проверка 1.2. Измените вашу первую программу так, чтобы она выводила `Привет, <Ваше имя>!`. Перезагрузите её в GHCi и протестируйте вывод.

Ответ 1.2. Измените файл следующим образом:

```
main = do
  putStrLn "Привет, Уилл!"
```

И загрузите его в GHCi:

```
GHCi> :l hello.hs
GHCi> main
Привет, Уилл!
```

1.4. Написание кода на Haskell и работа с ним

Одной из наиболее разочаровывающих вещей для начинающих программистов на Haskell является то, что базовые операции ввода-вывода довольно сложны. Очень часто при изучении нового языка распространённой практикой является вывод некой информации на каждом шаге выполнения, для того чтобы понять, как работает программа. В Haskell же этот способ отладки несколько затруднён. Обычное дело — столкнуться с ошибкой в программе на Haskell и запутанным сообщением о ней, абсолютно при этом не понимая, как действовать дальше.

Усугубляет эту проблему то, что чудесный компилятор Haskell достаточно строго проверяет корректность вашего кода. Если вы привыкли быстро написать программу, запустить её, а потом по-быстрому исправить сделанные ошибки, то Haskell вас разочарует. Haskell поощряет медитативное сидение и обдумывание встреченных проблем до запуска программы. После того как вы наберётесь опыта, мы уверены, это разочарование превратится в вашу любимую особенность Haskell. Оборотной стороной одержимости корректностью на этапе компиляции является то, что ваши программы будут работать гораздо более предсказуемо, чем вы, возможно, привыкли.

Секрет спокойного написания кода без частых столкновений с ошибками состоит в том, чтобы писать его маленькими кусочками и тестировать эти кусочки в процессе написания. Чтобы продемонстрировать этот способ работы, давайте возьмём сложную программу на Haskell и почистим её так, чтобы стали понятны отдельные фрагменты. Например, давайте разработаем консольное приложение для составления электронных писем с благодарностями читателям от авторов. Ниже находится первая, плохо написанная версия этой программы.

Листинг 1.2 Черновая версия first_prog.hs

```
messyMain :: IO ()
messyMain = do
    putStrLn "Кто получатель этого письма?"
    recipient <- getLine
    putStrLn "Название книги:"
    title <- getLine
    putStrLn "Кто автор этого письма?"
    author <- getLine
    putStrLn ("Дорогой " ++ recipient ++ "!\n"
          ++ "Спасибо за то, что купили \""
```

```
        ++ title ++ "\"!\nС уважением,\n" ++ author)
```

Ключевой момент в этой программе — одна большая цельная функция `messyMain`. Рекомендация писать модульный код универсальна при разработке программного обеспечения, но в Haskell особенно важно писать код, который вы сможете понимать и инспектировать. Эта программа работает, хотя и выглядит неряшливо. Если вы измените `messyMain` на `main`, то сможете её скомпилировать и запустить. Но вы также можете загрузить этот код в GHCi без изменений, при условии что находитесь в том же каталоге, что и first_prog.hs:

```
$ ghci
GHCi> :l first_prog.hs
[1 of 1] Compiling Main    ( first_prog.hs, interpreted)
Ok, modules loaded: Main.
```

Если GHCi вывел «Ok», то становится понятно, что код был успешно скомпилирован и нормально работает. Обратите внимание, что GHCi не волнует наличие `main`. Теперь можно устроить тестовый прогон программы (вводимые пользователем строки выделены полужирным шрифтом):

```
GHCi> messyMain
Кто получатель этого письма?
Читатель
Название книги:
Программируй на Haskell
Кто автор этого письма?
Уилл
Дорогой Читатель!
Спасибо за то, что купили "Программируй на Haskell"!
С уважением,
Уилл
```

Всё нормально, но было бы гораздо удобнее работать с кодом, разбитым на меньшие части. Основная задача — создать текст для письма, но можно заметить, что письмо состоит из трёх частей, связанных вместе: имя получателя, основная часть и подпись. Начнём с того, что сделаем отдельные функции для этих частей. Следующие строки нужно добавлять в файл first_prog.hs. Да и большинство функций и значений, которые вы встретите в книге, предназначаются для тех файлов, с которыми вы работаете в данный момент. Начнём с функции `toPart`:

```
toPart recipient = "Дорогой" ++ recipient ++ "!\n"
```

В данном случае вы легко сможете написать все три функции разом, но лучше работать вдумчиво, тестируя каждую функцию по мере написания. Для проверки загрузим файл в GHCi:

```
GHCi> :l first_prog.hs
[1 of 1] Compiling Main    ( first_prog.hs, interpreted )
Ok, modules loaded: Main.
GHCi> toPart "Читатель"
"ДорогойЧитатель!\n"
GHCi> toPart "Боб Смит"
"ДорогойБоб Смит!\n"
```

Этот метод работы с кодом, когда он сначала набирается в редакторе, а потом перезагружается в GHCi, будет основным при работе с данной книгой. Чтобы избежать повторений, будем предполагать, что команда `:l first_prog.hs` используется всегда, когда это необходимо.

