MANNING

JavaScript

Луис Атенсио

Функциональное программирование на JavaScript

Как улучшить код JavaScript-программ

Функциональное
программирование
на JavaScript

Functional Programming in JavaScript

Luis ATENCIO

fl

MANNING

Manning Publications Co.
220 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Функциональное программирование на JavaScript

Луис Атенсио

Москва • Санкт-Петербург • Киев
2018

ББК 32.973.26-018.2.75
А92
УДК 681.3.07

Компьютерное издательство "Диалектика"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:
info@dialektika.com, http://^^.dialektika.com

Атенсио, Луис.

А92 Функциональное программирование на JavaScript: как улучшить код JavaScript-программ.: Пер. с англ. — СПб.: ООО “Альфа-книга", 2018.- 304 с.: ил. — Парал. тит. англ.

ISBN 978-5-9909445-8-9 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Луне Атенсио

Функциональное программирование на JavaScript: как у^^шить код JavaScript-программ

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. П,рдиенко*

Подписано в печать 25.10.2017. Формат 70x100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 24,51. Уч.-изд. л. 16,1
Тираж 500 экз. Заказ № 7526

Отпечатано в АО “Первая Образцовая типография"
Филиал “Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО “Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9909445-8-9 (рус.)
ISBN 978-1617-29282-8 (англ.)

Оглавление

Содержание

Моей замечательной жене Ане. Благодарю за безоговорочную поддержку и неисчерпаемый источник страстного увлечения и вдохновения в моей жизни.

Предисловие

Когда я учился в колледже и магистратуре, мой план занятий был направлен на объектно-ориентированное проектирование как единственную методологию планирования и построения архитектуры программных систем. И, как и многие разработчики, я начинал свою карьеру с написания объектно-ориентированного кода и построения по этому принципу целых систем.

На протяжении всей своей карьеры разработчика я изучал языки программирования и внимательно следил за их развитием не только потому, что мне хотелось узнать что-то новое и полезное, и но и потому, что меня интересовали проектные решения и основные подходы к программированию, принятию которых способствует каждый язык. Как новый язык, так и новая парадигма позволяет по-новому взглянуть на подход к решению задач разработки программного обеспечения. Несмотря на то что объектно-ориентированный подход продолжает оставаться основной методикой разработки программного обеспечения, изучение функционального программирования позволит открыть новые методики, которые можно применять как по отдельности, как и в сочетании с любой другой подходящей парадигмой.

Функциональному программированию уже немало лет, но меня оно привлекало мало. Мне приходилось слышать и читать о преимуществах языков Haskell, Lisp, Scheme, а позднее — Scala, Clojure и F#, в частности, их выразительности и высокой

производительности. И даже в языке Java, который традиционно считается многословным, теперь имеются средства функционального программирования, способствующие лаконичности прикладного кода. В конечном итоге мне пришлось обратить более пристальное внимание на возможности функционального программирования. И как оказалось, широко распространенный объектно-ориентированный язык JavaScript можно применять совершенно иначе — функционально, извлекая из него наибольшую пользу. Чтобы обнаружить это, мне потребовалось немало времени, и в этой книге мне хотелось бы ознакомить вас с функциональными возможностями JavaScript, чтобы вы больше не удивлялись, почему ваш кодJavaScript становится таким сложным.

Занимаясь разработкой, я постепенно научился применять принципы функционального программирования, чтобы создавать модульный, выразительный, надежный, легко доступный для понимания и простой для тестиро

вания прикладной код. И без сомнения, это совершенно изменило меня как разработчика программного обеспечения. Поэтому я решил каким-то образом поделиться своим опытом — например, в книге. Я, естественно, обратился к издательству Manning с предложением написать книгу по функциональному программированию на языке Dart. В то время я экспериментировал с языком Dart и подумал, что было бы неплохо воспользоваться его средствами вместе со своими знаниями и опытом функционального программирования, чтобы исследовать эту неизведанную область. Я написал свое предложение и через неделю был приглашен на собеседование в издательство Manning. В ходе собеседования выяснилось, что в издательстве ищут автора, готового написать книгу по функциональному программированию на JavaScript. А поскольку я был просто одержим языком JavaScript, то с большой охотой ухватился за такую возможность. Написанием этой книги я надеюсь помочь вам выработать в себе такие же, как и у меня, навыки искать новые направления в разработке программного обеспечения.

Благодарности

Написание книги — дело непростое. Чтобы выпустить в свет книгу, которую вы читаете, мне пришлось неустанно сотрудничать со многими людьми, обладающими разнообразными дарованиями.

Сотрудники издательства Manning проявили огромное внимание и оказали всестороннюю помощью, на которую вправе рассчитывать всякий автор, за что я искренне им благодарен. Без них эта книга не состоялась бы. Особая благодарность Марджан Бэйс (Maijan Bace) и Майку Стивенсу (Mike Stephens) — за то, что они поверили в мое предложение написать эту книгу и в меня как ее автора; Марине Майклс (Marina Michaels) — за то, что она дала мне путеводную нить, чтобы не заблудиться в лабиринте трудностей, возникавших при написании книги; Сьюзен Конант (Susan Conant) — за то, что она подгоняла меня и дала мне первые уроки в написании технической литературы; Берту Бэйтсу (Bert Bates) — за то, что он зажег во мне первые искры творческой инициативы и поделился своими замечательными мыслями, как следует обучать программированию; а также всем сотрудникам редакции и издательства Manning, включая Мэри Пирджис (Mary Piergies), Джанет Вэйл (Janet Vail), Кевина Салливана (Kevin Sullivan), Тиффани Тэйлор (Tiffany Taylor), Кати Теннант (Katie Tennant), Денниса Далинника (Dennis Dalinnik) и многих других скромных сотрудников издательства, незаметно делающих свое дело.

Огромной благодарности заслуживает замечательная группа технических рецензентов во главе с Александаром Драгосавлевичем (Dragosavljevic): Эми Тенг (Amy Teng), Эндрю Мередит (Andrew Meredith), Беки Хьюетт (Becky Huett), Дэниел Ламб (Daniel Lamb), Дэвид Баркол (David Barkol), Эд Грибел (Ed Griebel), Эфран Кобиси (Efran Cobisi), Эзра Симелофф (Ezra Simeloff), Джон Ши (John Shea), Кен Фукуяма (Ken Fukuyama), Питер Эдвардс (Peter Edwards), Субхазис Гош (Subhasis Ghosh), Тэннер Слейтон (Tanner Slayton), Торстен Шуцкус (Thorsten Szutzkus), Вильфредо Манрике (Wilfredo Manrique), Уильям Е. Уилер (William E. Wheeler), Йи Линг Лу (Yiling Lu), а также все одаренные участники форума. Они внесли свой вклад в выявление технических ошибок, погрешностей в терминологии и опечаток, а также немало дельных предложений по поводу материала книги. Каждый из них внимательно просмотрел рукопись книги и своими отзывами на форуме способствовал ее формированию в нечто, достойное издания.

Моя особая признательность Дину Иверсону (Dean Iverson), исполнявшему обязанности технического редактора, Дэниелу Ламбу — корректору рукописи, а также Брайану Ханафи (Brian Hanafee) — за его основательную и глубокую оценку всей книги. Лучших технических редакторов книги трудно найти.

И, наконец, хотя и не в последнюю очередь, благодарю свою жену, которая всегда меня поддерживает, а также мою семью, побуждающую меня становиться лучше с каждым днем и не нарекавшую на недостаточное внимание с моей стороны в процессе работы над книгой. Благодарю также своих коллег за приобретение ранних выпусков отдельных глав книги. Мне очень приятно работать с такими замечательными людьми.

Об этой книге

Обуздать сложность — нелегко, и вряд ли нам удастся сделать это полностью. Она всегда будет одной из неотъемлемых особенностей разработки программного обеспечения. Мне пришлось провести бесчисленное количество часов и затратить безмерное количество энергии на мозговой штурм, чтобы понять назначение конкретного фрагмента кода. А секрет кроется в том, чтобы контролировать сложность, не давая ей увеличиваться пропорционально размеру кодовой базы, и здесь на помощь может прийти функциональное программирование. Ныне приходится писать код на JavaScript больше, чем когда-либо прежде, переходя от создания небольших процедур обработки событий на стороне клиента к крупным клиентским архитектурам и законченным изоморфным (серверным и клиентским) приложениям на JavaScript. Функциональное программирование — это не средство, а способ мышления, который можно в равной степени применять в любой из этих сред.

Эта книга призвана научить вас применять методики функционального программирования в своем коде, используя стандарт ECMAScript 6 языка JavaScript. Ее материал излагается постепенно, охватывая теоретические и практические вопросы функционального программирования. А для подготовленных пользователей предоставляются дополнительные сведения, чтобы помочь им основательнее разобраться в некоторых более трудных для усвоения понятиях.

Структура книги

Эта книга состоит из восьми глав, разделенных на три части. Такая структура книги позволяет постепенно раскрыть ее материал, проводя читателей от фундаментальных стандартных блоков к более развитым и практическим примерам применения функционального программирования.

В части I, “Умение мыслить функционально”, дается общая картина функциональных свойств языка JavaScript. В ней также обсуждаются основные вопросы функционального применения JavaScript и поясняется, как научиться мыслить как программирующий функционально.

- В главе 1 представлены некоторые из основных понятий функционального программирования, более подробно поясняющихся в последующих главах и подготавливающих читателя к переходу на функциональное

мышление. В ней даются основы функционального программирования, включая чистые функции, побочные эффекты и декларативное программирование.

- В главе 2 задается необходимый уровень для читателей, обладающих начальным и промежуточным опытом программирования на JavaScript, а для более опытных читателей она служит лишь напоминанием о том, что им должно быть уже известно. Кроме того, в ней рассматриваются основные понятия функционального программирования, чтобы подготовить читателя к усвоению методик, обсуждаемых в части II.

В части II, "Погружаемся в функциональное программирование", главное внимание уделяется основным методиками функционального программирования, включая соединение функций в цепочки, карринг, композицию, монады и пр.

- В главе 3 представлены цепочки функций, а также исследуется методика написания программ в виде определенного сочетания рекурсивных функций с функциями высшего порядка, реализующими такие операции, как map, filter и reduce. Эти понятия рассматриваются на конкретных примерах, в которых применяется библиотека Lodash.js.
- В главе 4 рассматриваются распространенные методики карринга и композиции, повышающие модульность прикладного кода. Композиция служит связующим материалом, координирующим все решение на JavaScript, вырабатываемое с помощью функциональной библиотеки Ramda.js.
- В главе 5 углубленно обсуждаются более теоретические вопросы функционального программирования. Здесь функторы и монады всесторонне и постепенно рассматриваются в контексте обработки ошибок.

В части III, "Расширение функциональных навыков", раскрываются практические преимущества, которые дает функциональное программирование в разрешении реальных трудностей.

- В главе 6 выявляется присущая функциональным программам простота, с которой они поддаются модульному тестированию. Кроме того, в ней представлен строгий режим автоматизированного тестирования, называемый *тестированием на основе свойств.*
- В главе 7 рассматривается модель памяти, принятая в JavaScript и предназначенная для поддержки вычисления функций. В этой главе обсуждаются также методики, помогающие оптимизировать время выполнения функциональных приложений на JavaScript.
- В главе 8 представлены некоторые из основных трудностей, с которыми приходится часто сталкиваться разработчикам, когда они имеют дело с обработкой событий и асинхронным поведением их приложений на JavaScript. Здесь также обсуждаются изящные решения, которые может предложить функциональное программирование, чтобы сделать менее сложными существующие императивные решения, опираясь на соответствующую парадигму, называемую *реактивным программированием* и реализуемую с помощью библиотеки RxJS.

Кому адресована книга

Эта книга написана для разработчиков, имеющих хотя бы основные понятия об объектно-ориентированных программах и общее представление о тех трудностях, которые представляет разработка современных веб-приложений на JavaScript. Благодаря широкой распространенности языка JavaScript читатель может извлечь наибольшую пользу из этой книги, изучая основы функционального программирования и пользуясь знакомым синтаксисом вместо того, чтобы осваивать такой малораспространенный язык, как Haskell. (Эта книга вряд ли окажет помощь в освоении Haskell, поскольку у каждого языка свои особенности, разобраться в которых лучше, изучая его непосредственно.)

Эта книга поможет читателям, имеющим начальный и промежуточный опыт программирования, повысить свои навыки обращения с функциями высшего порядка, замыканиями, каррингом функций, композицией и новыми языковыми средствами JavaScript, внедренными в стандарт ES6, включая лямбда-выражения, итераторы, генераторы и обязательства. А опытные разработчики извлекут для себя пользу, изучая подробно рассматриваемые в этой книге монады и принципы реактивного программирования, что поможет им реализовать новаторские пути решения трудных задач написания управляемого событиями и асинхронного кода, в полной мере используя преимущества и возможности платформы JavaScript.

Как пользоваться этой книгой

Если у вас имеется лишь начальный или промежуточный опыт разработки приложений на JavaScript, а функциональное программирование для вас внове, начните чтение с главы 1. А если у вас достаточный опыт программирования на JavaScript, можете пропустить главу 1, бегло просмотреть главу 2 и сразу же перейти к главе 3, с которой начинается рассмотрение цепочек функций и общей концепции функционального проектирования.

Более опытные пользователи функциональных свойств JavaScript, как правило, хорошо разбираются в понятиях чистых функций, карринга и композиции. Поэтому они могут бегло просмотреть главу 4 и сразу же перейти к главе 5.

Примеры исходного кода

Исходный код примеров, приведенных в этой книге, написан на языке JavaScript версии ECMAScript 6 и может в равной степени выполняться как на стороне сервера (на платформе Node.js), так и на стороне клиента. В некоторых примерах демонстрируются интерфейсы API для ввода-вывода и браузерной модели DOM, хотя и без учета несовместимости браузеров. При этом предполагается, что у читателя имеется опыт взаимодействия с HTML-страницами и консолью на самом элементарном уровне. В этих примерах отсутствует код JavaScript, характерный для отдельных браузеров.

В примерах из этой книги интенсивно применяются такие функциональные библиотеки и каркасы JavaScript, как Lodash.js, Ramda.js и пр. Сведения об их установке и документации на них приведены в приложении к данной книге.

В тексте книги приводится немало листингов исходного кода, где методики функционального программирования сравниваются, там, где это уместно, с методиками императивного программирования. Исходный код всех примеров из этой книги можно

загрузить со страницы веб-сайта издельства Manning по адресу https://www.manning.com/books/functional-programming- in-javascript или из хранилища GitHub по адресу https://github.com/ luijar/functional-programming-js.

Условные обозначения

В тексте приняты следующие условные обозначения.

- *Курсивом* выделяются важные термины.
- Моноширинным шрифтом выделяется исходный код листингов, а также элементы и атрибуты разметки, имена методов, классов, функций и прочих средств программирования.
- Комментарии к исходному коду некоторых листингов указываются по обеим сторонам каждого листинга и поясняют важные понятия.

Об авторе

Луис Атенсио (Qluijar) работает штатным инженером-разработчиком программного обеспечения в компании Citrix Systems, г. Форт-Лодердейл, шт. Флорида. Он имеет степени бакалавра и магистра в области вычислительной техники и полностью занят разработкой и построением архитектуры приложений на платформах JavaScript, Java и PHP. Луис является активным участником местных собраний и конференций, на которых он часто выступает с докладами. Он ведет блог, посвященный разработке программного обеспечения, на странице по адресу http: //luisatencio.net, пишет статьи для журналов и сайта Dzone, а также является одним из авторов книги *RxJS in Action,* выход которой планируется в издательстве Manning в 2017 году.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`
WWW: `http://www.dialektika.com`

Наши почтовые адреса:
в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116 в Украине: 03150, Киев, а/я 152

Умение мыслить функционально

Вполне возможно, что весь ваш опыт написания профессиональных приложений сводится к программированию на объектно-ориентированном языке. Вам, вероятно, приходилось слышать или читать о функциональном программировании в других книгах, блогах, форумах и журналах, но вы, скорее всего, никогда не писали функциональный код. И вы в этом не одиноки, что не удивительно, поскольку и я большую часть своей карьеры разработчика провел в объектно-ориентированной среде. Написать функциональный код не так трудно, как научиться мыслить функционально, избавляясь от старых привычек. Основная цель части I — заложить прочное основание и подготовить ваше мышление к восприятию методик функционального программирования, представленных в частях II и III.

В главе 1 поясняется, что собой представляет функциональное программирование, а также тип мышления, который требуется для его восприятия. В ней также представлен ряд наиболее важных методик, основанных на чистых функциях, неизменяемости, побочных эффектах и ссылочной прозрачности. Эти понятия образуют основную структуру всего функционального кода и помогут вам легче перейти к функциональному программированию. Они служат также руководящими принципами, подготавливающими многие проектные решения, рассматриваемые в последующих главах.

В главе 2 дается первое представление о JavaScript как о языке функционального программирования. В силу повсеместности и широкой распространенности этого языка он идеально подходит для изучения функционального программирования. Если у вас недостаточно опыта разработки приложений на JavaScript, материал этой главы позволит вам быстро усвоить все, что требуется знать о функциональных свойствах языка JavaScript, включая функции высшего порядка, замыкания и правила соблюдения области видимости.

Основы функционального программирования

В этой главе...

- Приобретение навыков мыслить функциональными категориями
- Пояснение назначения и оснований для функционального программирования
- Представление принципов неизменяемости и чистых функций
- Рассмотрение методик функционального программирования и их влияния на общее проектное решение

Объектно-ориентированное программирование делает код понятным, инкапсулируя перемещенные части кода.
А функциональное программирование делает код понятным, сводя перемещенные части кода к минимуму.

Майкл Физерс (Michael Feathers; Twitter)

Если вы читаете эту книгу, то, скорее всего, разрабатываете приложения на JavaScript, имея знания и практический опыт объектно-ориентированного или структурированного проектирования, но теперь вы заинтересовались функциональным программированием. Вполне возможно, что вы уже пытались изучать его прежде, но так и не сумели удачно применить его на работе или в личных проектах. В любом случае ваша главная цель — развить свои навыки разработки программного обеспечения и повысить качество своего кода. И эта книга поможет вам достичь поставленной цели.

Быстрые темпы развития веб-платформ, браузеров, а самое главное — рост требований конечных пользователей оказали глубокое влияние на современ

ный порядок разработки веб-приложений. В частности, пользователи требуют, чтобы веб-приложения были в большей степени похожи на приложения для настольных систем и мобильных устройств с богатым набором оперативно реагирующих виджетов. И естественно, что эти требования вынуждают разработчиков, программирующих на JavaScript, осмысливать более обширное пространство решений и внедрять подходящие парадигмы программирования и нормы передовой практики разработки, дающие наилучшие из возможных решения.

Разработчики обычно тяготеют к тем каркасам и библиотекам, которые помогают им создавать расширяемые и ясные архитектуры приложений. Тем не менее сложность кодовой базы по-прежнему выходит из-под контроля, и разработчикам приходится пересматривать основные принципы разработки прикладного кода. Кроме того, Всемирная паутина (или просто веб) сейчас совершенно иная, чем прежде, для разработчиков веб-приложений на JavaScript, поскольку теперь они могут делать многое из того, что было технически невозможно раньше. В частности, они могут разрабатывать крупные серверные приложения на платформе Node.js или переносить большую часть бизнес-логики на сторону клиента, оставляя сервер "тонким". Но в любом случае разработчикам приходится иметь дело с технологией хранения информации, порождать асинхронные процессы, обрабатывать события и делать многое другое.

Объектно-ориентированное программирование (ООП) помогает решить лишь часть задачи, но в силу динамического характера языка JavaScript с множеством общих состояний не приходится долго ждать, чтобы накопилась достаточная сложность, делающая прикладной код громоздким и неудобным для сопровождения. Безусловно, ООП служит путеводителем в нужном направлении, но этого недостаточно. Вам, вероятно, приходилось не раз слышать за последние годы термин *реактивное программирование.* Эта парадигма программирования призвана облегчить обработку потоков данных и распространение изменений. Она крайне важна в JavaScript , когда приходится иметь дело с асинхронным или запускаемым событиями кодом. Таким образом, требуется парадигма программирования, побуждающая тщательно анализировать данные и взаимодействующие с ними функции. Обдумывая проектное решение приложения, необходимо задать себе ряд следующих вопросов, касающихся принципов проектирования.

- **Расширяемость.** Приходится ли постоянно реорганизовывать код для поддержки дополнительных функциональных возможностей?
- **Простота модуляризации.** Если внести изменения в один файл, то повлияет ли это на другой файл?
- **Повторное использование.** Много ли имеется дублирующегося кода?
- **Тестируемость.** Есть ли стремление к модульному тестированию функций?
- **Простота понимания.** Насколько код неструктурирован и труден для сопровождения?

Если вы ответите утвердительно или не знаете, что ответить на эти вопросы, значит, вы выбрали нужную книгу в качестве руководства на пути к повышению производительности и вам требуется парадигма функционального программирования (ФП). И хотя ФП основывается на довольно простых принципах, оно требует перехода к совершенно иному

осмыслению решаемых задач. ФП — это не какое-то новое инструментальное средство или интерфейс API, а совершенно иной подход к решению поставленных задач, который станет интуитивным, как только будут уяснены основные его принципы.

В этой главе дается определение ФП и поясняется, чем и почему оно полезно и важно для вас. В ней представлены базовые принципы неизменяемости и чистых функций, а также рассматриваются методики ФП и их влияние на подход к проектированию программ. Эти методики позволяют легко усвоить реактивное программирование и применять его для решения сложных задач BjavaScript. Но прежде необходимо пояснить, почему так важно научиться мыслить функционально и как это поможет справляться со сложностями программ HaJavaScript.

1.1. Чем может помочь функциональное программирование

Изучение функционального программирования стало как нельзя более актуальным. В сообществе разработчиков и основных компаниях, выпускающих программное обеспечение, начинают осознавать преимущества применения методик ФП для усиления их коммерческих приложений. В настоящее время в большинстве основных языков программирования (Scala, Java 8, F#, Python, JavaScript и многих других) имеется поддержка ФП (встроенная или на основе интерфейса API). Следовательно, навыки ФП весьма востребованы сейчас и будут оставаться таковыми в ближайшие годы.

В контексте JavaScript тип мышления ФП может быть использован для формирования невероятно выразительного характера языка и оказания помощи в написании ясного, модульного, тестируемого и лаконичного кода, с которым можно работать более продуктивно. Многие годы разработчики пренебрегали тем обстоятельством, что HaJavaScript можно программировать более эффективно в функциональном стиле. Такое пренебрежение отчасти объясняется общим недопониманием возможностей языка JavaScript, а также отсутствием в нем собственных конструкций для надлежащего управления состоянием. Ведь эта динамическая платформа накладывает бремя ответственности за управление таким состоянием на разработчиков — главных виновников программных ошибок, вносимых в их приложения. И если это может оказаться вполне допустимым для небольших сценариев, то по мере разрастания кодовой базы контролировать ее становится все труднее. В известной степени ФП ограждает вас от самого JavaScript. Подробнее этом речь пойдет в главе 2.

Написание функционального кода HaJavaScript позволяет разрешить большинство подобных вопросов. Используя ряд проверенных методик и норм практики, основанных на чистых функциях, можно написать код, который нетрудно понять в свете постоянно растущей сложности. Программировать на JavaScript функционально выгодно вдвойне, поскольку вы не только повышаете качество своего приложения в целом, но и приобретаете дополнительные профессиональные навыки и лучше понимаете сам язык JavaScript.

Как упоминалось выше, ФП — это не каркас или инструментальное средство, а способ написания кода, поэтому функциональное мышление радикально отличается от мышления объектно-ориентированными категориями. Как стать программирующим функционально и как начать мыслить функционально? ФП становится интуитивным, стоит лишь уяснить его

сущность. Самое трудное — избавиться от старых привычек, что может стать коренным переломом для тех, у кого имеется прежний опыт ООП. Но прежде чем научиться мыслить функционально, нужно уяснить, что собой представляет ФП.

1.2. Сущность функционального программирования

Проще говоря, функциональное программирование — это стиль разработки программного обеспечения, где основной акцент делается на функции. Но вы можете возразить, что и так пользуетесь функциями в повседневной практике. В чем же тогда различие? Как упоминалось выше, ФП требует несколько иного типа мышления, когда приходится решать поставленные задачи. Для этого недостаточно применять функции, чтобы добиться нужного результата. А цель состоит в том, чтобы *абстрагировать* потоки управления и операции над данными с помощью функций и тем самым *исключить* побочные эффекты и *сократить* изменение состояния в приложении. Очевидно, что такого пояснения явно недостаточно. Ведь нужно еще пояснить каждый из этих терминов, что и будет сделано далее, поскольку именно на них опирается изложение материала всей этой книги.

Как правило, пояснение основ ФП в литературе начинается с примера вычисления чисел Фибоначчи, но мы начнем с простой программы на JavaScript, отображающей текст на HTML-странице. И для этой цели как нельзя лучше подойдет старое доброе приветствие "Hello World" (Здравствуй, мир):

```
document.querySelector('#msg').innerHTML = '<hl>Hello World</hl>';
```

Примечание

Как уже не раз упоминалось ранее, функциональное программирование — это не конкретное инструментальное средство, а способ написания кода, который можно применять для разработки как клиентских (на основе браузеров), так и серверных (на платформе Node.js) приложений. Открыть окно браузера и ввести в нем какой-нибудь код — это простейший способ запустить на выполнение сценарий JavaScript. И этого должно быть достаточно для выполнения примеров исходного кода, приведенных в данной книге.

Эта программа проста, но поскольку все в ней жестко закодировано, то с ее помощью нельзя отображать сообщения в динамическом режиме. Допустим, требуется изменить форматирование, содержимое, а возможно, и целевой элемент разметки. Для этого придется переписать все приведенное выше выражение. Если же потребуется заключить весь этот код в тело функции и сделать точки изменения параметрами, такой код можно написать один раз, чтобы использовать его в любой конфигурации, как показано ниже.

```
function printMessage(elementId, format, message) {
    document.queryselector('#${elementId}').innerHTML =
            '<${format}>${message}</${format}>';
}

printMessage('msg', 'hl','Hello World');
```

Улучшение вполне очевидно. Тем не менее этот фрагмент кода еще не вполне пригоден для повторного использования. Допустим, сообщение требуется вывести в файл, а не на HTML-страницу. С этой целью придется перенести простой мыслительный процесс создания параметризированных функций на совсем другой уровень, где параметры могут быть не только скалярными величинами, но и функциями, предоставляющими дополнительные функциональные возможности. Функциональное программирование в какой-то степени похоже на преувеличенное внимание к применению функций, поскольку в этом случае преследуется единственная цель — вычислить и объединить вместе многие функции, чтобы добиться более совершенного поведения. Забегая несколько вперед, рассмотрим приведенную ниже ту же самую программу, написанную с применением функционального подхода.

Листинг 1.1. Функциональная версия программы printMessage

```
var printMessage = run(addToDom('msg'), hl, echo);

printMessage('Hello World');
```

Без сомнения, эта версия рассматриваемой здесь программы радикально отличается от ее первоначальной версии. Прежде всего, параметр hl теперь является не скалярной величиной, а функцией подобно addToDom и echo. На первый взгляд, это похоже на создание одной функции из других, более мелких.

И для такого безрассудства имеется веское основание. В листинге 1.1 зафиксирован сначала процесс декомпозиции (т.е. разложения) программы на более мелкие, пригодные для повторного использования, надежные и понятные части, а затем их объединения для формирования всей программы, которую легче осмыслить в целом. Этому основополагающему принципу следует всякая функциональная программа. А в рассматриваемой здесь программе для последовательного вызова ряда функций addToDom (), hl () и echo () применяется

некая волшебная функция run (), которая более подробно поясняется далее[1]. По существу она связывает внутренним образом каждую вызываемую функцию в цепочку, где значение, возвращаемое из предыдущей функции, передается в качестве параметра следующей функции. В данном случае символьная строка "Hello World", возвращаемая из функции echo (), передается функции hl (), а результат разметки в конечном итоге передается функции addToDom ().

Почему же функциональное решение выглядит именно так, а не иначе? По существу, его лучше рассматривать как параметризацию прикладного кода, которую можно легко изменить, ничего не нарушая, подобно коррекции начальных условий алгоритма. Заложив столь прочное основание, теперь можно без особого труда расширить возможности программы printMessage, чтобы повторить сообщение дважды, воспользоваться заголовком h2 и организовать вывод на консоль вместо модели DOM. И для этого не придется переписывать ничего из внутренней логики работы данной программы, как показано ниже.

Листинг 1.2. Расширенная версия программы printMessage

```
var printMessage = run(console.log, repeat(2), h2, echo);

printMessage('Get Functional');
```

Такой внешне отличающийся подход к программированию неслучаен. Сравнивая функциональное и нефункциональное решение, вы, возможно, обратили внимание на радикальное отличие и в стиле программирования. В обоих решениях выводится одинаковый результат, хотя выглядят они совсем по- разному. И это объясняется декларативным характером разработки, который присущ ФП. Чтобы полностью усвоить ФП, нужно сначала изучить следующие принципы и понятия, на которых оно основывается.

- Декларативное программирование.
- Чистые функции.
- Ссылочная прозрачность.
- Неизменяемость.

1.2.1. Декларативный характер функционального программирования

Функциональное программирование действует по принципам *декларативного* программирования, где ряд операций выражается таким образом, чтобы не обнаруживать, как они реализованы и как через них проходят потоки данных. Но в настоящее время более распространены *императивные VUWL процедурные модели,* которые поддерживаются в большинстве языков структурного и объектно-ориентированного программирования, таких как Java, C#, C++ и пр. В императивной модели программирования компьютерная программа рассматривается лишь как следующий сверху вниз ряд операторов, изменяющих состояние системы с целью вычислить результат.

Рассмотрим простой пример императивного программирования. Допустим, требуется

[1] **Более подробные сведения о функции** `run ()` **см. по адресу** `https: / /gist. github. com/luijar/ce6b96fl3e31cbl53093#file-chOl-magic-run-j s.`

возвести в квадрат все числа в массиве. Соответствующая императивная программа выглядит следующим образом:

```
var array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] ; for(let i = 0; i < array.length;
i++) {
    array[i] = Math.pow(array[i], 2);
}
array; //-> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

При императивном программировании компьютер получает подробные инструкции, *как* выполнить определенную задачу (в данном случае применить формулу возведения в квадрат к каждому числу на отдельном шаге цикла). Это самый распространенный способ написания подобного кода и, вероятнее всего, самый первый подход к решению данной задачи.

А при декларативном программировании описание программы отделяется от вычисления. При этом основное внимание уделяется использованию выражений для описания логики программы, не обязательно указывая поток ее управления или изменение состояния. Характерным примером декларативного программирования служат операторы SQL. В частности, запросы SQL состоят из операторов, описывающих результат выполнения запроса, абстрагируясь от внутреннего механизма извлечения данных. В главе 3 будет продемонстрирован пример применения SQL-подобного наложения на функциональный код с целью придать смыл как приложению, так и проходящим через него данным.

Чтобы перейти к функциональному подходу для решения той же самой задачи, достаточно позаботиться о применении правильного поведения к каждому элементу, уступив управление циклом другим частям системы. В частности, большую часть трудоемкой работы можно поручить методу Array. map ():

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9].map(
    function (num) {
        return Math, pow (num, 2);
    }).
```

Методу map () передается в качестве параметра функция, возводящая каждое число в квадрат

```
И -> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

По сравнению с предыдущим примером данный код освобождает от ответственности за надлежащее управление счетчиком цикла и доступом к массиву по индексу. Проще говоря, чем больше кода, тем больше в нем мест для возникновения программных ошибок. Кроме того, стандартные циклы не подлежат повторному использованию, если только они не абстрагированы с помощью функций. Именно это мы собираемся сделать далее. Так, в главе 3 демонстрируется, как исключить организуемые вручную циклы из прикладного кода в пользу функций первого класса и высшего порядка, которые реализуют операции map, reduce, filter и которым передаются другие функции в качестве параметров, чтобы сделать прикладной код в большей степени пригодным для повторного использования, расширяемым и декларативным. Именно это и было сделано с помощью волшебной функции run () в листингах 1.1 и 1.2.

Абстрагирование циклов с помощью функций позволяет с выгодой пользоваться *лямбда-выражениями* или стрелочными *функциями,* внедренным в стандарт ES6 языка

JavaScript. В частности, лямбда-выражения служат лаконичной альтернативой анонимным функциям и могут быть переданы функции в качестве аргумента, чтобы писать меньше кода:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9].map(num => Math.pow(num, 2));

// -> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Преобразование обозначения лямбда-выражения в обозначение обычной функции

Лямбда-выражения дают громадное синтаксическое преимущество по сравнению с обозначениями обычных функций, поскольку они сводят структуру вызова функции лишь к самым важным частям. Так, следующее лямбда-выражение:

```
num => Math.pow(num, 2)
```

равнозначно функции

```
function(num) {
    return Math.pow(num, 2);
}
```

А зачем вообще исключать циклы из прикладного кода? Цикл является императивной управляющей структурой, которая с трудом поддается повторному использованию и встраиванию в другие операции. Кроме того, он подразумевает код, который постоянно изменяется или модифицируется, реагируя на новые шаги цикла. Как будет показано далее, в функциональных программах нужно стремиться к *отсутствию сохранения состояния* и *неизменяемости* в как можно большей степени. Все дело в том, что если код не зависит от состояния программы, то ничто не может нарушить его работу в глобальном масштабе. Для достижения этой цели используются функции, в которых исключаются побочные эффекты и изменения состояния и поэтому называемые *чистыми*.

1.2.2. Чистые функции и трудности, связанные с побочными эффектами

Функциональное программирование основывается на той предпосылке, что неизменяемые программы строятся из чистых функций как стандартных блоков. Чистая функция обладает следующими свойствами.

- Зависит только от предоставляемых входных данных, а не от любого скрытого или внешнего состояния, которое может измениться во время ее вычисления или в промежутках между ее вызовами.
- Не вносит изменения за пределами области своей видимости, например, не изменяет глобальный объект или параметр, передаваемый по ссылке.

Нетрудно догадаться, что любая функция, не удовлетворяющая этим требованиям, считается “нечистой”. Программирование с учетом неизменяемости может показаться на первый взгляд необычным. Ведь весь смысл императивного проектирования, к которому так привыкли разработчики, состоит в том, чтобы заявить, что переменные подлежат

модификации от одного оператора к другому, поскольку они все-таки “переменные”. Вносить изменения в переменные — вполне естественно для разработчиков. Рассмотрим в качестве примера следующую функцию: `var counter = 0; function increment() { return ++counter; }`

Эта функция является “нечистой” потому, что она читает или модифицирует внешнюю переменную counter, которая не является локальной в области видимости данной функции. В общем, можно сказать, что у функции имеются побочные эффекты, если в ней читаются данные из внешних источников или записываются данные во внешние источники (рис. 1.1). Другим примером тому служит распространенная функция Date. now (). Очевидно, что выводимый ею результат непредсказуем и непостоянен, так как всегда зависит от постоянно изменяющегося фактора времени.

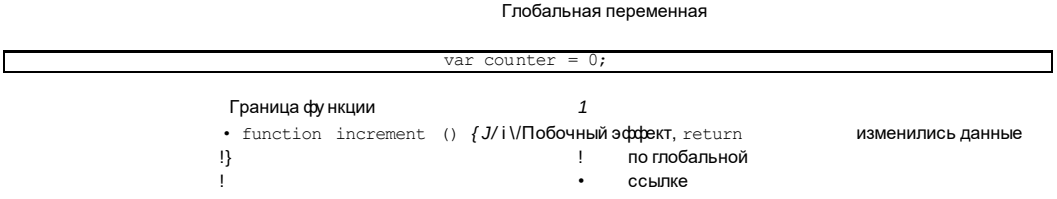


Рис. 1.1. Функция increment () вызывает побочные эффекты от чтения или модификации внешней переменной counter. Результат ее выполнения непредсказуем, поскольку переменная counter может измениться в любой момент в промежутках между вызовами данной функции

В данном случае переменная counter доступна через неявную глобальную переменную (в браузерном сценарии это объект window). Еще один типичный побочный эффект возникает при доступе к данным экземпляра по ключевому слову this. Ссылка this ведет себя в JavaScript иначе, чем в других языках программирования, поскольку она определяет динамический контекст функции. А это зачастую приводит к появлению кода, который трудно понять, чего еле-

дует избегать при всякой возможности. Мы еще вернемся к этому вопросу в следующей главе. Побочные эффекты могут возникать во многих случаях, включая следующие.

- Глобальное изменение переменной, свойства или структуры данных.
- Изменение первоначального значения аргумента функции.
- Обработка вводимых пользователем данных.
- Генерирование исключения, если только оно не перехватывается в той же самой функции.
- Вывод сообщений на экран или в журнал регистрации.
- Запрос HTML-документов, браузерных cookie-файлов или баз данных.

Если в программе нельзя создавать и модифицировать объекты или выводить информацию на консоль, то какая практическая польза от такой программы? Безусловно, чистыми функциями трудно пользоваться там, где преобладает динамический режим и модификация. Но на практике функциональное программирование накладывает ограничение не на *все* изменения состояния. Оно лишь предоставляет каркас, помогающий управлять ими и сокращать их, позволяя в то же время отделять "чистое" от "нечистого". "Нечистый" код производит *внешне видимые* побочные эффекты вроде перечисленных выше, и в этой книге рассматриваются способы обращения с ними.

Чтобы обсуждать эти вопросы более конкретно, допустим, что вы работаете разработчиком в команде, реализующей приложение для обработки данных учащихся школы. В листинге 1.3 демонстрируется небольшая императивная программа, выполняющая поиск записи об учащемся по его номеру социального страхования и отображающая ее в окне браузера (применение браузера в данном примере несущественно; ведь с тем же успехом запись можно вывести на консоль, в базу данных или файл). Мы будем еще не раз обращаться к данной программе на протяжении всей книги, расширяя ее как типичный пример реального сценария, в котором возникают побочные эффекты при взаимодействии с внешним хранилищем локальных объектов (вроде массива объектов) и выполнении ввода-вывода на определенном уровне.

Листинг 1.3. Императивная функция showstudent () с побочными эффектами

