Оглавление

Предисловие

Гай Стил

Я пишу программы уже более 52 лет. И мне это все еще интересно, потому что всегда появляются новые проблемы, которые нужно решить, и новые штуки, которые нужно изучать. За прошедшие годы мой стиль программирования серьезно менялся в процессе изучения новых алгоритмов, новых языков программирования и новых методов реализации кода.

Когда я учился программировать в 1960-х годах, считалось, что перед написанием кода следует нарисовать блок-схему программы. Каждое вычисление в программе изображалось прямоугольным блоком, каждое решение — ромбом, а каждая операция ввода/вывода — еще какой-нибудь фигурой. Блоки соединялись стрелками, представляющими передачу управления между блоками. Написание программы сводилось к написанию кода для каждого блока в определенном порядке. Каждый раз, когда стрелка указывала куда-либо за пределами следующего блока, который вы должны были запрограммировать, программист также писал команду `goto` для обозначения необходимой передачи управления. Проблема была в том, что блок-схемы были двумерными, а код — одномерным, и даже если структура блок-схемы на бумаге была красивой и аккуратной, понять ее после записи в виде кода могло быть достаточно сложно. Если провести в коде стрелку от каждой команды `goto` к ее точке передачи, результат часто напоминал клубок спагетти, поэтому в те дни часто говорили о сложностях понимания и сопровождения «спагетти-кода».

Первые изменения в моем стиле программирования были вызваны движением «структурного программирования» в начале 1970-х. Обращаясь к прошлому, я вижу *две* важные идеи, выработанные в ходе обсуждения в сообществе. Обе относятся к средствам **организации потока управления**.

Первая идея получила большую популярность. Она заключалась в том, что большая часть логики передачи управления может быть выражена несколькими

простыми паттернами: последовательным выполнением, многовариантными решениями (например, командами `if-then-else` и `switch`) и повторным выполнением (например, циклы `for` и `while`). Иногда эта идея чрезмерно упрощается до девиза «Никаких команд `goto`!», но здесь важны паттерны, при последовательном использовании которых выяснялось, что необходимость в `goto` возникает крайне редко. Вторая идея, не столь популярная, но не менее важная, гласила, что последовательные команды могут группироваться в блоки, которые должны правильно вкладываться друг в друга, а нелокальная передача управления может осуществляться к концу блока или из блока (команды `break` и `continue`), но не может входить в блок извне.

Когда я впервые ознакомился с идеями структурного программирования, у меня не было доступа к подобному языку. Однако я обнаружил, что стал писать код на Fortran чуть более аккуратно, организуя его по принципам структурного программирования. Даже низкоуровневый код на ассемблере я писал так, словно был компилятором, преобразующим структурный язык программирования в машинные команды. Оказалось, что эта дисциплина упрощает написание и сопровождение моих программ. Да, я все еще писал команды `goto` и команды ветвления, но почти всегда делал это по одному из стандартных паттернов. Код получался намного более понятным.

В старые недобрые времена, когда я программировал на Fortran, все переменные, которые могли понадобиться программе, приходилось объявлять заранее в одном месте, а за ними следовал исполняемый код. (В языке COBOL эта конкретная организация была жестко формализована: переменные объявлялись в «разделе данных» программы, который начинался со слов DATA DIVISION. Далее следовал код, который всегда начинался со слов PROCEDURE DIVISION.) К любой переменной можно было обратиться из любой точки кода. Программисту было труднее понять, как именно можно обратиться к каждой конкретной переменной и изменить ее.

На мой стиль программирования также сильно повлияло «объектно-ориентированное программирование», которое в моем представлении является сочетанием и высшей точкой развития более ранних идей объектов, классов, «сокрытия информации» и «абстрактных типов данных». Оглядываясь назад, я снова вижу, как этот грандиозный синтез породил две выдающиеся идеи, и *обе* были связаны с **организацией доступа к данным**. Первая идея заключается в том, что переменные должны каким-то образом инкапсулироваться, чтобы программисту было проще видеть, что их чтение или запись возможны только из определенных частей кода. Возможны разные варианты инкапсуляции, от простого объявления локальных переменных в блоке (а не в начале программы) до объявления переменных в классе (или модуле), чтобы они были доступны только для методов этого класса (или процедур внутри модуля). Классы или модули могут гарантировать, что группы переменных обладают определенными характеристиками целостности — методы или процедуры могут быть запрограммированы так, чтобы при обновлении одной переменной связанные с ней переменные также обновлялись соответ-

ствующим образом. Вторая идея, наследование, обозначает, что программист может определить более сложный объект путем расширения более простых объектов через добавление новых переменных или методов и, возможно, переопределение существующих методов. Вторая идея становится возможной благодаря первой.

В то время, когда я узнал об объектах и абстрактных типах данных, я писал много кода на языке Lisp. И хотя Lisp не является чистым объектно-ориентированным языком, на нем достаточно просто реализуются структуры данных и обращения к ним только через проверенные методы (реализованные как функции Lisp). Уделяя внимание организации данных, я мог пользоваться многими преимуществами объектно-ориентированного программирования даже с учетом того, что программировал на языке, который не принуждал к соблюдению этой дисциплины.