Теперь, когда вы загрузили файл в GHCi, можно заметить небольшую ошибку — пропущенный пробел между словом «Дорогой» и именем получателя. Давайте посмотрим, как её можно исправить.

Листинг 1.3 Исправленная функция toPart

```
toPart recipient = "Дорогой " ++ recipient ++ "!\n"
```

И протестируем в GHCi:

```
GHCi> toPart "Джуд Лоу"
"Дорогой Джуд Лоу!\n"
```

Выглядит нормально. Теперь определим оставшиеся две функции, реализовав их вместе, но не забывая, что лучше всего реализовывать функции по одной, загружать их в GHCi для тестирования и только потом продвигаться дальше.

Листинг 1.4 Определения функций bodyPart и fromPart

```
bodyPart bookTitle = "Спасибо за то, что купили \""
                     ++ bookTitle ++ "\"!\n"
fromPart author = "С уважением,\n" ++ author
```

Эти функции вы также можете протестировать:

```
GHCi> bodyPart "Программируй на Haskell"
"Спасибо за то, что купили \"Программируй на Haskell\"!\n"
GHCi> fromPart "Уилл"
"С уважением,\nУилл"
```

Всё выглядит прекрасно! Теперь потребуется функция, которая свяжет всё воедино.

Листинг 1.5 Определение функции createEmail

```
createEmail recipient bookTitle author = toPart recipient ++
                                         bodyPart bookTitle ++
                                         fromPart author
```

Обратите внимание на то, как выровнены вызовы функций. Пробельные символы в Haskell значимы (хотя и не настолько существенно, как в Python). Любое форматирование отнюдь не случайно: если какие-то секции кода выровнены, то для этого есть причины. Большинство редакторов может автоматически следить за этим, если поставить плагины для работы с Haskell.

Вооружившись всеми написанными функциями, можно протестировать `createEmail`:

```
GHCi> createEmail "Читатель" "Программируй на Haskell" "Уилл"
"Дорогой Читатель!\nСпасибо за то, что купили
          ↳ \"Программируй на Haskell\"!\nС уважением,\nУилл"
```

Функции работают как положено, поэтому теперь можно соединить их в `main`.

Листинг 1.6 Версия first_prog.hs с аккуратной main

```
main = do
    putStrLn "Кто получатель этого письма?"
    recipient <- getLine
    putStrLn "Название книги:"
    title <- getLine
    putStrLn "Кто автор этого письма?"
    author <- getLine
    putStrLn (createEmail recipient title author)
```

Всё готово к компиляции, но всегда лучше сначала протестировать программу в GHCi:

```
GHCi> main
Кто получатель этого письма?
Читатель
Название книги:
Программируй на Haskell
Кто автор этого письма?
```

```
Уилл
Дорогой Читатель!
Спасибо за то, что купили "Программируй на Haskell"!
С уважением,
Уилл
```

Похоже, что все части вместе работают нормально, к тому же нам удалось протестировать их по отдельности и убедиться, что они работают так, как предполагалось. Наконец-то можно скомпилировать получившуюся программу:

```
$ ghc first_prog.hs
[1 of 1] Compiling Main    ( first_prog.hs, first_prog.o )
Linking first_prog ...
$ ./first_prog
Кто получатель этого письма?
Читатель
Название книги:
Программируй на Haskell
Кто автор этого письма?
Уилл
Дорогой Читатель!
Спасибо за то, что купили "Программируй на Haskell"!
С уважением,
Уилл
```

Только что вы написали свою первую рабочую программу на Haskell. Теперь, обладая основными знаниями о процессе разработки, вы можете погрузиться в удивительный мир Haskell!

Итоги

В этом уроке нашей главной целью было приступить к работе с Haskell. Начали мы с установки Haskell Platform, соединяющей в себе инструменты, которые мы будем использовать в данной книге. Эти инструменты включают: GHC, компилятор Haskell; GHCi, интерактивный интерпретатор, и stack, инструмент для сборки программ, который потребуется позже. В оставшейся части урока рассматривались вопросы написания и оформления кода на Haskell, а также способы взаимодействия с ним. Давайте проверим, насколько хорошо вы во всём этом разобрались.

Задача 1.1. Посчитайте в GHCi: 2^{123}.

Задача 1.2. Измените код каждой из функций в файле first_prog.hs, тестируя их в GHCi в процессе модификации, а затем скомпилируйте новую версию программы для генерации шаблонов электронных писем, указав `email` в качестве имени исполняемого файла.