```
                                                  Получить доступ к хранилищу объектов для поиска учащегося по его
    номеру социального страхования. Будем пока что считать эту операцию синхронной. О том, как обращаться с асинхронным
function showStudent(ssn) { war      кодом, речь пойдет далее в книге
    student = db.find(ssn);
    if(student !== null) {                                          Обратиться к внеш-
        document.queryselector('#${elementId}').innerHTML           ней функции для
            '${student.ssn},                                        чтения элемента по
             ${student.firstname},                                  идентификатору
             ${student.lastname}';
    } else {
      throw new Error ('Student not found!'); ◄---- iСгенерировать исключение для
    j                                                  > недействительного учащегося
}                                                  |Выполнить эту программу для студента с номером
                                                   *социального страхования 444-44-4444 и вывести showStudent ('
444-44-4444 ' ) ;                                  на веб-странице подробные сведения об учащемся
```

Проанализируем исходный код данной программы. По существу это функция, выявляющая следующий ряд побочных эффектов, распространяющихся за пределы области ее видимости.

- Эта функция взаимодействует с внешней переменной (db) для доступа к данным, поскольку в ее сигнатуре внешняя переменная не объявлена как параметр. Но в любой момент ссылка на внешнюю переменную может стать пустой (null) или измениться в промежутке между вызовами функции, что может привести к совершенно другим результатам или нарушению целостности программы.
- Глобальная переменная elementId может измениться в любой момент, выйдя из под контроля.
- Элементы HTML-разметки модифицируются непосредственно. А сам HTML-документ (его модель DOM) является изменяемым, общим, глобальным ресурсом.
- Эта функция может сгенерировать исключение, если учащийся не найден, что может привести к сворачиванию стека вызовов и аварийному завершению программы.

Функция из листинга 1.3 зависит от внешних ресурсов, что делает ее исходный код негибким, неудобным для работы и трудным для тестирования. А "чистые" функции имеют как часть своих сигнатур ясные контракты, четко описывающие все формальные параметры функции (т.е. ряд входных данных), что упрощает понимание и применение таких функций.

А теперь попробуем сменить стиль программирования на функциональный, воспользовавшись в реальном сценарии теми уроками, которые были извлечены из примера программы printMessage. По мере усвоения стиля функционального программирования на протяжении данной книги у вас будет возможность совершенствовать эту реализацию рассматриваемого здесь сценария с помощью новых методик. А в настоящий момент в него можно внести два простых усовершенствования.

- Разделить длинную функцию на более короткие функции с отдельным назначением.
- Сократить количество побочных эффектов, явно определив все аргументы, которые требуются функциям для выполнения их заданий.

Начнем с отделения действий по извлечению записи об учащемся от ее отображения на экране. Разумеется, побочные эффекты от взаимодействия с внешней системой хранения и моделью DOM неизбежны, но их можно, по

крайней мере, сделать более контролируемыми, обособив от основной логики. С этой целью можно воспользоваться распространенной в ФП методикой, называемой *каррингом.* С помощью карринга можно частично задать некоторые аргументы функции, чтобы свести их к одному. Как демонстрируется в листинге 1.4, применив функцию curry (), можно свести find () и append () к унарным функциям, которые нетрудно объединить посредством функции run ().

Листинг 1.4. Декомпозиция программы showStudent

```
const find = curry((db, id) => {
    let obj = db. find (id);
    {
      throw new Error('Object not found!'); }
    return obj;
}) •

const csv = student =>
    '${student.ssn}, ${student.firstname},
${student.lastname};

const append = curry((selector, info) => {
    document.queryselector(selector).innerHTML = info;
});

var showStudent =
run(
    append (' #student-info '),
    find (db) ) ;
showStudent('444-44-4444');
```

Для работы функции поиска find () т^буется ссылка на хранилищ объектов и идентификатор учащгося

Преобразовать объект со сведениями об учащемся в разделяемый запятыми список значений

Для отображения подробных сведений об учащемся на странице требуется идентификатор элемента разметки и данные об учащемся

Задать в этой части идентификатор элемента HTML-разметки, чтобы воспользоваться им в функции

Задать в этой части объект доступа к данным, указывающий на таблицу учащихся

Несмотря на то что рассматриваемая здесь программа была усовершенствована незначительно, в ней начинают проявляться следующие немалые преимущества.

- Программа стала намного более гибкой, поскольку теперь она состоит из трех повторно используемых компонентов.
- Повторное использование мелкоструктурных функций направлено на повышение производительности, поскольку оно позволяет значительно сократить объем кода, требующий активного сопровождения.
- Придерживаясь декларативного стиля программирования, дающего ясное представление о стадиях, выполняемых данной программной на верхнем уровне, можно существенно повысить удобочитаемость исходного кода.
- Еще важнее, что взаимодействие с объектами HTML-разметки переносится в отдельную функцию, полностью отделяя "чистое" поведение от "нечистого". Более подробное пояснение карринга и управление "чистыми" и "нечистыми" частями программ приведено в главе 4.

Данная программа все еще имеет ряд недоработок, которые требуется устранить. Тем не менее она станет в меньшей степени подверженной изменению внешних условий благодаря сокращению побочных эффектов. Внимательно проанализировав функцию find (), можно заметить, что в ней имеется оператор ветвления по условию проверки на пустое значение null, которое может вызвать исключение. По целому ряду рассматриваемых далее причин целесообразно гарантировать возврат из функции согласованного значения, благодаря чему результат ее выполнения становится постоянным и предсказуемым. Данное свойство "чистых" функций называется *ссьиочной прозрачностью.*

1.2.3. Ссылочная прозрачность и подстановочность

Ссылочная прозрачность является более формальным способом определения "чистой" функции. "*Чистота*[9] в этом смысле означает существование "чистого" соответствия аргументов функции ее возвращаемому значению. Так, если функция постоянно возвращает один и тот же результат по тем же самым входным данным, то она считается *ссьиючно-прозрачной.* Например, рассмотренная ранее функция increment (), сохраняющая состояние, не является ссылочно-прозрачной, поскольку возвращаемое ею значение сильно зависит от внешней переменной counter, как еще раз показано ниже.

```
var counter = 0;

function increment() { return ++counter;
}
```

Чтобы сделать эту функцию ссылочно-прозрачной, из нее придется исключить зависимое состояние (т.е. внешнюю переменную), превратив его в формальный параметр, явно определяемый в сигнатуре данной функции. Для этого ее можно преобразовать в форму лямбда-выражения следующим образом:

```
var increment = counter => counter + 1;
```

Теперь данная функция устойчива и всегда возвращает один и тот же результат, если она получает те же самые входные данные. В противном случае на значение, возвращаемое функцией, будет оказывать влияние какой-нибудь внешний фактор.

Данное свойство желательно в функциях потому, что оно упрощает не только тестирование кода, но и *понимание всей* программы в *целом.* Понятие ссылочной прозрачности, иначе называемой *эквациональной корректностью,* унаследовано из математики, но функции в языках программирования ведут себя совсем иначе, чем математические функции, и поэтому достижение ссылочной

прозрачности полностью зависит от программирующего. С помощью все той же волшебной функции run () на рис. 1.2 для сравнения показано применение функциональной и императивной версий функции increment ().

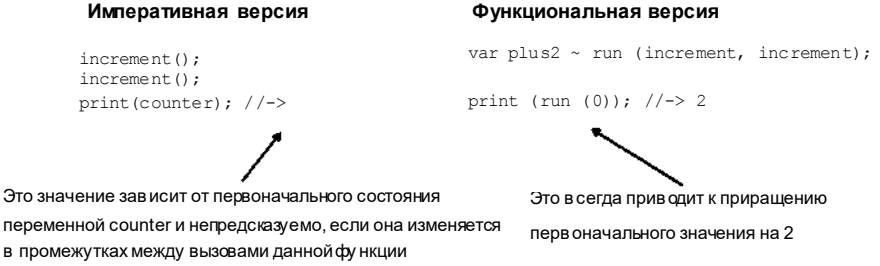


Рис. 1.2. Сравнение функциональной и императивной версий функции increment (). Результат выполнения императивной версии непредсказуем и может быть непостоянным, поскольку внешняя переменна counter способна измениться в любой момент, нарушив результат последующих вызовов данной функции. А ссылочно-прозрачная функциональная версия всегда эквационально корректна и не оставляет места для ошибок

Программы, написанные подобным образом, намного легче понять, поскольку в этом случае можно составить в уме модель состояния системы и добиться желаемого результата посредством *переписывания* или *подстановки*. Рассмотрим такую возможность на более конкретном примере, допустив, что любую программу можно определить в виде ряда функций, обрабатывающих заданные входные данные и получающих конечный результат. Такое допущение можно выразить в псевдоформе следующим образом:

```
Program = [Input] + [fund, func2, func3, ...] -> Output
```

Если указанные функции [fund, func2, func3, ...] являются “чистыми”, то данную программу можно переписать, подставив вместо функций, получаемые в них значения [vail, val2, val3, . . .], не изменяя результат. Рассмотрим в качестве следующего простого примера вычисление средней оценки учащегося: `var input = [80, 90, 100]; var average = (arr) => divide(sum(arr), size(arr)); average (input); //-> 90`

Благодаря ссылочной прозрачности функций sum() и size () выражение `var average = (arr) => divide(sum(arr), size(arr));`

можно переписать для заданных входных данных следующим образом:

```
var average = divide(270, 3); //-> 90
```

Функция divide () всегда “чистая”, и поэтому ее можно переписать далее, используя ее математическое обозначение. Таким образом, для тех же самых входных данных средняя оценка всегда равна 270/3 = 90. Ссылочная прозрач-

ность всегда дает возможность анализировать программы подобным систематическим и почти математическим образом. А вся программа может быть реализована следующим образом:

```
var sum = (total, current) => total + current;
var total = arr => arr.reduce(sum);
var size = arr => arr.length;
var divide = (a, b) => a / b;
var average = arr => divide(total(arr), size (arr));
average(input); // -> 90
```

Еще одна новая функция reduce (). Как и функция тар (), она перебирает всю коллекцию. Если передать ей функцию sum (), с ее помощью можно подытожить результаты сложения чисел в массиве

В главе 4 будут рассмотрены способы объединения функции average () в композицию

Несмотря на то что в данной книге не предполагается применять эквацио- нальные рассуждения к каждому примеру программы, важно понять, что они подразумеваются в любой чисто функциональной программе и что это было бы просто невозможно, если бы функции имели побочные эффекты. Мы еще вернемся в главе 6 к этому важному принципу, рассмотрев его в контексте модульного тестирования функционального кода. Определив заранее все аргументы функции, можно избежать побочных эффектов в большинстве случаев, когда задаются скалярные значения. Но если функции передаются объекты по ссылке, следует принять особые меры предосторожности, чтобы не модифицировать эти объекты неумышленно.

1.2.4. Сохранение данных неизменяемыми

Неизменяемыми называются такие данные, которые нельзя изменить после их создания. Как и во многих других языках программирования, все примитивные типы данных (String, Number и пр.) BjavaScript, по существу, являются неизменяемыми, но объекты других типов (например, массивы) таковыми не являются. Даже если они передаются в качестве входных данных функции, изменение их первоначального содержимого может вызвать побочный эффект. Рассмотрим в качестве простого примера следующий код сортировки массива:

```
var sortDesc = arr => { return arr.sort( (a, b) => b - a ) ;
};
```

На первый взгляд, приведенный выше код идеально чист от побочных эффектов. Он делает именно то, что от него требуется, обрабатывая получаемый массив и возвращая его отсортированным по убывающей:

```
var arr = [1, 2,3, 4,5, 6, 7,8, 9];
sortDesc(arr); // -> [9,8,7,6,5,4,3,2,1]
```

Но, к сожалению, функция Array. sort () сохраняет состояние, вызывая побочный эффект сортировки массива на месте, в результате чего изменяется первоначальная ссылка. И это серьезное препятствие в языке JavaScript , которое мы попробуем преодолеть в последующих главах.

Итак, бросив беглый взгляд на основополагающие принципы функционального программирования (декларативность, чистоту и неизменяемость), его сущность можно выразить в более краткой форме.

Функциональное программирование означает декларативное вычисление "чистых"

функций для создания неизменяемых программ, исключая внешне наблюдаемые побочные эффекты.

Такое определение ФП может показаться не совсем ясным, поскольку оно лишь вскользь затрагивает практические преимущества написания функциональных приложений. Но теперь у вас должно уже складываться ясное представление о том, что значит мыслить функционально.

Большинство трудностей, с которыми сталкиваются разработчики приложений на JavaScript , обусловлены интенсивным применением крупных функций, которые в значительной степени зависят от внешних общих переменных, выполняют немало операций ветвления и не обладают ясной структурой. К сожалению, подобная ситуация характерна для многих современных приложений на JavaScript . Ведь даже удачные приложения обычно состоят их многих исполняемых вместе файлов, образуя общую сеть изменяемых, глобальных данных, которые с трудом поддаются отслеживанию и отладке.

Заставляя себя мыслить категориями "чистых" операций, рассматривая функции как герметичные *единицы работы,* которые вообще не модифицируют данные, можно существенно сократить возможность появления программных ошибок. Эти базовые принципы важно понимать, чтобы выгодно пользоваться теми преимуществами, которые функциональное программирование привносит в прикладной код, направляя по верному пути преодоления его сложности.

1.3. Преимущества функционального программирования

Чтобы извлечь выгоду из функционального программирования, необходимо научиться мыслить функционально, имея в своем распоряжении подходящие инструментальные средства. В этом разделе представлены некоторые из основных методик, образующих обязательный набор инструментальных средств для выработки *функционального восприятия* — инстинктивного стремления анализировать решаемые задачи в виде определенного сочетания простых функций, составляющих вместе готовое решение. Вопросы, рассматриваемые в этом разделе, служат кратким введением в ряд последующих глав. Если какое-нибудь понятие вам трудно усвоить поначалу, не отчаивайтесь — оно постепенно разъяснится по мере чтения остальных глав.

А теперь рассмотрим в общих чертах те преимущества, которые ФП привносит в приложения НаJavaScript. В последующих разделах поясняется, каким образом ФП способно

- побуждать к декомпозиции решаемых задач на простые функции;
- обрабатывать данные, используя текучие цепочки;
- уменьшать сложность управляемого событиями кода, применяя принципы реактивного программирования.

1.3.1. Побуждение к декомпозиции сложных задач

В самых общих чертах функциональное программирование, по существу, означает определенное сочетание декомпозиции (разложения программ на мелкие части) и композиции (соединения отдельных частей в единое целое). Благодаря именно этой двойственности функциональные программы становятся модульными и эффективными. Как упоминалось ранее, единицей модульности, или *единицей работы,* является сама функция. Функциональное мышление, как правило, начинается с декомпозиции, побуждающей учиться разделять отдельную задачу на логические подзадачи (т.е. функции), как демонстрируется на примере декомпозиции программы showStudent, приведенном на рис. 1.3.

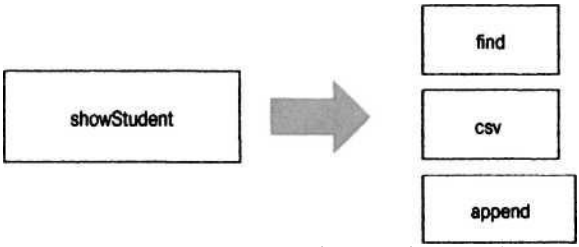


Рис. 1.3. В процессе декомпозиции программа showStudent разбивается на мелкие части — независимые подзадачи, которые проще понять, а сочетая их вместе, легче решить крупную задачу

При необходимости эти задачи могут быть дополнительно подвергнуты декомпозиции до тех пор, пока не будут достигнуты более простые “чистые” функции, каждая из которых является независимой единицей работы. Напомним, что именно такого мыслительного процесса мы придерживались при реорганизации программы showStudent в листинге 1.4. Модуляризация в ФП тесно связана с принципом *единственности,* который гласит, что функции должны иметь единственное назначение. И это было ясно продемонстрировано ранее на примере исходного кода функции average (). Чистота и ссылочная прозрачность побуждают мыслить именно таким образом, поскольку соединяемые вместе функции должны быть согласованы по типам входных и выходных данных. В частности, из ссылочной прозрачности следует, что сложность функции иногда связана непосредственно с количеством аргументов, который ей передаются (это лишь практическое наблюдение, а не формальное понятие, обозначающее следующее: чем меньше параметров у функции, тем она проще).

Изначально мы пользовались волшебной функцией run (), чтобы соединить вместе функции, образующие целую программу. А теперь настало время раскрыть тайны ее волшебства. На самом деле run — это псевдоним *композиции* — одной из самых важных методик ФП. Композиция двух функций дает еще одну функцию, которая получается в результате подключения выхода одной функции на вход другой. Допустим, имеются две функции: f () и *д* (). Формально их композицию можно выразить следующим образом:

```
f • g = f (g (x) )
```

Данная формула означает "композицию f из д", при которой образуется слабая типизированная взаимосвязь значения, возвращаемого функцией д (), с аргументом функции f (). Требование совместимости обеих функций состоит в том, что они должны быть согласованы по количеству аргументов и их типам. Более подробно об этом речь пойдет в главе 3, а до тех пор рассмотрим блок- схему композиции программы showStudent, приведенную на рис. 1.4, где на этот раз использовано подходящее имя функции compose:

```
var showStudent = compose(append('#student-info'), csv, find(db));

showStudent('444-44-4444');
```

showStudent

Рис. 1.4. Поток данных при композиции двух функций. Значение, возвращаемое из функции find (), должно быть совместимым по типу и арности (т.е. количеству аргументов) с функцией csv (), возвращающей, в свою очередь, информацию, которая может быть использована в функции append (). Обратите внимание на то, что порядок вызова этих функций был изменен на обратный, чтобы сделать более понятным поток данных

Понимание принципа действия функции compose () имеет решающее значение для изучения особенностей реализации модульности и повторного использования в функциональных приложениях. Более подробно об этом речь пойдет в главе 4. В результате композиции функций получается код, в котором назначение всего выражения можно понять из назначения отдельных его частей. Такое качество кода трудно получить, применяя другие принципы и парадигмы программирования.

Кроме того, композиция функций повышает уровень абстракции до такой степени, которая позволяет ясно обозначить все стадии выполнения кода, не раскрывая никаких нижележащих подробностей. Функции compose () передаются другие функции в качестве аргументов и поэтому она называется *функцией высшего порядка.* Но композиция является далеко не единственным способом создания текучего, модульного кода. В этой книге поясняется также, как составлять последовательности операций, соединяя их подобно цепочке.

1.3.2. Обработка данных с помощью текучих цепочек

Помимо функции tap (), ассортимент функций высшего порядка можно импортировать

в любой проект HaJavaScript через ряд эффективных и оптимизированных функциональных библиотек. В главах 3 и 4 дается краткий обзор многих функций высшего порядка, реализованных в таких весьма распространенных инструментальных средствах функционального программирования на JavaScript, как Lodash.js и Ramda.js. Их возможности во многих отношениях перекрываются, но каждое из них привносит свои особенности, упрощающие составление функций в цепочки.

Если у вас имеется некоторый опыт написания кода средствами библиотеки JQuery, то вы, вероятно, знакомы с идиомой *цепочки (chain)* — последовательностью вызовов функций, из которых возвращается один и тот же общий объект в качестве возвращаемого значения (например, объект $ или jQuery). Аналогично композиции, эта идиома позволяет писать выразительный и краткий код и зачастую применяется в библиотеках как функционального, так и реактивного программирования HaJavaScript (подробнее об этом речь пойдет далее). Чтобы продемонстрировать возможности цепочек, рассмотрим решение другой задачи. Допустим, требуется написать программу, вычисляющую среднюю оценку учащихся, зачисленных в несколько классов. Так, если имеется следующий массив с данными зачисления: `let enrollment = [`

```
   {enrolled: 2, grade: 100},
   {enrolled: 2, grade: 80},
   {enrolled: 1, grade: 89}
];
```

то императивный подход к решению данной задачи может выглядеть следующим образом:

```
var totalGrades = 0;
var totalStudentsFound = 0;
for(let i = 0; i < enrollment.length; i++) {
     let student = enrollment [i];
     if(student !== null) {
       if(student.enrolled > 1) {
          totalGrades+= student.grade;
          totalstudentsFound++;
       }
     } } var average = totalGrades / totalStudentsFound; // -> 90
```

Как и прежде, произведя декомпозицию данной задачи по типу функционального мышления, можно выявить следующие главные стадии.

- Выбор подходящего ряда учащихся, зачисленных в несколько классов.
- Извлечение их оценок.
- Вычисление их средней оценки.

Для соединения функций, представляющих эти стадии, можно теперь воспользоваться библиотекой Lodash, сформировав из них цепочку, как показано в листинге 1.5 (полное описание назначения каждой из этих функций можно найти в документации, ссылка на которую приведена в приложении). Цепочка функций является программой с *отложенным вычислением.* Это означает, что ее вычисление откладывается до тех пор, пока оно не потребуется, что положительно сказывается на производительности, экономя ценное процессорное время, поскольку позволяет избежать выполнения целых

последовательностей кода, которые нигде больше не используются. Подобным образом, по существу, имитируется режим *вызова только по требованию,* встроенный в другие языки функционального программирования.

Листинг 1.5. Программирование с помощью цепочек функций

```
  chain(enrollment)
.filter(student => student.enrolled > 1)
.pluck (' grade' )          С вызова функции_. value ()
. average ()                начинается выполнение всех
.value (); // -> 90 ◄ ---- 1 операций в цепочке
```

Не особенно беспокойтесь, если вам не все понятно, что происходит в приведенном выше коде. Пока вам достаточно сравнить его с представленным ранее императивным вариантом решения рассматриваемой здесь задачи, обратив внимание на то, каким образом можно исключить потребность в объявлении и изменении переменных, циклов и условных операторов if-else. Как будет показано в главе 7, многие императивные механизмы потоков управления вроде циклов и ветвлений повышают уровень сложности функций, поскольку они выполняются по разным путям в зависимости от определенных условий, что невероятно затрудняет их тестирование.

Но, откровенно говоря, в данном примере отсутствует немалая доля кода обработки событий, имеющаяся в типичных реальных программах. Как упоминалось ранее, генерирование исключений является причиной побочных эффектов. Исключения отсутствуют в академических языках функционального программирования, но в реальности избежать их не удается. Цель состоит в том, чтобы реализовать чистую обработку ошибок в как можно большей степени, допустив генерирование исключений в действительно исключительных ситуациях, подобных описанным ранее.

Правда, применяя некоторые исключительно функциональные проектные шаблоны, не придется приносить в жертву такой уровень выразительности, чтобы снабдить свой код надежной логикой обработки событий. Обсуждению этой темы в основном посвящена глава 5.

До сих пор было показано, каким образом ФП может помочь в создании модульных, тестируемых, расширяемых приложений. Но насколько оно пригодно в тех случаях, когда требуется взаимодействие с асинхронными или основанными на событиях данных, вводимыми пользователем или поступающими из удаленных запросов, файловых систем или постоянных хранилищ?

1.3.3. Реагирование на сложность асинхронных приложений

Если вы вспомните, когда в последний раз вам приходилось извлекать удаленные данные, обрабатывать вводимые пользователем данные или взаимодействовать с локальным хранилищем, то в вашей памяти, вероятно, всплывут целые разделы написанной вами бизнес-логики, оформленной во вложенные последовательности функций обратного вызова. Такой образец обратного вызова нарушает линейный поток выполнения прикладного кода и становится неудобочитаемым, поскольку он загроможден вложенными формами логики обработки удачного и неудачного (с ошибками) исхода выполнения кода.

Все это должно быть изменено.

Как пояснялось ранее, крайне важно изучать функциональное программирование, и особенно разработчикам приложений HaJavaScript. При написании крупных приложений немало внимания переносится с объектно-ориентированных библиотек вроде Backbone js на библиотеки, в которых поддерживается парадигма реактивного программирования. Такие веб-ориентированные библиотеки, как Angular.js, по-прежнему широко применяются в настоящее время. Но вместе с тем появились и новые библиотеки (например, RxJS), где ФП эффективно используется для решения довольно трудных задач.

Реактивное программирование относится к едва ли не самым захватывающим и интересным приложениями функционального программирования. Оно позволяет значительно уменьшить сложность асинхронного и управляемого событиями кода, с которым постоянно приходится иметь дело разработчикам как клиентских, так и серверных приложений HaJavaScript.

Главное преимущество, которое дает принятие парадигмы реактивного программирования, заключается в том, что она повышает уровень абстракции прикладного кода, позволяя сосредоточиться на конкретной бизнес-логике и забыть о трудном стереотипном коде, связанном с настройкой асинхронных и управляемых событиями программ. Кроме того, эта перспективная парадигма позволяет в полной мере использовать возможности ФП составлять функции в композиции или цепочки.

События наступают по самым разным причинам: от щелчков кнопками мыши, при изменениях в текстовых полях, смене фокуса ввода, обработке новых HTTP-запросов, поступлении запросов в базу данных, записи в файлы и т.д. Допустим, требуется прочитать и проверить достоверность номера социального страхования учащегося. Типичный императивный подход к решению этой задачи может выглядеть так, как демонстрируется в листинге 1.6.

Листинг 1.6. Императивная версия программы, читающей и проверяющей достоверность номера социального страхования учащегося

```
var valid = false;
var elem = document.queryselector('#student-ssn');
elem.onkeyup = function(event) {
r>var val = elem.value;
  if(val !== null && val.length !== 0) { val =
      val. replace (Ms* I\s*$ I \-s/g, ' ');
      if(val.length === 9) {
          console.log(4Valid SSN: ${val}!');
  —invalid = true;


  else {
        console.log('Invalid SSN: ${val}!');
```

Побочные эффекты, возникающие при доступе к данным за пределами области видимости функции

Усечение и очистка входных данных, модифицирующая их на месте

Вложенная логика ветвления

Такое решение столь простой задачи выглядит довольно сложным, а реализующему его коду недостает желательного уровня модульности, чтобы разместить всю бизнес-логику в одном месте. Кроме того, функцию в данном примере нельзя использовать повторно из-за ее зависимости от внешнего состояния. А поскольку реактивное программирование опирается на функциональное программирование, то в нем можно выгодно воспользоваться "чистыми" функциями для обработки данных с помощью тех же самых операций вроде шар и reduce и лаконичных лямбда-выражений. Следовательно, изучить функциональное программирование — это лишь полдела, чтобы овладеть реактивным программированием!

Парадигма реактивного программирования активизируется посредством очень важного средства, называемого *наблюдаемым объектом (observable)*. Такие объекты позволяют подписаться на поток данных, который можно обработать, изящно составив операции вместе в единую цепочку. В листинге 1.7 демонстрируется практическое применение наблюдаемого объекта с целью подписаться на простое поле ввода номера социального страхования учащегося.

Листинг 1.7. Функциональная версия программы, читающей и проверяющей достоверность номера социального страхования учащегося

Заметили ли вы сходство исходного кода из листинга 1.7 с исходным кодом из листинга 1.5? Тем самым демонстрируется, что, независимо от того, обрабатываются ли элементы коллекции или вводимые пользователем данные, все операции абстрагируются и интерпретируются одним и тем же способом. (Подробнее об этом речь пойдет в главе 8.)

Из всего изложенного выше можно сделать следующий едва ли не самый важный вывод: все операции, выполняемые в версии программы из листинга 1.7, являются совершенно неизменяемыми, а вся бизнес-логика выделена в отдельные функции. Пользоваться функциональным программированием вместе с реактивным совсем

необязательно, но к этому *побуждает* стремление мыслить функционально. И в этом случае раскрывается совершенно замечательная архитектура, основанная на *функционально-реактивном программировании* (ФРП).

Функциональное программирование — это парадигма, позволяющая существенно преобразить подход к решению любых трудных задач программирования. Так способно ли ФП заменить столь распространенное ООП? К счастью, ФП не предполагает категорический, не допускающий никаких отступлений подход к написанию прикладного кода, как метко заметил Майкл Физерс (см. цитату, приведенную в начале этой главы). На самом деле при разработке многих приложений можно извлечь выгоду как из ФП, так и из объектно-ориентированной архитектуры. Благодаря жесткому контролю неизменяемости и общего состояния ФП отличается еще и тем, что упрощает многопоточное программирование. Но поскольку платформа JavaScript является однопоточной, то об этом можно не особенно беспокоиться, что и сделано в данной книге. В следующей главе будет уделено некоторое внимание пояснению главных отличий ФП и ООП, что поможет вам легче овладеть умением мыслить функционально.

В этой главе были вкратце затронуты вопросы, которые будут более подробно рассмотрены далее в книге по мере того, как вы будете все глубже погружаться в функциональный образ мышления. Если вы внимательно следовали описанию всех представленных в этой главе понятий — это хорошо, но не особенно беспокойтесь, если вы упустили что-нибудь из виду. Это лишь означает, что вы выбрали нужную вам книгу. При традиционном ООП вы, вероятно, привыкли программировать в императивном/процедурном стиле. Смена этого стиля потребует от вас коренного перелома в мыслительных процессах, как только вы начнете решать поставленные задачи функциональным способом.

Резюме

Из этой главы вы узнали следующее.

- Код, в котором применяются “чистые” функции, неспособен изменить или нарушить глобальное состояние, что помогает сделать код более удобным для тестирования и сопровождения.

- Функциональное программирование осуществляется в декларативном стиле, упрощающем понимание прикладного кода. Тем самым повышается общий уровень удобочитаемости прикладного кода, который становится более лаконичным благодаря сочетанию функций с лямбда-выражениями.
- Обработка данных, хранящихся в элементах коллекции, плавно выполняется по цепочкам функций, связывающих вместе такие операции, как шар и reduce.
- В функциональном программировании функции рассматриваются как стандартные блоки с учетом того, что функции первого класса и высшего порядка улучшают модульность и повторное использование прикладного кода.
- Сложность управляемых событиями программ можно уменьшить, сочетая

функциональное программирование с реактивным.

Сценарий высшего порядка

в этой главе...

- Причины, по которым JavaScript является подходящим языком для функционального программирования
- Язык JavaScript допускает разработку со множеством парадигм
- Неизменяемость и правила внесения изменений
- Представление о функциях первого класса и высшего порядка
- Исследование понятий замыканий и областей видимости
- Практическое применение замыканий

Преобладающая парадигма отсутствует не только в естественном языке, но и в JavaScript. Разработчики могут выбирать из массы подходов — процедурных, функциональных и объектно-ориентированных, — применяя их в любом подходящем сочетании.

Из книги *If Hemingway WroteJavaScript* (Если бы Хемингуэй писал HaJavaScript) *Ангуса Кролла (Angus Croll)*

По мере разрастания приложений увеличивается их сложность. Независимо от ваших личных качеств, беспорядок неизбежен, если у вас нет подходящих моделей программирования. В главе 1 пояснялись побуждающие причины, по которым следует принять парадигму функционального программирования. Но сами парадигмы являются лишь моделями программирования, которые требуют появления подходящего языка их реализации.

В этой главе дается краткий обзор гибридного языка JavaScript, сочетающего в себе принципы ООП и ФП. Разумеется, этот обзор ни в коей мере не претендует на обширное исследование языка. Напротив, оно сосредоточено

на том, что именно дает возможность пользоваться языком JavaScript функционально, а также на том, что препятствует этому. Характерным тому примером служит отсутствие в JavaScript поддержки неизменяемости. В этой главе рассматриваются также функции высшего порядка, позволяющие писать код JavaScript в функциональном стиле. Итак, без лишних слов приступим к делу.

2.1. Причины для выбора JavaScript

Мы начинали с пояснения причин, по которым требуется мыслить функционально в процессе программирования. А теперь попытаемся объяснить причины для выбора JavaScript. Такой выбор просто объясняется вездесущностью этого языка. В частности, JavaScript является динамически типизированным, объектно-ориентированным, универсальным языком с необычайно выразительным синтаксисом. Это едва ли не самый распространенный из всех когда- либо создававшихся языков программирования, поскольку он применяется для разработки приложений для мобильных устройств, веб-сайтов, веб-серверов, настольных и встроенных систем и даже баз данных. Принимая во внимание необычайно широкую область внедрения JavaScript как *языка для веб,* можно смело утверждать, что JavaScript применяется в ФП чаще всех остальных когда- либо создававшихся языков программирования.

Несмотря на свой C-подобный синтаксис, JavaScript немало заимствовал из других языков ФП наподобие Lisp и Scheme. Их сходство проявляется в поддержке функций высшего порядка, замыканий, литералов массивов и прочих языковых средств, благодаря которым JavaScript служит превосходной платформой для применения методик ФП. На самом деле функции являются основными единицами работы в JavaScript, а это означает, что они служат не только для управления режимом работы приложений, но и для определения объектов, создания модулей и обработки событий.

Язык JavaScript активно развивается и совершенствуется. В его очередном выпуске ES6 по стандарту ECMAScript (ES) внедрено немало новых языковых средств, в том числе стрелочные функции, константы, итераторы, обязательства, иначе называемые обещаниями, и прочих средств, пригодных и для ФП.

Несмотря на то что в JavaScript имеется немало эффективных функциональных средств, очень важно подчеркнуть, что JavaScript является одновременно языком как ООП, так и ФП. К сожалению, возможности JavaScript для ФП нередко упускаются из виду. В частности, большинство разработчиков пользуются изменяемыми операциями, императивными управляющими структурами, а также изменениями состояния экземпляров объектов, т.е. всеми теми языковыми средствами, которые просто исключаются из арсенала тех, кто принимает функциональный стиль программирования. Тем не менее здесь важно уделить сначала немного внимания обсуждению объектно-ориентированных свойств JavaScript, чтобы лучше оценить главные отличия парадигм ООП и ФП. Благодаря этому вам будет легче перейти к функциональному стилю программирования.

2.2. ФП в сравнении с ООП

Для разработки систем от среднего до крупного масштаба можно применять как функциональное программирование (ФП), так и объектно-ориентированное

программирование (ООП). Обе парадигмы программирования могут сочетаться в одном языке, и примером тому служат гибридные языки вроде Scala и F#. Аналогичными возможностями обладает и язык JavaScript, поэтому овладение им подразумевает приобретение навыков использования ФП и ООП в определенном сочетании. Выбор пределов такого сочетания зависит от личных предпочтений и требований, которые ставит решаемая задача. Поэтому ясное представление о том, где пересекаются функциональный и объектно- ориентированный подходы к программированию и где они расходятся, поможет свободно переходить от одного к другому или к мышлению категориями того или другого.

Рассмотрим в качестве примера простую модель системы управления обучением, включая объект типа Student. С точки зрения иерархии классов или типов вполне естественно рассматривать тип Student как подтип, производный от типа Person, охватывающего основные свойства вроде Ф.И.О., адрес и т.д.

Объектно-ориентированный характер JavaScript

Если отношение между объектами в этой книге определяется термином *подтип* или *производный тип,* то имеется в виду *прототипное* отношение, существующее между объектами. В этой связи очень важно подчеркнуть, что в JavaScript отсутствует поддержка *классического* наследования, имеющаяся в других языках вроде Java, несмотря на то, что язык JavaScript считается объектно-ориентированным.

В стандарте ES6 механизм установления прототипных связей между объектами был (по мнению многих —- ошибочно) приукрашен такими ключевыми словами, как class и extends. И хотя это упрощает программирование наследования объектов, тем не менее, скрывает настоящее действие и потенциал применяемого в JavaScript механизма прототипов. В данной книге не рассматриваются объектно-ориентированные возможности JavaScript, но в конце этой главы упоминается книга, в которой они обсуждаются достаточно подробно наряду с другими возможностями JavaScript.

Чтобы ввести дополнительные функциональные возможности, можно произвести далее от типа Student более конкретный тип, например College Student. По существу, объектно-ориентированные программы отличаются созданием новых производных типов объектов в качестве основных средств для достижения повторного использования кода. В данном случае в типе Collegestudent будут повторно использоваться все данные и виды поведения его родительских типов. Но ввести дополнительные функциональные возможности в существующие объекты будет нелегко, если они должны применяться не ко всем производным от них объектам. Так, если свойства firstname и lastname применяются как в самом объекте типа Person, так и в объектах про

изводных от него типов, то свойство workAddress, безусловно, более уместно для объекта типа Employee, производного от типа Person, чем для объекта типа Student. Такая модель рассматривается здесь потому, что главное отличие объектно-ориентированных приложений от функциональных заключается в организации подобных данных (т.е. свойств объектов) и поведении (т.е. функций).

Объектно-ориентированные приложения, которые в большинстве своем являются императивными, в значительной степени опираются на инкапсуляцию объектов, чтобы сохранить целостность их изменяемого (как непосредственного, так и наследуемого) состояния, а также раскрывать это состояние или манипулировать им через методы экземпляра. В итоге устанавливается тесная связь данных объектов и с их довольно конкретным поведением, образуя связный пакет. В этом, собственно, и состоит главная цель объектно-ориентированных программ и причина, по которой центральной формой абстракции в них является объект.

А функциональное программирование исключает потребность скрывать данные от вызывающего кода и, как правило, оперирует более мелким набором очень простых типов данных. И поскольку в ФП ничто не изменяется, объектами можно манипулировать непосредственно, но на этот раз через обобщенные функции, находящиеся за пределами области видимости объекта. Иными словами, данные слабо связаны с поведением объектов. Как показано на рис. 2.1, вместо вполне конкретных методов экземпляра функциональный код опирается на более крупные операции, которые можно выполнять над многими типами данных. В такой парадигме *функции становятся главной формой абстракции.*

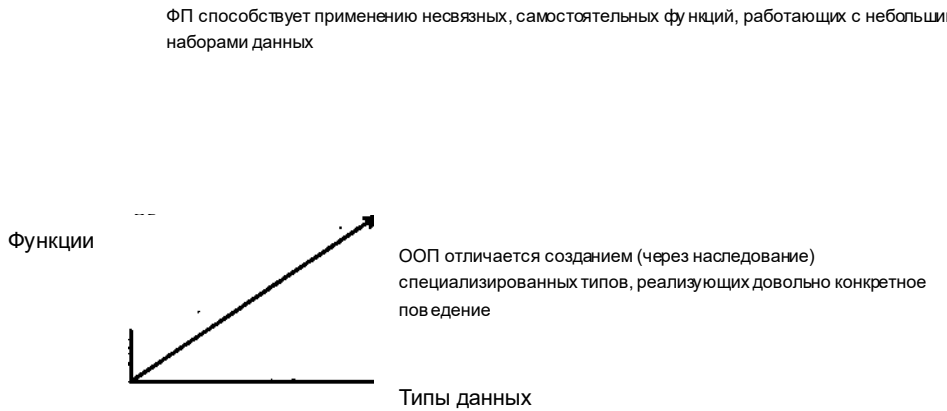


Рис. 2.1. ООП способствует установлению логической связи многих типов данных со специализированным поведением, тогда как ФП сосредоточено на связывании операций с этими типами данных через композицию. Имеется удобное место, где обе парадигмы программирования могут использоваться продуктивно. И это вполне возможно в таких гибридных языках, как Scala, F# и JavaScript

Глядя на рис. 2.1, можно заметить, что обе парадигмы программирования начинают заметно отличаться по мере перемещения по стрелке вправо вверх.

На практике в некоторых лучших образцах объектно-ориентированного кода можно обнаружить совместное применение обеих парадигм в точке их пересечения. Для этого объекты следует рассматривать как неизменяемые сущности или значения, разделяя их функциональные возможности на отдельные функции, оперирующие этими объектами. Таким образом, из следующего метода для объекта типа Person:

```
get fullname() {
    return [this._firstname, this._lastname].join(' '); }
```

В методах рекомендуется пользоваться ссылкой this для доступа к состоянию объектов

можно выделить функцию таким образом:

```
var fullname =
    person => [person.firstname, person.lastname].join(' ');
```

Ссылка this, по существу, заменена переданным объектом

Как известно, JavaScript является динамически типизированным языком, а это означает, что рядом со ссылками на объекты типы данных явно не указываются. Следовательно, метод fullname () будет оперировать объектом любого типа, производного от типа Person (или же любым объектом со свойствами firstname и lastname), как показано на рис. 2.2. Принимая во внимание динамический характер JavaScript, в этом языке поддерживается применение

```
var person = new Student('Alonzo', 'Church', '444-44-4444', 'Princeton'); p.fullname; //-> Alonzo
Church _
```

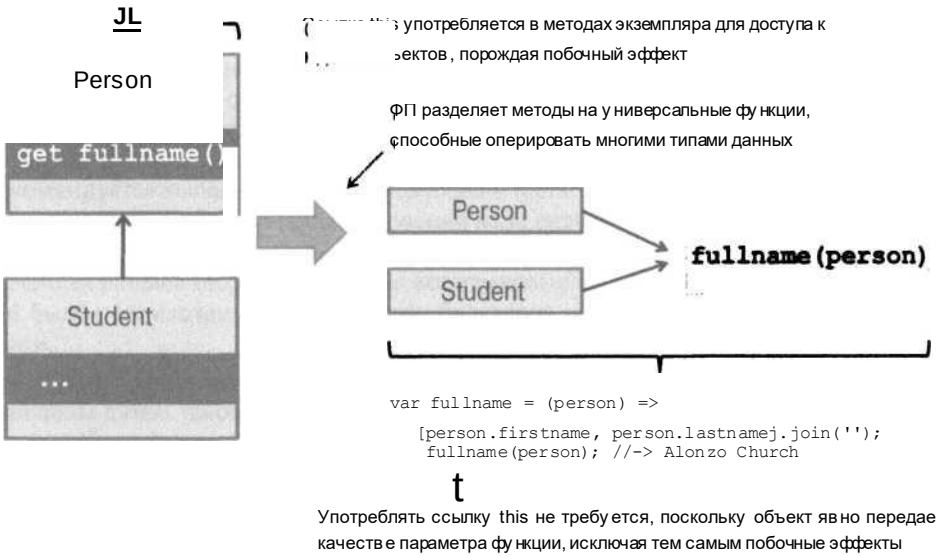


Рис. 2.2. ООП сосредоточено на создании иерархий наследования (например, типа Student от типа Person) с методами и тесно связанными вместе данными, а ФП — на создании универсальных полиморфных функций, оперирующих разными типами данных, исключая применение ссылки this

обобщенных полиморфных функций. Иными словами, те функции, где используются ссылки на базовые типы данных (например, Person), могут оперировать объектами производных типов (например, Student или Collegestudent).

Как следует из рис. 2.2, выделение из метода fullname () самостоятельной функции побуждает избегать употребления ссылки this для доступа к данным объекта. Использование ссылки this затруднительно потому, что она предоставляет доступ к данным экземпляра объекта за пределами области видимости метода, что вызывает побочные эффекты. А если применяется ФП, то данные объекта не связываются непосредственно с отдельными частями прикладного кода, и поэтому их намного удобнее использовать повторно и сопровождать.

Вместо того чтобы создавать целый ряд производных типов, можно расширить поведение функции, передав ей другие функции в качестве аргументов. Как пример рассмотрим определение простой модели данных в листинге 2.1. Эта модель содержит класс Student, производный от класса Person, и применяется в большинстве примеров, приведенных в данной книге.

Листинг 2.1. Определение классов Person и Student

```
class Person {
    constructor(firstname, lastname, ssn) {
        this._firstname = firstname;
        this._lastname = lastname;
        this._ssn = ssn;
        this._address = null;
        this._birthYear = null;
}

    get ssn() {
        return this._ssn;
    }

    get firstname() {
        return this._firstname;
    }

    get lastname() {
        return this._lastname;
    }

    get address() {
        return this._address;
    }

    get birthYearO {
        return this._birthYear;
    }
```

Употребление методов установки совсем не означает поддержку модификации объектов, но служит удобным способом создания объектов с разными свойствами, не прибегая к помощи очень длинных конструкторов. Как только объекты будут созданы и заполнены данными, их состояние больше не меняется (о том, как обращаться с этим, речь пойдет далее в главе)

```
set birthYear(year) {
    this._birthYear = year;

set address(addr){
    this, address = addr;
                        toStringO {
  return 'Person(${this. firstname}, ${this._lastname})';


class Student extends Person { constructor{firstname, lastname, ssn, school) {
super(firstname, lastname, ssn);
    this, school = school;


get school() { return this._school;
}
```

Как найти, загрузить примеры исходного кода из этой книги и пользоваться ими

Примеры исходного кода из этой книги можно найти на веб-странице по адресу www.manning.com/books/functionalprogramming-in-javascript или https://github.com/luijar/functional-programming-js. Можете свободно пользоваться этим кодом, чтобы практиковаться в ФП. С этой целью рекомендуется выполнять любые модульные тесты и экспериментировать с разными программами. Для преобразования кода версии ES6 в эквивалентный код версии ES5 в примерах из этой книги был использован транспилятор Babel, называвшийся раньше 6to5, поскольку не все языковые средства JavaScript из версии ES6 были реализованы в большинстве браузеров на момент написания данной книги.

Некоторые языковые средства не требуют транспиляции и могут быть активизированы путем такого параметра настройки браузера, как, например, флажок Enable Experimen tal JavaScript (Активизировать экспериментальный режим работы JavaScript) в браузере Chrome. Если вы работаете в экспериментальном режиме, то активизируйте *строгий режим* работы JavaScript, введя оператор ' use strict'; в самом начале исходного файла JavaScript.

Если известно конкретное лицо, то задача состоит в том, чтобы найти всех его друзей, проживающих в той же стране. А если известен конкретный учащийся, то необходимо найти других учащихся, проживающих в той же стране и посещающих ту же школу. С одной стороны, в приведенном ниже объектно

ориентированном решении этой задачи операции тесно связаны по ссылкам this и super с текущим и родительским объектом соответственно.

```
// Класс Person
peoplelnSameCountry(friends) { var result = [];
    for (let idx in friends) {
        var friend = friends [idx];
        if (this.address.country === friend.address.country) {
            result.push(friend);
        }
    } return result;
};

// Класс Student studentsInSameCountryAndSchool(friends) {
    var closeFriends = super.peoplelnSameCountry(friends); ◄— var
    result = [];
    for (let idx in closeFriends) {
        var friend = closeFriends[idx];
        if (friend.school === this.school) { result.push(friend);
        }
    }
    return result;
```

Использовать ссылку super для запроса данных из родительского класса

С другой стороны, опираясь на принятые в ФП принципы чистоты и ссылочной прозрачности, позволяющие обособить поведение от состояния, можно ввести дополнительные операции, определив и объединив новые функции, оперирующие упомянутыми выше типами данных. В конечном счете можно получить простые объекты, ответственные за хранение данных, а также универсальные функции, способные оперировать этими объектами как аргументами, которые могут быть составлены путем композиции для достижения специальных функциональных возможностей. И хотя речь о композиции пойдет лишь в главе 4, важно подчеркнуть еще одно фундаментальное отличие обеих рассматриваемых здесь парадигм программирования. По существу, наследованию в ООП соответствует композиция в ФП с точки зрения нового поведения, применяемого к разным типам данных[2]. Чтобы выполнить приведенный выше код, потребуется следующий набор данных:

```
var curry = new Student('Haskell', 'Curry',
                        '111-11-1111', 'Penn State');
curry.address = new Address('US');
var turing = new Student('Alan', 'Turing', '222-22-2222', 'Princeton');
turing.address = new Address('England');

var church = new Student('Alonzo', 'Church', '333-33-3333', 'Princeton');
church.address = new Address('US');
```

[2] **Эта сноска касается в большей степени тех, кто применяет ООП на практике, чем самой парадигмы ООП. Многие авторы в данной области, включая и знаменитую "Банду четырех", предпочитают больше композицию объектов, чем наследование классов, основанное на принципе подстановки Барбары Дисков.**

```
var kleene = new Student('Stephen', 'Kleene', '444-44-4444', 'Princeton');
kleene.address = new Address('US');
```

В объектно-ориентированном решении данной задачи для поиска всех остальных учащихся, посещающих ту же самую школу, применяется следующий метод из класса Student:

```
church.studentsInSameCountryAndSchool([curry, turing, kleene]); // -> [kleene]
```

А в функциональном решении данная задача разделяется на мелкие функции следующим образом:

```
function selector (country, school) {
    return function (student) {
        return student .address, country () === country &&
               student.school() === school;
    j.
.   '
```

Создать функцию selector (), которой должно быть известно, как сравнивать учащихся по стране проживания и посещаемой школе

Перемещаться по графам объектов. Далее в этой главе будет продемонстрирован лучший способ доступа к свойствам объекта

```
var findStudentsBy = function (friends, selector) {
    friends. filter (selector) ;
j.
```

Применить операцию фильтрации return к массивам и внедрить специальное поведение через функцию selector ()

```
findStudentsBy([curry, turing, church, kleene], selector('US', 'Princeton'));

// -> [church, kleene]
```

Применяя функциональное программирование, можно создать совершенно новую функцию findStudentsBy (), с которой намного проще работать. Однако эта новая функция оперирует любыми объектами, связанными с типом Person, а также с любым сочетанием названий школы и страны.

Данный пример наглядно демонстрирует отличия обеих парадигм программирования. В частности, ООП сосредоточено на характере данных и отношениях между ними, тогда как ФП — на операциях, выполняемых над данными (т.е. на поведении). В табл. 2.1 сведены также остальные отличия, которые следует подчеркнуть в связи с тем, что они будут рассматриваться далее в этой и других главах.

Таблица 2.1. Сравнение важных свойств ООП и ФП. Эти свойства обсуждаются на протяжении всей книги

	ФП	ООП
Единицы композиции	Функции	Объекты (классы)
Стиль программирования	Декларативный	Императивный
Данные и поведение	Слабо связанные в чистые, самостоятельные функции	Тесно связанные в классы с методами
Управление состоянием	Объекты считаются неизменяемыми значениями	Объекты модифицируются через методы экземпляра
Поток управления	Функции и рекурсия	Циклы и условные операторы
Потокобезопасность	Разрешается параллельное программирование	Трудно достижима
Инкапсуляция	Не требуется, поскольку ничто не изменяется	Требуется для сохранения целостности данных

Несмотря на заметные отличия обеих парадигм программирования, их сочетание при написании приложений может оказаться весьма эффективным. С] одной стороны, такой подход позволяет создать богатую модель данных с естественными отношениями между составляющими ее типами, а с другой стороны, получить в свое распоряжение ряд чистых функций, способных манипулировать этими типами. Где положить предел такому сочетанию обеих парадигм программирования, зависит от того, насколько удобно вам пользоваться каждой из них. Вследствие того что язык JavaScript является одновременно объектно-ориентированным и функциональным, для его функционального применения требуется уделять особое внимание контролю над изменениями состояния.

2.2.1. Управление состоянием объектов в JavaScript

Состояние программы можно определить как моментальный снимок данных, хранящихся во всех ее объектах в любой момент времени. Но что касается сохранения состояния объекта, то JavaScript, к сожалению, — самый неподходящий для этой цели язык. Ведь объект в JavaScript весьма динамичен, а следовательно, его свойства могут быть изменены, добавлены или удалены в любой момент времени. Так, если вы считаете, что в исходном коде из листинга 2.1 можно инкапсулировать свойство address (использование в его имени знака подчеркивания носит исключительно синтаксический характер) средствами класса Person, то вы ошибаетесь. Имея полный доступ к этому свойству за пределами данного класса, вы можете сделать с ним все, что угодно, в том числе удалить его.

Всякая свобода действий предполагает большую ответственность. И хотя вы вольны выполнить немало ловких операций вроде динамического создания свойств, такая свобода действий может также привести к появлению кода, который чрезвычайно трудно сопровождать в программах как среднего, так крупного масштаба.

Как упоминалось в главе 1, применение чистых функций упрощает понимание и сопровождение исходного кода. А есть ли что-нибудь подобное чистому объекту? Неизменяемый объект, содержащий неизменяемые функциональные средства, можно считать чистым. Рассуждение в отношении функций вполне подходит к простым объектам.

Управление состоянием BjavaScript имеет решающее значение для поиска путей его применения в качестве функционального языка. Имеется ряд норм практики и шаблонов, которыми можно пользоваться для управления неизменяемостью, и об этом речь пойдет в последующих разделах. Но в стремлении соблюсти неизменяемость придется немало потрудиться, чтобы добиться полной инкапсуляции и защиты данных.

2.2.2. Обращение с объектами как со значениями

Символьные строки и числа относятся к числу тех типов данных, с которыми проще всего обращаться в любом языке программирования. А почему это так? Отчасти это объясняется тем, что по традиции эти примитивные данные, по существу, неизменяемы, что позволяет спокойно предположить, что другие определяемые пользователем типы данных таковыми не являются. В функциональном программировании типы данных с подобным поведением называются *значениями*. В главе 1 пояснялось, как следует понимать неизменяемость и что для этого любой объект требуется, по существу, рассматривать как значение. Это дает возможность работать с функциями, передающими объекты, не беспокоясь об изменениях в них.

Несмотря на все синтаксические удобства, которые дает внедрение классов в стандарте ES6, объекты BjavaScript являются не более чем вместилищами свойств, которые можно добавить, удалить и изменить в любой момент. Как же исправить это положение? Во многих языках программирования поддерживаются конструкции, позволяющие сделать свойства объекта неизменяемыми. Характерным тому примером служит ключевое слово final в Java. А в таких языках, как F#, переменные являются неизменяемыми по умолчанию, если только они не объявлены иначе. В настоящее время такой роскошью JavaScript не обладает. И хотя примитивные типы BjavaScript не подлежат изменению, тем не менее, можно изменить состояние переменной, ссылающейся на примитивный тип. Следовательно, требуется какая-то возможность предоставить или хотя бы эмулировать неизменяемые ссылки на данные, чтобы определяемые пользователем объекты вели себя так, как будто они неизменяемы.

В стандарте ES6 появилось ключевое слово const, предназначенное для создания константных ссылок. А это уже позволяет двигаться в нужном направлении, поскольку константы нельзя переназначать или переопределять. В практике функционального программирования ключевым словом const можно пользоваться как средством внедрения простых конфигурационных данных (строк URL, имен баз данных и прочих) в функциональную программу по мере надобности. И хотя чтение из внешней переменной вызывает побочный эффект, на платформе JavaScript предоставляется специальная семантика для констант, чтобы они не изменялись неожиданно в промежутках между вызовами функций. Ниже приведен характерный пример объявления константного значения в JavaScript.

```
const gravityms = 9.806;      . Интерпретатор JavaScript не
                                допустит переназначение
gravity_ms = 20; ◄ -------- ' этой константы
```

Но это не разрешает затруднения, связанные с изменяемостью, в такой степени, в какой того требует ФП. Переназначение переменной можно предотвратить, но как воспрепятствовать изменению состояния объекта? Так, следующий фрагмент кода вполне

допустим:

```
const student = new Student('Alonzo', 'Church',
                            '666-66-6666', 'Princeton');

student, lastname = 'Mourning';◄---------- 1 Это свойство изменилось
```

Следовательно, требуются более строгие правила соблюдения неизменяемости, и для защиты от модификаций вполне подходит инкапсуляция. В качестве неплохой альтернативы для простых структур данных можно принять проектный шаблон “Объект-значение” (Value Object), где объект-значение — это такой объект, равенство которого зависит не от идентичности или ссылки, а только от его значения. После того как объект-значение объявлен, изменить его состояния нельзя. Помимо чисел и символьных строк, примерами объектов-значений служат типы tuple, pair, point, zipCode, coordinate, money, date и пр. Ниже приведена реализация объекта-значения типа zipCode.

```
function zipCode(code, location) {
    let _code = code;
    let -location = location || '';

    return {
        code: function () {
        return _code;
    },
    location: function () {
        return -location;
    },
    fromString: function (str) {
        let parts = str.split('-');
        return zipCode(parts[0], parts [1]);
    },
    toString: function () {
        return _code + '-' + -location;
    } };
} const princetonZip = zipCode('08544', ’3345’); princetonZip.toString(); //->
’08544-3345'
```

Функциями BjavaScript можно пользоваться для защиты доступа к внутреннему состоянию объекта-значения почтового индекса, возвращая *интерфейс объектного литерала,* обеспечивающий вызывающему коду доступ к небольшому ряду методов и обращающийся со свойствами code и location как с псевдо- закрытыми переменными. Такие переменные доступны в объектном литерале только через *замыкания,* рассматриваемые далее в этой главе.

Возвращаемый объект, по существу, ведет себя как примитивный тип, у которого отсутствуют модифицирующие методы[3]. Следовательно, метод toString () ведет себя как чистая функция, хотя он ею не является и служит чистым строковым представлением данного объекта. Объекты-значения просты и удобны для работы как в ФП, так и в ООП. А сочетая их с ключевым словом const, можно создавать объекты с такой же семантикой, как и символьных строк и чисел. Рассмотрим еще один пример реализации объекта-

[3] **Внутреннее состояние объекта может быть защищено, но его поведение по- прежнему подвержено модификации, поскольку имеется возможность динамически удалять или заменять любые его методы.**

значения:
```
function coordinate(lat, long) { let _lat = lat; let _long = long;
return { latitude: function () { return _lat;
        }, longitude: function () { return _long;
        } t
        translate: function (dx, dy) {
            return coordinate (_lat + dx, _long + dy) ; < . —
        }, toString: function () { return '(' + _lat + ', ' + _long + ')';
         } };
}
```
Возвратить новую копию с преобразованными координатами

```
const greenwich = coordinate(51.4778, 0.0015);
greenwich.toString(); // -> '(51.4778, 0.0015)'
```

Еще один способ реализовать неизменяемость — возвратить новые копии объектов, как это сделано выше в операции translate. В результате выполнения операции преобразования над данным объектом получается новый объект типа coordinate:

```
greenwich.translate(10, 10).toString(); // -> '(61.4778, 10.0015)'
```

Проектный шаблон "Объект-значение" возник под влиянием функционального программирования. И это еще один пример того, насколько изящно парадигмы ФП и ООП дополняют одна другую. Но, несмотря на то что данный проектный шаблон идеально подходит для ФП, его явно недостаточно для моделирования всей реальной предметной области. На практике в прикладном коде, скорее всего, потребуется обрабатывать иерархические данные, как, например, представленные ранее типы Person и Student, а также взаимодействовать с унаследованными объектами. Правда, в JavaScript имеется механизм для эмуляции такого поведения с помощью функции Ob j ect. f reeze ().

2.2.3. Глубокое замораживание подвижных частей

В новом для JavaScript синтаксисе классов не определены ключевые слова для отметки полей как неизменяемых. Тем не менее в JavaScript для этой цели поддерживается внутренний механизм, управляющий рядом скрытых метасвойств объекта вроде writable. Если установить в таком метасвойстве логическое значение false, доступная в JavaScript функция Object. freeze () может предотвратить изменение состояния объекта. В качестве первого примера ниже демонстрируется замораживание объекта person из листинга 2.1.

```
var person = Object.freeze(new Person('Haskell',
                                'Curry', '444-44-4444'));
person, firstname = 'Bob'; < --- Запрещно
```

В результате выполнения приведенного выше кода свойства объекта person становятся доступными только для чтения. И любая попытка изменить их (в данном случае — свойство _f irstname) приведет к следующей ошибке:

```
TypeError: Cannot assign to read only propertyfl
           '_firstname' of #<Person>
(Ошибка соблюдения типов: присваивание значения доступному только для чтения
свойству _firstname объекта типа Person запрещено)
```

С помощью функции Object. freeze () можно заморозить и наследуемые свойства. Например, замораживание экземпляра объекта типа Student происходит точно таким же образом: следуя по цепочке прототипов данного объекта, оно защищает все его свойства,

наследуемые от типа Person. По таким механизмом нельзя воспользоваться для замораживания вложенных свойств объектов, как показано на рис. 2.3.

В качестве еще одного примера приведено определение типа Address:

```
class Address {
  constructor(country, state, city, zip, street) { this._country = country;
      this._state = state;
      this._city = city;
      this._zip = zip;
      this._street = street;
  }
```

```
    get street() {
        return this._street; }

    get city() {
        return this._city;
    }

    get state () {
        return this._state; }

    get zip() {
        return this._zip;
    }

    get country() { return this._country;
    }
```

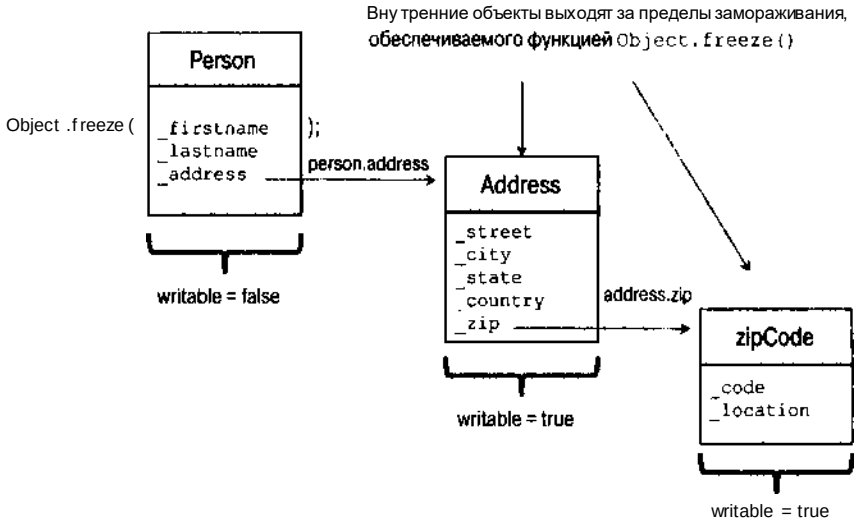


Рис. 2.3. Несмотря на то что объект типа Person заморожен, его внутренние свойства (например, _address) не были заморожены. Таким образом, вложенное свойство person. address. country подвержено изменениям. Такое замораживание оказывается мелким, поскольку заморожены только свойства верхнего уровня иерархии

И, к сожалению, никаких ошибок в следующем фрагменте кода не возникнет:

```
var person = new Person('Haskell', 'Curry', '444-44-4444');
person.address = new Address('US', 'NJ', 'Princeton',
                             zipCode('08544','1234'), 'Alexander St.');
```

```
    person = Object.freeze(person);

person.address.^country = 'France'; // -> Разрешено! person.address.country; // -
> 'France'
```

Функция Object. freeze () выполняет операцию неглубокого замораживания. В качестве обходного приема вложенную структуру объекта придется заморозить вручную, как показано в листинге 2.2.

Листинг 2.2. Рекурсивная функция для замораживания объекта

```
var isObject = (val) => val && typeof val === 'object';

function deepfreeze(obj) {
  if(isObject(obj)
     && ! Object. isFrozen (obj ) ) {

    Object.keys(obj). forEach(name => deepfreeze (obj [name] ) ) ;

    Object.freeze(obj);
```

Пройтись по всем свойствам незамороженного объекта и рекурсивно вызвать для них функцию Object.freeze() (подробнее о функции преобразования перечисляемых элементов return obj ; см. в главе 3).

Допустимы любые функции JavaScript, хотя формально функции в JavaScript могут быть модифицированы, основное внимание нужно уделить свойствам объектов

◄---

{-<—| Игнорировать уже замороженные объекты; заморозить те объекты, которые еще не заморожены

*0 > Вызывать себя рекурсивно (подробнее о рекурсии см. в главе 3)

Итак, мы рассмотрели ряд методик, с помощью которых можно усиливать степень неизменяемости в прикладном коде, но было бы нереалистично ожидать, что можно создавать целые приложения, вообще не изменяя их состояние. Следовательно, при создании новых объектов из первоначальных (например, с помощью функции coordinate. translate ()) чрезвычайно полезно установить строгие правила, пытаясь умерить сложности и хитросплетения в приложениях HaJavaScript. А теперь обсудим другой функциональный подход под названием *линзы* в качестве наилучшей альтернативы централизованному управлению неизменяемостью изменений в объектах.

2.2.4. Перемещение по графам объектов и их модификация с помощью линз

Обычно в ООП принято вызывать методы, изменяющие внутреннее содержимое сохраняющего свое состояние объекта. Недостаток такого подхода заключается в том, что нет никакой гарантии, что в результате извлечения состояния не нарушится функционирование какой-нибудь части системы, где ожидается, что объект останется в неизмененном состоянии. Можно, конечно, попытаться реализовать собственную стратегию *копирования при записи,* чтобы возвращать новые объекты из каждого вызываемого метода, хотя это трудоемкий и чреватый ошибками процесс. В таком случае простой метод-установщик из класса Person мог бы выглядеть следующим образом:

```
set lastname(lastname) {
    return new Person(this._firstname, lastname, this._ssn); •<----------------
'
```

Состояние всех свойств приходится вручную копировать в новый экземпляр, что просто ужасно!

А теперь представьте, что все это придется проделать с каждым свойством в отдельности каждого типа в вашей модели предметной области. Очевидно, требуется другое решение задачи модифицирования объектов, сохраняющих свое состояние, чтобы делать это неизменяемо, ненавязчиво и без жесткого повсеместного кодирования стереотипного кода. Таким решением являются *линзы,* иначе называемые *функциональными ссылками.* Они применяются в ФП для доступа к свойствам типов данных, сохраняющих свое состояние, чтобы манипулировать ими неизменяемым способом. Внутренний механизм действия линз аналогичен стратегии копирования при записи. Для этого используется внутренний компонент хранения, которому известно, как управлять состоянием надлежащим образом и копировать его. Этот механизм не нужно реализовывать самостоятельно, а можно воспользоваться функциональной библиотекой Ramda.js HaJavaScript (подробнее об этой и других библиотеках см. в приложении). По умолчанию функциональные средства библиотеки Ramda.js доступны через глобальный объект R. Вызвав метод R. lens Prop (), можно, например, создать линзы, заключающие в оболочку свойство lastname объекта типа Person, как показано ниже.

```
var person = new Person('Alonzo', 'Church', '444-44-4444'); var lastnameLens =
R.lenseProp('lastName');
```

А вызвав метод R. view (), можно прочитать содержимое данного свойства следующим образом:

```
R.view(lastnameLens, person); // -> 'Church'
```

По существу, это все равно, что вызвать метод get lastname (), что не особенно впечатляет. Когда же дело доходит до метода установки, тогда и начинается самое интересное. Так, если вызвать метод R, set (), как показано ниже, он создаст и возвратит совершенно новую копию объекта, содержащую новое значение и сохраняющую первоначальное состояние экземпляра. (Семантика копирования при записи не стоит ничего!)

```
var newPerson = R.set(lastnameLens, 'Mourning', person); newPerson.lastname; // -
> 'Mourning' person.lastname; //-> 'Church'
```

Линзы ценны тем, что они предоставляют вам ненавязчивый механизм манипулирования объектами, даже если они являются унаследованными или не находятся под вашим контролем. Кроме того, линзы поддерживают вложенные свойства, аналогичные свойству address из класса Person:

```
person.address = new Address('US', 'NJ', 'Princeton',
                             zipCode('08544','1234'), 'Alexander St.');
```

В качестве примера ниже показано, как создать линзу для перемещения по свойству address.zip.

```
var zipPath = ['address', 'zip'];                                    Определить по
var zipLens = R. lens (R. path (zipPath), R. assocPath (zipPath) ); ^— ведениеметодов
R.view(zipLens, person); // -> zipCode('08544', '1234')              новки*"^ и
```

Благодаря тому что линзы реализуют неизменяемые методы установки, имеется возможность изменять вложенные объекты и в то же время возвращать новый объект типа Person:

```
var newPerson = R.set(zipLens, zipCode('90210', '5678'), person); var
newZip = R.view(zipLens, newPerson);
                                // -> zipCode('90210', '5678') var originalZip =
R.view(zipLens, person);
                                // -> zipCode('08544', '1234') newZip.toString() !==
originalZip.toString(); // -> true
```

И это замечательно, поскольку теперь в вашем распоряжении имеется семантика методов получения и установки свойств объектов функциональным способом. Помимо того, что линзы предоставляют защитную неизменяемую оболочку, они очень хорошо согласуются с принятым в ФП принципом обособления логики доступа к свойствам от самого объекта, исключая необходимость полагаться на ссылку this и предоставляя эффективные функции, которым известно, как получить доступ к содержимому любого объекта и манипулировать им.

А теперь, когда стало понятно, как манипулировать объектами надлежащим образом, сменим тему, обратившись к функциям. Функции приводят в действие подвижные части приложения и составляют саму суть функционального программирования.

2.3. Функции

Функции являются основными единицами работы в функциональном программировании, а это означает, что вокруг них сосредоточено все остальное. *Функция*— это любое вызываемое выражение, которое может быть вычислено, если применить к нему операцию (). Функции могут возвращать вызывающему коду вычисляемое или неопределенное (undefined в функции типа void) значение. Л поскольку ФП действует во многом по принципам математики, то функции имеют смысл лишь в том случае, если они выдают *полезный,* а не пустой (null) или неопределенный (undefined) *результат.* В противном случае предполагается, что они модифицируют внешние данные и вызывают побочные эффекты. В целях изложения материала этой книги проводится различие между *выражениями* (т.е. функциями, выдающими значения) и *операторами* (т.е. функциями, *не* выдающими никаких значений). Императивное и процедурное программирование состоит, главным образом, из упорядоченных последовательностей операторов, а функциональное программирование — полностью из

выражений. Поэтому следует избегать функций, которые не отвечают назначению этой парадигмы программирования.

Функции в языке JavaScript обладают двумя важными характеристиками, образующими практическую основу его функционального стиля: они относятся к первому классу и высшему порядку. Обе эти характеристики функций будут подробно рассмотрены далее.

2.3.1. Функции первого класса

Термин *первый класс* употребляется в JavaScript потому, что функции, по существу, являются в этом языке объектами, которые также относятся к первому классу. Вам, вероятно, не раз встречались объявления функций, аналогичные следующему: `function multiplier(a,b) { return a * b; }`

Но в JavaScript предоставляются и другие возможности функций. Как и объекты, функции могут присваиваться:

- переменным в качестве анонимной функции или лямбда-выражения (подробнее о лямбда-выражениях речь пойдет в главе 3):

```
var square = function (x) { return x * x;
}                                    Анонимная функция

var square = x => x * x;  --------— Лямбда-выражение
```

- свойствам объектов в качестве методов:

```
var obj = {
    method: function (x) { return x * x; }
};
```

Если для вызова функции служит операция (), как, например, square (2), то функциональный объект вызывается следующим образом:

```
square;
// function (x) {
// return x * x;
// }
```

Имеется также возможность получать экземпляры функций через конструкторы. И хотя эта практика не особенно распространена, она лишний раз доказывает, что функции в JavaScript относятся к первому классу. Для получения экземпляра функции конструктору передается ряд формальных параметров и ее тело, а также указывается ключевое слово new:

```
var multiplier = new Function('a', 'b', 'return a * b');

multiplier(2, 3); // -> 6
```

Каждая функция BjavaScript является экземпляром типа Function. С помощью свойства функции length можно извлекать количество ее формальных параметров, а с помощью методов apply () и call () — вызывать функции вместе с их контекстами (подробнее об этом — в следующем разделе).

В правой части выражения анонимной функции указывается функциональный объект с

пустым свойством name. Используя анонимные функции, можно расширить или конкретизировать поведение функции, передав их в качестве аргументов. Рассмотрим в качестве примера встроенную BjavaScript функцию Array. sort (comparator), которой в качестве параметра передается функциональный объект компаратора. По умолчанию функция sort () преобразует сортируемые значения в символьные строки, используя их значения в Юникоде в качестве критерия сортировки. Но это ограничивает сортировку и не всегда отвечает конкретным потребностям. Ниже приведены два примера применения функции sort ().

```
var fruit = [ 'Coconut', 'apples'];              | Прописные буквы в Юникоде всегда
fruit, sort (); //->[' Coconut', 'apples'] ◄--- 1 предшествуют строчным буквам

var ages = [1, 10, 21, 2];                  | Числа преобразуются в символьные строки и
ages. sort(); // - > [ 1, 10, 2, 21] ◄——| сравниваются по их кодовым точкам в Юникоде
```

В итоге поведение функции sort () нередко определяют критерии, реализуемые в функции comparator (), которая сама по себе практически бесполезна. Указав специальный аргумент функции, можно добиться надлежащего выполнения операций числового сравнения, отсортировав людей по возрасту, как показано ниже.

```
people.sort(( pl, p2) => pl.getAgeO - p2.getAge());
```

Функции comparator () передается два параметра, pl и p2, со следующим контрактом.

- Если функция comparator () возвратит значение меньше нуля, то параметр pl должен следовать прежде параметра p2.
- Если функция comparator () возвратит нулевое значение, то порядок следования параметров pl и p2 остается без изменения.
- Если функция comparator () возвратит значение больше нуля, то параметр pl должен следовать после параметра p2.

Помимо того, что функции BjavaScript можно присваивать, им могут передаваться другие функции в качестве аргументов подобно упомянутой выше функции sort (). И в этом отношении они относятся к категории *функций высшего порядка.*

2.3.2. Функции высшего порядка

Если функции ведут себя как обычные объекты, то совершенно естественно предположить, что их можно передавать другим функциям в качестве аргументов и возвращать из других функций. Поэтому они называются *функциями высшего порядка.* Примером тому служит функция comparator (), которая передается функции Array. sort () в качестве аргумента. Рассмотрим вкратце другие примеры применения функций высшего порядка.

В приведенном ниже фрагменте кода демонстрируются функции, которые могу!' быть переданы другим функциям. В частности, функции apply Operation () передается два аргумента и любая операторная функция, выполняющая нужную нам операцию над этими аргументами.

```
function applyOperation(a, b, opt) {  ------ 1 Функцию opt () можно передать
    return opt (a, Ь) ;                      * в качестве аргумента другим функциям
}
```

```
var multiplier = (a, b) => a ★ b;

applyOperation(2, 3, multiplier); // -> 6
```

В следующем примере функции add () передается аргумент, а она возвращает функцию, которой, в свою очередь, передается второй аргумент, значение которого складывается с первым аргументом:

```
function add(a) {
    return function (b) { ◄----- j Функция,возвращаемая
        return a + b;             I из другой функции
    } } add(3)(3); // -> 6
```

Благодаря тому что функции в JavaScript относятся к первому классу и высшему порядку, они могут вести себя *как значения.* Это означает, что функция в данном случае оказывается не более чем *просто вычисленным значением,* которое однозначно (и поэтому неизменяемо!) зависит только от значений входных данных функции. Этот принцип соблюдается во всем, что приходится делать в функциональном программировании, особенно при связывании функций в цепочки, как поясняется в главе 3. При построении цепочек функций для указания на отдельные части программы, которые должны выполняться как части целого выражения, всегда используются имена функций.

Функции высшего порядка можно соединять вместе для составления осмысленных выражений из более мелких частей, упрощая тем самым многие программы, которые иначе было бы труднее написать. Допустим, требуется вывести список людей, проживающих в США. Первое решение этой задачи, вероятно, будет подобно следующему императивному коду:

```
function printPeoplelnTheUs(people) {
    for (let i = 0; i < people.length; i++) {
        var thisPerson = people[i];
        if (thisPerson.address. country === 'US') { I Вызвать метод toS tring ()
            console.log(thisPerson);          ------ * для каждого объекта
        }
    }
}                                              | Параметры pl, p2, и p3 являются
printPeoplelnTheUs ( [pl, p2, p3] );-<—! экземплярами типа Person
```

А теперь допустим, требуется обеспечить также вывод списка людей, проживающих в других странах. С помощью функций высшего порядка можно изящно абстрагировать действие, выполняемое для каждого человека (в данном случае — вывод сведений о нем на консоль). С этой целью функции высшего порядка printPeople () можно свободно передать любую функцию, выполняющую нужное действие, определяемое параметром action:

```
function printPeople(people, action) {
    for (let i = 0; i < people.length; i++) { action (people[i]);
    }
}

var action = function (person) { if(person.address.country = console.log(person);
                                    •US') {

printPeople(people,action);
```

Для таких языков, как JavaScript, характерен порядок обозначать имена функций пассивными существительными вроде multiplier, comparator и action. А поскольку они являются функциями первого класса, то их можно присваивать переменным и выполнять в последующие моменты времени. Ниже приведен реорганизованный вариант функции printPeople (), где в полной мере используются функции высшего порядка.