Третьим фактором, повлиявшим на мой стиль программирования, стало «функциональное программирование», суть которого иногда упрощается до девиза «Никаких побочных эффектов!». Впрочем, это невозможно. На самом деле функциональное программирование предоставляет средства для **упорядочения побочных эффектов**, чтобы они не возникали *где угодно,* — **именно это и является темой книги**.

И здесь мы снова имеем дело с двумя главными идеями, которые работают в сочетании друг с другом. Первая: отделение *вычислений*, которые не влияют на внешнее окружение и выдают один и тот же результат при многократном выполнении, от *действий*, которые могут выдавать разные результаты при каждом выполнении и могут создавать побочные эффекты для внешнего окружения (например, выводить текст на экран или запускать ракеты). Программу будет проще понять, если она организована на базе стандартных паттернов. С такими паттернами программист быстро разберется, какие части могут иметь побочные эффекты, а какие являются «всего лишь вычислениями». Стандартные паттерны можно разделить на две подкатегории: типичные для однопоточных программ (последовательное выполнение) и типичные для многопоточных программ (параллельное выполнение).

Вторая глобальная идея включает набор методов обработки больших коллекций данных (массивов, списков, баз данных) по принципу «все сразу», а не элемент за элементом. Такие методы оказываются наиболее эффективными тогда, когда элементы могут обрабатываться независимо друг от друга, когда на них не влияют побочные эффекты. Получается, что вторая идея снова работает лучше благодаря первой.

В 1995 году я помогал в написании первой полной спецификации языка программирования Java, а на следующий год уже участвовал в создании первого стандарта JavaScript (ECMAScript). Оба этих языка очевидно являются объектно-ориентированными: в Java вообще нет такого понятия, как глобальная переменная, — каждая переменная должна быть объявлена внутри некоторого класса или метода. Кроме того, в обоих языках нет команды `goto`: разработчики языка пришли к выводу, что идеология структурного программирования достиг-

ла успеха и в `goto` более нет надобности. В наши дни миллионы программистов прекрасно обходятся без `goto` и глобальных переменных.

Но как насчет функционального программирования? Существуют чисто функциональные языки, например Haskell, которые широко применяются на практике. Haskell можно использовать для вывода текста на экран или для запуска ракет, но для использования побочных эффектов в Haskell устанавливаются очень жесткие ограничения. Как следствие, вы не можете просто вставить команду вывода в любую точку в программе Haskell, чтобы понять, что в ней происходит.

С другой стороны, Java, JavaScript, C++, Python и многие другие языки программирования не являются чисто функциональными, но они взяли на вооружение многие идеи из функционального программирования, упрощающие их использование. В этом и суть: если вы понимаете ключевые принципы организации побочных эффектов, эти простые идеи можно использовать в любом языке программирования. Книга показывает, как это делается. На первый взгляд она кажется длинной, но это доступная литература, наполненная практическими примерами и врезками с объяснением технических терминов. Я обратил на нее внимание, получил большое удовольствие и узнал пару новых фишек, которые мне не терпится применить в своем коде.

Надеюсь, вам она тоже понравится!

Джессика Керр

Когда я только начинала изучать программирование, оно нравилось мне своей предсказуемостью. Каждая из моих программ была незатейливой: небольшая, работающая на одной машине и простая для использования одним человеком (мной). Разработка современных программных продуктов — совсем другое дело. Программные продукты огромны. Они не пишутся одним человеком. Они работают на многих машинах и во многих процессах. Ими пользуются разные люди, в том числе и те, которых вообще не интересует, как работает программа.

Полезные программы не могут быть простыми.

Что же делать программисту?

Методы функционального программирования в течение 60 лет развивались в умах теоретиков. Исследователи вообще любят делать позитивные заявления относительно того, чего в принципе быть не может.

Последнее десятилетие или два разработчики применяли эти методы в коммерческих программных продуктах. И это было своевременно, потому что соответствовало доминированию веб-приложений: каждое приложение представляет собой распределенную систему, загружаемую на неизвестные компьютеры, с которой работают неизвестные люди. Функциональное программирование отлично подходит для таких целей. Целые категории трудноуловимых ошибок становятся в принципе невозможными.

Однако перенос функционального программирования из академической области в коммерческую разработку не проходит легко и мирно. Мы не работаем на Haskell и не начинаем с нуля. Мы зависим от библиотек и исполнительных сред, которые нам неподконтрольны. Наши программные продукты взаимодействуют с множеством других систем — просто вывести ответ недостаточно. Путь от мира ФП до коммерческого программирования долог.

Эрик прошел этот путь за нас. Он глубоко изучил функциональное программирование, выявил в нем все самое полезное и делится этим с нами.

В прошлом осталась строгая типизация, «чистые» языки, теория категорий. Вместо них мы видим код, который взаимодействует с окружающим миром; разработчик сознательно оставляет данные без изменений и осуществляет логическое разбиение кода для большей ясности. На смену возвышенным абстракциям приходят степени и уровни абстракции. На смену отказу от состояния приходят способы безопасного хранения состояния.