```
function printPeople(people, selector, printer) {
    people. forEach (function (person) { -<——i Функция f orEach() написана if (selector
        (person) ) {                          в функциональном стиле, что
            printer (person) ;                предпочтительнее организации циклов;
                                                подробнее об этом см. в следующей главе

    }); } var inUs = person => person.address.country === 'US';

printPeople(people, inUs, console.log);
```

Благодаря применению функций высшего порядка начинает проявляться декларативный характер кода. Из этого выражения совершенно ясно назначение программы

Именно такой тип мышления вам нужно выработать, чтобы полностью освоить функциональное программирование. В данном примере наглядно показано, что прикладной код стал намного более гибким, чем в первоначальном варианте, поскольку он позволяет быстро сменить (или настроить) критерии выбора, а также изменить место вывода получаемых результатов. Этой теме посвящены главы 3 и 4, где рассматривается также применение специальных библиотек для связывания операций в текучие цепочки и построения сложных программ из простых частей.

Забегая вперед

Сделаем краткий перерыв в обсуждении основных языковых средств JavaScript, чтобы еще больше усовершенствовать рассматриваемый в этом разделе пример программы, опираясь на некоторые понятия, которых мы вкратце коснулись ранее. Представленный здесь способ написания программ может показаться вам немного сложным, но вскоре вы научитесь писать свои программы подобным способом, применяя методики ФП. В частности, функции, обеспечивающие доступ к свойствам объекта, можно создать с

помощью линз следующим образом:

```
var countryPath = ['address', 'country'];
var country!. = R.lens(R.path(countryPath), R.assocPath(countryPath)); var
inCountry = R.curry((country, person) =>
      R.equals(R.view(country!, person), country));
```

Такой код выглядит намного более функционально, чем прежде:

```
people.filter(inCountry('US')).map(console.log);
```

Как видите, название страны становится еще одним параметром, который можно изменить как угодно. Подробнее об этом речь пойдет в последующих главах.

Функции в JavaScript не только вызываются, но и применяются. Рассмотрим подробнее эту характерную особенность механизма вызова функций BjavaScript.

2.3.3. Способы вызова функций

Механизм вызова функций BjavaScript является весьма интересной частью этого языка, отличаясь от аналогичных механизмов в других языках программирования. В языке JavaScript предоставляется полная свобода для определения динамического контекста, в котором вызывается функция. Речь идет о том, что в момент вызова функции можно изменить значение ссылки this, которая используется в теле функции. Функции BjavaScript можно вызывать самыми разными способами.

- **Как глобальную функцию.** В этом случае ссылка this устанавливается на глобальный объект (global) или вообще имеет неопределенное значение (undefined) в строгом режиме:

```
function doWorkO {
    this.myVar = 'Some value'; ◄— при вызове функции doWork()
}                                  как глобальной ссылка this будет
doWork () ;              <     .... указывать на глобальный объект
```

- **Как метод.** В этом случае ссылка this будет указывать на владельца метода. Это очень важная черта объектно-ориентированного характера JavaScript:

```
var obj = {
    prop: 'Some property',
```

```
        getProp: function () {return this.prop}
};
obj.getProp();
```

При вызове метода для объекта ссылка this будет указывать на владеющий этим методом объект

- **Как конструктор, предваряемый ключевым словом new.** В этом случае неявно возвращается ссылка на вновь созданный объект:

```
function MyType(arg) {
    this.prop = arg;
}

var someVal = new MyType('some argument');
```

При вызове функции с ключевым словом now ссылку this будет указывать на создаваемый и неявно возвращаемый объект

Как следует из приведенных выше примеров, в отличие о других языков программирования, значение ссылки this устанавливается в зависимости от того, каким образом используется функция (как глобальная, как метод объекта, как конструктор и т.д.), а не от ее лексического контекста (т.е. ее местоположения в исходном коде). Это может затруднить понимание исходного кода, поскольку в данном случае необходимо уделить пристальное внимание тому контексту, в котором выполняется функция.

Материал данного раздела включен в эту главу потому, что всякому разработчику приложений HaJavaScript очень важно знать, каким образом вызываются функции. Но, как не раз отмечалось ранее, ссылка this редко применяется в функциональном программировании, и на самом деле всегда стремятся избежать ее использования любой ценой. Тем не менее она интенсивно применяется разработчиками библиотек и инструментальных средств в особых случаях, когда требуется подчинить языковой контекст, чтобы добиться невероятных успехов. И с этой целью нередко применяются методы функций apply () и call ().

2.3.4. Методы функций

В языке JavaScript поддерживается вызов функций через их методы call () и apply (), подобные метафункциям и относящиеся к прототипу вызываемой функции. Оба метода широко применяются при построении каркасного кода с тем, чтобы пользователи интерфейса ЛР1 могли создавать новые функции из уже существующих. Ниже в качестве краткого примера приведено определение функции negate ().

```
function negate (func) {
    return function () {
        return ! func. apply (null, arguments) ;
    };
}

function isNull(val) { return
    val === null;
}
var isNotNull = negate (isNull) ;
// -> false isNotNull({}); // -> true
```

Создать функцию высшего порядка negate (), которой в качестве параметра передается другая функция и возвращающую еще одну функцию, отрицающую результат ее выполнения

1 Вызвать метод Funct ion. apply (), чтобы выполнить данную функцию с ее первоначальными аргументами

Определить функцию isNullO

Определить функцию isNotNull () как отрицание функции isNull () isNotNull(null);

Функция negate () создает новую функцию, запускающую ту функцию, которая была передана функции negate () в качестве аргумента, а затем логически отрицающую

полученный результат. В данном примере используется метод apply (), но с тем же успехом можно было бы применить и метод call (). Отличие заключается в том, что методу call () передается список аргументов, тогда как методу apply () — массив аргументов. Первым аргументом thisArg можно воспользоваться для изменения контекста функции в случае необходимости. Ниже приведены сигнатуры обоих методов.

```
Function.prototype.apply(thisArg, [argsArray])

Function.prototype.call(thisArg, argl,arg2,...)
```

Если аргумент thisArg ссылается на объект, то в нем хранится ссылка объект, для которого вызывается данный метод. А если аргумент thisArg принимает пустое значение null, то в качестве контекста функции задается глобальный объект, и она ведет себя как простая глобальная функция. Но если данный метод оказывается функцией в строгом режиме, то ему фактически передается пустое значение null.

Манипулирование контекстом функции через аргумент thisArg открывает путь для внедрения различных методик. Но в функциональном программировании это не поощряется, поскольку в нем вообще не учитывается состояние контекста (напомним, что функциям все данные передаются в качестве аргументов). Поэтому не будем больше тратить время и место на обсуждение подобной возможности.

Несмотря на то что бесполезно упоминать общий глобальный контекст или же контекст объекта в свете функционального характера JavaScript, имеется один особый контекст, который все же следует иметь в виду. Это контекст функции. Но чтобы понять его, нужно уяснить понятия замыканий и областей видимости.

2.4. Замыкания и области видимости

До появления JavaScript замыкания существовали только в тех языках ФП, которые находили специальное применение. И только BjavaScript замыкания были внедрены в массовую разработку, существенно изменив порядок написания кода. В качестве примера рассмотрим еще раз определение типа zipCode: `function zipCode(code, location) { let _code = code; let -location = location || ''; return { code: function () {`

```
        return _code;
    },
    location: function () { return -location;
    },
```

Внимательно проанализировав приведенный выше код, можно прийти к выводу, что функция zipCode () возвращает объектный литерал, у которого, по-видимому, имеется полный доступ к переменным, объявленным вне области его видимости. Иными словами, по завершении функции zipCode () получаемому в итоге объекту по-прежнему может быть доступна информация, объявленная в этой объемлющей его функции, как показано ниже.

```
const princetonZip = zipCode('08544', '3345');
princetonZip.code(); //-> '08544'
```

Это кажется несколько обескураживающим, но все происходит благодаря замыканию, которое образуется вокруг объявлений объектов и функций в JavaScript. Такой способ доступа к данным находит немало примеров практического применения, и в этом разделе мы рассмотрим применение замыканий для эмулирования закрытых переменных, извлечения данных из сервера и принудительного задания переменных с блочной областью видимости.

Замыкание — это структура данных, привязывающая функцию к ее окружению в момент ее объявления. Оно основывается на текстовом местоположении объявления функции и поэтому иначе называется *статической* или *лексической областью видимости,* окружающей определение функции. А поскольку замыкание предоставляет функции доступ к окружающему ее состоянию, оно делает исходный код ясным и удобочитаемым. Как будет показано ниже, замыкания служат важным подспорьем не только для работы с функциями высшего порядка в функциональных программах, но и для обработки событий и обратных вызовов, эмулирования закрытых переменных и обхода некоторых препятствий, скрытых BjavaScript.

Правила, определяющие поведение замыкания функции, тесно связаны с принятыми BjavaScript правилами соблюдения области видимости. В области видимости группируется ряд привязок переменных и определяется участок кода, где объявляются переменные. По существу, замыкание функции наследует области видимости подобно тому, как метод объекта получает доступ к переменным экземпляра, которые наследует этот объект. В обоих случаях имеются ссылки на родительские элементы. Замыкания нетрудно обнаружить во вложенных функциях. Ниже приведен краткий тому пример.

```
function makeAddFunction (amount) { function add (number) {
                                    ◄---- Функция add() лексически связана
                                          с функцией makeAddFunction ()
    return number + amount;               и получает доступ к переменной
                                          amount
```

```
return add; }

function makeExponentialFunction(base) {
    function raise (exponent) {
        return Math.pow(base, exponent);

    return raise;
}
var addTenTo = makeAddFunction(10);
addTenTo(10); // -> 20 var raiseThreeTo = makeExponentialFunction(3);
raiseThreeTo(2); // -> 9
```

Функция raise () лексически связана с функцией makeExponentialFunction() и получает доступ к переменной base

Следует особо подчеркнуть, что в данном примере переменные amount и base больше не находятся в активной области видимости обеих функций. Тем не менее они доступны из возвращаемой функции при ее вызове. По существу, вложенные функции add () и raise () можно рассматривать как такие, которые не только упаковывают свои вычисления, но и делают моментальные снимки всех окружающих их переменных. В более общем смысле, как показано на рис. 2.4, замыкание включает в себя следующее.

- Все параметры функции (в данном случае — params и params2).
- Все переменные из внешней области видимости, в том числе и все глобальные переменные, а также те переменные, которые были объявлены после функции (в данном случае — переменная additionalVars).

« Внешняя область видимости (глобальная) » function makerInner(params) {

« Внутренняя область видимости » return function inner(params2) {

Г<< Тело функции »

var additionalVars; }

Рис. 2.4. Замыкание включает в себя переменные, находящиеся во внешней (глобальной) области видимости, внутренней области видимости родительской функции, а также параметры родительской функции и дополнительные переменные, объявленные после объявления функции. В коде, образующем тело функции, могут быть доступны переменные и объекты, определенные в каждой из этих областей видимости. Все функции разделяют общую глобальную область видимости

В листинге 2.3 демонстрируется, каким образом действует замыкание.

Листинг 2.3. Действие замыкания

```
                                    Объявить глобальную
                                    переменную outerVar
var outerVar    'Outer'                                  Объявить переменную innerVar,
function makeInner(params) {                             являющуюся локальной для функции
    var innerVar = 'Inner'; <■                           makeInner()
    function inner() { ◄ -----   Объявить функцию inner (), в замыкание которой
      console.log(               включаются переменные outerVar и innerVar
          'I can see: ${outerVar}, ${innerVar}, and ${params}');

    return inner;
}                                   Вызвать функцию makeInner (), чтобы
                                    возвратить функцию inner ()
var inner = makeInner('Params'); ◄-
inner () ; ◄---- Функция inner () достоверна и благополучно переживает выполнение ее внешней функции
```

В результате выполнения исходного кода из листинга 2.3 на экран выводится следующее сообщение:

```
'I can see: Outer, Inner, and Params'
```

На первый взгляд, действие замыкания может показаться нелогичным и даже таинственным. Ведь локальные переменные (в данном случае — innerVar) обычно прекращают свое существование или собираются в "мусор" поле возврата из функции (в данном случае — makeInner ()), а следовательно, должен быть выведен неопределенный (undefined) результат. Тем не менее они продолжают существовать благодаря чудесному действию внутреннего механизма замыканий. В частности, функция, возвращаемая из функции makeInner (), запоминает все переменные из своей области видимости в тот момент, когда она объявляется, а также препятствует их утилизации. Глобальная область видимости также включается в это замыкание, обеспечивая тем самым доступ к переменной outerVar. Мы еще вернемся в главе 7 к замыканиям и тому, что входит в контекст функции.

В связи с изложенным выше может возникнуть вопрос: каким образом переменные (вроде additionalVars), объявляемые после объявления функции, могут быть также включены в ее замыкание? Для ответа на этот вопрос нужно знать, что BjavaScript имеются три формы определения области видимости: глобальная, область видимости функции и псевдоблока.

2.4.1. Трудности, связанные с соблюдением глобальной области видимости

Глобальная область видимости является простейшей, но и самой худшей формой определения области видимости. Любые объекты и переменные, объявленные на самом внешнем уровне сценария и не входящие ни в одну из функций, включаются в *глобальную область видимости* и доступны для всего кода JavaScript. Напомним, что в функциональном программировании преследуется цель предотвратить распространение любых наблюдаемых изменений за пределы функций. Но в глобальной области видимости каждая выполняемая строка кода приводит к видимым изменениям.

Пользоваться глобальными переменными соблазнительно, но они оказываются общими

для всех сценариев, загружаемых на странице, а это может легко привести к конфликтам пространств имен, если код JavaScript не упакован в модули. Засорение глобального пространства имен может вызвать осложнения, поскольку переменные и функции, объявленные в разных файлах, могут быть переопределены.

Глобальные переменные оказывают вредное влияние, затрудняя понимание программ, поскольку в этом случае приходится держать в уме состояние всех этих переменных в любой момент времени. И это одна из главных причин, по которым сложность программы возрастает по мере увеличения объема исходного кода. Наличие глобальных переменных способствует также проявлению побочных эффектов в функциях, поскольку при чтении или записи данных в них неизбежно образуются внешние зависимости. И уже теперь должно быть очевидно, что если писать код в стиле ФП, то употребления глобальных переменных следует избегать любой ценой.

2.4.2. Область видимости функций JavaScript

В языке JavaScript это наиболее предпочтительная форма определения области видимости. Любые переменные, объявленные в функции, оказываются локальными для нее и невидимыми (т.е. недоступными) больше нигде. Кроме того, при возврате из функции любые объявленные в ней локальные переменные удаляются. Так, в следующем примере:

```
function doWorkO {
     let student = new Student
     let address = new Address(...);
   // выполнить дополнительные действия
};
```

переменные student и address привязаны к функции doWor к () и недоступны за ее пределами. Как показано на рис. 2.5, имена переменных разрешаются аналогично именам в цепочке прототипов, как пояснялось ранее. Этот процесс начинается с внутренней области видимости и продолжается наружу. Механизм определения области видимости BjavaScript действует следующим образом.

1. Проверяется область видимости функции, в которой встретилось имя переменной.
2. Если значение переменной отсутствует в локальной области видимости, происходит переход к окружающей ее лексической области видимости, где ссылка на переменную ищется до тех пор, пока не будет достигнута глобальная область видимости.
3. Если значение переменной так и не было найдено, то возвращается неопределенное значение undefined.

Рассмотрим следующий пример кода: `var x = 'Some value';`

```
function parentFunction() { function innerFunction() { console.log(x);
    } return innerFunction;
}

var inner = parentFunction();
inner();
```

Когда вызывается функция inner (), интерпретатор JavaScript запускает процесс поиска значения переменной x в последовательности, приведенной на рис. 2.5.

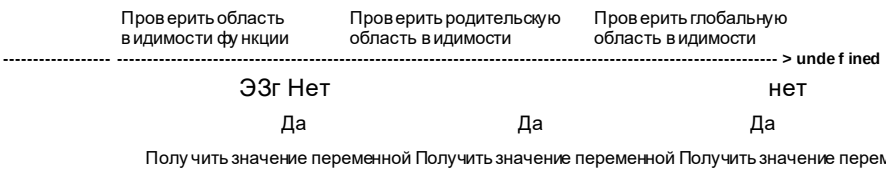


Рис. 2.5. Порядок разрешения имен, принятый в JavaScript. Сначала поиск значения переменной x производится в ближайшей, окружающей ее области видимости, а затем продолжается наружу. При этом сначала проверяется область видимости функции (т.е. локальная область видимости), затем происходит переход к родительской области видимости, если таковая существует, и, наконец, — к глобальной области видимости. Если переменная x не найдена, функция возвратит неопределенное значение undefined

Если у вас имеется опыт программирования на каком-нибудь другом языке, вы, вероятнее всего, привыкли пользоваться областью видимости функции. Но принимая во внимание C-подобный синтаксис языка JavaScript, вы вправе ожидать, что аналогичным образом действуют области видимости блока кода.

2.4.3. Область видимости псевдоблока

К сожалению, в стандарте ES5 языка JavaScript не поддерживается область видимости на уровне блока кода, заключаемого в фигурные скобки {} в таких управляющих структурах, как операторы for, while, if и switch. Исключением из этого правила является переменная ошибки, передаваемая блоку catch. Область видимости блока может соблюдаться в операторе with, но он больше не рекомендован для применения и исключается в строгом режиме. В других C-подобных языках переменная, объявляемая в блоке условного оператора if (в приведенном ниже примере — переменная myVar), недоступна за пределами этого блока кода.

```
if (someCondition) {
    var myVar = 10;
```

Это может смутить тех разработчиков, которые привыкли к подобному стилю программирования и только начинают осваивать JavaScript. В связи с тем что BjavaScript соблюдается исключительно область видимости функции, любые переменные, объявляемые в блоке кода, доступны в любом месте функции. Данное ограничение может стать настоящим кошмаром для программирующих HaJavaScript, но это препятствие все-таки можно преодолеть. Рассмотрим следующий пример кода: `function doWorkO { if (JmyVar) { var myVar = 10;`

```
    }
    console.log(myVar); //-> 10
} doWork();
```

В данном примере переменная myVar объявлена в блоке условного оператора if, тем не менее, она доступна за переделами этого блока. Но, как ни странно, в результате выполнения кода из данного примера на экран выводится значение 10. Такой результат может озадачить, особенно тех разработчиков, которые привыкли к более распространенной области видимости на уровне блока кода. Внутренний механизм JavaScript поднимает объявления переменных и функции до верхнего уровня текущей области видимости (в данном случае — области видимости функции). И это может сделать рискованным написание циклов. Обратите особое внимание на следующий пример исходного кода, приведенный в листинге 2.4.

Листинг 2.4. Проблема неоднозначности счетчика цикла

```
var arr = [1, 2, 3, 4] ; function processArrO {

    function multipleBylO(val) { i = 10; return val * i;
    }

    for(var i = 0; i < arr.length; i++) { arr[i] = multipleBylO(arr[i]);
    } return arr;
}

processArr(); // -> [10, 2, 3, 4]
```

В данном примере счетчик цикла i перемещен наверх функции multiple By 10 () и включается в ее замыкание. Забыв указать ключевое слово var в объявлении переменной i, мы не сможем создать эту переменную с локальной областью видимости в функции multipleBylO (), а значение счетчика цикла случайно изменится на 10. Сначала происходит поднятие объявления переменной i и установка в ней неопределенного значения undefined, а затем ей присваивается значение 0 при выполнении цикла. В главе 8 будет продемонстрировано повторение этой же проблемы неоднозначности при вычислении неблокирующих операций в циклах.

Отчасти разрешить подобные затруднения помогают хорошие интегрированные среды разработки (ИСР) и средства контроля качества кода, но и они оказываются бессильными перед сотнями строк кода. В следующей главе мы рассмотрим более совершенные решения, которые оказываются не только изящными, но и менее подверженными ошибкам, чем стандартные циклы. Это методики, в которых в полной мере используются функции высшего порядка, что помогает обойти подобные препятствия. В примерах из этой главы не раз демонстрировалось применение ключевого слова let, внедренного в стандарт ES6 B3biKaJavaScript. Оно помогает разрешить проблему неоднозначности счетчика цикла благодаря надлежащей привязке этого счетчика к охватывающему его блоку кода, как показано ниже.

```
for (let i = 0; i < arr. length; i++) {   ---- Ключевое слово let разрешает проблему
    // …                                        поднятия переменой i и правильно определяет
I                                            область ее видимости. За пределами цикла
                                             переменная 1 не определена i;  //  i  ===
```

```
undefined
```

Это шаг в правильном направлении и причина, по которой лучше пользоваться ключевым словом let, чем ключевым словом var при объявлении переменных с ограниченной областью видимости, хотя у организуемым вручную циклов имеются другие недостатки, о устранении которых речь пойдет в следующей главе. Итак, рассмотрев, что именно включается в замыкание функции и каким образом это взаимосвязано с механизмами определения областей видимости, перейдем к практическим примерам применения замыканий.

2.4.4. Практические примеры применения замыканий

Замыкания находят немало практических примеров применения, которые очень важно иметь в виду, реализуя крупные программы HaJavaScript. Они характерны не только для функционального программирования, хотя в них выгодно используется механизм функций JavaScript. Замыкания применяются в следующих целях.

- Эмулирование закрытых переменных.
- Организация асинхронных вызовов на стороне сервера.
- Эмулирование переменных с блочной областью видимости.

Эмулирование закрытых переменных

В отличие от JavaScript, во многих других языках программирования предоставляется встроенный механизм для определения внутренних свойств объекта с помощью модификаторов доступа вроде private. В языке JavaScript от

сутствует собственное ключевое слово для объявления закрытых переменных и функций, доступных только в области видимости объекта. Инкапсуляция может пойти на пользу неизменяемости, поскольку нельзя изменить то, что недоступно.

Но такое поведение можно сэмулировать с помощью замыканий. Характерным тому примером служит возврат объекта подобно тому, как это сделано в упоминавшихся ранее функциях zipCode () и coordinate (). Эти функции возвращают объектные литералы с методами, которые имеют доступ к любым переменным из внешних функций, но не раскрывают эти переменные, по существу, делая их локальными, или закрытыми.

Замыкания могут также предоставить способ управления глобальным пространством имен, чтобы избежать глобально видимых общих данных. Создатели библиотек и модулей находят лучшее применение замыканиям, скрывая закрытые методы и переменные в целых модулях. Это шаблон "Модуль" (Module), называемый так потому, что для инкапсуляции внутренних переменных в нем применяется *немедленно вызываемое функциональное выражение* (IIFE), но в то же время допускается экспортировать нужные функциональные средства во внешний мир и значительно сократить количество глобальных ссылок.

Примечание

В качестве общей нормы передовой практики весь функциональный код рекомендуется упаковывать в хорошо инкапсулированные модули. Все основные принципы функционального программирования, рассматриваемые в этой книге, могут быть вполне перенесены и на уровень модулей.

Ниже приведен краткий пример заготовки модуля[4].

```
var MyModule = (function MyModule(export) {  -<
    let _myPrivateVar = ...;

    export.methodi = function () { < //
    выполнить нужные действия };

    export ,method2 = function () { «<—
        // выполнить нужные действия
    };
   return export;
}(MyModule II {}));
```

Присвоить имя функции снова, чтобы в любых данных трассировки стека, выводимых в результате ошибки, можно было ясно определить выражение IIFE

т Закрытые переменные, недоступные за пределами I данной функции, но доступные для обоих методов

Методы, глобально экспортируемые в пределах области видимости объекта, что приводит к созданию псевдопространства имен

Одиночный объект, охватывающий частным образом все скрытое состояние и методы. В частности, метод methodi () можно вызвать по ссылке MyModule.methodi()

[4] **Более подробно различные типы шаблонов модулей поясняются в статье** *JavaScript Module Pattern: In-Depth,* **опубликованной Беном Черри (Ben Cherry) в его блоге** *Adequately Good* **12 марта 2010 года** (http://www.adequatelygood.com/JavaScript-Module- Pattern-In-Depth.html).

Объект MyModule создается глобально и передается функциональному выражению, составляемому с помощью ключевого слова function и выполняемому сразу же после загрузки сценария. Благодаря определению области видимости BjavaScript переменная myPrivateVar и любые другие закрытые переменные оказываются локальными для охватывающей функции. Именно замыкание, охватывающее оба экспортируемых метода, позволяет объекту благополучно получать доступ ко всем внутренним свойствам модуля. Это необходимо для того, чтобы свести к минимуму следы глобальной деятельности и в то же время раскрыть объект с инкапсулированным в значительной степени состоянием и поведением. Такой шаблон модуля был внедрен во всех функциональных библиотеках, упоминаемых в этой книге.

Организация асинхронных вызовов на стороне сервера

Функции первого класса и высшего порядка можно передавать BjavaScript другим функциям в виде обратных вызовов. Обратные вызовы используются в качестве перехватчиков событий для их обработки малозаметным способом. Допустим, требуется сделать запрос к серверу с уведомлением о получении ответных данных. Традиционный подход к решению этой задачи состоит в том, чтобы предоставить функцию обратного вызова, которая обработает запрос, как показано ниже.

```
getJSON('/students', (students) => {
                grades =>                                  Обработать оба ответа
        getJSON('/students/grades',                                  Обработать ошибки
                processGrades(grades),                               извлечения оценок
                error => console.log(error.message));*<-
    }, (error) => console.log(error.message)
                                                  I Обработать ошибки извлечения 1
                                                  данных об учащихся
```

Функция get JSON () относится к категории высшего порядка. В качестве аргументов ей передаются две функции обратного вызова: для обработки запроса при удачном исходе и обработки ошибок при неудачном исходе. Общая закономерность, наблюдаемая в связи с асинхронным кодом и обработкой событий, заключается в том, что глубоко вложенные вызовы функций могут легко завести в тупик любого квалифицированного разработчика. Это приводит к неприятной проблеме “неизбежной пирамиды обратных вызовов”, когда требуется сделать несколько последовательных запросов к серверу. Вы, вероятно, по собственному опыту знаете, насколько трудно отследить глубоко вложенный код. Поэтому в главе 8 будут представлены нормы передовой практики, позволяющие преобразовать такой код в более текучие и декларативные выражения, которые допускают связывание в цепочку вместо вложения.

Эмулирование переменных с блочной областью видимости

Используя замыкания, можно предоставить альтернативное решение проблемы неоднозначности счетчика цикла, демонстрировавшейся в примере из листинга 2.4. Как упоминалось ранее, в основе этой проблемы лежит отсутствие BjavaScript семантики для области видимости блока, и поэтому ее приходится каким-то образом создавать (или эмулировать) искусственно. С помощью ключевого слова let можно разрешить многие затруднения, возникающие в традиционном механизме организации циклов, но

функциональный подход заключается в том, чтобы выгодно воспользоваться замыканиями и определяемой BjavaScript областью видимости, а также рассмотреть возможность для применения функции f orEach (), как показано ниже. Вместо того чтобы заботиться о привязке счетчика цикла и других переменных к области видимости, можно заключить тело цикла в оболочку, сэмулировав блок области видимости функции в операторе цикла. Как поясняется далее, это поможет вызвать асинхронное поведение при переборе коллекций.

```
arr.forEach(function(elem, i) {

});
```

В этой главе были рассмотрены лишь самые основы JavaScript, чтобы помочь вам лучше понять присущие этому языку ограничения, когда он применяется функционально, а также подготовить вас к освоению методик ФП, представленных в последующих главах книги. Если вам требуется более основательное представление о JavaScript, на эту тему имеется довольно обширная литература, где подробно поясняются такие понятия, как объекты, механизм наследования и замыкания.

Желаете стать ниндзя JavaScript?

Усвоение рассмотренных в этой главе понятий объектов, функций, определения области видимости и замыканий играют важную роль в становлении знатока JavaScript. Но здесь они были рассмотрены лишь в общих чертах, чтобы подготовить почву, уравняв тем самым шансы читателей всех категорий и уделив основное внимание только функциональному программированию в остальных главах книги. А для того чтобы изучить JavaScript более основательно и стать настоящим знатоком этого языка, достигнув уровня ниндзя, рекомендуется прочитать второе издание книги *Секреты JavaScript ниндзя* Джона Резига, Беэра Бибо и Иосипа Мараса, издательство Manning, 2016 г. (www.manning.com/books/ secrets-of-the-javascript-ninja-second-edition). Ее перевод на русский язык вышел в издательстве "Диалектика" в 2017 г, ISBN 978-5-9908911-8-0.

Итак, заложив прочный фундамент знаний по JavaScript, в следующей главе мы перейдем к рассмотрению вопросов обработки данных с помощью таких распространенных операций, как map, reduce, filter, а также рекурсии.

Резюме

Из этой главы вы узнали следующее.

- Язык JavaScript имеет универсальный характер и обладает эффективными средствами как для ООП, так и для ФП.
- Реализация неизменяемости в ООП позволяет изящно сочетать эту парадигму программирования с ФП.
- Функции высшего порядка и первого класса BjavaScript обеспечивают прочную опору для написания кода HaJavaScript в функциональном стиле.
- Замыкания находят немало примеров практического применения для сокрытия информации, разработки модулей и передачи параметризиро- ванного поведения

более крупным функциям через разные типы данных.

Погружаемся в функциональное программирование

В части I были даны ответы на два самых главных вопроса: зачем программировать в функциональном стиле и почему для этой цели подходит язык JavaScript? А теперь, когда стало ясно, что именно функциональное программирование вносит в разработку приложений HaJavaScript, вам предстоит подняться на несколько ступенек выше в освоении ФП. В части II обсуждаются все понятия и методики, которые потребуются вам для решения практических задач с применением функционального программирования. Из этой части вы узнаете, что означает получить навыки функционального программирования.

В главе 3 бегло рассматриваются некоторые универсальные функциональные программы, в которых применяются декларативные абстракции вроде map, reduce и filter с целью создать легко усваиваемый код. В ней обсуждается также применение рекурсии как средства для выполнения обхода разных форм данных в функциональном стиле.

В главе 4 понятия, рассмотренные в главе 3, применяются для построения из функций конвейеров с целью ускорить разработку и писать код в бесточечном стиле. Из этой главы вы узнаете, что самое главное для построения функционального кода — разбить сложные задачи на более мелкие независимые составляющие, которые можно соединять вместе, чтобы составлять целые решения по принципу композиционности. В итоге получается модульная и повторно используемая кодовая база.

И, наконец, в главе 5 будут представлены основополагающие проектные шаблоны, предназначенные для борьбы с растущей сложностью приложений и обработки ошибок. Композиция функций становится более надежной и прочной благодаря абстрактным типам данных вроде функторов и монад, обеспечивающих такой уровень абстракции, который делает код устойчивым к сбоям и исключительным ситуациям.

Применение методик, представленных в части II, позволит вам полностью преобразить свой стиль написания кода HaJavaScript. Кроме того, в этой части подготавливается почва для перехода к части III, посвященной применению методик функционального программирования для решения более сложных задач HaJavaScript, включая обработку асинхронных данных и событий.

Меньше структур данных и больше операи^и

В этой главе...

- Общее представление о потоке управления программой
- Эффективный анализ кода и данных
- Раскрытие истинного потенциала операций map, reduce и filter
- Знакомство с цепочками и библиотекой Lodash.js
- Умение мыслить рекурсивно

> **Вычислительные процессы являются абстрактными существами, населяющими компьютеры. В ходе своего развития процессы манипулируют другими абстрактными сущностями, называемыми данными.**
>
> Из книги *Structure and Interpretation of Computer Programs* (Структура и интерпретация компьютерных программ) *Гарольда Абелъсона (Harold Abelson)* и *Джеральда Джея Сассмана (Gerald Jay Sussman),* издательство MIT Press, 1979 г.

В части I этой книги были достигнуты две важные цели: во-первых, научить вас мыслить функционально и представить инструментальные средства, требующиеся для функционального программирования; а во-вторых, дать краткий обзор самых основных языковых средств JavaScript, в том числе функций высшего порядка, которые будут часто применяться в примерах из этой и остальных глав. Теперь, когда вы знаете, как сделать функции чистыми, настало время научиться связывать их вместе.

В этой главе представлен ряд практически полезных операций вроде tap, reduce и filter, которые позволяют выполнять обход и преобразование структур данных в последовательном порядке. Эти операции настолько важны, что они так или иначе применяются буквально во всех функциональных программах. Они упрощают также исключение организуемых вручную циклов из прикладного кода, поскольку большинство циклов являются лишь частными случаями применения этих операций.

Из этой главы вы узнаете также, как пользоваться библиотекой Lodash.js на JavaScript, которая позволяет обрабатывать и понимать не только структуру приложения, но и структуру данных. В этой главе обсуждается также важная роль, которая принадлежит рекурсии в функциональном программировании, а также те преимущества, которые дает умение мыслить рекурсивно. Опираясь на эти понятия и принципы, вы научитесь писать краткие, выразительные и декларативные программы, где поток управления ясно отделяется от основной логики программы.

3.1. Общее представление о потоке управления прикладной программой

Путь, который программа проходит к своему решению, называется *потоком* ее *управления.* В императивной программе поток ее управления описывается довольно подробно с раскрытием всех стадий, необходимых для решения задачи, поставленной перед программой. Эти стадии обычно включают в себя немало циклов и ветвлений, а также переменные, изменяемые каждым оператором. В самых общих чертах простую императивную программу можно описать следующим образом:

```
var loop = optC(); while(loop) {
    var condition = optA();
    if(condition) {
      optBl();
    }
    else {
        optB2();
    }
    loop = optC();
} optDO ;
```

На рис. 3.1 показана простая блок-схема данной программы.

А в декларативных программах, особенно функциональных, уровень абстракции заметно повышается благодаря применению минимально структурированного потока, состоящего из независимых операций, заключаемых в “черный ящик” и связываемых вместе в простую топологию. Эти связанные вместе операции являются ничем иным, как функциями высшего порядка, не-

реносящими состояние из одной операции в другую, как показано на рис. 3.2. Функциональный порядок обработки таких структур данных, как массивы, способствует именно такому стилю разработки, при котором данные и поток управления программой рассматриваются как простые связи между компонентами высокого уровня.

Рис. 3.1. Императивная программа, состоящая из последовательного ряда операций (или операторов), выполняемых в ветвлениях и циклах

Рис. 3.2. Функциональное управление связанных вместе операций, заключаемых в "черный ящик". Информационные потоки проходят независимо от одной операции к другой, причем операции являются отдельными чистыми функциями. При такой организации потока управления программой ветвления и циклы, по существу, сводятся к минимуму или вообще исключаются в пользу абстракций

В конечном итоге код получается более или менее похожим на следующий:

```
optA().optB().optC().optD();
```

Соединение через точки подразумевает наличие общего объекта, содержащего эти методы

Такой порядок связывания операций в цепочку приводит к тому, что программы становятся лаконичными, текучими и выразительными, что позволяет отделить поток управления программой от логики выполняемых в ней вычислений. Таким образом, прикладной код и данные легко поддаются эффективному анализу.

3.2. Связывание методов с цепочку

Связывание методов в цепочку является шаблоном ООП, позволяющим вызывать несколько методов в одном операторе. Если же все эти методы принадлежат одному и тому же объекту, в таком случае связывание методов с цепочку называется *каскадированием методов,* И хотя этот шаблон наблюдается, главным образом, в объектно-ориентированных приложениях, при определенных уело- виях, например, при манипулировании неизменяемыми объектами, он оказывается пригодным и для функционального программирования. Но поскольку модификация объектов в функциональном коде запрещается, то возникает вопрос: как такое вообще возможно? Для ответа на этот вопрос рассмотрим следующий пример манипулирования символьными строками:

```
'Functional Programmingsubstring(0, 10).toLowerCase() + ' is fun';
```

В данном примере методы substring () и toLowerCase () служат для обработки символьных строк, оперируя (по ссылке this) владеющим ими строковым объектом и возвращая новые символьные строки. Операция сцепления символьных строк, обозначаемая знаком +, перегружается в JavaScript ради синтаксического удобства и также производит новую строку. В результате выполнения указанных выше преобразований получается символьная строка, никак не связанная с первоначальной строкой, которая остается незатронутой. И этого следовало ожидать, поскольку символьные строки по определению неизменяемы. С точки зрения ООП это само собой разумеется, но с точки зрения ФП это идеально, поскольку для обработки символьных строк не требуются линзы.

Если реорганизовать приведенную выше строку кода в более функциональном стиле, то она примет следующий вид:

```
concat(toLowerCase(substring('Functional Programming', 1, 10))), '
                                is fun');
```

Этот код следует принципиальному положению ФП о том, что все параметры должны быть явно определены в объявлении функции. В этом коде отсутствуют побочные эффекты и не модифицируется первоначальный объект. Бесспорно, написание этого кода навыворот не делает его таким же текучим, как связывание методов в цепочку. Такой код намного менее удобочитаем, поскольку приходится постепенно снимать оболочку с каждой функции, чтобы понять, что же на самом деле происходит в этом коде.

Связывание в цепочку методов, относящихся к одному экземпляру объекта, находит свое применение в функциональном программировании, при условии, что соблюдается правило внесения изменений. Но было бы неплохо приспособить этот шаблон для применения к массивам. Поведение, наблюдаемое в символьных строках, было расширено и для обработки массивов BjavaScript, но оно мало кому известно, и поэтому многие прибегают к бесхитростным циклам for.

3.3. Связывание функций в цепочку

Наследование применяется в объектно-ориентированных программах в качестве основного механизма для повторного использования кода. Как было показано в предыдущей главе, тип Student наследует от типа Person все состояние и методы как порожденный. Такой образец наследования можно наблюдать, главным образом, более отчетливо в объектно-ориентированных языках, особенно в их реализациях структур данных. Например, в языке Java имеется целая иерархия конкретных классов типа List для всяких потребностей: ArrayList, LinkedList, DoublyLinkedList, CopyOnWriteArrayList и прочие классы, реализующие основной интерфейс List и производные от общих родительских классов, в которых вносятся собственные функциональные возможности.

А в функциональном программировании принят другой подход. Вместо того чтобы создавать классы для новых структур данных с целью удовлетворить конкретным потребностям, в ФП применяются общие структуры данных вроде массивов и выполняемые над ними крупные операции высшего порядка, которые не зависят от базового представления данных. Такие операции предназначены для выполнения следующих

действий.

- Принятие аргументов функций с целью внедрить специальное поведение, позволяющее решить конкретную задачу.
- Замена традиционных механизмов, предназначенных для организации циклов вручную, предполагающих модификацию временных переменных и вызывающих побочные эффекты. Благодаря этому упрощается сопровождение кода и становится меньше мест, где могут возникнуть ошибки.

Рассмотрим эти особенности операций более подробно. В примерах из этой главы применяется приведенная ниже коллекция объектов типа Person. Ради краткости в ней объявлены лишь четыре объекта, но те же самые принципы применимы и к более крупным коллекциям.

```
const pl = new Person(’Haskell’, 'Curry*, '111-11-1111'); pl.address = new
Address('US');
pl.birthYear = 1900;

const p2 = new Person(*Barkley*, 'Rosser', '222-22-2222');
p2.address = new Address('Greece');
p2.birthYear = 1907;

const p3 = new Person('John', 'von Neumann', '333-33-3333');
p3.address = new Address('Hungary');
p3.birthYear = 1903;

const p4 = new Person('Alonzo', 'Church', '444-44-4444');
p4.address = new Address('US');
p4.birthYear = 1903;
```

3.3.1. Общее представление о лямбда-выражениях

Лямбда-выражения впервые начали применяться в функциональном программировании, а BjavaScript они известны под названием *стрелочных функций*. Они служат для написания однострочных анонимных функций с более

кратким синтаксисом, чем у традиционного объявления функции. И хотя лямбда-выражения могут быть многострочными, чаще всего они применяются в однострочной форме, как демонстрировалось в примерах из главы 2. Выбор синтаксиса лямбда-выражений или традиционных объявлений функций зависит от удобочитаемости прикладного кода, но внутренний механизм их действия одинаков. Ниже приведен пример объявления с помощью синтаксиса лямбда-выражений простой функции, предназначенной для извлечения имени конкретного лица.

```
const name - (p) => p.fullname;
console.log(name(pl)); // -> 'Haskell Curry'
```

Краткое обозначение (p) => p. full name служит лишь для синтаксического удобства объявления функции, которой передается единственный параметр p и неявно возвращающей ссылку на свойство p. fullname. Структура этого синтаксического нововведения наглядно показана на рис. 3.3.

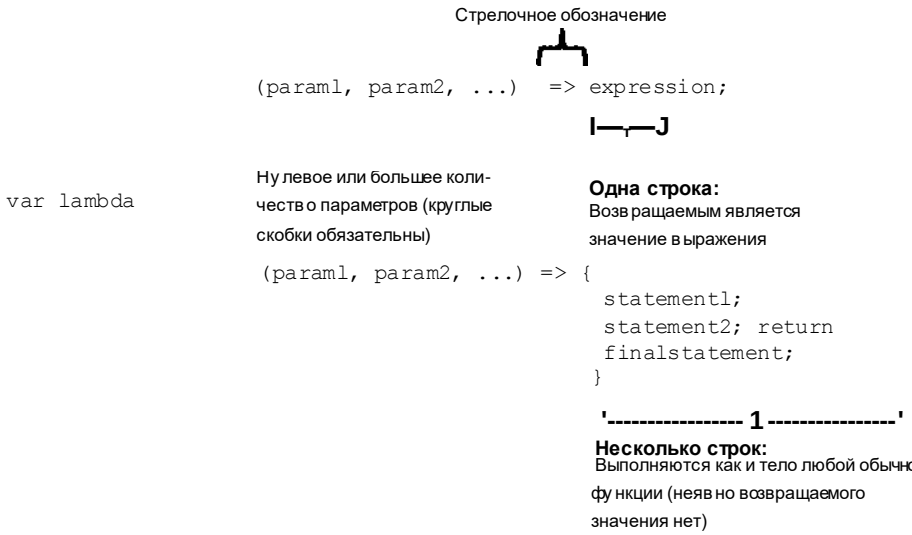


Рис. 3.3. Разбор структуры стрелочных функций. В правой части лямбда-выражения указывается единственное выражение или ряд операторов, заключаемых в фигурные скобки

Лямбда-выражения поддерживают функциональный характер определения функции, поскольку они побуждают всегда возвращать значение. Для однострочных выражений возвращаемое значение на самом деле получается из значения, вычисляемого в теле функции. Следует также обратить внимание на взаимосвязь между функциями первого класса и лямбда-выражениями. В приведенном выше примере ссылка name указывает не только на конкретное значение, но и (отложенно) на описание порядка его получения. Иными ело- вами, ссылка name указывает на стрелочную функцию, которой известно, как вычислить указанные данные. Именно поэтому в функциональном программировании можно пользоваться функциями как значениями. Данный принцип ФП более подробно рассматривается далее в этой главе, а функции с отложенными вычислениями, буквально

называемыми “ленивыми”, — в главе 7, “Оптимизация функционального кода”.

Более того, ФП способствует применению трех главных функций высшего порядка, реализующих операции map, reduce и filter. Эти функции предназначены для применения вместе с лямбда-выражениями. Большая часть функционального кода JavaScript основывается на обработке списков данных, откуда и происходит название первоначального языка функционального программирования LISP (list processing, т.е. обработка списков) — предшественника JavaScript. В версии JavaScript 5.1 предоставляются собственные варианты этих функций, называемые *дополнениями к массивам,* но для получения готовых решений, которые могут включать в себя другие подобные типы операций, рекомендуется пользоваться их реализациями из функциональной библиотеки Lodash.js. В этой библиотеке предоставляются важные средства, способствующие написанию функциональных программ, а также богатый набор служебных функций для решения многих типичных задач программирования. Установив библиотеку Lodash.js (подробнее об этом. см. в приложении к данной книге), можно получить доступ к ее функциональным средствам через глобальный объект с префиксомИтак, начнем с операции tap.

Знак подчеркивания в библиотеке Lodash

В библиотеке Lodash знак подчеркивания употребляется потому, что ее разработка была начата как ответвление известного проекта широко распространенной библиотеки Undesrscore.js (http://underscorejs.org/). В библиотеке Lodash до сих пор можно найти следы интерфейса API из библиотеки Underscore, причем до такой степени, что он может служить в качестве прямой замены. Но внутренний механизм библиотеки Lodash полностью переписан в пользу более изящных способов составления функций в цепочки и некоторых приемов повышения производительности, как поясняется в главе 7.

3.3.2. Преобразование данных с помощью операциитар

Допустим, требуется преобразовать все элементы крупной коллекции данных. Так, если имеется список объектов, представляющих учащихся, то из него необходимо извлечь Ф.И.О. каждого учащегося. Для решения подобных задач вам, вероятно, не раз приходилось писать следующую последовательность операторов:

```
var result = [];
var persons = [p1, p2, p3, p4];
for(let i = 0; i < persons.length; i++) {
     var p = persons[i];
```

```
    if(p !== null && p !== undefined) {
    result.push(p.fullname);Ч ■— }
}
```

При императивном подходе предполагается, что fullname обозначает метод из класса Student

Операция tap, иначе называемая collect, реализуется в виде функции высшего порядка, применяющей функцию-итератор для перебора каждого элемента массива по очереди и возвращающей новый массив равной длины. Ниже приведено решение той же самой задачи, но на этот раз в функциональном стиле с помощью операции _.tap.

```
.tap (persons,
    s => (s !== null && s !== undefined) ? s.fullname : ' ' <----
,
```

Используя функции высшего порядка, удалось исключить все объявления переменных

Формальное определение данной операции выглядит следующим образом: `map(f, [dO, dl, d2...]) -> [rO, rl, r2...]; где: f(dn) = rn`

Операция map особенно удобна для синтаксического анализа целых коллекций элементов, не прибегая к необходимости организовывать цикл или разрешать затруднения, связанные с нетипичным определением области видимости. Кроме того, данная операция является неизменяемой, поскольку в результате ее выполнения получается совершенно новый массив. Операции tap передается в качестве параметров функция f и коллекция из п элементов. В результате возвращается новый массив длиной п, элементы которого получаются в результате применения функции f к каждому элементу коллекции слева направо, как показано на рис. 3.4.

Рис. 3.4. В операции **tap** функция-итератор f применяется к каждому элементу исходного массива и возвращается новый массив равной длины

В приведенном выше примере операция _. tap служит для перебора массива объектов, представляющих учащихся, чтобы извлечь из него их Ф.И.О. В качестве функции-итератора в данном случае служит лямбда-выражение, и это довольно типичное явление. При этом исходный массив не изменяется, а возвращается новый массив, содержащий следующие элементы:

```
['Haskell Curry', 'Barkley Rosser', 'John von Neumann', 'Alonzo Church']
```

Рассмотрим, каким образом можно реализовать операцию _. tap, поскольку всегда полезно знать, что происходит на один уровень ниже абстракции. В листинге 3.1 приведен исходный код для реализации данной операции.

Листинг 3.1. Реализация операции преобразования

Этой функции передается в качестве параметров функция-итератор и массив. Заданная функция применяется к каждому элементу массива и возвращается новый массив такой же длины, как и у исходного массива

```
function map(arr, fn) { ◄----
    const len = arr.length,
                        result = new Array(len);
    for (let idx = 0; idx < len; ++idx) { result[idx]
    = fn(arr[idx], idx, arr);< }
    return result;
```

: Результат: массив такой же J длины, как и исходный массив

Применить функцию f n к каждому элементу исходного массива и сохранить результат в новом массиве

Как видите, внутренне операция .тар основывается на стандартных циклах. Она выполняет рутинную операцию по обходу элементов массива вместо вас. От вас же требуется только обеспечить требуемую функциональность для обработки каждого элемента массива в функции-итераторе. Вам больше не нужно каждый раз писать один и тот же код, к котором инкрементируется переменная цикла и проверяются границы массива. Это наглядный пример того, как функциональные библиотеки переносят прикладной код на уровень языков программирования, имеющих более отчетливый функциональный характер.

Операция тар выполняется исключительно слева направо. Если же требуется выполнить ее справа налево, то для этого придется сначала обратить массив, как показано ниже. Ради совместимости функция .reverse () из библиотеки Lodash совместима с предоставляемой BjavaScript функцией Array, reverse (), но это означает, что она модифицирует исходный массив на месте. Не следует, однако, упускать из виду случаи, когда у функций могут быть побочные эффекты.

```
_(persons).reverse().map(
    p => (p !== null && p !== undefined) ? p.fullname : '' ) ;
```

Обратите внимание на применение несколько иного синтаксиса в данном примере кода. В библиотеке Lodash предоставляется изящный, ненавязчивый способ интегрировать ее в прикладной код. Для манипулирования объектами их достаточно заключить в круглые скобки, употребив обозначение _(...). Это дает возможность получить полный контроль над всем арсеналом эффективных функциональных средств данной библиотеки, чтобы выполнять любые требующиеся преобразования.

Применение функции преобразования к контейнерам

Принцип, по которому функция применяется к структуре данных (в данном случае — к массиву) для преобразования составляющих ее значений, имеет далеко идущие последствия. Как будет показано в главе 5, "Проектные шаблоны и сложность", функцию можно применить для преобразования любого объекта, а не только массива.

Итак, научившись применять функцию преобразования к исходным данным, полезно также уметь делать выводы или извлекать определенные результаты, исходя из новой структуры. Для этой цели служит функция, реализующая операцию reduce.

3.3.3. Получение результатов с помощью операции reduce

Теперь вы знаете, как преобразовывать исходные данные. Но как извлечь из этого существенные результаты? Допустим, требуется определить страну с наибольшим количеством объектов типа Person, подсчитанных в коллекции. С этой целью можно воспользоваться функцией, реализующей операцию reduce.

Операция reduce реализуется в виде функции высшего порядка, сводящей массив элементов к единственному значению. Это значение вычисляется из накапливаемого результата вызова функции для каждого элемента массива, которой передается в качестве параметра значение аккумулятора, как наглядно демонстрирует схема, приведенная на рис.3.5.

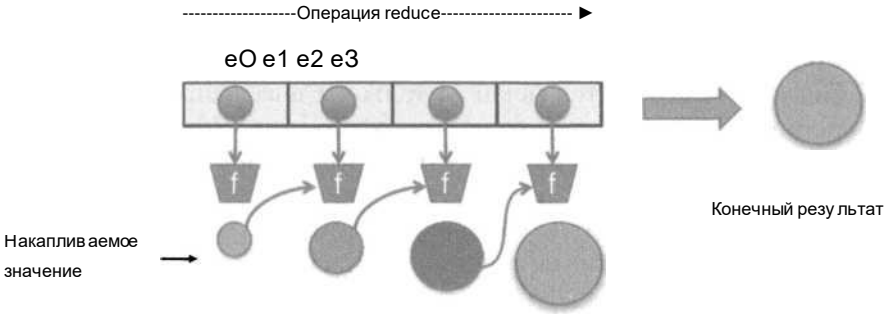


Рис. 3.5. Сведение массива к единственному значению. На каждом шаге итерации возвращается значение, накапливаемое исходя из предыдущего результата. Это накапливаемое значение хранится до тех пор, пока не будет достигнут конец массива. Окончательным результатом операции reduce всегда оказывается единственное значение

Более формально схема на рис. 3.5 может быть выражения следующим образом:

```
reduce(f, [e0, e1, e2, e3],ассиш) -> f(f(f(f(acc, e0), e1, e2, e3)))) -> R
```

В листинге 3.2 приведена упрощенная реализация внутреннего механизма выполнения операции reduce.

Листинг 3.2. Реализация операции сведения

```
function reduce(arr, fn, accumulator) {
let idx = -1, len = arr.length;
```

```
if (!accumulator && len > 0) { accumulator = arr[++idx];
}
return accumulator; «<■
```

Если значение аккумулятора не указано, использовать для его инициализации первый элемент массива

Возвратить единственное накопленное значение

```
while (++idx < len) { accumulator
    = fn(accumulator, arr[idx],
    idx, arr);
}
```

Вызвать функцию fn для каждого элемента массива, передав ей накапливаемое значение, а также значение текущего элемента массива

Функции, реализующей операцию reduce, передаются следующие параметры:

- fn — функция-итератор, вызываемая для каждого элемента массива, которой в качестве параметров передаются: накапливаемое значение, текущее значение в массиве, индекс массива и сам массив.
- accumulator — начальное значение аккумулятора, которое используется затем для хранения накапливаемого результата и передается при каждом последующем вызове функции.

Рассмотрим в качестве примера программу, собирающую статистические данные о ряде объектов типа Person. Допустим, требуется подсчитать количество людей, проживающих в некоторых странах. В листинге 3.3 приведен исходный код программы, решающей эту задачу.

Листинг 3.3. Подсчет количества людей, проживающих в некоторых странах

Извлечь страну проживания отдельного лица

Создать элемент массива для каждой страны, инициализировав его значением 1, а затем инкрементируя его при подсчете каждого лица, проживающего в данной стране

```
(persons).reduce((stat, person) => { const country =
person.address.country; ◄— stat[country] =
_.isUndefined(stat[country] ) stat[country] + 1;
```

Возвратить накопленный объект

```
  return stat;-<
```

Начать процесс сведения с пустого объекта (инициализировать аккумулятор)

В результате выполнения приведенного выше кода входной массив преобразуется в единственный объект, содержащий представление количества лиц, проживающих в отдельных странах:

```
'US' : 2, 'Greece' : 1, 'Hungary': 1
```

Чтобы еще больше упростить решение данной задачи, можно реализовать повсеместно употребляемое сочетание операций шар и reduce. Связывая вместе функции, реализующие эти операции, можно расширить их поведение до специальных его видов, предоставив последние в качестве параметров. Так, поток управления рассматриваемой здесь программой будет на самом верхнем уровне абстракции иметь следующую структуру:

```
_(persons) .map (fund) .reduce(func2) ;
```

где функции fund и func2 реализуют конкретное требующееся поведение. Отделив функции от главного потока управления, можно в конечном итоге получить исходный код,

приведенный в листинге 3.4.

Листинг 3.4. Сочетание операций tap и reduce для подсчета статистики

```
const getCountry = person => person.address.country;

const gatherStats = function (stat, criteria) {
    stat[criteria] = _.isUndefined(stat[criteria]) ? 1 : stat[criteria] + 1;
    return stat;
};

(persons).map(getCountry).reduce(gatherStats, {});
```

В исходном коде из листинга 3.4 операция tap служит для предварительной обработки массива объектов и извлечения всех стран. Затем в нем выполняется операция reduce для накапливания конечного результата. Вместо прямого доступа к свойствам можно воспользоваться линзой (из библиотеки Ramda), чтобы сосредоточиться на свойстве address.city объекта, представляющего конкретное лицо:

```
const cityPath = ['addresscity'];
const cityLens = R.lens(R.path(cityPath), R.assocPath(cityPath));
```

И так же просто можно подсчитать количество людей, проживающих в отдельных городах:

```
-(persons).map(R.view(cityLens)).reduce(gatherStats, {});
```

С другой стороны, можно воспользоваться операцией _. groupBy, чтобы добиться того же самого результата еще более лаконичным способом:

```
_.groupBy(persons, R.view(cityLens));
```

В отличие от операции map, операция reduce зависит от накапливаемого результата и поэтому может вести себя по-разному, когда выполняется слева направо или справа налево, если только выполняемая функцией-итератором операция не является коммутативной. Чтобы продемонстрировать эту особенность операции reduce, рассмотрим в качестве примера следующую простую программу, суммирующую числа в массиве:

```
([0,1,3,4,5]).reduce( .add); // -> 13
```

Того же самого результата можно добиться, выполнив сведшие в обратном порядке с помощью операции reduceRight. Эта операция действует именно так, как и следовало предполагать, поскольку операция сложения является коммутативной. Но она может привести к совершенно другим результатам для операций вроде деления, которые не являются коммутативными. Используя те же самые обозначения, что и прежде, операцию _.reduceRight можно формально определить следующим образом:

```
reduceRight(f, [eO, el, e2, e3],accum) -> f(eO, f(el, f(e2, f(e3,accum)))) -> R
```

Например, обе программы в правой и левой частях следующей операции сравнения дадут совершенно разные результаты от применения операции .divide:

```
([1,3,4,5]).reduce(_.divide) !== ([1,3,4,5]).reduceRight(_.divide);
```

Более того, операция reduce применяется сразу ко всем элементам массива, а это означает, что ее нельзя сократить, чтобы не обрабатывать весь массив. Допустим, требуется проверить достоверность списка значений для входных параметров. Проверка массива параметров на достоверность означает, что нам нужно вычислить единственное логическое значение, отражающее корректны ли все значения в массиве или нет.

Но воспользоваться для этой цели операцией reduce было бы неэффективно, поскольку пришлось бы обойти все значения в массиве. Обнаружив в ходе проверки одно недостоверное значение, было бы нецелесообразно продолжать ее дальше. Поэтому рассмотрим более эффективную функцию проверки достоверности, в которой применяются операции .some, .isUndefined и _. isNull, как показано ниже. Когда операция some применяется к каждому элементу массива, она возвращает результат (логическое true), как только обнаружит хотя бы одно прошедшее проверку значение.

```
const isNotValid = val => _.isUndefined(val) I I _.isNull (val) ; <-
const notAHValid = args => _(args).some(isNotValid);
<-------------------------------------------------------

notAHValid( ['string', 0, null, undefined]); // -> true
notAHValid (['string', 0, {}]);              // -> false
```

Значение недостоверно, если оно не определено или пустое

Функция some () возвратит результат, как только он окажется истинным. Это удобно для проверки хотя бы одного достоверного значения

Воспользовавшись операцией _. every, можно получить и функцию all Valid (), выполняющую действие, логически обратное действию функции notAHValid(), как показано ниже. Эта функция проверяет, возвращает ли заданная предикатная функция логическое значение true для всех элементов массива.

```
const isValid = val => isUndefined(val) && !_.isNull(val); const allValid = args
=> _(args).every(isValid);

allValid(['string', 0, null]); // -> false allValid(['string', 0, {}]);   // ->
true
```

Как было показано ранее, в операциях map и reduce предпринимается попытка обойти весь массив. Но зачастую обрабатывать все элементы массива или иной структуры данных не нужно, поскольку сначала нужно пропустить в нем любые пустые (null) или неопределенные (undefined) элементы или объекты. Поэтому было бы неплохо иметь в своем распоряжении механизм для исключения или отсеивания определенных элементов из структуры данных, прежде чем выполнять над ними вычисления. И для этой цели служит рассматриваемая далее операция _.filter.

3.3.4. Исключение ненужных элементов с помощью операции filter

В процессе обработки крупных коллекций данных нередко требуется исключить из нее те элементы, которые не задействуются в вычислениях. Допустим, требуется подсчитать количество людей, проживающих в европейских странах или родившихся в определенном году. Вместо того чтобы загромождать исходный код этой задачи условными операторами if-else, для ее решения можно воспользоваться операцией filter.

Операция filter, иначе называемая select, реализуется в виде функции высшего порядка, перебирающей элементы массива и возвращающей новый массив, являющийся подмножеством исходного массива и содержащий значения, для которых предикатная функция p возвращает истинный результат (логическое значение true). Формальное определение этой операции выглядит следующим образом (см. также рис. 3.6):

```
filter(p, [d0, di, d2, d3...dn]) ->
       [d0,dl,...dn] (подмножество исходного массива данных)
       ----------------------------------------------------------------Операция filter
```

Рис. 3.6. Функции, реализующей операцию filter, передается в качестве параметра исходный массив данных и функция-итератор p; она применяет к нему заданные в функции p критерии отбора, чтобы получить в итоге потенциально меньшее подмножество исходного массива. Критерии отбора p называются также *предикатной функцией*

Одна из возможных реализаций операции filter приведена в листинге 3.5.

Листинг 3.5. Реализация операции фильтрации `function filter(arr, predicate) { let idx = -1, len = arr. length,` **i Получаемый в итоге массив содержит** `result = [];'` **подмножество элементов исходного массива**

```
    while (++idx < len) {
        let value = arr[idx];
        if (predicate(value, idx, this)) {-<
            result.push(value);
        }
    }
    return result;
```

Вызвать предикатную функцию. Если результат истинный, элемент массива сохраняется, а иначе — пропускается

Помимо исходного массива данных, функции, реализующей операцию filter, передается предикатная функция predicate (), предназначенная для проверки каждого элемента массива на включение в результирующий массив. Если эта функция возвращает логическое значение true, элемент остается в результирующем массиве, а иначе он пропускается. Именно поэтому операция filter широко применяется для удаления недостоверных данных из массива, как показано ниже.

```
_(persons).filter(isValid).map(fullname);
```

Но эта операция способна на гораздо большее. Допустим, из коллекции объектов типа Person требуется извлечь данные только о тех людях, которые родились в 1903 году. Для решения этой задачи намного проще и яснее употребить операцию _. filter, чем условные операторы, как демонстрируется в следующем примере кода:

```
const bornlnl903 = person => person.birthYear === 1903;

_(persons).filter(bornlnl903).map(fullname).join(' and ');

// -> 'John von Neumann and Alonzo Church'
```

Генерирование массивов

Операции map и filter реализуются в виде функций высшего порядка, возвращающих новые массивы, формируемые из существующих массивов. Они имеются во многих языках ФП, включая Haskell, Clojure и пр. Альтернативой сочетанию операций tap и filter служит понятие *генерирования массивов (array comprehension),* называемое иначе *генерированием списков (list comprehension).* Это языковое средство ФП, инкапсулирующее функциональные возможности операций tap и filter в лаконичном синтаксисе, где используются ключевые слова for.. .of и if соответственно:

```
[for (x of итерируемый объект) if (условие) x]
```

На момент написания данной книги существовало предложение внедрить средства генерирования массивов в стандарт ECMAScript 7. Они позволят составлять лаконичные выражения для построения новых массивов (именно поэтому все приведенное выше выражение заключено в квадратные скобки). Например, используя синтаксис генерирования массивов, исходный код из предыдущего примера можно реорганизовать следующим образом:

```
[for (p of people) if (p.birthYear === 1903) p.fullname]
    .join(' and ');
```

Применение всех рассмотренных выше методик ФП опирается на расширяемые и эффективные функции, которые позволяют не только писать более ясный код, но и лучше

уяснить данные. Пользуясь декларативным стилем программирования, можно уделить больше внимания выводимому из приложения результату, чем способу его получения. Тем самым становится легче глубоко анализировать разрабатываемое приложение.

3.4. Анализ прикладного кода

Напомним, что тысячи строк кода, разделяющие в JavaScript общее глобальное пространство имен, могут быть сразу загружены на одной странице. В последнее время проявляется немалый интерес к созданию модулей упорядочения бизнес-логики, но по-прежнему в эксплуатации находятся тысячи проектов, где этого не сделано.

Что же означает анализ прикладного кода? В предыдущих главах этот термин употреблялся в широком смысле для обозначения способности анализировать любую часть программы и легко строить в уме модель происходящего. Эта модель содержит такие динамические части, как состояние всех переменных и возвращаемые функциями результаты, а также статические части, включая степень удобочитаемости и выразительность разработанного кода. Обе эти части важны в равной степени. И в этой книге поясняется, насколько неизменяемость и чистые функции упрощают построение такой модели.

Ранее уже отмечалось, насколько ценно уметь связывать вместе операции высокого уровня для написания программ. Потоки управления императивной и функциональной программами отличаются коренным образом. В частности, поток управления функциональной программой дает ясное представление о ее назначении, не вникая в подробности работы ее внутреннего механизма. Это дает возможность более глубоко анализировать прикладной код, а также прохождение входящих и исходящих потоков данных на разных стадиях для получения искомых результатов.

3.4.1. Декларативный характер цепочек функций с отложенными вычислениями

Как упоминалось в главе 1, функциональные программы состоят из простых функций, которые сами по себе ничего особенного не делают, но совместно они способны решать сложные задачи. В этом разделе представлен способ написания целой программы путем связывания вместе ряда функций.

Декларативный характер модели функционального программирования побуждает рассматривать программы как вычисление независимых чистых функций, что способствует построению необходимых абстракций с целью добиться текучести и выразительности прикладного кода. Подобным образом можно составить онтологию или словарь, ясно выражающие назначение разрабатываемого приложения. Построение чистых функций из таких строитель

ных блоков, как операции map, reduce и filter, приводит к выработке стиля написания легко анализируемого и сразу же понимаемого кода.

Сильная сторона такого повышения уровня абстракции заключается в том, что вы начинаете рассматривать операции независимо от применяемых базовых структур данных. Рассуждая теоретически, семантический смысл вашей программы не меняется, манипулируете ли вы массивами, связными списками, двоичными деревьями или иными структурами данных. Именно поэтому в функциональном программировании операциям уделяется больше внимания, чем структурам данных.

Допустим, требуется прочитать список имен, нормализовать их, удалить любые дубликаты и отсортировать конечный результат. Рассмотрим сначала императивный вариант решения этой задачи, а затем попробуем решить ее в функциональном стиле.

Список имен можно выразить в виде следующего массива с неравномерно отформатированными символьными строками:

```
var names = [’alonzo church’, ’Haskell curry’, 'stephen_kleene',
             'John Von Neumann', 'stephen_kleene'];
```

В листинге 3.6 приведен исходный код императивного варианта программы, решающей поставленную выше задачу.

Листинг 3.6. Выполнение последовательных операций над массивами (императивный подход)

```
var result = [];
for (let i = 0; i < names. length; i++) {◄----   Обработать в цикле все имена в массиве
    var n = names[i];
    if (n !== undefined && n !== null) { ◄-----   Проверить, все ли слова достоверны
        var ns = n.replace(/_/, ' ').split(' ');
        for(let j = 0; j < ns.length; j++) {
            var p = ns[j];
            p = p.charAt(0).toUpperCase() + p.slice(1);
            ns[ j] = p;
        }
        if (result.indexOf(ns.join(' ')) < 0) {-<
            result.push(ns.join(' '));
        }

result. sort () ; ◄ -1
```

Этот массив содержит несогласованно отформатированные данные; на данной стадии нормализуется (исправляется) каждый элемент массива

Устранить дубликаты, проверив, существует ли имя в конечном результате

Отсортировать массив

Выполнение кода из листинга 3.6 дает следующий желательный результат: `['Alonzo Church', 'Haskell Curry', 'John Von Neumann', 'Stephen Kleene']`

Недостаток императивного кода из листинга 3.6 заключается в том, что он нацелен на эффективное решение конкретной задачи. Этим кодом можно воспользоваться для решения только данной конкретной задачи. Следовательно, он выполняется на гораздо более низком уровне абстракции, чем функциональный код. А ведь чем ниже уровень абстракции, тем меньше вероятность повторного использования кода, но тем больше его сложность и вероятность появления в нем ошибок.

С другой стороны, для реализации функционального варианта поставленной выше задачи достаточно соединить вместе компоненты в виде "черного ящика", возложив всю ответственность на эти вполне устоявшиеся и проверенные компоненты интерфейса API, как демонстрируется в листинге 3.7. Обратите внимание, насколько каскадный порядок следования вызовов функций упрощает чтение исходного кода из листинга 3.7.

Листинг 3.7. Выполнение последовательных операций над массивами (функциональный подход)

```
                                | Инициализировать цепочку функций, как поясняется далее
_. cha i n (names)                      недостоверные значения
    .filter(isValid)  --- J
    .map(s => s. replace (/_/, ' ') ) ◄  Нормализовать значения
    .uniq()              ◄ Исключить дубликаты
    .map(_.startcase) <<—
    .sort()                    Задать регистр букв
    .value();

// -> ['Alonzo Church', 'Haskell Curry', 'Jon Von Neumann',
  'Stephen Kleene']
```

Функции, реализующие операции filter и .map, берут на себя всю черновую работу для обхода массива names по достоверным индексам. А на оставшихся стадиях процесса нам остается лишь реализовать специальное поведение программы. Так, функция, реализующая операцию _. uniq, служит для исключения дублирующихся элементов, а функция, реализующая операцию .startcase, — для выделения прописной буквы в каждом слове. И, наконец, все полученные результаты сортируются в нужном порядке.

Писать и читать такие программы, как приведенная в листинге 3.7, намного приятнее, не так ли? Такой стиль программирования позволяет не только значительно сократить объем кода, но и сделать его структуру простой и понятной.

Итак, продолжим исследование библиотеки Lodash. В следующем примере снова рассматривается исходный код из листинга 3.4, где подсчитываются все страны их массива объектов типа Person. Для целей данного примера усовершенствуем немного функцию gatherStats (), как показано ниже.

```
const gatherStats = function (stat, country) {
    if(!isValid(stat[country])) {
        stat[country] = {'name': country, 'count': 0};
    }
    stat[country].count++;
    return stat;
};
```

Теперь эта функция возвращает объект со следующей структурой: {

```
  'US' : {'name': 'US', count: 2},
  'Greece' : {'name': 'Greece', count: 1},
  'Hungary': {'name': 'Hungary', count: 1}
}
```

Такая структура гарантирует однозначность элементов для каждой строки. Но ради интереса попробуем ввести дополнительные данные в первоначальный массив объектов типа Person, приведенный в начале этой главы:

```
const p5 = new Person('David', 'Hilbert', '555-55-5555');
p5.address = new Address('Germany');
p5.birthYear = 1903;

const p6 = new Person('Alan', 'Turing', '666-66-6666');
p6.address = new Address('England');
p6.birthYear = 1912;

const p7 = new Person('Stephen', 'Kleene', '777-77-7777'); p7.address = new
Address('US');
p7.birthYear = 1909;
```

Следующая задача состоит в написании программы, возвращающей страну, в которой проживает наибольшее количество людей из приведенного выше массива данных. Решим ее снова, связав вместе нужные функции, начиная с функции, реализующей операцию _. chain, как показано в листинге 3.8.

Листинг 3.8. Демонстрация цепочки функций с отложенными вычислениями из библиотеки Lodash

```
. chain (persons)        // Составить цепочку из функций с отложенными вычислениями для обработки предоставленного массива
    .filter(isValid)
    .map(_.property('address.country'))   // Воспользоваться функцией, реализующей операцию _. property, чтобы извлечь свойство address. country из объекта заданного лица. Это эквивалентная, но более скромная по функциям версия R. view () из библиотеки Ramda
    .reduce(gatherStats, {})
    .values()
    .sortBy('count')
    .reverse()
    .first()
    . value ()           // Выполнить все функции из цепочки
    .name; // -> 'US'
```

Функцию, реализующую операцию chain, можно использовать для расширения состояния исходного объекта, соединяя вместе операции, преобразующие входные данные в желательные выходные данные. В отличие от заключения массивов в круглые скобки с помощью краткого обозначения объектов такой подход оказывается более эффективным потому, что он позволяет явно сделать соединяемой в цепочку любую функцию из заданной последовательности. Несмотря на всю сложность рассматриваемой здесь программы, в ней можно обойтись без создания переменных и полностью исключить циклы.

Еще одно преимущество, которое дает применение функции, реализующей операцию _. chain, заключается в возможности писать сложные программы с отложенным поведением, когда ничего не выполняется до тех пор, пока не будет вызвана последняя функция value (). Это может оказать огромное влияние на приложение, поскольку дает возможность пропускать выполнение целых функций, если возвращаемые ими результаты не нужны (подробнее об отложенных вычислениях речь пойдет в главе 7). Поток управления рассматриваемой здесь программой приведен на рис. 3.7.

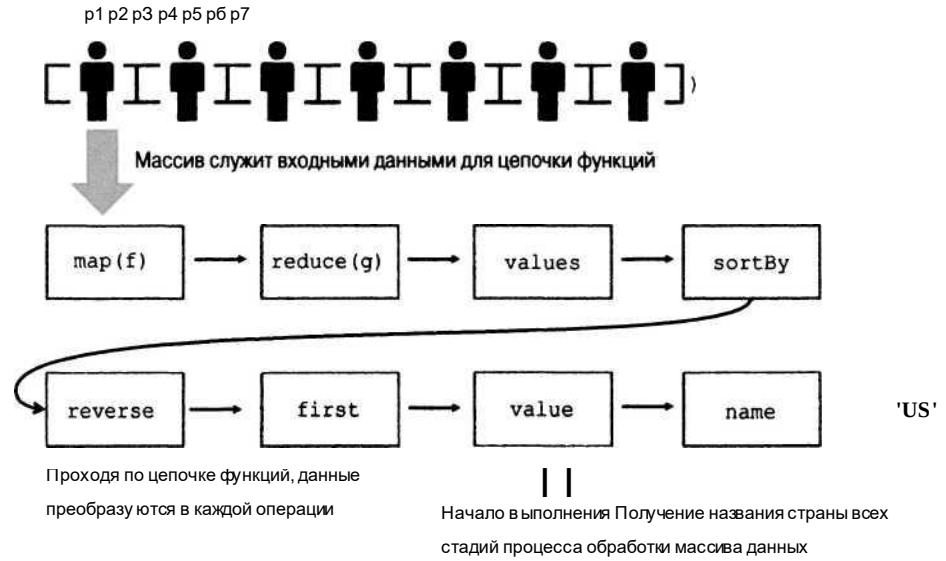


Рис. 3.7. Поток управления программой, составленной из цепочек функций библиотеки Lodash.
Массив объектов типа Person обрабатывается в ходе выполнения последовательного ряда операций, проходя через которые данные в конечном итоге сводятся к единственному значению

На данном примере постепенно становится ясно, почему функциональные программы более совершенны. Анализ недостатков императивного варианта решения рассматриваемой здесь задачи оставляется на ваше рассуждение. А плавность работы функционального варианта из листинга 3.8 объясняется тем, что он опирается на основополагающие принципы ФП — чистые и свободные от побочных эффектов функции. Каждая последующая функция в цепочке неизменяемо оперирует новыми массивами, построенными в предыдущих функциях. Начиная цепочку с вызова функции, реализующей операцию _. chain, библиотека Lodash извлекает выгоду из этого шаблона, чтобы предоставить универсальный набор служебных функций для удовлетворения большинства потребностей в ФП. Это помогает постепенно перейти к стилю *бесточечного* (или *комбинаторного)* программирования, который характерен только для ФП и будет представлен в следующей главе.

Возможность определять программные конвейеры в отложенном режиме дает намного больше преимуществ, чем только их удобочитаемость. Программы с отложенными

вычислениями определяются прежде, чем они вычисляются, поскольку они могут быть оптимизированы с помощью таких методик, как повторное использование структур данных и объединение методов. Но подобные методики оптимизации не сокращают время, которое требуется для выполнения функций в чистом виде, а скорее помогают исключить излишние вызовы. Более подробно они рассматриваются в главе 7 при обсуждении вопросов производительности функциональных программ.

В исходном коде из листинга 3.8 данные проходят от одного звена цепочки к следующему. Благодаря декларативному характеру использования функций высшего порядка становится очевидным, каким образом данные преобразуются в каждом звене цепочки, обнаруживая больше ценных сведений о данных.

3.4.2. ЗОЬподобные данные: функции как данные

На протяжении всей этой главы демонстрировался разнообразный ряд функций, реализующих операции map, reduce, filter, groupBy, sortBy, uniq и т.д. Словарь, сформировавшийся вокруг этих функций, может быть использован для того, чтобы ясно экстраполировать сведения, относящиеся к данным. Если помыслить нестандартно, отвлекшись на секунду от приведенных выше рассуждений, то можно заметить, что эти функции напоминают запросы к базе данных на языке SQL, и это неслучайно.

Разработчики привыкли пользоваться языковыми средствами SQL, чтобы уяснить и экстраполировать назначение данных. Например, объекты типа Person из коллекции можно представить так, как показано в табл. 3.1.

Таблица 3.1. Табличное представление данных из списка объектов типа Person

id	firstname	lastname	country	birthYear
0	Haskell	Curry	US	1900
i	Barkley	Rosser	Greece	1907
2	John	Von Neumann	Hungary	1903
3	Alonzo	Church	US	1903
4	David	Hilbert	Germany	1862
5	Alan	Turing	England	1912
6	Stephen	Kleene	US	1909

Оказывается, что мышление категориями языка запросов при написании программ очень похоже на мышление категориями операций, применяемых к массивам в функциональном программировании. По существу, это означает применение общего словаря или, если угодно, алгебры, побуждающее к глубокому анализу характера данных и их структуры. Так, из следующего запроса SQL:

```
SELECT p.firstname FROM Person p
WHERE p.birthYear > 1903 and p.country IS NOT 'US'
ORDER BY p.firstname
```

вполне очевидно, каким образом должны выглядеть данные после выполнения этого запроса. Прежде чем реализовать версию этого запроса HaJavaScript, попробуем реализовать ряд псевдонимов функций, чтобы стало понятнее, о чем здесь идет речь. В библиотеке Lodash поддерживаются средства, называемые *примесями* и предназначенные

для расширения основной библиотеки дополнительными функциями, которые можно также связывать в цепочку:

```
_.mixin({'select': _.map,
         'from':   chain,
         'where':  filter,
         'sortBy':   sortByOrder});
```

Применив этот объект примеси, можно написать программу, исходный код которой приведен в листинге 3.9.

Листинг 3.9. Написание SQL-подобной программы на JavaScript

```
   from(persons)
   .where(p => p.birthYear > 1900 && p.address.country !== 'US')
   .sortBy(['firstname']) .select(p => p.firstname) .value();

// -> ['Alan', 'Barkley', 'John']
```

В программе из листинга 3.9 создаются псевдонимы, преобразующие ключевые слова SQL в соответствующие функции. Этот пример позволяет лучше понять, насколько функциональный код похож на запрос SQL.

Примеси в JavaScript

Примесь является объектом, определяющим абстрактное подмножество функций, относящихся к конкретному типу (в данном случае — команде SQL). Этот объект не применяется непосредственно в исходном коде, а только расширяет поведение другого объекта аналогично *трейтам* в других языках программирования. Целевой объект заимствует у примеси все функциональные возможности.

Повторно использовать код в ООП можно и по-другому, не прибегая к наследованию и не имитируя множественное наследование в тех языках программирования, где оно не поддерживается (к их числу относится и JavaScript). В этой книге примеси не рассматриваются, хотя они весьма эффективны, если применяются надлежащим образом. Для более подробного изучения примесей рекомендуется прочитать статью, доступную по адресу `https://javascriptweblog, wordpress.com/2011/05/`.

Все сказанное выше должно убедить вас, что функциональное программирование может служить эффективной абстракцией над императивным кодом. Что может быть лучше, чем обрабатывать и анализировать синтаксически данные, пользуясь семантикой языка запросов? Подобно запросу SQL, данные в приведенном выше коде JavaScript моделируются в форме функций, что фактически означает применение *функций как данных* и поэтому так и называется. Благодаря своему декларативному характеру эта форма описывает *результат* обработки данных, а не *порядок* его достижения. В приведенных до сих пор примерах из этой главы мы обходились вообще без обыкновенных операторов, предназначенных для организации циклов, и намерены продолжить в том же духе и в остальных главах, пользуясь абстракциями, заменяющими на высоком уровне организацию циклов.

Еще одной методикой, нередко применяемой вместо организации циклов, служит рекурсия. Ею можно пользоваться как средством абстрагирования от итерации, когда

приходится решать задачи “самоподобного” характера. Для решения таких задач последовательные цепочки функций не подходят и малоэффективны. А рекурсия по-своему реализует способы обработки данных, поручая всю черновую работу по стандартной организации циклов интерпретатору языка или его исполняющей среде.

3.5. Умение мыслить рекурсивно

Иногда задача оказывается сложной и трудноразрешимой, чтобы приступить к ней вплотную. В таком случае следует сразу же попытаться найти способы ее декомпозиции. Если задача разделяется на более мелкие варианты решения ее самой, то, решив их по отдельности, можно решить и задачу в целом. Рекурсия имеет существенное значение для обхода элементов массива исключительно средствами языка функционального программирования, подобного Haskell, Scheme или Erlang, поскольку в этих языках просто отсутствуют конструкции для организации циклов.

И в языке JavaScript рекурсия находит немало примеров применения, включая синтаксический анализ XML- или HTML-документов, графов и т.д. В этом разделе сначала поясняется, что такое рекурсия, а затем приводится упражнение, проработав которое вы сможете научиться мыслить рекурсивно. А после этого будет сделан краткий обзор некоторых структур данных, которые можно синтаксически анализировать, пользуясь рекурсией.

3.5.1. Что такое рекурсия

Рекурсия — это методика, предназначенная для решения задач программирования путем их декомпозиции на более мелкие самоподобные задачи, объединяя которые вместе можно прийти к решению всей исходной задачи в целом. Рекурсивная функция состоит из следующих основных частей:

- основные варианты, называемые также *условиями возврата,*
- рекурсивные варианты.

Основные варианты представляют собой набор входных данных, на основании которых рекурсивная функция вычисляет конкретный результат, не прибегая непосредственно к рекурсии. А рекурсивные варианты имеют отношение к набору входных данных, который должен быть меньше исходного и для которого функция вызывает сама себя. Если же такой набор входных данных оказывается не меньше исходного, то рекурсия выполняется бесконечно и вплоть до аварийного завершения программы. В процессе рекурсивного выполнения функции набор входных данных безусловно уменьшается, достигая в конечном итоге величины, для которой выбирается основной вариант, и на этой величине весь процесс завершается.

Напомним, что в некоторых примерах из главы 2 рекурсия применялась для глубокого замораживания всей вложенной структуры объекта. Основной вариант выбирался, когда встречался примитивный или уже замороженный объект. В противном случае рекурсивная стадия продолжалась обходом структуры объекта до тех пор, пока еще обнаруживались незамороженные объекты. В данном случае рекурсия была уместна потому, что на каждом

уровне решаемая задача оказывалась совершенно одинаковой. Но научиться мыслить рекурсивно нелегко, поэтому начнем именно с этого.

3.5.2. Как научиться мыслить рекурсивно

Усвоить принцип действия рекурсии непросто. Как и в функциональном программировании, самое трудное — разучиться мыслить привычно. Эта книга не призвана научить вас полностью овладеть рекурсией. И хотя эта методика программирования применяется нечасто, она важна для овладения. Поэтому ниже предлагается упражнение для вашего ума, которое поможет научиться лучше анализировать рекурсивные задачи.

Рекурсивное мышление принимает во внимание самое себя или свой модифицированный вариант. Рекурсивный объект является самоопределяемым, и примером тому служит составление ветвей дерева. У каждой ветви имеются листья и другие ветви, а у тех — свои листья и ветви. Этот процесс ветвления продолжается бесконечно и останавливается только при наличии одного ограничивающего фактора — размера дерева (в данном случае).

Принимая во внимание все сказанное выше, рассмотрим в качестве разминочного упражнения простую задачу сложения всех чисел, хранящихся в массиве, постепенно перейдя от императивного варианта ее решения к функциональному. Та сторона вашего ума, которая привыкла мыслить императивно, естественно представляет решение данной задачи в том, чтобы обойти массив и сохранить накопленное суммарное значение, как показано ниже.

```
var ace = 0;
for(let i = 0; i < nums.length; i++) { acc += nums[i];
}
```

Ум побуждает вас к необходимости воспользоваться аккумулятором, без которого просто не обойтись, если требуется хранить промежуточное суммарное значение. Но нужно ли действительно организовывать для этого цикл вручную? В этот момент вы осознаете, что в вашем распоряжении имеются и другие средства из арсенала ФП и, в частности, приведенная ниже операция reduce. `_(nums).reduce((ace, current) => ace + current, 0);`

Поставив ручную итерацию в определенные рамки, можно абстрагировать от нее свой прикладной код. Но можно добиться и большего, передав итерацию полностью платформе. Функция, реализующая операцию reduce, наглядно показывает, что можно вообще не беспокоиться об организации цикла и даже о размере обрабатываемого массива. Чтобы получить желаемый результат, достаточно сложить последовательно первый элемент массива с остальными его элементами, а следовательно, достичь рекурсивного образа мышления. Этот мыслительный процесс можно расширить до общего представления о суммировании чисел как о выполнении последовательности операций следующим образом, называемым *нестандартным мышлением:* `sum[1, 2,3,4,5,6,7,8,9] = 1 + sum[2,3, 4,5, 6, 7,8,9]`

```
= 1 + 2 + sum[3, 4,5, 6, 7,8,9]
= 1 + 2 + 3 + sum[4,5, 6, 7,8,9]
```

Итерация и рекурсия — две стороны одной медали. В отсутствие модификации рекурсия предлагает более выразительную, эффективную и отличную альтернативу итерации. На самом деле в языках исключительно функционального программирования вообще отсутствуют стандартные конструкции для организации циклов вроде do, for, и while, поскольку это делается полностью рекурсивно. Кроме того, рекурсия приводит к написанию кода, который легче понять, поскольку она основывается на многократном повторении одних и тех же действий над постепенно уменьшающимся набором входных данных. В рекурсивном решении рассматриваемой здесь задачи, приведенном в листинге 3.10, применяются функции, реализующие операции first и rest из библиотеки Lodash, для доступа к первому элементу массива или ко всем его элементам, кроме первого, соответственно.

Листинг 3.10. Рекурсивное суммирование чисел

```
function sum(arr) { . ~,        . «   . ,      I Основной вариант
    if(_.isEmpty(arr)) { ◄ --- (условиевозврата)
        return 0;

    return _.first(arr) + sum(_.rest(arr));

sum([]); //-> 0
sum( [1,2,3, 4,5, 6, 7,8,9] ) ; //
-> 45
```

Итеративный вариант: функция вызывает сама себя для набора входных данных, постепенно уменьшаемого с помощью операцийfirst irest

При сложении элементов пустого массива выбирается основной вариант выполнения приведенной выше рекурсивной функции, из которой, естественно, возвращается нулевое суммарное значение. В противном случае выполнение данной функции для обработки непустого массива продолжается рекурсивно, когда первый элемент массива складывается с остальными его элементами. Рекурсивные вызовы данной функции размещаются во внутреннем стеке друг над другом.

И как только алгоритм достигнет условия возврата, все операторы возврата выполняются по мере сворачивания стека с целью обеспечить суммирование элементов массива. Такой механизм применяется при рекурсии для того, чтобы передать организацию циклов интерпретатору языка. Ниже приведено пошаговое представление реализованного выше алгоритма суммирования элементов массива.

```
1 +       3 4,5 6,7,8,9]
1 + 2 + sum[3,4,5,  7,8,9]
1 + 2 + 3 + sum[4,5,6,7,8,9]
1 + 2 + 3 + 4 + sum[5, 6, 7,8,9]
1 + 2 + 3 + 4 + 5 + sum[6,7,8,9]
1 + 2 + 3 4- 4 + 5 + 6 + sum[7,8,9]
1 + 2 + 3 + 4 + 5 + 6 + 7 + sum[8,9]
1 + 2 + 3 4- 4 + 5 + 6 + 7 + 8 + sum[9]
1 + 2 + 3 4- 4 + 5 + 6 + 7 + 8 + 9 + sum[]
1 + 2 + 3 + 4 + 5 + 6+7+8+9+0-> остановка, сворачивание стека
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
1+2 + 3   + 4 + 5 + 6 + 7 + 17
1 + 2 + 3 + 4 + 5 + 6 + 24
1 + 2 + 3 + 4 + 5 + 30
1 + 2 + 3 + 4 + 35
1 + 2 + 3 + 39
1 + 2 +   >
1 + 44
45
```

В данный момент вполне естественно возникает мысль о производительности рекурсии в сравнении с итерацией вручную. Ведь компиляторы стали довольно развитыми логически, чтобы оптимизировать циклы. Так, в стандарте ES6 языка JavaScript внедрено средство, называемое *оптимизацией хвостовых вызовов (tail-call optimization)* и способное сблизить производительность итерации и рекурсии. В связи с этим рассмотрим следующий, несколько иной вариант реализации функции sum(): `function sum(arr, ace = 0) { if(_.isEmpty(arr)) { return 0;`

```
    }                                                    ! Рекурсивный вызов
    return sum(_.rest(arr), acc + _. first (arr) );      ' из хвостовой позиции
}
```

В этом варианте рекурсивный вызов размещается на последней стадии выполнения тела функции, иначе называемой *хвостовой позицией*. Преимущества такого подхода будут дополнительно рассмотрены в главе 7, когда речь пойдет о видах оптимизации функционального кода.

3.5.3. Рекурсивно определяемые структуры данных

Вас, вероятно, могли удивить имена, переданные объектам типа Person, употреблявшимся в приведенных ранее примерах в качестве образцовых данных. В 1900-е годы активно развивался математический аппарат, положенный в основу функционального программирования (лямбда-исчисление, теория категорий и пр.).

Большая часть трудов, опубликованных в данной области математики, основана на общих идеях и теоремах, выдвинутых и доказанных учеными ведущих университетов под

опекой таких профессоров, как Алонзо Чёрч. На самом деле многие выдающиеся математики, такие как Беркли Россер, Алан Тьюринг, Стивен Клини, были докторантами профессора Чёрча. И у них самих были докторанты. На рис. 3.8 представлена древовидная структура отношений ученичества между этими и другими выдающимися математиками, внесшими заметный вклад в развитие математического аппарата, применяемого в функциональном программировании.

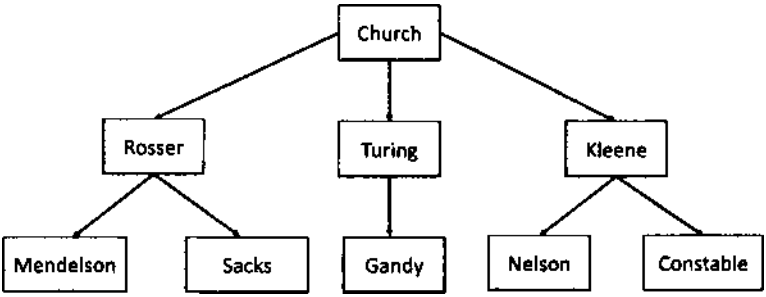


Рис. 3.8. Влиятельные математики, внесшие заметный вклад в развитие математического аппарата, применяемого в функциональном программировании. Линии, соединяющие родительские и порожденные узлы древовидной структуры, представляют отношения ученичества

Подобные древовидные структуры весьма распространены в программном обеспечении и могут' служить для моделирования XML-документов, файловых систем, таксономий, категорий, виджетов меню, фасетной навигации, социальных графов и многого другого. На рис. 3.8 представлен ряд узлов, соединяемых линиями, обозначающими отношения научного руководителя и докторанта. До сих пор рассматривались методики ФП и соответствующие операции, предназначенные для синтаксического анализа плоских структур данных вроде массивов. Но эти методики и операции непригодны для обработки древовидных структур данных. А поскольку BjavaScript отсутствует встроенный древовидный объект, то приходится самостоятельно создавать простые структуры данных, основанные на узлах дерева, где *узел* — это объект, содержащий значение, ссылку на свой родительский узел и массив порожденных узлов. Так, узел Rosser на рис. 3.8 связан со своим родительским узлом Church и порожденными узлами Mendelson и Sacks. Если же у какого-нибудь узла дерева отсутствует родительский узел, то такой узел считается корневым (в данном случае — узел Church). В листинге 3.11 приведено определение типа Node, представляющего древовидную структуру.

Листинг 3.11. Определение типа Node

```
class Node { constructor(val) { this._val = val; this._parent = null; this.-
Children = [];
    }

    isRootO { return isValid (this. _parent); ◄      Эта функция была создана прежде
    }

    get children() { return this.-Children;
    }
```

```
    hasChildren() {
        return this.-Children.length > 0;
    }

    get value() { return this._val;
    }

    set value(val) {
        this, val = val;
    }

    append(child) {
        chi Id. parent = this;
        this.-Children.push (child) ;
        return this;
    j
    toString() { return 'Node (val: ${this._val}, children: ${this.-
        Children.length})';
    }
```

Задать родительский узел для данного узла

Добавить этот порожденный узел в список порожденных узлов

[5] Возвратить тот же самый узел, что удобно для каскадирования методов

Новые узлы дерева можно создать следующим образом:

```
const church = new Node (new Person (
                          'Alonzo', 'Church', ' 111-11-1111')); -<-------- ---- *
```

Повторить эту операцию для каждого узла дерева I

Деревья являются рекурсивно определяемыми структурами данных, содержащими корневой узел, как показано ниже.

```
class Tree {
    constructor (root) {
        this, root = root;
    I          ~

    static map(node, fn, tree = null) {' j
        = fn (node, value) ;
        if (tree === null) {
            tree = new Tree (node);
        }

        i f (node. hasChildren () ) { <0——J
            .map(node.children, function (child)
                Tree.тар(child,  fn, tree);  —
            });
```

Использовать статический метод во избежание путаницы с более распространенным методом Array .pro to type, map (). Этот статический метод можно также применять как, по существу, самостоятельную функцию

Вызвать функцию-итератор и обновить `node, value` **значение в узловом элементе дерева**

Действует аналогично методу Ar г ау .prototype. » map (), а результате получается новая структура да иных

| Если у данного узла отсутствуют порожденные узлы, то продолжать не стоит (базовый вариант)

{4 Вызвать предоставляемую функ- цию для каждого порожденного I узла

```
        }                    Сделать рекурсивный вызов для каждого порожденного узла

        return tree;
    get root() {
        return this._root;
    }
}
```

Основная логика работы узла находится в методе append (). Когда к данному узлу присоединяется порожденный узел, в нем устанавливается ссылка parent на данный узел как на родительский, и введенный узел добавляется в список порожденных узлов. Дерево создается путем связывания узлов с другими порожденными узлами приведенным ниже образом, начиная с корневого узла church.

```
church.append(rosser).append(turing).append(kleene);
kleene.append(nelson).append(constable);
rosser.append(mendelson).append(sacks); turing.append(gandy);
```

Каждый узел отвечает за заключение объекта типа Person в оболочку. Рекурсивный алгоритм выполняет обход всего дерева в ширину, начиная с корневого узла и продолжая далее ко всем его порожденным узлам. Вследствие самоподобного характера такого обхода дерева, начиная с корневого узла, он подобен обходу дерева, начиная с любого узла, что, по существу, составляет определение рекурсии. Для этой цели служит метод Tree .шар () как функция высшего порядка с семантикой, подобной методу Array. prototype. map (), которому передается функция, вычисляющаяся для значения каждого узла. Как видите, независимо от типа структуры, применяемой для моделирования подобных данных (в данном случае — древовидной структуры), семантика данной функции должна оставаться той же самой. По существу, функцию преобразования можно применить к любым типам данных, сохранив их структуру. Более формально вопрос применения функций преобразования к типам данных с сохранением их структуры будет рассмотрен в главе 5.

Обход рассматриваемого здесь дерева в ширину осуществляется в несколько перечисленных ниже стадий, начиная с корневого узла.

1. Отображение части данных, хранящихся в корневом узле.
2. Обход левого поддерева в ширину путем рекурсивного вызова соответствующей функции.
3. Обход правого поддерева аналогичным образом.

На рис. 3.9 показан путь, по которому следует алгоритм обхода дерева в ширину.

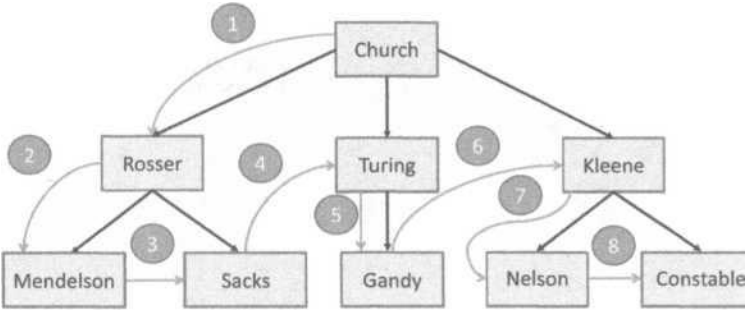


Рис. 3.9. Рекурсивный характер обхода дерева в ширину начинается с корневого узла и продолжается сначала влево, а затем вправо

Методу Tree .tap () передаются следующие обязательные для него параметры: корневой узел, с которого, по существу, начинается дерево, а также функция-итератор, преобразующая значение в каждом узле дерева:

```
Tree.map(church, p => p.fullname);
```

В результате обхода данного дерева в ширину и применения заданной функции преобразования к каждому узлу будет получен следующий результат:

```
'Alonzo Church', 'Barkley Rosser', 'Elliot Mendelson',
'Gerald Sacks', 'Alan Turing', 'Robin Gandy',
'Stephen Kleene', 'Nels Nelson', 'Robert Constable'
```

Такое представление об инкапсуляции данных для управления доступом к ним считается главным в функциональном программировании, когда дело доходит до обработки типов данных с учетом их неизменяемости и отсутствия у них побочных эффектов. О том, как расширить его дальше, речь пойдет в главе 5. Синтаксический анализ структур данных относится к числу самых главных вопросов разработки программного обеспечения вообще и функционального программирования в частности. В этой главе был углубленно рассмотрен функциональный стиль разработки программного обеспечения с использованием языковых средств JavaScript для ФП, реализованных в расширяемой функциональной библиотеке Lodash. Такой стиль отличается наличием рационализированной модели, где операции высокого уровня могут быть составлены в цепочку как последовательность стадий общего процесса, включая бизнес-логику, требующуюся для достижения желаемого результата.

Бесспорно, написание потокового кода приносит также выгоды для его повторного использования и модуляризации, хотя мы коснулись здесь этих вопросов лишь вскользь. Такое представление о потоковом программировании будет выведено на новый уровень в главе 4, где основное внимание будет уделено созданию настоящих конвейеров функций.

Резюме

Из этой главы вы узнали следующее.

- Используя функции высшего порядка, реализующие операции tap, reduce и filter, можно писать расширяемый код.

- Библиотека Lodash служит незаменимым средством для обработки данных и создания программ через управляющие цепочки, где ясно разграничены потоки данных и преобразования.
- Декларативный стиль функционального программирования позволяет создавать код, который легче понимать и анализировать.
- При сопоставлении абстракций высокого уровня со словарем языка SQL обнаруживается более глубокое представление о данных.
- Рекурсия позволяет решать самоподобные задачи и обычно требуется для синтаксического анализа рекурсивно определяемых структур данных.

На пути к повторно используемому, модульному коду

В этой главе...

- Сравнение цепочек функций с конвейерами
- Введение в функциональную библиотеку Ramda.js
- Исследование понятий карринга, частичного применения и связывания функций
- Создание модульных программ с помощью композиции функций
- Усовершенствование потока управления программой с помощью комбинаторов функций

Сложная система, которая работает, несомненно, получила свое развитие из простой системы, которая некогда работала.

Из книги *The Systems Bible* (Настольная книга по системам) *Джона Голла (John Gall),* издательство General Systemantics Press, 2012 г.

Модульность относится к одним из самых важных свойств крупных программных проектов. Она представляет ту степень, до которой программы могут быть разделены на более мелкие независимые части. Модульные программы выгодно отличаются тем, что их назначение нетрудно определить на основании их составных частей. Эти части (или подпрограммы) становятся повторно используемыми компонентами, которые могут быть полностью или частично перенесены в другие системы. Благодаря этому прикладной код становится более удобным для сопровождения и чтения, что в конечном итоге повышает производительность труда разработчиков. Применение модульности

наглядно демонстрирует следующий простой пример составления конвейера команд в оболочке системы Unix:

```
tr 'A-Z' 'a-z' <words.in | uniq | sort
```

Даже в отсутствие всякого опыта программирования на языке оболочки системы Unix нетрудно заметить, что этот код включает в себя последовательность стадий общего процесса преобразования слов из верхнего регистра букв в нижний, удаления дубликатов и сортировки того, что осталось. Канальная операция, обозначаемая знаком |, связывает вместе отдельные команды. Примечательно, что, имея четкие контракты, описывающие входные и выходные данные, небольшие программы можно соединять вместе для решения сложных задач. Если попытаться представить себе, как написать такую программу в традиционном стиле HaJavaScript, то на ум сразу придет несколько циклов, операций сравнения символьных строк, а возможно, и несколько условных операторов и глобальных переменных для слежения за всем процессом обработки текстовых данных. Очевидно, что такой стиль написания программы трудно назвать модульным. В программировании мы предпочитаем решать задачи, разбивая их на более мелкие части и воссоздавая из них целое решение.

В примерах из главы 3, “Меньше структур данных и больше операций”, для решения аналогичных задач применялись функции высшего порядка, тесно связываемые в цепочки методов, располагаемых каскадом над единственным объектом-оболочкой. А в этой главе мы расширим подобное представление дальше, чтобы создавать слабо связанные конвейеры с помощью композиции функций, что позволит с большей гибкостью создавать целые программы из независимых компонентов. Эти компоненты могут быть такими же мелкими, как функции, или крупными, как целые модули, которые сами по себе не представляют никакой ценности, но вместе придают смысл единому целому.

Создание модульного кода — дело непростое. Поэтому здесь будут представлены такие весьма важные методики ФП, как частичное применение и композиция. С помощью функциональной библиотеки Ramda.js они придают коду нужный уровень абстракции, позволяющий выражать решения в бесточечном стиле через декларативные конвейеры функций.

4.1. Цепочки методов в сравнении с конвейерами функций

В главе 3 мы остановились на цепочках методов, применяемых для последовательного соединения функций, раскрыв тем самым стиль функционального программирования, заметно отличающийся от любого другого стиля разработки программного обеспечения. Но имеется и другой подход к соединению функций, называемый *конвейеризацией*.

Изучая функции, полезно описывать их с точки зрения входных и выходных данных (рис. 4.1). Такой способ обозначения функций при их определении применяется, например, в языке Haskell и весьма распространен в сообществе программирующих функционально, поэтому он будет еще не раз продемонстрирован в последующих примерах.

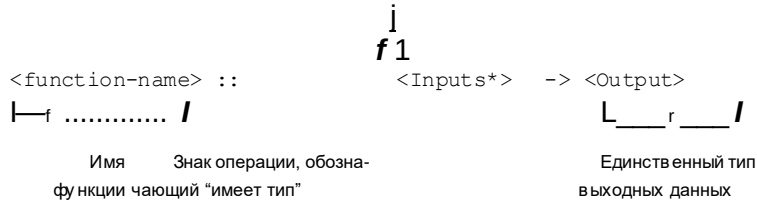


Рис. 4.1. Принятое в языке Haskell обозначение функции при ее определении. В этом обозначении описывается имя функции, после которого следует операция, задающая типы входных и выходных данных этой функции

Напомним, что в функциональном программировании функция служит для математического преобразования типов входных данных в типы выходных данных, как показано на рис. 4.2. Например, простая функция вроде isEmpty (), которой передается символьная строка и возвращающая логическое значение, может быть выражена в данном обозначении следующим образом:

```
isEmpty :: String -> Boolean
```

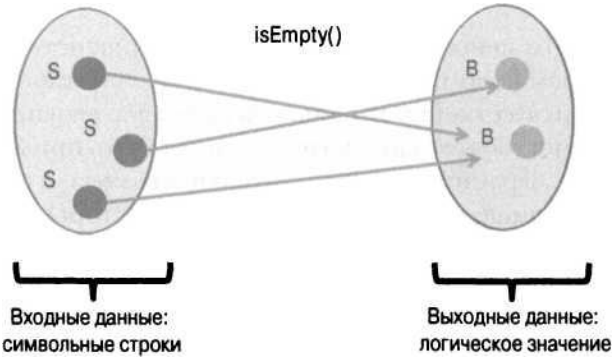


Рис. 4.2. Функция **isEmpty ()** служит для ссылочно-прозрачного преобразования всех строковых значений в логические значения

Данная функция выполняет ссылочно-прозрачное преобразование всех входных значений типа String в выходные значения типа Boolean. Ниже приведена форма лямбда-выражения данной функции BjavaScript вместе с ее сигнатурой.

```
// isEmpty :: String -> Boolean const isEmpty = s => Is || Is.trimO;
```

Представляя функции в виде преобразований типов данных, следует знать, каким образом они могут быть соединены в цепочку и конвейер. Для этого имеются следующие способы.

- Связывание методов в цепочку (тесная связь, но ограниченная выразительность).
- Организация конвейеров функций (слабая связь, но значительная гибкость).

4.1.1. Связывание методов в цепочку

Как упоминалось в главе 3, функциям, реализующим операции tap и filter, передается исходный массив в качестве входных данных, а они возвращают новый массив в качестве выходных данных. Эти функции могут быть тесно связаны в цепочку через неявно определяемый в библиотеке Lodash объект-оболочку, который внутренним образом управляет созданием новых структур данных. Ниже приведен пример такого связывания, взятый из главы 3.

```
_. chain (names)                    ◄ -- — После каждой точки можно
    .filter(isValid)                      вызвать только другие методы
    ,map(s => s.replaced /, • '))         « цепочки, управляемой
                                          средствами библиотеки Lodash

    .map(_.startcase)
    .sort()
    .value();
```

Очевидно, что это явное синтаксическое усовершенствование по сравнению с императивным кодом, заметно повышающее его удобочитаемость. Но, к сожалению, оно искусственно и тесно связано с владеющим объектом, ограничивающим количество методов, которые могут быть применены в цепочке, что, в свою очередь, ограничивает выразительность кода. В данном случае приходится обходиться лишь тем набором операций, которые предоставляет библиотека Lodash, но нельзя так же просто соединить функции из других (или собственных) библиотек в одной программе.

Примечание

Имеются способы расширить объект дополнительными функциональными средствами, используя примеси, но в этом случае приходится брать на себя ответственность за управление объектом примеси. В этой книге примеси не рассматриваются, но о них можно подробнее узнать из статьи, доступной по адресу https://javascriptweblog.wordpress.com/2011/05/.

Обобщенно простую последовательность методов обработки массивов можно представить так, как показано на рис. 4.3. Но было бы лучше разбить, так сказать, цепочку, чтобы получить намного больше свободы для организации последовательности независимых функций. И этой цели можно достичь с помощью конвейеров функций.

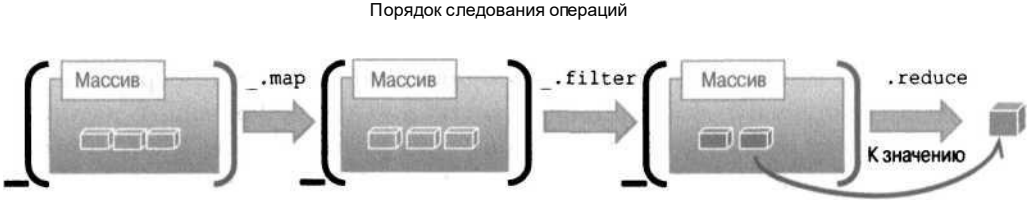


Рис. 4.3. Цепочка массивов, состоящая из методов, последовательно вызываемых через владеющий ими объект. Внутри ее каждый метод возвращает новый массив, содержащий результат вызова каждой функции

4.1.2. Организация функций в конвейеры

Функциональное программирование устраняет ограничения, присущие связыванию методов цепочку, и обеспечивает гибкость соединения любого набора функций независимо от их происхождения. Для этой цели служит *конвейер* — направленная последовательность функций, располагаемых свободно, но так, чтобы выходные данные из одной функции служили входными данными для другой. Общее представление об организации конвейера дает рис. 4.4, где соединены вместе функции, обрабатывающие разнотипные объекты.

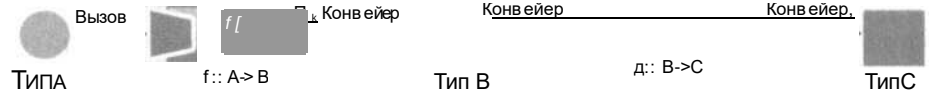


Рис. 4.4. Конвейер функций, начинающийся с функции f, которой передаются входные данные типа A и которая формирует объект типа B, который затем передается функции д, выводящей объект типа C как конечный результат. Функции f и g могут относиться к любой библиотеке или быть вашими собственными функциями

В этой главе представлены методики, позволяющие организовывать вызовы функций на высоком уровне в лаконичные конвейеры по такой же схеме, как и на рис. 4.4. Если схема покажется вам знакомой, это можно объяснить тем, что она следует объектно-ориентированному проектному шаблону “Каналы и фильтры” (Pipes and Filters), наблюдаемому во многих приложениях масштаба предприятия и навеянному принципами функционального программирования (фильтрами в данном случае служат отдельные функции).

При сравнении рис. 4.3 и 4.4 обнаруживается следующее главное отличие обоих рассматриваемых здесь способов соединения функций: при связывании в цепочку образуются тесные связи через методы объекта, тогда как конвейер соединяет входы и выходы любых функций, что приводит к слабо связанным компонентам. Но для того чтобы такая связь стала возможной, соединение функций должно быть совместимым с точки зрения количества аргументов и типов данных. Именно об этом и пойдет речь далее.

4.2. Требования к совместимости функций

Конвейеры применяются в объектно-ориентированных программах спорадически, в отдельных сценариях (зачастую при аутентификации/авториза- ции), тогда как в основу функционального программирования конвейеры положены как единый способ написания

программ. В зависимости от решаемой задачи зачастую возникает довольно заметный разрыв межу постановкой задачи и предлагаемым решением, поэтому вычисления должны выполняться на вполне определенных стадиях. Эти стадии представлены функциями, которые выполняются при условии, что их входные и выходные данные совместимы по следующим двум критериям.

- **Ъш.** Тип, возвращаемый предыдущей функцией, должен совпадать с типом аргумента последующей функции.
- **Количество аргументов.** Последующая функция должна быть объявлена хотя бы с одним аргументом или параметром, чтобы обработать значение, возвращаемое из вызова предыдущей функции.

4.2.1. Совместимые по типу функции

При проектировании конвейеров функций очень важно обеспечить соответствующий уровень совместимости данных, возвращаемых и передаваемых функциям. Что касается совместимости по типу данных, то обеспечить ее в JavaScript легче, чем в языках со строгой типизацией, поскольку}ауа5спр1 — это язык со слабой типизацией. Так, если поведение объекта на практике похоже на определенный тип, то он относится именно к этому типу. Это так называемая *утиная типизация* по следующему принципу: “Если это похоже на походку и кряканье утки, значит, это и есть утка”.

Примечание

Строго типизированные языки программирования обладают тем преимуществом, что в них применяются системы типов для предупреждения о возможных осложнениях без выполнения прикладного кода. Системы типов являются важной темой в функциональном программировании, но в этой книге они не рассматриваются.

Механизм динамической диспетчеризации, применяемый BjavaScript, пытается обнаружить свойства и методы объектов безотносительно к сведениям об их типе. И хотя это очень удобно, нередко требуется все же знать, какие именно типы значений предполагаются в функции. Если ясно определить их (возможно, задокументировав в коде с помощью принятого в языке Haskell обозначения), то прикладные программы будет легче понять.

Формально функции f и g считаются совместимыми по типу, если выходные данные функции f относятся к типу, эквивалентному тому типу, с которым за

даны входные данные функции д. В качестве примера ниже приведена простая программа, обрабатывающая номер социального страхования учащегося.

```
                                    I Удалить начальные и конечные пробелы
trim : : String -> String ◄---- *
normalize :: String -> String  ------ j Удалить любые знаки тире из входной символьной строки
```

Теперь можно проследить соответствие входных данных функции normalize () и выходных данных функции trim(), чтобы вызывать их в простой, организуемой вручную конвейерной последовательности, как демонстрируется в листинге 4.1.

Листинг 4.1. Построение конвейера из функций trim() и normalize () вручную

```
// trim :: String -> String const trim = (str) =>
str.replace(/^\s*I\s*$/g, // normalize :: String -> String
const normalize = (str) => str.replace(/\-/g, '');

normalize(trim(' 444-44-4444 ')); //-> '444444444'
```

Вызвать обе функции вручную в простом последовательном конвейере(далее поясняется, как автоматизировать этот процесс). Данная функция намеренно вызывается с начальными и конечными пробелами

Безусловно, совместимость функций по типам важна. Но BjavaScript она не так важна, как совместимость по количеству аргументов, передаваемых функциям.

4.2.2. Функции и арность: вариант для кортежей

Арность можно определить как количество аргументов, которые передаются функции. Она также означает длину функции. В других парадигмах программирования арность воспринимается как нечто само собой разумеющееся, но в функциональном программировании, как следует из принципа ссылочной прозрачности, количество аргументов, с которыми объявляется функция, нередко оказывается прямо пропорциональным ее сложности. Например, функция, обрабатывающая единственную символьную строку, вероятнее всего, окажется намного более простой, чем функция, которой передаются три или четыре аргумента:

```
// isValid :: String -> Boolean
function isValid (str) { ◄--- ----  Этой функцией пользоваться проще
```

```
// makeAsyncHttp:: String, String, Array -> Boolean  А этой функцией пользоваться
function makeAsyncHttp (method, url, data) {     < --  труднее, так как сначала должны
                                                       быть вычислены все ее аргументы
```

Пользоваться чистыми функциями, которым передается единственный аргумент, проще потому, что они служат только одной цели — единственной ответственности. А наша цель — пользоваться функциями с как можно меньшим количеством аргументов, поскольку они более гибкие и универсальные, чем те, что зависят от многих аргументов. Но, к сожалению, получить унарные функции нелегко. На практике такая функция, как isValid () в приведенном ниже примере, может быть дополнительно наделена сообщениями об ошибках, ясно описывающих, что произошло при ее выполнении.