Эрик предлагает взглянуть на существующий код с новых точек зрения. Новые диаграммы, новые «запахи кода», новые эвристики. Да, все это вышло из мира функционального программирования, но когда он находит новые способы выражения своих мыслей, чтобы мы тоже могли ими воспользоваться, он создает нечто новое. Нечто такое, что поможет всем нам в работе над нашими творениями.

Мои простые программы были бесполезными для окружающего мира. Полезные программы никогда не будут простыми. Тем не менее мы можем сделать код проще, чем он был до этого. И мы можем управлять критическими частями, которые взаимодействуют с миром. Эрик распутывает все эти противоречия за нас.

После прочтения этой книги вы будете лучше разбираться в программировании — и более того, в разработке программного обеспечения.

Вступление

Я впервые познакомился с функциональным программированием (ФП) при изучении языка Common Lisp в 2000 году, во время прохождения курса по искусственному интеллекту в университете. По сравнению с другими объектно-ориентированными языками, к которым я привык, Lisp поначалу казался каким-то непривычным и нестандартным. Но к концу семестра я реализовал на Lisp уже достаточно учебных заданий и перестал ощущать дискомфорт. Я ощутил вкус ФП, хотя только начинал понимать его.

С годами я все больше разбирался в функциональном программировании. Написал собственную реализацию Lisp. Читал книги о Lisp. Начал писать на Lisp учебные задания из других курсов. Вскоре я познакомился с Haskell, а в 2008 году и с Clojure. И в Clojure я обрел свою музу. Он строился на базе 50-летней традиции Lisp, но на современной и практичной платформе. А сообщество щедро выдавало идеи о вычислениях, природе данных и практике разработки больших программных систем. Оно было благоприятной почвой для философии, компьютерной теории и технического проектирования. И я был поглощен этим. Я вел блог, посвященный Clojure, и в конечном итоге основал компанию по обучению Clojure.

Тем временем также повышалась популярность Haskell. Я несколько лет профессионально работал на Haskell. У него много общего с Clojure, но существует и ряд различий. Как определить *функциональное программирование*, чтобы определение включало как Clojure, так и Haskell? С формулирования этого вопроса и началась история появления данной книги.

Основной была идея действий, вычислений и данных как главного отличия парадигмы функционального программирования. Если вы спросите любого функционального программиста, он согласится с тем, что это отличие критично для практики ФП, хотя лишь немногие согласятся с тем, что оно является определяющим аспектом парадигмы. Все это отдает когнитивным диссонансом. Как известно, люди склонны учить так, как когда-то учили их. В этом когни-

тивном диссонансе я увидел возможность помочь людям изучать ФП по новым принципам.

Я проработал много черновых вариантов этой книги. Один был излишне теоретическим. В другом демонстрировались впечатляющие возможности ФП. Третий отличался чрезмерной дидактичностью. Четвертый был полностью повествовательным. Но в конечном итоге после всех наставлений со стороны редактора я пришел к нынешнему варианту, в котором функциональное программирование рассматривается как совокупность навыков. По сути, речь идет о навыках, стандартных в кругах ФП, но редко встречающихся за их пределами. И когда я определился с этим подходом, планирование книги свелось к нахождению таких навыков, их упорядочению и расстановке приоритетов. После этого работа пошла очень быстро.

Конечно, в книге не получится рассмотреть все возможные навыки. Функциональному программированию не менее 60 лет. Мне не хватило места для описания многих методов, которые определенно стоило бы описать. Надеюсь, что навыки, рассмотренные в книге, станут отправной точкой для обсуждения и дальнейшего обучения для других авторов.

Моя главная цель при написании книги заключалась в том, чтобы по крайней мере запустить процесс легитимизации функционального программирования как прагматичного варианта для профессиональных программистов. Когда программист хочет изучить объектно-ориентированное программирование, он найдет множество книг по теме, написанных именно для него — начинающего профессионала. В этих книгах описываются паттерны, принципы и практики, на основе которых учащийся может формировать свои навыки. У функционального программирования такой учебной литературы нет. Существующие книги в основном имеют академическую природу, а тем, которые пытаются ориентироваться больше на практику, по моему мнению, не удается объяснить основные концепции. Однако все необходимые знания и опыт есть у тысяч функциональных программистов. Надеюсь, эта книга будет способствовать расцвету литературы о функциональном программировании.

Благодарности

Прежде всего, мне хотелось бы поблагодарить Рика Хики и все сообщество Clojure — неиссякаемый источник философских, научных и технических идей, относящихся к программированию. Вы вдохновляли меня и были моим стимулом.

Я должен поблагодарить свою семью, особенно Вирджинию Мединилья, Оливию Норманд и Изабеллу Норманд, поддерживавших меня во время написания книги своим терпением и любовью. Также я благодарю Лиз Уильямс, которая постоянно помогала мне своими советами.

Спасибо Гаю Стилу и Джесси Керр за их внимание к книге. Видеть суть вещей дано не каждому, но я считаю, что вы правильно поняли замысел этой книги. И конечно, спасибо за личные впечатления, которыми вы поделились во вступлении.

Наконец, хочу поблагодарить сотрудников издательства Manning. Берт Бэйтс, спасибо за бесчисленные часы обсуждений, часто менявших направление, — благодаря им эта книга все-таки была завершена. Спасибо за постоянные наставления относительно того, как лучше учить. Спасибо за терпение и поддержку в то время, когда я разбирался, какой должна быть эта книга. Благодарю Дженни Стаут за то, что она вела проект в нужном направлении.

Спасибо Дженнифер Хаул за прекрасный дизайн книги. Спасибо всем остальным работникам Manning, которые участвовали в работе. Я знаю, что эта книга была непростой во многих отношениях. Хочу упомянуть всех рецензентов: Майкл Эйдинбас, Джеймс Дж. Билецки, Хавьер Колладо, Тео Деспудис, Фернандо Гарсиа, Клайв Харбер, Фред Хит, Колин Джойс, Оливье Кортен, Джоэл Лукка, Филипп Мешан, Брайан Миллер, Орландо Мендес, Нага Паван Кумар Т., Роб Пачеко, Дэн Пози, Аншуман Пурохит, Конор Редмонд, Эдвард Рибейро, Дэвид Ринк, Армин Сейдлин, Кай Стрем, Кент Спиллнер, Серж Симон, Ричард Таттл, Айвен Фелизо и Грег Райт — ваши предложения помогли улучшить эту книгу.

О книге

Для кого написана эта книга

Книга написана для программистов с практическим опытом от 2 до 5 лет. Предполагается, что вы уже знаете хотя бы один язык программирования. Также желательно, чтобы вы построили хотя бы одну достаточно крупную систему, чтобы представлять, с какими проблемами разработчики сталкиваются при масштабировании. Примеры написаны в стиле JavaScript, направленном на читаемость кода. Если вы понимаете код C, C#, C++ или Java, у вас не будет особых сложностей.

Структура издания

Книга состоит из двух частей и 19 глав. В каждой части описан некоторый фундаментальный навык, а затем исследуются другие связанные с ним навыки. Каждая часть завершается описанием принципов проектирования и архитектуры в контексте функционального программирования. В части I, начинающейся с главы 3, вводятся различия между действиями, вычислениями и данными. Часть II, начинающаяся с главы 10, знакомит читателя с идеей первоклассных значений.

- В главе 1 приводится общая информация о книге и основных концепциях функционального программирования.
- В главе 2 приведен краткий обзор возможностей, которые откроет перед вами использование этих навыков.

Часть I. Действия, вычисления и данные.

- Глава 3 открывает первую часть книги: в ней представлены практические навыки, которые позволят вам различать действия, вычисления и данные.
- Глава 4 показывает, как проводить рефакторинг кода в вычисления.

- Из главы 5 вы узнаете, как усовершенствовать действия в том случае, если они не могут быть преобразованы в вычисления посредством рефакторинга.
- В главе 6 представлен важный механизм неизменяемости — так называемое копирование при записи.
- В главе 7 описан дополняющий механизм неизменяемости, называемый защитным копированием.
- В главе 8 представлен способ организации кода в соответствии со смысловыми уровнями.
- Глава 9 помогает анализировать уровни в соответствии с принципами сопровождения кода, тестирования и повторного использования.

Часть II. Первоклассные абстракции.

- Глава 10 открывает вторую часть книги описанием концепции первоклассных значений.
- Глава 11 показывает, как наделить любую функцию суперспособностями.
- Из главы 12 вы узнаете, как создавать и использовать средства перебора массивов.
- Глава 13 помогает строить сложные вычисления из средств, описанных в главе 12.
- В главе 14 представлены функциональные средства для работы с вложенными данными, а также кратко затрагивается тема рекурсии.
- Глава 15 знакомит читателя с концепцией временных диаграмм как средства анализа выполнения вашего кода.
- Глава 16 показывает, как организовать безопасное совместное использование ресурсов между временными линиями без создания ошибок.
- Глава 17 показывает, как манипуляции с порядком и повторением действий могут использоваться для предотвращения ошибок.
- Глава 18 завершает часть II обсуждением двух архитектурных паттернов для построения сервисов в функциональном программировании.
- Глава 19 завершает книгу ретроспективным обзором и рекомендациями по дальнейшему обучению.

Книгу следует читать с самого начала и по порядку. Каждая глава строится на материале предыдущей. Не пропускайте упражнения. Думайте над упражнениями, пока не найдете ответ. Упражнения включены для того, чтобы помочь вам формулировать собственное мнение по субъективным вопросам. У упражнений «Ваш ход!» есть ответы. Они включены в книгу для того, чтобы дать вам возможность потренироваться и укрепить усвоенные навыки на реалистичных сценариях. Ничто не мешает вам прервать чтение в любой момент — еще никто не освоил ФП одним чтением книг. Если вы узнали что-то важное, отложите

книгу и дайте материалу закрепиться в памяти. Книга подождет до того момента, когда вы будете готовы к ней вернуться.