```
isValid :: String -> (Boolean, String) ◄—I Возвратить структуру, хранящую состояние проверки I
                                          достоверности, а возможно, и сообщение об ошибке
```

```
isValid(' 444-444-44444'); // -> (false, 'Input is too long!')
```

Но как возвратить из функции два разных значения? В языках функционального программирования для этой цели поддерживается структура, называемая *кортежем (tuple)*. Это конечный, упорядоченный список элементов, обычно сгруппированных из двух или трех значений и обозначаемых как (a, Ь, c). Опираясь на понятие кортежа, можно воспользоваться им как значением, возвращаемым из функции isValid (), сгруппировав в нем состояние проверки достоверности вместе с возможным сообщением об ошибке, чтобы возвратить их как единое целое, а если потребуется, то соответственно передать другой функции. Рассмотрим кортежи более подробно.

Кортежи являются неизменяемыми структурами, упаковывающими вместе разнотипные элементы, чтобы их можно было передавать другим функциям. Имеются и другие способы возврата произвольных данных, в том числе объектные литералы и массивы:

```
return {
    status : false, или return [false, 'Input is too long!'];
    message: 'Input is too long!'
I;
```

Но что касается обмена данными между функциями, то в этом отношении кортежи предоставляют следующие дополнительные преимущества.

- **Неизменяемость.** Как только кортеж будет создан, изменить его внутреннее содержимое уже нельзя.
- **Отсутствие потребности создавать специальные типы.** В кортежи могут быть объединены значения, вообще не связанные друг с другом. Поэтому определение и получение экземпляров новых типов только для группировки данных излишне усложняет их модель.
- **Отсутствие потребности создавать неоднородные массивы.** Обработка массивов, состоящих из разнотипных элементов, затруднена потому, что для этого требуется написать код с большим количеством защитных проверок типов. По традиции массивы предназначены для хранения объектов одного типа.

Более того, кортежи ведут себя как объекты-значения, упоминавшиеся в главе 2, “Сценарий высшего порядка”. Конкретным тому примером служит простой тип данных Status, состоящий из признака проверки состояния и сообщения: (false, ’ Some error occurred! ’). В отличие от других языков

функционального программирования, например Scala, BjavaScript отсутствует собственная поддержка типа данных Tuple для кортежей. Так, если имеется следующее определение кортежа в Scala: `var t = (30, 60, 90)`

то доступ к отдельным его составляющим можно получить следующим образом:

```
var sumAnglesTriangle = t._1 + t._2 + t._3 = 180
```

Тем не менее BjavaScript предоставляются все стандартные средства, требующиеся для реализации собственной версии типа данных Tuple, как демонстрируется в листинге 4.2.

Листинг 4.2. Реализация типа данных Tuple

```
const Tuple = function ( /* типы */ ) {

   const typeinfo = Array.prototype.slice.call (arguments, 0);

  const _T = function( /* значения */ ) {

    const values = Array.prototype.slice.call (arguments, 0);

    if (values. some (
      val ==> val === null || val === undefined)) {
      throw new ReferenceError (' Tuples may not
        have any null values');
    }

    if (values. length !== typeinfo, length) {
      throw new TypeError ('Tuple arity does
        not match its prototype');
    }

    values .map ( (val, index) => {
      this['_' + (index + 1)] = checkType(typeinfo[index])(val);
      }, this);

    Object.freeze(this)

  ; };

  _T.prototype.values = ()=>{
    return Object.keys(this)
      .map(k => thisfk], this);
  };
  return T;
```

Прочитать предоставляемые типы i аргументов, содержащихся в кортеже I

Объявить внутренний тип_Т, отвечающий за проверку соответствия типов и значений

Извлечь значения, которые должны храниться в кортеже I

Выполнить проверку на непустые значения. Функциональные типы данных не должны допускать проникновение п»етмхзначений

Проверить на соответствие количество аргументов | в кортеже и количество определенных типов |

Проверить, соответствует ли каждое переданное значение правильному типу в определении кортежа, используя представленную далее функцию checkType (). Каждый элемент кортежа будет преобразован в свойство кортежа, обозначаемое как_п, где n — индекс элемента, начиная с 1

Сделать кортеж неизменяемым экземпляром

Извлечь все значения из кортежа в виде массива. Это можно сочетать вместе с деструктурированным присваиванием, внедренным в версии ES6, чтобы преобразовать значения из кортежа в переменные

Объект типа Tuple, определение которого приведено в листинге 4.2, является неизменяемой структурой фиксированной длины, применяемой для хранения неоднородного множества значений *n* типов, которыми могут обмениваться функции. Этим объектом можно, например, воспользоваться для быстрого построения объектов-значений типа Status:

```
const Status = Tuple(Boolean, String);
```

А теперь завершим предыдущий пример программы, обрабатывающей номер социального страхования учащегося, выгодно воспользовавшись кортежами (листинг 4.3).

Листинг 4.3. Применение кортежей в функции isValid ()

```
// trim :: String -> String
const trim = (str) => str.replace(/^\s*|\s*$/g, '');

// normalize :: String -> String
const normalize = (str) => str.replace(/\-/g, '');

// isValid :: String -> Status
const isValid = function (str) {
    if (str. length === 0) {
        return new Status (false,
                'Invald input. Expected non-empty value!');
    }
    else {
        return new Status(true, 'Success!');
    }
}

isValid(normalize(strim('444-44-4444'))); // -> (true, 'Success!')
```

Объявить тип Status, чтобы хранить значения для состояния проверки (типа Boolean) и сообщения (типа String)

Двухэлементные кортежи настолько часто встречаются в программном обеспечении, что их стоит сделать объектами первого класса. А если воспользоваться еще и *деструктурированным присваиванием,* поддержка которого реализована в стандарте ES6, то значения из кортежа можно преобразовывать в переменные ясным и понятным образом. Так, с помощью кортежей в примере кода из листинга 4.4 создается объект типа StringPair.

Листинг 4.4. Определение типа StringPair

```
const StringPair = Tuple(String, String);
const name = new StringPair('Barkley', 'Rosser');

[first, last] = name.values!);
first; // -> 'Barkley'
last; // -> 'Rosser'

const fullname = new StringPair('J', 'Barkley', 'Rosser');
```

Здесь возникает ошибка в связи с несоответствием количества аргументов

Кортежи служат одним из средств для сокращения количества аргументов функции, но существует и лучшая альтернатива на те случаи, когда кортежи оказываются неэффективными. Придадим обсуждению большей остроты, представив методику *карринга функций,* которая не только позволяет абстрагировать количество аргументов, но и способствует модульности и повторному использованию кода.

4.3. Вычисление каррированных функций

Передать значение, возвращаемое из функции, в качестве входных данных унарной функции нетрудно. Но что, если целевой функции передается больше параметров? Чтобы уяснить назначение карринга BjavaScript, нужно сначала разобраться, чем *каррированная*

функция отличается от обычной (некарриро- ванной) функции в плане вычислений. В языке JavaScript разрешается вызывать обычную (т.е. некаррированную) функцию для выполнения с отсутствующими аргументами. Иными словами, если определить функцию f (a, Ь, c) и вызвать ее только с аргументом a, то ее вычисление будет продолжено, а интерпретатор JavaScript автоматически установит неопределенное (undefined) значение для аргументов Ь и c, как показано на рис. 4.5. Это прискорбная и наиболее вероятная причина, по которой карринг не является встроенным языковым средством JavaScript. Нетрудно представить, что если не объявить какие-нибудь аргументы и полагаться только на объект arguments в теле функции, то создавшееся положение только усугубится.

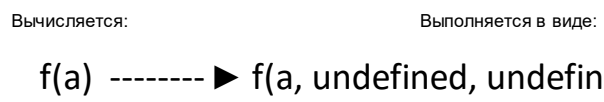


Рис. 4.5. Вызов некаррированной функции с отсутствующими аргументами приводит к автоматическому вычислению недостающих параметров и присвоению им неопределенных (undefined) значений

А каррированной называется функция, где все аргументы определены явно. Поэтому, если вызвать такую функцию с неполным набором аргументов, то вместо результата она возвратит новую функцию, которой нужно будет передать недостающие аргументы перед тем, как она возвратит результат вычисления. Порядок вычисления каррированной функции наглядно показан на рис. 4.6.

Карринг — это методика преобразования функции со многими переменными в постепенно формируемую последовательность унарных функций путем приостановки или "откладывания" ее выполнения до тех пор, пока не будут предоставлены все аргументы, что может произойти позднее. Ниже приведено формальное определение карринга трех параметров функции.

```
curry(f) :: ((a,b,c) -> d) -> a -> b -> c -> d
```

В этом формальном определении предполагается, что curry — это постепенное преобразование одних функций в другие с целью разобрать входные данные (a,Ь, c) на вызовы отдельных функций с единственным аргументом. В таких языках исключительно функционального программирования, как Haskell, карринг является встроенным языковым средством и автоматически

включается в определения всех функций. А BjavaScript карринг функций не выполняется автоматически, и поэтому приходится писать какой-нибудь поддерживающий код для активизации карринга. В связи с этим рассмотрим пример реализации карринга двух аргументов вручную, прежде чем перейти к автокар- рингу. Соответствующий исходный код приведен в листинге 4.5.

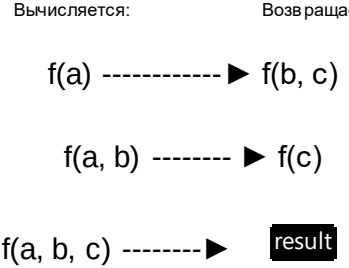


Рис. 4.6. Вычисление каррированной функции f. Эта функция выдает конкретный результат только в том случае, если ей переданы все аргументы. В противном случае она возвращает другую функцию, ожидающую получить остальные аргументы перед своим выполнением

Листинг 4.5. Реализации карринга двух аргументов вручную

```
function curry2(fn) {                          // Зафиксировать первый аргумент при
  return function (firstArg) { ◄ -- 1 первом вызове функции curry2 ()
      return function(secondArg) {             Зафиксировать второй аргумент при
          return fn(firstArg, secondArg);      втором вызове данной функции
      };                                         Возвратить результат применения данной
  };                                             функции
                                                 // с обоими аргументами
```

Как видите, карринг оказывается другим вариантом лексической области видимости (или замыкания), где возвращаемые функции являются ничем иным, как обычными вложенными функциями-обертками, фиксирующими аргументы для последующего их использования. Ниже приведен простой пример применения карринга функций, реализованного в листинге 4.5.

```
const name = curry2(function (last, first) => { return new StringPair(last,
    first);
                                                 Если указаны оба аргумента,
                                                 то функция вычисляется
                                                 полностью
[first, last] = name('Curry')('Haskell').values();*<■
first;// -> 'Curry' last; // -> 'Haskell' name('Curry');
// -> Функциям
                                  А если указан только один аргумент, то возвращается
                                  другая функция вместо вычисления данной функции с
                                  неопределенным значением отсутствующего аргумента
```

Рассмотрим еще один пример применения функции curry2 () для реализации функции checkType (), употреблявшейся в исходном коде из листинга 4.2

при реализации типа Tuple. В данном примере используются функции из еще одной функциональной библиотеки под названием Ramda.js.

Еще одна функциональная библиотека?

Подобно библиотеке Lodash, в библиотеке Ramda.js предоставляется немало полезных функций для соединения функциональных программ, а также средства, позволяющие программировать исключительно в функциональном стиле. Данной библиотекой рекомендуется пользоваться потому, что ее средства удобно организованы для упрощения реализации карринга, частичного применения и композиции, о которых речь пойдет далее в этой главе. Подробнее об установке библиотеки Ramda.js см. в приложении.

Установив библиотеку Ramda.js, можно воспользоваться глобальной переменной R для доступа ко всем ее функциональным средствам, включая функцию R. is ():

```
// checkType :: Type -> Object -> Object
const checkType = R.curry((typeDef, obj) => {
  if(!R.is(typeDef, obj)) {                    Вызвать функцию R. is () для
    n                                          I проверки сведений о типе
    let type = typeof obj;
    throw new TypeError('Type mismatch. Expected
      [${typeDef}] but found [${type}]');

  return obj;



checkType(String)('Curry');   // ->'Curry'
checkType(Number)(3);         // -> 3
checkType(Number)(3.5);       // -> 3.5

let now = new Date(); checkType(Date)(now); checkType(Object)({});
checkType(String)(42);        // -> now
                              //->{}
                              // -> Ошибка типа TypeError
```

Функция curry2 () вполне подходит для решения простых задач, но стоит приступить к созданию более сложных функциональных средств, вам сразу же потребуется автоматическая обработка произвольного количества аргументов. Как правило, в примерах из этой книги внутреннее устройство функций демонстрируется полностью, но функция curry () слишком длинная и сложная для пояснения, и поэтому, избавив вас от лишних хлопот, перейдем к более полезному обсуждению (конкретную реализацию функции curry () и ее разновидностей curryRight (), curryN () и т.д. можно найти в библиотеках Lodash и Ramda).

С помощью функции R.curry () можно сымитировать механизм автоматического карринга, применяемый в языках исключительно функционального программирования для обработки произвольного количества аргументов. Автоматический карринг можно представить как искусственно созданные области видимости вложенных функций, соответствующие количеству объявленных аргументов. В следующем примере осуществляется автоматический карринг функции fullname ():

```
// fullname :: (String, String) -> String const fullname = function (first, last)
{
```

```
}
```

Несколько аргументов данной функции преобразуются в унарные функции, определяемые в следующей форме:

```
// fullname :: String -> String -> String const fullname = function (first) {
return function (last) {

    }
}
```

А теперь перейдем к некоторым практическим примерам применения карринга. В частности, его можно использовать для реализации следующих весьма распространенных проектных шаблонов.

- Эмуляция функциональных интерфейсов.
- Реализация повторно используемых, модульных шаблонов функций.

4.3.1. Эмуляция фабрик функций

Интерфейсы в ООП — это абстрактные типы, применяемые для определения контракта, который должен быть реализован в классах. Так, если создать интерфейс с функцией findStudent (ssn), то она должна быть реализована в конкретных классах, реализующих этот интерфейс. Рассмотрим следующий "краткий" пример, демонстрирующий этот принцип реализации интерфейсов в языке Java:

```
public interface Studentstore {
    Student findStudent(String ssn);
}

public class DbStudentStore implements Studentstore {
    public Student findStudent(String ssn) {
        // ...
        ResultSet rs = jdbcStmt.executeQuery(sql);
        while(rs.next()){
            String ssn = rs.getString("ssn");
            String name = rs.getString("firstname") +
                          rs.getString("lastanme"); return new Student(ssn,
            name);
        }
    }
}
public class CacheStudentStore implements Studentstore { public Student
findStudent(String ssn) { // ...
        return cache.get(ssn);
    }
}
```

Извините за столь длинный фрагмент кода, но ведь Java — многословный язык! В этом примере кода демонстрируются две реализации одного и того же интерфейса: одна — для

чтения сведений об учащихся из базы данных, а другая — для чтения тех же самых сведений из кеша. Но для выполнения кода имеет значение лишь вызов метода, а не происхождение объекта. Эта замечательная особенность ООП воплощается в проектном шаблоне "Фабричный метод" (Factory Method). Используя фабрику функций, можно получить надлежащую реализацию следующим образом:

```
Studentstore store = getStudentStore(); store.findStudent("444-44-4444");
```

Чтобы не покидать область ФП, можно выбрать карринг в качестве подходящего решения. В частности, преобразуя кодДауа в код}ауа5спр1, можно создать функцию, производящую поиск объектов учащихся как в информационном хранилище, так и в массиве. Ниже приведены обе ее реализации.

```
// fetchStudentFromDb :: DB -> (String -> Student)
                                                    Искать в базе данных
const fetchStudentFromDb = R. curry (function (db, ssn) { — отитов return find(db,
    ssn);
}) ;

// fetchStudentFromArray :: Array -> (String -> Student)
const fetchStudentFromArray = R. curry (function (arr, ssn) { < ---- Искать в массиве
    return arr[ssn];
}) ;
```

Благодаря тому что функции каррированы, определение функции можно отделить от ее вычисления с помощью обобщенного фабричного метода findStudent (), подробности реализации которого можно было бы вывести из одной из следующих реализаций:

```
const findStudent = useDb ? fetchStudentFromDb(db)
                          : fetchStudentFromArray(arr); findStudent('444-44-
4444');
```

Теперь фабричный метод findStudent () можно передать другим модулям, а вызывающему его коду даже не потребуется знать конкретную его реализацию (это обстоятельство будет иметь большое значение для имитации взаимодействия с информационным хранилищем в ходе модульного тестирования, рассматриваемого в главе 6). Что же касается повторного использования кода, то карринг позволяет также создать целое семейство шаблонов функций.

4.3.2. Реализация повторно используемых шаблонов функций

Допустим, требуется настроить разные функции протоколирования на обработку различных состояний в приложении, в том числе ошибок, предупреждений, отладки и т.д. В шаблонах функций определяется семейство связанных вместе функций, исходя из количества аргументов, каррированных в момент создания. В следующем примере применяется весьма распространенная библиотека Log4js, обладающая намного большими возможностями для протоколирования состояния кода JavaScript, чем типичная функция console. log (). Подробные сведения об ее установке см. в приложении. Ниже приведена основная установка режима протоколирования.

```
const logger = new Log4js.getLogger('StudentEvents');
logger.info('Student added successfully!');
```

Но библиотека Log4js способна на гораздо большее. Допустим, протокольные

сообщения требуется отображать во всплывающем окне на экране. С этой целью можно настроить соответствующий аппендер следующим образом:

```
logger.addAppender(new Log4j s.JSAlertAppender());
```

Имеется также возможность изменить компоновку, настроив поставщик компоновки таким образом, чтобы выводить сообщения в формате JSON, а не простым текстом:

```
appender.setLayout(new Log4j s.JSONLayout());
```

В данной библиотеке имеется немало настраиваемых параметров, но копирование и вставка кода их настройки в каждый файл приводит к значительному дублированию. Вместо этого воспользуемся каррингом, чтобы определить повторно используемый шаблон функций (модуль регистратора, если угодно), чтобы добиться наибольшей степени гибкости и повторного использования. Соответствующий исходный код приведен в листинге 4.6.

Листинг 4.6. Создание шаблона функций для регистратора

```
const logger = function(appender, layout, name, level, message) {
    const appenders = {                    ◀I----- -----j Определить ряд готовых аппендеров
        'alert': new Log4js.JSAlertAppender(),
        'console': new Log4js.BrowserConsoleAppender()
    };
    const layouts = {    ————-1 Определить ряд готовых поставщиков компоновки
        'basic': new Log4js.BasicLayout(),
        'json': new Log4js.JSONLayout(),
        'xml' : new Log4js.XMLLayout()
    };
    const appender = appenders[appender];
    appender.setLayout(layouts[layout]);
    const logger = new Log4j s. getLogger (name) ; .выдать команду протоколиро-logger.
    addAppender (appender) ;                    вания со всеми примененны-
    logger. log (level, message, null);  —— - - - J ми параметрами настройки
};
```

Если теперь произвести карринг функции logger (), то можно централизованно управлять регистраторами и повторно использовать их в каждом отдельном случае, как показано ниже.

```
const log = R.curry(logger)('alert', ' json', *FJS');^ ----- < Вычислить все
                                                              аргументы, кроме
log('ERROR', 'Error condition detected!!');                    двух последних

// -> это приведет к появлению предупреждающего окна
// с запрашиваемым сообщением
```

Если же требуется реализовать несколько операторов обработки ошибок в одной функции или файле, то для этой цели имеется удобная возможность установить все параметры, кроме последнего:

```
const logError = R.curry(logger)('console', 'basic', 'FJS', 'ERROR');
logError('Error code 404 detected!!');
logError('Error code 402 detected!!');
```

Внутри данной функции делаются последовательные вызовы функции curry (), чтобы в

конечном итоге получить унарную функцию. Благодаря возможности создавать новые функции из существующих и передавать им любое количество параметров упрощается постепенное создание функций по мере определения их аргументов.

Помимо извлечения немалых выгод из повторного использования кода, как упоминалось выше, принципиальное новшество, которое вносит карринг, состоит в преобразовании функций со многими аргументами в унарные функции. Альтернативой каррингу служат *частичное применение функций* и *привязка* их *параметров.* Обе эти методики находят умеренную поддержку в языке JavaScript и предназначены для получения функций с меньшим количеством аргументов, но вполне пригодных для соединения в конвейеры.

4.4. Частичное применение и привязка параметров

Частичное применение (partial application) — это операция, инициализирующая фиксированными значениями подмножество параметров функции с постоянным количеством аргументов, создавая в итоге функцию с меньшим количеством аргументов. Проще говоря, если имеется функция с пятью параметрами, а ей предоставляются лишь три аргумента, то в конечном счете получается функция, ожидающая получить два оставшихся параметра.

Подобно каррингу, частичное применение может служить для непосредственного сокращения длины функции, хотя и несколько иным образом. Каррированная функция, по существу, является частично применяемой функцией, и поэтому обе эти методики часто путают. Главное их отличие заключается во внутреннем механизме и контроле над передачей параметров. В частности:

- карринг формирует вложенные унарные функции при каждом частичном вызове. Конечный результат формируется внутренним образом из постепенной композиции этих унарных функций. Кроме того, он дает возможность полного контроля над тем, как и когда происходит вычисление функции;
- частичное применение привязывает аргументы функции к предопределенным значениям (или присваивает их) и формирует новую функцию с меньшим количеством аргументов. Получающаяся в итоге функция содержит фиксированные параметры в своем замыкании и *полностью* вычисляется при последующем вызове.

Итак, прояснив главные отличия обеих методик, перейдем к исследованию возможной реализации функции partial (), как демонстрируется в листинге 4.7.

Листинг 4.7. Реализация функции partial ()

```
function partial () {
    let fn = this, boundArgs = Array.prototype.slice.call(arguments);
    let placeholder = «partialPlaceholderObj»; ■---------
    let bound = function () {
        let position = 0, length = boundArgs.length;
        let args = Array(length);
        for (let i = 0; i < length; i++) {
            args[i] = boundArgs[i] === placeholder -----
                ? arguments[position++] : boundArgs[i];
        }
        while (position < arguments.length) {
            args.push(arguments[position++]); } return fn.
            apply (this, args);^-----------------------
        I.
        Function.apply(), return bound;
    }) e
```

В реализациях функции partial () из таких библиотек, как Lodash, в качестве заполнителей применяются объекты из библиотеки Underscore.js, а в других специальных реализациях — неопределенные значения, указывающие на то, что данный параметр следует пропустить

Создать новую функцию со всеми частично применяемыми параметрами

Объект-заполнитель partialPlaceholderObj пропускает определение параметра функции для дальнейшего вызова, поэтому можно выбрать, какие параметры привязать и какие из них предоставить как часть вызова (см. приведенные далее примеры)

Применить метод чтобы вызвать данную функцию с подходящим контекстом и всеми привязанными аргументами

Для целей обсуждения методик частичного применения и связывания функций мы снова воспользуемся библиотекой Lodash, поскольку связывание функций в ней поддерживается несколько лучше, чем в библиотеке Ramda. Но, по существу, функция, реализующая операцию _. partial, применяется аналогично функции R.curry (), причем в обеих функциях поддерживаются аргументы-заполнители с соответствующими объектами-заполнителями. Обратившись снова к демонстрировавшейся ранее функции logger (), можно частично применить определенные параметры, чтобы реализовать более конкретное поведение:

```
const consoleLog = partial(logger, 'console',
                           'json', 'FJS Partial');
```

Воспользуемся данной функцией, чтобы лишний раз подчеркнуть отличия карринга от частичного применения. После применения трех аргументов получающаяся в итоге функция consoleLog () ожидает получить два других аргумента, когда она вызывается, причем не постепенно, а все сразу. Таким образом, в отличие от карринга, вызов функции consoleLog () с единственным аргументом приведет не к возврату новой функции, а к

вычислению данной функции с установленным неопределенным (undefined) значением последнего аргумента. Но ничто не мешает и далее частично применять аргументы к функции consoleLog () с помощью все той же операции .partial:

```
const consolelnfoLog = partial(consoleLog, 'INFO'); consolelnfoLog('INFO logger
configured with partial');
```

Карринг можно рассматривать как автоматический способ частичного применения, и в этом заключается его главное отличие от частичного применения. Имеется еще одна возможность — *связывание функций,* которое BjavaScript реализуется в виде метода Function.prototype .bind () J Но она действует несколько иначе, чем частичное применение, как показано ниже.

```
const log =_.bind(logger, undefined, 'console',
                  'json', 'FJS Binding');
log('WARN', 'FP is too awesome!');
```

А что означает неопределенное (undefined) значение второго аргумента для операции _. bind? Связывание позволяет создавать связываемые функции, которые можно выполнять в контексте владеющего объекта, поэтому передачей неопределенного значения второго аргумента интерпретатору предписывается привязать данную функцию к глобальному контекс гу. Рассмотрим следующие примеры практического применения операций .partial и .bind.

- Расширение базовых языковых средств.
- Связывание отложенных функций.

4.4.1. Расширение базовых языковых средств

Частичное применение может служить для расширения базовых типов данных вроде String и Number полезными служебными функциями, повышающими выразительность языка. Не следует, однако, забывать, что подобное расширение базовых языковых средств может сделать прикладной код менее переносимым на обновленные версии платформы, если дополнить эти средства новыми конфликтующими методами. Рассмотрим следующие примеры подобного расширения:

[1] **См. описание метода** `Function.prototype .bind ()` **на странице документации Mozilla Developer Network по адресу** `https : //developer .mozilla. org/ru/docs/Web/ JavaScript/Reference/Global_Objects/Function/bind.`

```
// Извлечь первые N символов
String.prototype.first = _.partial(String.prototype.substring, 0, _); ◄-------

'Functional Programmingfirst(3); // -> 'Fun'

// Преобразовать любое имя в формат "Фамилия, Имя"
String.prototype.asName =
    _.partial(String.prototype.replace, /(\w+)\s(\w+)/,    '$2, $1');    ◄ ----

'Alonzo Church'.asName(); // -> 'Church, Alonzo'

// Преобразовать символьную строку в массив
String.prototype.explode =
      partial(String.prototype.match, /[\w]/gi);

'ABC'.explode(); //-> ['A', 'B', 'C']

// Проанализировать синтаксически простой URL
String.prototype.parseUrl =     partial (String, prototype, match,
       /(http[s]?I ftp):\/\/([Л:\/\s]+)\.(Г:\/\s]{2,5})/);

[ 'http://example.com', 'http', 'example', 'com' ]
```

Используя заполнитель, можно частично применить подстроку, начиная с нулевого индекса, и создать функцию, ожидающую величину смещения

Частично применить определенные параметры, чтобы создать конкретное поведение

Частично применить совладение с отдельными регулярными выражениями, чтобы преобразовать символьную строку в массив, содержащий конкретные данные

Прежде чем реализовывать собственную функцию, убедитесь, что она совместима с последними обновлениями языка:

```
if(!String.prototype.explode) {
    String.prototype.explode =
        _.partial(String.prototype.match, /[\w]/gi); }
```

Иногда частичное применение не действует, например, в тех случаях, когда применяются отложенные функции вроде setTimeout (). В таких случаях следует воспользоваться связыванием функций.

4.4.2. Связывание отложенных функций

Связывание функций для установки объекта контекста приобретает особое значение в тех случаях, когда приходится работать с функциями и методами, в которых ожидается наличие определенного владеющего объекта. Например, в браузере применяются функции setTimeout () и setinterval (), ожидающие установки ссылки this на глобальный контекст (т.е. объект window), а иначе они просто не работают. Установить такой контекст можно, передав соответствующей функции аргументе неопределенным (undefined) значением. Например, с помощью функции setTimeout () можно создать простой объект планировщика для выполнения отложенных заданий. В следующем примере демонстрируется применение операций .bind и _. partial:

```
const Scheduler = (function () {
   const delayedFn = _.bind(setTimeout, undefined, _, _);
    return {
        delay5: _.partial(delayedFn, 5000), delaylO: _.partial(delayedFn, 10000),
        delay:     partial(delayedFn, _, _)
    };
}) () ;
```

```
Scheduler.delay5(function () {
    consoleLog('Executing After 5 seconds'') });
```

Используя объект типа Scheduler, можно вызвать любой фрагмент кода, заключенный в тело функции, с определенной задержкой (время задержки такого таймера не гарантируется интерпретатором, но это уже другой вопрос). Функции bind () и partial () возвращают другие функции, и поэтому их нетрудно сделать вложенными. Как следует из приведенного выше примера кода, каждая отложенная операция составляется из композиции привязанной и частично применяемой функций. Связывание функций не столь полезно в функциональном программировании, как их частичное применение, да и пользоваться им труднее, поскольку оно требует устанавливать каждый раз контекст функции. А здесь оно рассматривается на тот случай, если вам придется иметь с ним дело, самостоятельно изучая данный вопрос.

Обе рассмотренные выше методики карринга и частичного применения функций в равной степени полезны. Карринг чаще всего применяется для создания функций-оберток, абстрагирующих поведение функции для предварительной установки ее аргументов или же для частичного их вычисления. Это приносит выгоды потому, что чистые функции с меньшим количеством аргументов более удобны в обращении, чем функции со многими аргументами. Любая из этих методик упрощает предоставление подходящих аргументов, чтобы функциям не приходилось явным образом обращаться к объектам за пределами их области видимости, в то же время сводя их к унарным функциям. Благодаря обособлению логики получения этих необходимых данных повышается степень повторного использования функций, а главное — упрощается их композиция.

4.5. Составление конвейеров функций

В главе 1, “Основы функционального программирования”, обсуждалось, насколько важно уметь разбивать решаемую задачу на более мелкие подзадачи, чтобы, составив их вместе как в головоломке, прийти к искомому решению. Функциональные программы нацелены на получение требуемой структуры, приводящей к композиции, — главной опоры функционального программирования. Дойдя до этого раздела, вы должны ясно представлять, что именно понятия чистоты и отсутствия побочных эффектов у функций делают данную методику столь эффективной. Напомним, что побочные эффекты отсутствуют у такой функции, которая не зависит от любых внешних данных. Все, что

требуется такой функции, должно быть предоставлено в качестве аргументов. Чтобы пользоваться композицией надлежащим образом, следует освободить свои функции от побочных эффектов.

Более того, если программа построена из чистых функций, то и сама она становится чистой, допуская дальнейшую ее композицию как составной части еще более сложных решений, не противоречащей остальным частям проектируемой системы. Этот вопрос очень важно разобрать потому, что он станет главным для обсуждения в остальной части данной книги. Поэтому, прежде чем перейти к подробному рассмотрению особенностей композиции функций, рассмотрим конкретный пример компоновки виджетов на HTML-странице, чтобы дать ясное представление о самой композиции.

4.5.1. Представление о композиции HTML-виджетов

Понятие композиции самоочевидно и, безусловно, характерно не только для функционального программирования. Рассмотрим в качестве примера компоновку виджетов на HTML-странице. Сложные виджеты создаются из сочетания простых виджетов и, в свою очередь, могут составлять часть еще более крупных виджетов. Например, объединив три поля ввода текста с пустым контейнером, можно получить простую форму для заполнения сведений об учащемся, как показано на рис. 4.7.

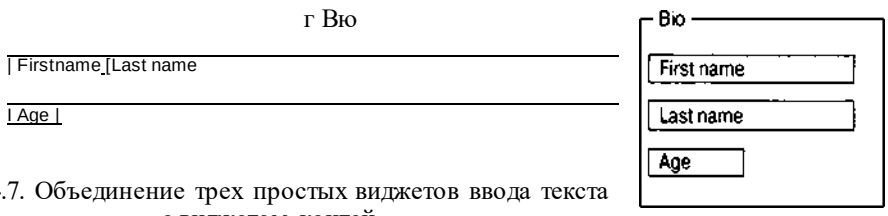


Рис. 4.7. Объединение трех простых виджетов ввода текста с виджетом контейнера для создания компонента биографической формы

Теперь форма для заполнения сведений об учащемся сама становится компонентом (или виджетом), который может быть составлен вместе с другими в более сложный компонент, чтобы создать полную консольную форму для заполнения сведений об учащемся (рис. 4.8). А далее виджет этой консольной формы может быть, если потребуется, внедрен в еще более крупную панель вроде личного кабинета. В данном случае консольная форма *составлена* из адресной и биографической форм. Объекты с простым поведением, не имеющие внешних зависимостей, удобны для композиции и могут быть использованы для построения сложных структур из простых подобно взаимосвязанным стандартным блокам.

В качестве примера создадим следующее рекурсивное определение кортежа Node:

```
const Node = Tuple(Object, Tuple);
```

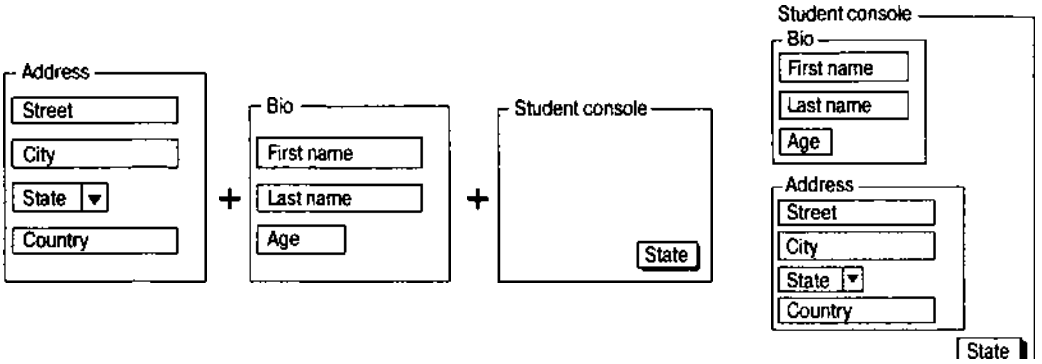


Рис. 4.8. Виджет консольной формы для заполнения сведений об учащемся, созданной из адресной и биографической форм, кнопки и контейнера

Такой кортеж может быть использован для хранения объекта и ссылки на другой кортеж (или узел). В конечном счете оказывается, что это, по существу, функциональное определение списка элементов, рекурсивно составляемого из головы и хвоста. Используя следующую каррированную функцию element (): `const element = R.curry((val, tuple) => new Node(val, tuple));`
можно создать список любого типа, завершающийся пустым (null) значением. В качестве примера на рис. 4.9 показан простой список чисел.