О примерах кода

В книге встречаются примеры кода. Код пишется на JavaScript в стиле, который на первое место ставит ясность. Я использую JavaScript вовсе не потому, чтобы показать вам, что на JavaScript можно заниматься функциональным программированием. Собственно, JavaScript не блещет в области ФП. Но именно потому, что в нем не реализована серьезная поддержка ФП, этот язык отлично подходит для обучения. Многие функциональные конструкции приходится строить самостоятельно, что позволит нам глубже понять их. Кроме того, вы будете больше ценить такие конструкции, предоставляемые языком (таким, как Haskell или Clojure).

Части текста, содержащие элементы кода, сразу видны. Для ссылок на переменные и другие фрагменты синтаксиса, встроенные в текст, используется моноширинный шрифт — так они выделяются в обычном тексте. Тот же шрифт используется в листингах. Иногда код выделяется цветом, чтобы показать изменения по сравнению с предыдущим шагом. Переменные верхнего уровня и имена функций выделяются **жирным** шрифтом. Я также использую подчеркивание для привлечения внимания к важным фрагментам кода.

Исходный код примеров этой книги можно загрузить по адресу https://www.manning.com/books/grokking-simplicity.

Другие ресурсы в интернете

Сетевых и автономных ресурсов, посвященных функциональному программированию, слишком много, чтобы перечислить их здесь. И никакие ресурсы не каноничны настолько, чтобы заслуживать особого упоминания. Я рекомендую поискать локальные группы по функциональному программированию. Мы лучше учимся тогда, когда делаем это вместе с другими.

Если вам нужны дополнительные материалы, относящиеся конкретно к этой книге, ссылки на ресурсы и другие материалы доступны по адресу https://grokkingsimplicity.com.

Об авторе

Эрик Норманд — опытный функциональный программист, преподаватель, докладчик и автор книг, посвященных функциональному программированию. Он родился в Новом Орлеане и начал программировать на Lisp в 2000 году. Эрик создает учебные материалы по Clojure на сайте PurelyFunctional.tv. Он также консультирует компании, желающие использовать функциональное программирование для более эффективного решения своих бизнес-задач. Вы можете ознакомиться с его докладами на международных конференциях по

программированию. Его статьи, доклады, обучающие материалы и информация о консультациях доступны на сайте LispCast.com.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Добро пожаловать в мир функционального мышления

1

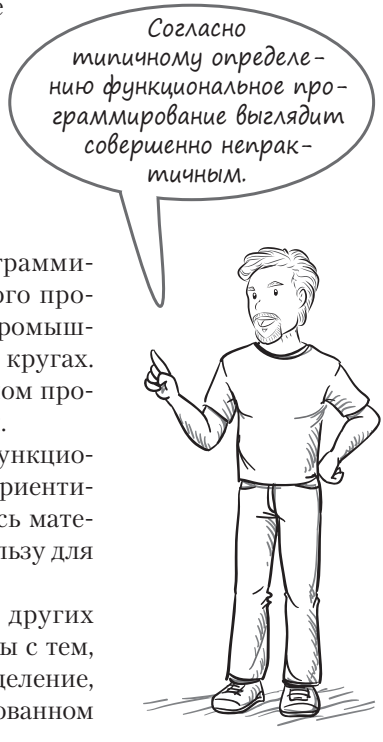
В этой главе вы

- ✓ Узнаете определение функционального мышления.
- ✓ Поймете, чем эта книга отличается от других книг по функциональному программированию.
- ✓ Познакомитесь с главной отличительной особенностью кода с точки зрения функциональных программистов.
- ✓ Решите, насколько эта книга подходит лично вам.

Мы начнем с того, что определим функциональное мышление и расскажем, как его главная отличительная особенность помогает программистам-практикам строить более качественные продукты. Кроме того, будет приведен обзор предстоящего путешествия в контексте двух главных идей, осознанных функциональными программистами.

Что такое функциональное программирование

Программисты постоянно спрашивают меня, что такое функциональное программирование (ФП) и для чего его лучше использовать. Мне трудно объяснить, для чего лучше подойдет ФП, потому что это парадигма общего назначения. Оно хорошо подходит для всего. А о том, в каких областях оно раскрывается по-настоящему, вы узнаете через несколько страниц.



Дать определение функциональному программированию тоже непросто. Мир функционального программирования огромен. Оно применяется в промышленном программировании и в академических кругах. Впрочем, большая часть книг о функциональном программировании написана именно теоретиками.

Эта книга отличается от типичных книг о функциональном программировании. Она безусловно ориентирована на промышленное применение ФП. Весь материал книги должен приносить практическую пользу для профессиональных программистов.

Возможно, вам встретятся определения из других источников, и важно понимать, как они связаны с тем, что мы обсуждаем в этой книге. Типичное определение, приведенное в Википедии, в слегка перефразированном виде выглядит так:

функциональное программирование (ФП), *сущ.*

1. Парадигма программирования, для которой характерно использование математических функций и предотвращение побочных эффектов.
2. Стиль программирования, использующий только чистые функции без побочных эффектов.

Подчеркнутым терминам необходимо дать определение

Разберем подчеркнутые термины.

К *побочным эффектам* относится все, что делает функция помимо возвращения значения: например, отправка электронной почты или изменение глобального состояния. Побочные эффекты могут создавать проблемы, потому что они происходят при каждом вызове вашей функции. Если вас интересует только возвращаемое значение, а не побочные эффекты, значит, в программе происходит что-то непреднамеренное.

Функциональные программисты обычно стараются избегать побочных эффектов, которые не являются абсолютно необходимыми.

Чистые функции — функции, которые зависят только от своих аргументов и не имеют побочных эффектов. С одними и теми же аргументами они всегда выдают одно возвращаемое значение. Можно называть их *математическими функциями*, чтобы отличить от функций как элемента в программировании. Функциональные программисты уделяют особое внимание использованию чистых функций, потому что они более просты для понимания и управления.

Определение подразумевает, что функциональные программисты полностью избегают побочных эффектов и используют только чистые функции. Тем не менее это не так. Функциональные программисты, работающие над реальными программами, используют побочные эффекты и функции с побочными эффектами.

Обычно программы запускаются как раз ради этого!

Типичные побочные эффекты

1. Отправка электронной почты.
2. Чтение файла.
3. Переключение светового индикатора.
4. Отправка веб-запроса.
5. Включение тормоза в машине.

Недостатки определения при практическом применении

Такое определение хорошо подойдет для академических кругов, но у него есть ряд недостатков с точки зрения программиста-практика. Еще раз присмотримся к определению:

функциональное программирование (ФП), *сущ.*

1. Парадигма программирования, для которой характерно использование математических функций и предотвращение побочных эффектов.
2. Стиль программирования, использующий только чистые функции без побочных эффектов.

Для наших целей такое определение создает три основные проблемы.

Проблема 1: ФП необходимы побочные эффекты

В определении сказано, что ФП стремится избегать побочных эффектов, но программы обычно запускаются как раз ради побочных эффектов. Какой прок от почтового клиента, который не отправляет электронную почту? Определение подразумевает, что мы должны полностью избегать их, тогда как на практике побочные эффекты используются там, где это необходимо.

Проблема 2: ФП неплохо справляется с побочными эффектами

Функциональные программисты знают, что побочные эффекты необходимы, хоть и проблематичны, поэтому существует значительное количество инструментов для работы с ними. Определение подразумевает, что мы используем только чистые функции. Напротив, мы достаточно часто применяем функции с побочными эффектами. Существует великое множество функциональных методов, которые упрощают их использование.

Загляни в словарь

Побочные эффекты — это любое поведение функции, кроме возврата значения.

Чистые функции зависят только от своих аргументов и не имеют побочных эффектов.

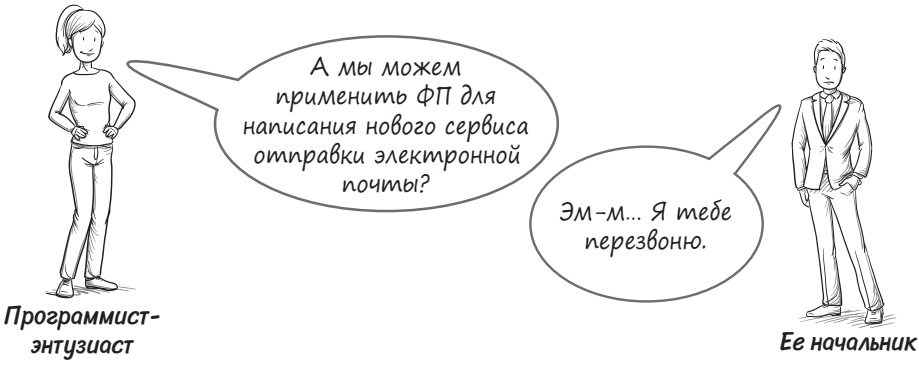
Проблема 3: Практичность ФП

Из определения может сложиться впечатление, что ФП имеет в основном математическую природу и для реальных программ оно непрактично. Тем не менее многие важные программные системы написаны с применением функционального программирования.

Определение особенно сильно сбивает с толку людей, которые знакомятся с ФП именно по нему. Представьте руководителя, действующего из лучших побуждений, который прочитал определение в Википедии.

Определение ФП сбивает с толку руководителей

Представьте, что Дженна, программист-энтузиаст, хочет использовать ФП для сервиса отправки электронной почты. Она знает, что ФП поможет спроектировать систему для улучшения общей надежности. Ее начальник не знает, что такое ФП, поэтому он находит определение в Википедии.



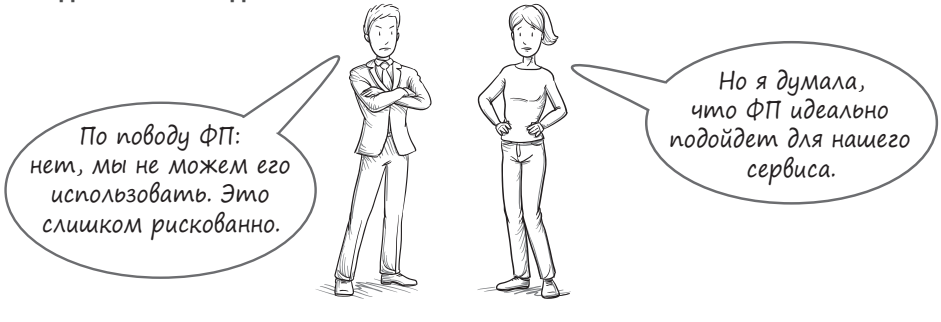
Начальник ищет «функциональное программирование» в Википедии:

. . . предотвращение побочных эффектов . . .

Он находит «побочный эффект» в Google. Типичные побочные эффекты:

- *отправка электронной почты*
- . . .

Позднее в тот же день...



Функциональное программирование рассматривается как совокупность навыков и концепций

Мы не будем использовать типичное определение в этой книге. ФП каждый понимает по-своему, это огромная область для научных исследований и практики.

Я говорил со многими функциональными программистами относительно того, какие свойства ФП кажутся им наиболее полезными. Книга «Грокаем функциональное мышление» стала квинтэссенцией навыков, логических умозаключений и перспектив функциональных программистов-практиков. В книгу были включены только самые практичные и эффективные идеи.

В книге вы не найдете данных последних исследований или эзотерических идей. Мы будем изучать только навыки и концепции, применяемые в наши дни. В ходе исследований я обнаружил, что самые важные концепции ФП применимы даже в объектно-ориентированном и процедурном коде, а также в разных языках программирования. Красота ФП по-настоящему проявляется в том, что ФП имеет дело с полезными, универсальными практиками программирования.

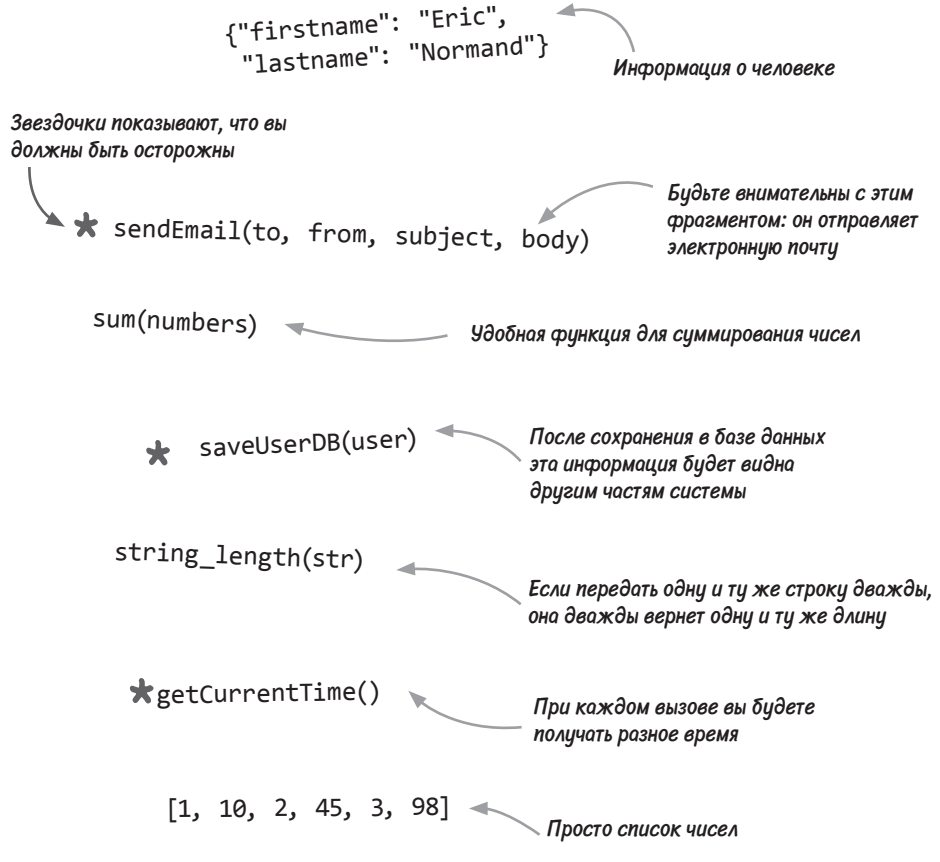
Взглянем на навык, важность которого не будет отрицать ни один функциональный программист: разграничение действий, вычислений и данных.

Действия, вычисления и данные

Когда функциональный программист смотрит на код, он немедленно классифицирует его на три категории:

1. Действия (actions — A).
2. Вычисления (calculations — C).
3. Данные (data — D).

Рассмотрим несколько примеров кода из существующей базы данных. Будьте особенно внимательны с фрагментами, помеченными звездочкой.

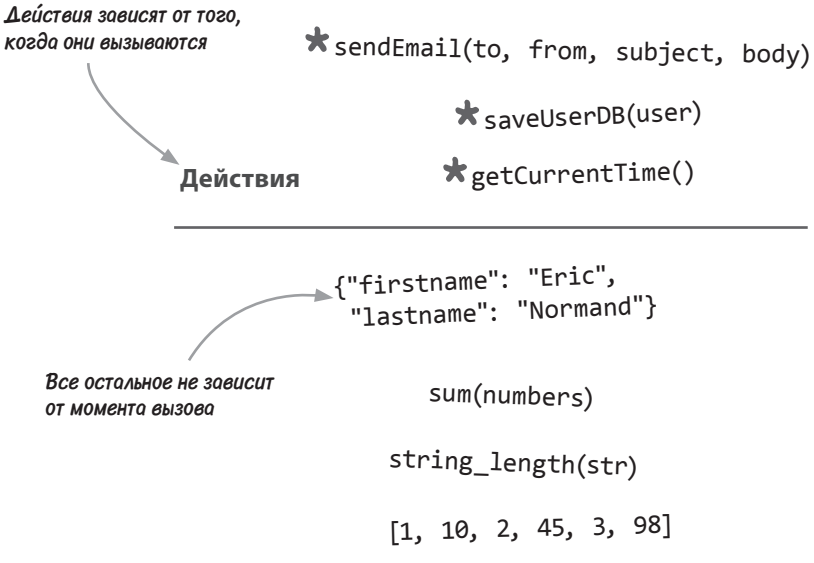


Функции со звездочкой требуют особого внимания, потому что они зависят от того, когда или сколько раз они вызываются. Например, важная электронная почта не должна отправляться дважды или отправляться ноль раз.

Фрагменты, помеченные звездочкой, относятся к *действиям*. Отделим их от остальных.

Функциональные программисты особо выделяют код, для которого важен момент вызова

Проведем линию и переместим все функции, зависящие от момента вызова, по одну сторону от линии:



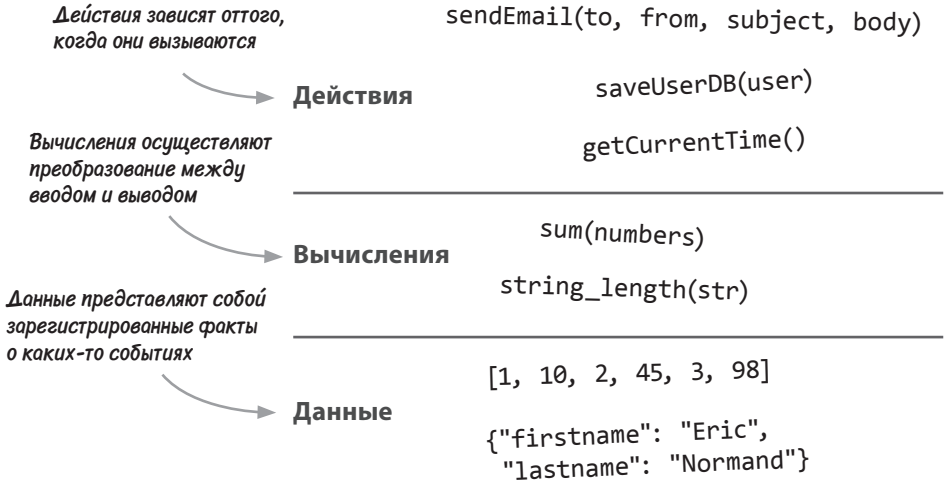
Это очень важный момент. Действия (все, что находится над линией) зависят от того, когда или сколько раз они вызываются. Они требуют особого внимания.

Однако с тем, что находится под линией, работать намного проще. Неважно, когда вы вызовете функцию `sum`, — она будет каждый раз вызывать правильный ответ. Неважно и то, сколько раз вы ее вызовете. Она не повлияет на остальные части программы или окружающий мир за пределами программы.

Однако существует еще одно различие: одна часть кода может выполняться, другая остается инертной. Проведем еще одну линию на следующей странице.

Функциональное программирование отличает инертные данные от работающего кода

Еще одна линия отделяет вычисления от данных. Ни вычисления, ни данные не зависят от того, сколько раз они будут использоваться. Отличие заключается в том, что вычисления могут выполняться, а данные выполняться не могут. Данные инертны и прозрачны. Вычисления непрозрачны в том смысле, что вы не знаете, что именно сделает вычисление до его запуска.



Различия между действиями, вычислениями и данными являются фундаментальными для ФП. Любой функциональный программист согласится с тем, что уметь различать их исключительно важно. Многие другие концепции и навыки в ФП строятся на базе этого навыка.

Важно подчеркнуть, что функциональные программисты не питают отвращения к использованию кода в любой из трех категорий, потому что все они важны. Однако при этом они понимают плюсы и минусы и пытаются выбрать оптимальный инструмент для своей работы. В общем случае они предпочитают данные вычислениям, а вычисления — действиям. Работать с данными проще всего.

Стоит еще раз подчеркнуть: **функциональные программисты видят эти категории каждый раз, когда они смотрят на любой код**. Это главная особенность точки зрения ФП. Эта классификация лежит в основе целого ряда навыков и концепций. Мы займемся их изучением в оставшихся главах части I.

Функциональные программисты предпочитают данные вычислениям, а вычисления — действиям.

Посмотрим, что эта классификация скажет о простом сервисе управления задачами.