```
var grades = element(1, element(2, element(3, element(4, null))));
```

Рис. 4.9. Выделение головной и хвостовой частей, образующих список чисел. Голова и хвост вполне доступны как функции для обработки массивов в языках ФП

Именно таким образом в большей или меньшей степени составляются списки в языках ФП вроде ML и Haskell. А с другой стороны, для композиции сложных объектов с высокой степенью связывания с другими внешними объектами отсутствуют ясные правила, и поэтому манипулировать ими крайне сложно. Функциональную композицию постигает та же участь, когда возникают побочные эффекты и модификации. А теперь перейдем непосредственно к обсуждению композиции функций.

4.5.2. Композиция функций: отделение описания от вычисления

По существу, *композиция функций* представляет собой процесс группирования сложного поведения, разделенного на более простые задачи. Вкратце она упоминалась еще в главе 1, "Основы функционального программирования", а теперь настало время пояснить ее подробно. С этой целью обратимся к еле- дующему краткому примеру кода, где для объединения двух чистых функций применяется функция R. compose () из библиотеки Ramda:

```
const str = 'We can only see a short distance
```

```
            ahead but we can see plenty there that needs to be done';
                                                    | Разбить предложение const explode =
(str) => str. split (/\s+/) ; ◄ ------------------' на массив слов

const count = (arr) => arr. length; ........  ] Подсчитать слова

const countwords = R.compose(count, explode);

countwords(str); //-> 19
```

Такой код, безусловно, легко читать, а его назначение нетрудно понять, глядя на составляющие части функции. Данная программа примечательна тем, что она не вычисляется до тех пор, пока не будет выполнена функция countwords (). Иными словами, функции, передаваемые по имени (в данном случае функции explode () и count ()), находятся в режиме ожидания в самой композиции. А в результате композиции получается еще одна функция (в данном случае countwords ()), ожидающая своего вызова с соответствующим аргументом. Таким образом, вся прелесть композиции функций состоит в *отделении описания функции от ее вычисления.*

А теперь поясним, что происходит в самой композиции. В результате вызова функции countwords (str) выполняется функция explode () над заданным ей предложением, передавая в итоге свои выходные данные (массив символьных строк, на которые разбито исходное предложение) функции count (), вычисляющей длину полученного массива. Композиция связывает выходы со входами функций, по существу, образуя из них конвейер. Исследуем далее более формальное определение композиции. С этой целью рассмотрим следующие функции f и g с соответствующими типами их входных и выходных данных:

```
                                ! Функция g преобразует
g • • A -> B <____               | Данные типа A в тип B

f :: B -> C                     | Функция f преобразует
                                ' данные типа B в тип C
```

На рис. 4.10 стрелками обозначены соединения всех групп, объединяемых в композиции. Этот абстрактный пример наглядно показывает, что обозначенной стрелкой функции f передается аргумент типа B, а она возвращает значение типа C. А другой обозначенной стрелкой функции g передается аргумент типа A и она возвращает значение типа B. В результате композиции функций д::А->Ви£ : : B -> C (произносится как “композиция f из д”) получается еще одна обозначаемая стрелкой функция A -> C (рис. 4.11). Более формально это описание данной композиции можно выразить следующим образом:

```
f • g = f(g) = compose :: ((B -> C), (A -> B)) -> (A -> C)
```

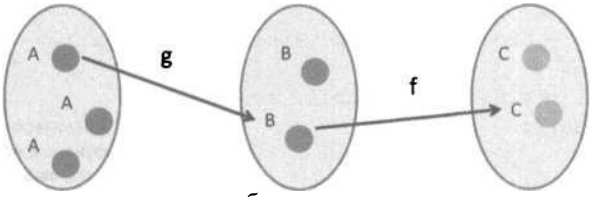


Рис. 4.10. Наглядная демонстрация преобразования типов входных и выходных данных в функциях f и д. В частности, функция д преобразует значения типа A в значения типа B, а функция f — значения типа B в значения типа C. Композиция происходит потому, что функции f и д совместимы по типам

Рис. 4.11. В результате композиции двух функций получается новая функция, непосредственно преобразующая входные данные первой функции в выходные данные второй. Композицию можно также рассматривать как ссылочно прозрачное преобразование входных данных в выходные

Напомним, что с точки зрения ссылочной прозрачности функции служат не более чем стрелками, соединяющими один объект из группы с другим. И это приводит к еще одному важному принципу разработки программного обеспечения, положенному в основу модульных систем. Благодаря тому что композиция слабо связывает совместимые по типу функции в пределах их границ (входных и выходных данных), она вполне удовлетворяет принципу *программирования на основе интерфейсов.*

Так, в предыдущем примере была продемонстрирована композиция функций explode : : String -> [String] и count : : [String] -> Number. Иными словами, каждой функции известен или важен интерфейс последующей функции, но не важна его реализация. И хотя функция compose () не входит в состав JavaScript, ее можно естественным образом выразить как функцию высшего порядка (листинг 4.8).

Листинг 4.8. Реализация функции compose ()

```
function compose(/*функции */) {
   let args = arguments;

    let start = args.length - 1;
    return function () {
```

На выходе функции compose () получается другая функция, которой передаются конкретные аргументы

```
    let i = start; let result = args[start].apply(this, arguments)
    while (i —)
        result = args [i] . call (this, result);^

    return result;
};
```

Динамически применить данную функцию для переданных аргументов

Циклически вызывать последующие функции, исходя из предыдущего возвращаемого значения

Рис. 4.12. Сложные функции могут быть созданы путем композиции простых функций. По тому же принципу композиции функций из разных модулей, содержащих дополнительные функции, могут быть написаны целые программы

Данный принцип распространяется не только на функции, но и на целые программы, которые могут быть написаны путем композиции других чистых и лишенных побочных эффектов программ или модулей. (Исходя из приведенного ранее определения функции, употребляемого повсюду в этой книге, термины *функция, программа* и *модуль* в широком смысле обозначают любой исполняемый блок кода с входными и выходными данными.)

Композиция является *конъюнктивной операцией,* а это означает, что она соединяет операнды, используя логическую операцию И. Например, функция isValidSsn() составлена из функций checkLengthSsn () *и* cleaninput (). В этом отношении программы являются производными от суммы всех их составных частей. А в главе 5 будут обсуждаться задачи, требующие дизъюнктивного поведения, чтобы выразить условия, при которых функции

Правда, в библиотеке Rainda предоставляется реализация функции R.compose (), которой можно воспользоваться, чтобы не реализовывать ее самостоятельно. Рассмотрим в качестве примера следующую программу, проверяющую достоверность номера социального страхования учащегося (употребляемые в ней вспомогательные функции применяются во многих примерах из этой книги):

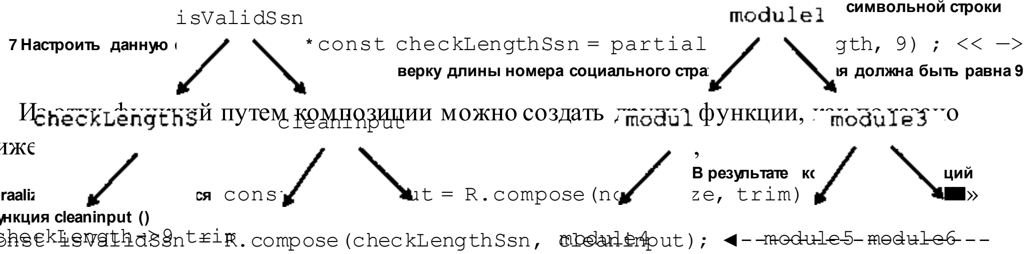
```
const trim = (str) => str. replace (/^\з* | \s*$/g, '');
const normalize = (str) => str.replace(/\-/д, '');
const validLength = (param, str) => str. length === param;
const checkLengthSsn = partial ...gth, 9) ;
```

1 Удалить любые пробелы до и после входных данных

Удалить все знаки тире

Проверить длину символьной строки

Настроить данную ... верку длины номера социального стра... ...я должна быть равна 9



Из этих функций путем композиции можно создать ... функции, ... ниже

```
const ...ut = R.compose(no...ze, trim)
const isValidSsn = R.compose(checkLengthSsn, cleaninput);
cleaninput (' 444-44-4444 '); //-> '444444444'
isValidSsnr 444-44-4444 '); //-> true
```

noraliz... ...ся функция cleaninput ()

В результате ко... ...ций

А в результате композиции функций cleaninput () и checkLengthSsn» получается еще одна новая функция

Расширив этот основополагающий принцип композиции, можно написать целые программы из сочетания простых функций, как показано на рис. 4.12.

могут возвращать один или два результата: А *ИЛИ*Ъ.

С другой стороны, доступный BjavaScript прототип Function можно расширить, введя функцию compose (). Как показано ниже, это можно сделать в таком же стиле, как и при связывании функций в цепочку, упоминавшемся в главе 3, “Меньше структур данных и больше операций”.

```
Function, prototype. compose = R. compose;
const cleaninput = checkLengthSsn.compose(normalize).compose(trim);
```

Функции снова связываются в цепочку с помощью точек

Если вам больше подходит именно такой способ, пользуйтесь им свободно. Как будет показано в следующей главе, подобный механизм связывания методов и функций в цепочку преобладает в функциональных алгебраических типах данных, называемых *монадами*. Рекомендуется все же придерживаться более функциональной формы композиции, поскольку она лаконичнее, гибче и лучше сочетается с функциональными библиотеками.

4.5.3. Композиция с помощью функциональных библиотек

Преимущество работы с такими функциональными библиотеками, как Ramda, заключается, в частности, в том, что все функции в них настроены надлежащим образом с помощью карринга, и благодаря этому они становятся универсальными средствами для композиции в конвейерах функций. Рассмотрим еще один пример, где составляется список учащихся класса с их оценками:

```
const students = [’Rosser', 'Turing', 'Kleene', 'Church'];
const grades = [80, 100, 90, 99];
```

Допустим, требуется найти учащегося с наивысшей оценкой в классе. Как пояснялось в главе 3, “Меньше структур данных и больше операций”, обращение с коллекциями данных является одним из краеугольных камней функционального программирования. Исходный код из листинга 4.9 составлен путем композиции нескольких каррированных функций, каждая их которых отвечает за преобразование приведенных выше массивов данных особым образом. Ниже вкратце поясняется назначение этих функций.

- Функция R. zip (). Создает новый массив, сопрягая содержимое смежных массивов. В данном случае сопряжение двух исходных массивов

приводит к появлению нового массива [['Rosser' , 80] , ['Turing' f 100], ...],

- Функция R.prop (). Указывает значение, применяемое при сортировке. В данном случае этой функции передается значение 1, указывающее на то, что вторым элементом массива является подмассив оценок.
- Функция R. sortBy (). Выполняет сортировку массива по заданному свойству в естественном порядке по нарастающей.
- Функция R. reverse (). Обращает весь массив, чтобы наибольшее число оказалось в первом его элементе.
- Функция R. pluck (). Строит массив, извлекая элемент по указанному индексу. Передаваемое ей нулевое значение указывает на элемент с именем учащегося.
- Функция R. head (). Принимает первый элемент массива.

Листинг 4.9. Определение самого успевающего учащегося

```
const smartestStudent = R.compose(
    R.head, R.pluck(0), R.reverse,
    R.sortBy(R.prop(l)), R.zip);

smartestStudent(students, grades); // -> 'Turing
```

Создать функцию smartestStudent (), определяемую как композицию ряда функций из библиотеки Ramda

Передача обоих массивов данной функции начинается с вызова функции R. zip (). На каждой стадии обработки этих массивов данные неизменяемо преобразуются из одного выражения в другое до тех пор, пока не будет получен конечный результат, извлекаемый с помощью функции R. head ()

Применение композиции может быть затруднено, особенно на начальной стадии освоения библиотеки Ramda или осмысления предметной области. Когда я пользуюсь композицией в своей работе, то нередко ловлю себя на мысли: с чего следует начать? И в этом случае самое трудное — разбить решаемую задачу на более мелкие подзадачи, после чего композиция сама принудит к воссоединению функций.

Что касается композиции функций, то вы быстро осознаете, и вам это начнет нравиться, насколько естественно и лаконично вы можете выразить все решение задачи в одной или двух строках. А поскольку вам придется создавать функции, подходящие для разных стадий реализуемого алгоритма, то начнете составлять онтологию, с помощью которой сможете состыковывать выражения, описывающие отдельные части решения, что позволит вашим коллегам быстрее понять ваш код. Исходный код из листинга 4.10 составлен аналогично упражнению из главы 3.

Листинг 4.10. Применение описательных псевдонимов функций

```
const first = R.head;
const getName = R.pluck(O); const reverse = R.reverse; const sortByGrade =
R.sortBy(R.prop(1));

const combine = R.zip;

R.compose(first, getName, reverse, sortByGrade, combine);
```

И хотя программу в таком виде легче читать, это не способствует ее повторному использованию, поскольку применяемые в ней функции подходят только для решаемой задачи. Вместо этого рекомендуется привыкнуть к словарю функций head (), pluck (), zip ()

и прочих, чтобы на практике приобрести всесторонние познания избранной библиотеки. Это поможет вам легче перейти к другим библиотекам или языкам ФП, поскольку в них зачастую применяются те же условные обозначения, что в конечном счете положительно скажется на производительности вашего труда.

В исходном коде из листингов 4.9 и 4.10 чистые функции применяются для выражения всего решения, но, как известно, такое возможно далеко не всегда. В разработке приложений возможны ситуации, когда приходится решать такие задачи, как чтение данных из локального хранилища или составление удаленных HTTP-запросов, что неизбежно приводит к побочным эффектам. В таком случае нужно суметь обособить и отделить нечистый код от чистого. И как будет показано в главе 6, это заметно упростит тестирование.

4.5.4. Меры борьбы с наличием нечистого и чистого кода

Нечистый код вызывает внешне наблюдаемые побочные эффекты после его выполнения и имеет внешние зависимости для доступа к данным за пределами области видимости составляющих его функций. Достаточно одной нечистой функции, чтобы нечистой стала вся программа.

Принимая во внимание сказанное выше, совсем не обязательно добиваться чистоты всех функций, чтобы пожать плоды функционального программирования. И хотя это идеальный случай, нужно научиться также мириться с чистым и нечистым поведением, ясно обособляя одно от другого — в идеальном случае, в отдельных функциях. А затем можно воспользоваться композицией, чтобы соединить вместе чистые и нечистые фрагменты кода. Напомним, что реализация требований к программе showStudent начиналась в главе 1, "Основы функционального программирования", со следующего выражения: `const showStudent = compose(append, csv, findstudent);`

Так или иначе, но большинство этих функций порождают побочные эффекты через те аргументы, которые они получают. В частности:

- в функции findStudent () используется ссылка на локальное хранилище объектов или какой-нибудь внешний массив;
- а функция append () непосредственно записывает или модифицирует элементы HTML-разметки.

Продолжим совершенствовать программу showStudent, воспользовавшись функцией curry () для частичного вычисления неизменяемых параметров

каждой функции. Введем также код для санобработки входных параметров и реорганизации HTML-операций с помощью более мелких функций. И, наконец, сделаем операцию find более функциональной, отделив ее от хранилища объектов. Усовершенствованная версия данной программы приведена в листинге 4.11.

Листинг 4.11. Версия программы showStudent с каррингом и композицией

```
// findObject :: DB -> String -> Object const
findObject = R.curry((db, id) => { const obj =
find(db, id);
    if(obj === null) {
      throw new Error('Object with ID [${id|]   not found');
    }
    return obj;
});

// findStudent : : String -> Student
const findStudent = f indOb ject (DB (' students ’));

const csv = ({ssn, firstname, lastname}) =>
    '${ssn}, ${firstname}, ${lastname}';

// append :: String -> String -> String const append = R.curry((elementld, info)
=> { document.queryselector(elementId).innerHTML = info; return info;
  });

    append(’#student-info’), csv, findStudent, normalize, trim);

showStudent(’44444-4444’); // -> 444-44-4444, Alonzo, Church
```

Реорганизованный метод f ind (), которому передается объект хранилища в качестве параметра с целью упростить композицию

Частично вычислить извлекаемую запись, указав хранилищ объектов учащихся и создав новую функцию findStudent()

Воспользоваться композицией, чтобы собрать всю программу в единый исполняемый блок

В исходном коде из листинга 4.11 определяются четыре функции, составляющие программу showStudent (их объявления дополнены сигнатурами типов, чтобы было легче проследить соответствие последовательных вызовов каждой из них). В данной программе все эти функции вызываются в обратном порядке, начиная с функции trim() и вплоть до функции append () и постепенно передают выходные данные из предыдущей на вход следующей функции. Но отвлечемся на мгновение и вспомним программу Unix, с которой начиналось обсуждение в этой главе. В этой программе каждая команда выполняется слева направо с помощью встроенной в Unix конвейерной операции |, как показано на рис. 4.13 А при соединении функций в конвейер составленные из них программы вычисляются в порядке, противоположном их композиции.

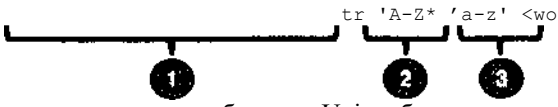


Рис. 4.13. Простая программа оболочки Unix, объединяющая в конвейер последовательность команд или других команд

Если обратный порядок выполнения функций по сравнению с их композицией покажется вам неестественным и вам потребуется наглядно представить свои программы в виде последовательности функций, выполняемых слева направо, воспользуйтесь для этой цели функцией pipe () из библиотеки Ramda, которая позволяет добиться того же самого

результата, вызывая функции, составляющие программу, в естественном порядке:

```
R.pipe(
     trim,
     normalize,
     findStudent,
     csv,
     append('#student-info'));
```

О том, насколько важен порядок выполнения функций, составляющих программу, свидетельствует встроенная в язык F# поддержка прямой конвейерной операции |>. Такая роскошь BjavaScript отсутствует, но имеется возможность надежно полагаться на функциональные библиотеки, чтобы эффективно добиваться желаемого результата. В отношении функций R. pipe () и R. compose () следует также заметить, что с их помощью создаются новые функции без явного объявления их формальных аргументов, как это обычно приходится делать. Композиция функций побуждает именно к такому стилю программирования, который так и называется *бесточечным,* а иначе — *комбинаторным.*

4.5.5. Введение в бесточечное программирование

Если внимательно проанализировать исходный код функции из листинга 4.10, то можно заметить, что в нем отсутствуют параметры всех ее составных функций, в отличие от их традиционного объявления. Приведем эту функцию еще раз:

```
R.compose(first, getName, reverse, sortByGrade, combine);
```

Применение функции R. compose () (или R.pipe ()) означает, что объявлять аргументы, называемые *точками* функции, вообще не нужно. Благодаря этому исходный код становится декларативным и более лаконичным, или *бесточечным.*

Бесточечное программирование приближает функциональный код Java Script к основному подходу, принятому в языке Haskell и системе Unix. Оно позволяет повысить уровень абстракции, побуждая мыслить категориями композиции компонентов на высоком уровне, а не беспокоиться о подробностях

вычисления функций на низком уровне. И здесь важную роль играет карринг, поскольку он дает удобную возможность частично определить все аргументы, кроме последнего, во встраиваемой ссылке на функцию. Такой стиль программирования называется также *неявным* или *комбинаторным* и во многом похож на порядок написания упомянутой выше программы Unix, пример реализации которой в бесточечном стиле HaJavaScript приведен в листинге 4.12.

Листинг 4.12. Версия написания программы Unix в бесточечном стиле на JavaScript с помощью функций из библиотеки Ramda

```
const runProgram = R.pipe(
    R.map(R.toLower),
    R.uniq,
    R. sortBy(R.identity));

runProgram(['Functional', 'Programming', 'Curry', 'Memoization', 'Partial',
            'Curry', 'Programming']);

// -> [curry, functional, memoization, partial, programming]
```

Воспользоваться функцией identity О, чтобы возвратить аргумент, с которым она была вызвана. Находит незначительное, но практическое применение, как поясняется в следующем разделе

Программа из листинга 4.12 состоит из бесточечных функциональных выражений, определяемых только по именам функций. И хотя их аргументы частично заданы, их типы не объявлены, а также не указан порядок соединения функций в более крупное выражение. Выполняя композицию в таком стиле программирования, не следует, однако, забывать, что злоупотребление им может сделать программы непонятными и запутанными. Не все в программе должно быть написано в бесточечном стиле. Иногда достаточно разбить композицию функций на две или три составляющие, чтобы добиться желаемого результата.

Бесточечный стиль написания кода может вызвать ряд вопросов, связанных с обработкой ошибок и отладкой программ. Иными словами, следует ли прибегать к возврату пустого значения null из составляемых функций из-за того, что генерирование исключений вызывает побочные эффекты? Проверка на пустое значение null в функциях вполне допустима, хотя она и приводит к появлению немалого объема дублирующегося стереотипного кода и предполагает возврат разумных значений по умолчанию, чтобы продолжить выполнение программы. А как отладить все команды и операторы, указанные в одной строке? Эти и другие вполне резонные вопросы разрешаются в следующей главе, где представлены дополнительные примеры бесточечных программ с автоматической поддержкой обработки ошибок.

Еще один очевидный вопрос возникает в тех случаях, когда требуется воспользоваться условной логикой или выполнять каким-то образом несколько функций подряд. В следующем разделе будут рассмотрены полезные служебные функции, предназначенные для организации потока управления прикладной программой.

4.6. Организация потока управления с помощью комбинаторов функций

В главе 3, “Меньше структур данных и больше операций”, было проведено сравнение потока управления программой в императивной и функциональной парадигмах и

подчеркнуты их существенные отличия. В императивном коде применяются процедурные механизмы управления вроде операторов if-else и for для организации потока выполнения программы, тогда как в функциональном коде они не применяются. Отказавшись от императивного стиля программирования, мы должны найти альтернативы, чтобы восполнить пробел в организации потока управления программой. И для этой цели мы можем воспользоваться *комбинаторами функций*.

Комбинаторы являются функциями высшего порядка, способными объединять более примитивные средства вроде других функций (или комбинаторов) и вести себя подобно управляющей логике. Как правило, в самих комбинаторах переменные не объявляются и не содержится бизнес-логика. Они предназначены для организации потока выполнения функциональной программой. Помимо функций compose () и pipe (), существует бесчисленное множество других комбинаторов, но здесь будут рассмотрены лишь некоторые из наиболее употребительных, в том числе следующие:

- identity()
- tap()
- alternation()
- sequence()
- fork () (join)

4.6.1. Тождественность (I-комбинатор)

Комбинатор identity () — это функция, возвращающая то же значение, которое было задано ей в качестве аргумента:

```
identity :: (a) -> a
```

Этот комбинатор широко применяется при проверке математических свойств функций, хотя он находит и другие примеры практического применения, в том числе следующие.

- Снабжение функций высшего порядка данными, которые требуются для вычисления аргумента функции, как это было сделано ранее при написании программы в бесточечном стиле (см. листинг 4.12).
- Модульное тестирование потока комбинаторов функций, где требуется простой результат выполнения функций, по которому делаются утверждения (подробнее об этом — в главе 6, “Отказоустойчивость приклад

ного кода"). Например, можно написать модульный тест для функции compose (), в которой применяются функции тождества.

- Функциональное извлечение данных из инкапсулированных типов (подробнее об этом — в следующей главе).

4.6.2. Ответвление (K-комбинатор)

Комбинатор tap () особенно удобен в качестве мостового перехода между пустыми функциями, в том числе протоколирующими или записывающими сообщения в файл или выводящими их на HTML-страницу, вставляя их в композицию с другими функциями и не требуя для этого написания дополнительного кода. С этой целью он получает одну функцию и возвращает другую. Ниже приведена его сигнатура как функции tap ().

```
tap :: (a -> *) -> a -> a
```

Этой функции передаются входной объект a и функция, выполняющая некоторое действие над этим объектом. Сначала она выполняет заданную функцию с предоставленным объектом, а затем возвращает этот объект. Например, функции R. tap () можно передать пустую функцию

```
const debugLog = partial(logger, 'console', 'basic', 'MyLogger', 'DEBUG');
```

и вставить ее в композицию других функций. Ниже приведены некоторые примеры.

```
const debug = R.tap(debugLog);
const cleaninput = R.compose(normalize, debug, trim);
const isValidSsn = R.compose(debug, checkLengthSsn, debug, cleaninput);
```

Вызов функции debug (), основанной на функции R. tap (), никоим образом не изменит результат выполнения программы. На самом деле этот комбинатор отбрасывает результат выполнения передаваемой ему функции, если таковой имеется. Так, в следующем примере кода вычисляется результат и по ходу дела выполняется отладка кода:

```
isValidSsn('444-44-4444');
// выводимый результат
MyLogger [DEBUG] 444-44-4444    // очищение входных данных
MyLogger [DEBUG] 444444444      // проверка длины
MyLogger [DEBUG] true           // конечный результат
```

4.6.3. Перемена (OR-комбинатор)

Комбинатор alt() позволяет выполнять простую условную логику, когда требуется обеспечить стандартное поведение в ответ на вызов функции. Этому комбинатору передается две функции, а он возвращает результат выполнения первой из них, если значение определено (т.е. не равно false, null или undefined). В противном случае он возвращает результат выполнения второй функции. Ниже показано, каким образом реализуется комбинатор alt (). `const alt = function (fund, func2) {`

```
    return function (val) {
        return fund (val) || func2(val);
    } };
```

С другой стороны, этот комбинатор можно написать более кратко, воспользовавшись

каррингом и лямбда-выражениями:

```
const alt = R.curry((fund, func2, val) => fund (val) || func2(val));
```

Комбинатор alt () можно применить в упоминавшейся ранее программе showStudent, чтобы справиться с ситуацией, когда операция выборки завершается неудачно, и в этом случае можно создать новый объект, представляющий учащегося:

```
const showStudent = R.compose(
     append('#student-info'), csv_f
     alt(findStudent, createNewStudent));

showStudent('444-44-4444');
```

Чтобы стало понятнее происходящее в этом коде, его лучше рассматривать как код, эмулирующий эквивалент простого оператора if-else в следующей императивной условной логике:

```
var student = findStudent('444-44-4444'); if(student !== null) {
     let info = csv(student);
     append('#student-info', info);
}
else {
     let newStudent = createNewStudent('444-44-4444');
     let info = csv(newStudent);
     append('#student-info', info);
}
```

4.6.4. Последовательность (S-комбинатор)

Комбинатор seq () служит для циклического выполнения последовательности функций. Ему передается две или три функции в качестве параметров, а он возвращает новую функцию, которая, в свою очередь, возвращает все эти функции в определенной последовательности относительно того же самого значения. Ниже приведена реализация комбинатора seq ().

```
const seq = function(/*функции*/) {
     const funcs = Array.prototype.slice.call(arguments);
     return function (val) {
          funcs.forEach(function (fn) { fn(val);
          });
     };
};
```

С помощью этого комбинатора можно выполнять последовательность связанных вместе, хотя и независимых операций. Например, как только будет найден объект, представляющий учащегося, можно воспользоваться комбинатором seq () как для воспроизведения этого объекта на HTML-странице, так и для его протокольного вывода на консоль, как показано ниже. Все функции будут выполняться именно в этом порядке относительно одного и того же объекта, представляющего учащегося.

```
const showStudent = R.compose(
```

```
    seq(
        append('#student-info'),
        consoleLog),
    csv,
    findStudent));
```

Комбинатор seq () не возвращает значение, а только выполняет следующие подряд действия. Если же его требуется внедрить в средину композиции, то можно воспользоваться комбинатором R. tap (), чтобы соединить его мостом с другими функциями.

4.6.5. Комбинатор разветвления

Комбинатор fork () оказывается удобным в тех случаях, когда требуется обработать данные из одного исходного ресурса двумя разными способами, а затем объединить полученные результаты. Этому комбинатору передаются три следующие функции: функция соединения и две оконечные функции, обрабатывающие предоставляемые им входные данные. Результат каждой разветвляемой функции в конечном итоге передается функции соединения join () в качестве одного из двух ее аргументов (рис. 4.14).

Рис. 4.14. Комбинатору fork() передается три функции: одна — join () и две — fund и func2 типа fork. В частности, функции типа fork выполняются относительно предоставляемых входных данных, а конечный результат объединяется в функции типа join

Примечание

Этот комбинатор не следует путать с каркасом разветвления (fork-join framework) в Java, который помогает организовать многопоточную обработку. Он реализует комбинатор fork (), который поддерживается в Haskell и других инструментальных средствах функционального программирования.

Комбинатор f or к () реализуется следующим образом:

```
const fork = function(join, fund, func2) {
    return function(val) {
        return join(fund(val), func2(val));
    };
};
```

А теперь покажем этот комбинатор в действии. С этой целью вернемся к представленному ранее примеру вычисления средней оценки из чисел в массиве. С помощью комбинатора f or k () можно скоординировать вычисление трех служебных функций следующим образом:

```
const computeAverageGrade =
    R.compose(getLetterGrade, fork(R.divide, R.sum, R.length));
```

```
computeAverageGrade([99, 80, 89]); // -> 'B'
```

В следующем примере проверяется, равны ли среднее и медиана оценки в исходной коллекции:

```
const eqMedianAverage = fork(R.equals, R.median, R.mean);
eqMedianAverage([80, 90, 100])); // -> Истинно
eqMedianAverage([81, 90, 100])); // -> Ложно
```

Некоторые усматривают в коллекции определенную ограниченность, но вы сами можете убедиться в совершенно противоположном: комбинаторы раскрепощают и облегчают бесточечное программирование. А поскольку комбинаторы являются чистыми функциями, то их можно составлять вместе с другими комбинаторами, что дает бесчисленное множество вариантов для выражения или преодоления сложностей в написании любого типа приложения. Их применение будет еще не раз продемонстрировано в последующих главах.

Благодаря основным принципам неизменяемости и чистоты функциональное программирование позволяет добиться ясной степени модульности и повторного использования функций, составляющих программу. Как пояснялось в главе 2, “Сценарий высшего порядка”, функции могут быть использованы BjavaScript и для реализации модулей. Опираясь на те же самые принципы, можно составлять и повторно использовать целые модули. Размышление над такой возможностью мы оставляем на ваше усмотрение.

Модульные функциональные программы состоят из абстрактных функций, которые можно уяснить и повторно использовать по отдельности и назначение которых выводится из правил, определяющих их композицию. Как пояснялось в этой главе, композиция чистых функций образует самую основу функционального программирования. В методиках карринга и частичного применения, представленных в этой главе, выгодно используется абстракция чистых функций с целью сделать их пригодными для композиции. До сих пор речь еще не шла об обработке ошибок, составляющей крайне важную часть любой надежной, отказоустойчивой прикладной программы, и поэтому данной теме посвящена следующая глава.

Резюме

Из этой главы вы узнали следующее.

- Цепочки и конвейеры функций позволяют соединять вместе повторно используемые и модульные программы, разделяемые на компоненты.
- Функциональная библиотека Ramda.js приспособлена для карринга и композиции и оснащена эффективным арсеналом служебных функций.
- Карринг и частичное применение могут быть использованы для сокращения количества аргументов чистых функций, позволяя частично вычислять подмножество аргументов функций, преобразуя их в унарные функции.
- Решаемую задачу можно разбить сначала на простые функции, затем составить их вместе, чтобы прийти к целому решению.
- С помощью комбинаторов функций можно организовать потоки выполнения сложных программ, решая любые практические задачи и программируя в

бесточечном стиле.

Проектные шаблоны и сложность

В этой главе...

- Недостатки императивных схем обработки ошибок
- Предотвращение доступа к недостоверным данным с помощью контейнеров
- Реализация функторов в качестве механизмов преобразования данных
- Представление о монадах как о типах данных, упрощающих композицию
- Объединение стратегий обработки ошибок с монадическими типами данных
- Чередование и композиция монадических типов данных

Пустые ссылки привели к ошибкам стоимостью в миллиарды долларов.

Тони Хоар (Tony Hoare), сайт InfoQ

Некоторые ошибочно считают функциональное программирование парадигмой, призванной решать лишь академические задачи, имеющие в основном числовой характер, где зачастую не принимается во внимание возможность отказов, возникающих в реальных системах. Но за последние годы разработчики обнаружили, что функциональное программирование позволяет организовать обработку ошибок более изящно, чем любой другой стиль разработки программного обеспечения.

В программном обеспечении может возникнуть немало осложнений в связи с тем, что данные неумышленно становятся пустыми (null) или неопределен-

ними (undefined), генерируются исключения или теряется связность узлов сети и возникают прочие неполадки. Поэтому в прикладном коде необходимо принимать во внимание возможность проявления подобных осложнений, что неизбежно усложняет дело. В итоге разработчикам приходится тратить бесчисленные часы рабочего времени, чтобы убедиться, что прикладной код надлежащим образом генерирует и перехватывает исключения, а также проверяет пустые значения null везде, где это только возможно. И в конечном счете все это еще больше усложняет прикладной код, который с трудом поддается масштабированию и пониманию по мере разрастания и повышения сложности приложений.

Труд разработчика должен быть более рациональным и менее тяжелым. Поэтому в текущей главе рассматриваются понятия *функторов* в качестве средств для создания простых типов данных, к которым могут быть приведены функции. В частности, функтор применяется к типам данных, которые называются *монадами* и обладают особым поведением в отношении разных способов обработки ошибок. Монады относятся к числу самых трудных для уяснения понятий в функциональном программировании, поскольку их теоретические основы глубоко укоренены в теорию категорий, которая здесь не рассматривается. В этой главе преследуется цель уделить основное внимание лишь практическим вопросам применения монад. Эта тема будет раскрыта здесь путем постепенного наслаивания ряда необходимых как предварительное условие понятий, после чего будет показано, как пользоваться монадами на практике для составления отказоустойчивых композиций так, как этого просто нельзя добиться с помощью императивных механизмов обработки ошибок.

5.1. Недостатки императивной обработки ошибок

Ошибки в коде JavaScript могут возникать в самых разных ситуациях, особенно в том случае, когда приложению не удается подключиться к серверу или же оно пытается получить доступ к свойству пустого (null) объекта. Кроме того, в сторонних библиотеках могут содержаться функции, генерирующие исключения для сигнализации об особых ошибочных условиях. Следовательно, нужно всегда быть готовым к наихудшему, проектируя систему с учетом возможных отказов, а не реагируя на них запоздало и впоследствии сожалея об этом. В императивном стиле программирования ошибки и исключения, как правило, обрабатываются ио принципу try-catch, т.е. пробы и перехвата.

5.1.1. Обработка ошибок в блоке операторов try-catch

Нынешний механизм обработки исключений и ошибок BjavaScript рассчитан не генерирование и перехват исключений с помощью широко распространенной конструкции try-catch, имеющейся в большинстве современных языков программирования и приведенной ниже.

```
try {
    // здесь следует код, способный сгенерировать исключение } catch (e) {
    // здесь следуют операторы для обработки любых исключений
    console.log('ERROR' + e.message);
}
```

Назначение этой языковой конструкции — охватить фрагмент кода, который считается ненадежным. Как только будет сгенерировано исключение, интерпретатор JavaScript резко остановит выполнение программы и сформирует трассировку стека со всеми вызовами функций вплоть до проблематичной инструкции. Как известно, конкретные подробности ошибки, в том числе сообщение, номер строки кода, имя файла, сохраняются в объекте типа Error и передаются в блок оператора catch, который становится надежным местом для потенциального возобновления нормальной работы программы. В качестве примера напомним рассматривавшиеся ранее функции findObject () и findStudent():

```
// findObject :: DB, String -> Object
const findObject = R.curry(function (db, id) { const result = find(db, id)
    if(!result) {
      throw new Error('Object with ID [' + id + '] not found');
    }
    return result;
});

// findStudent :: String -> Student
const findStudent = findObject(DB('students'));
```

Любая из этих функций способна сгенерировать исключение, и поэтому на практике их вызовы приходится заключать в блок операторов try-catch следующим образом:

```
try {
    var student = findStudent('444-44-4444');
}
catch (e) {
    console.log('ERROR' + e.message);
}
```

Но в функциях необходимо абстрагироваться, как и прежде от циклов и условных операторов, так и теперь от обработки ошибок. Очевидно, что функции, в которых применяются блоки операторов try-catch, как показано выше, нельзя составлять или связывать в цепочку, что оказывает заметное влияние на порядок разработки прикладного кода.

5.1.2. Причины не генерировать исключения в функциональных программах

Структурированный механизм генерирования и перехвата исключений в императивном коде JavaScript обладает многими недостатками и несовместим с функциональной методикой проектирования. Функциям, генерирующих исключения, присущи следующие недостатки.

- Их нельзя составлять или связывать в цепочки подобно другим средствам функционального программирования.
- Они нарушают принцип ссылочной прозрачности, выступающий за единственное, предсказуемое значение, поскольку генерирование исключений составляет еще один путь выхода из вызовов функций.
- Они вызывают побочные эффекты, поскольку непредвиденное сворачивание стека

оказывает влияние на всю систему, помимо вызова самой функции.

- Они нарушают принцип нелокальное™, поскольку код, применяемый для восстановления работоспособности после ошибки, отделен от кода, вызывающего функцию, как показано ниже. Когда возникает ошибка, функция покидает локальный стек и свое окружение.

```
try {
    var student = findStudent('444-44-4444');

    // здесь следуют дополнительные строки промежуточного кода
}
catch (e) {
    console.log('ERROR: not found');

    // обработать здесь ошибку
}
```

- Они накладывают на вызывающий код немалое бремя ответственности объявлять соответствующие блоки операторов catch, чтобы обрабатывать отдельные исключения вместо того, чтобы просто обеспечивать возврат из функции единственного значения.
- Их трудно применять, когда многие ошибочные условия приводят к появлению вложенных на разных уровнях блоков обработки исключений, как показано ниже.

```
var student = null;
try {
    student = findStudent('444-44-44444');
}
catch (e) {
    console.log('ERROR: Cannot locate students by SSN');
    try {
          student = findStudentByAddress(new Address(...));
    catch (e) {
       console.log('ERROR: Student is no where to be found!');
    }
}
```

В связи с изложенным выше может возникнуть следующий вопрос: "Не следует ли считать генерирование исключений неактуальным в функциональным программировании?" Вряд ли. На практике оно вообще не может быть неактуальным, поскольку имеется немало факторов, которые не поддаются контролю, но должны быть учтены при разработке. Кроме того, в прикладном коде может применяться сторонняя и не поддающаяся контролю библиотека, где реализуются исключения.

Применение исключений может оказаться довольно эффективным в некоторых крайних случаях. В функции checkType (), упоминавшейся в главе 4, исключение применялось для сигнализации о совершенно неверном применении интерфейса API. Исключения приносят пользу и для сигнализации о таких невосстановимых условиях, как, например, RangeError: Maximum call stack size exceeded (Ошибка при проверке границ: превышен максимальный размер стека вызовов), как подробнее поясняется в главе 7, "Оптимизация функционального кода". Для генерирования исключений всегда найдется место, но этим не следует

злоупотреблять. Как правило, BjavaScript возникает пресловутая ошибка соблюдения типов (Type-Error), обусловленная вызовом функции для пустого (null) объекта.

5.1.3. Затруднения, возникающие при проверке пустого значения

Чтобы неожиданно не завершать вызов функции можно просто возвратить из нее пустое значение null. По крайней мере, это гарантирует лишь один путь выхода из функции, хотя он и не самый лучший. Функции, возвращающие пустое значение null, накладывают на пользователей другое бремя ответственности: проверять наличие досадного пустого значения. В качестве примера рассмотрим следующую функцию getCountry (), отвечающую за чтение адреса и страны проживания учащегося:

```
function getCountry(student) {
    let school = student.getSchool();
    if(school !== null) {
        let addr = school.getAddress();
        if(addr !== null) {
            var country = addr.getCountry();
            return country;
        }
        return null;
    }
    throw new Error('Error extracting country info');
```

На первый взгляд, реализовать эту функцию было нетрудно. Ведь она всего лишь извлекает свойство объекта. Можно было бы, конечно, создать простую линзу, чтобы сосредоточить основное внимание на данном свойстве, поскольку линза обладает достаточно развитой логикой, чтобы возвратить неопределенное значение (undefined), если адрес окажется пустым (null). Но это никак не поможет вывести сообщение об ошибке.

Но в конечном счете пришлось написать немало строк кода, чтобы застраховаться от неожиданного поведения. Предусмотрительное заключение кода в блоки операторов try-catch или же его охват проверками на пустое значение null на самом деле служит признаком малодушия. Не лучше ли организовать эффективную обработку ошибок, избавившись от излишнего стереотипного кода?

5.2. Выработка лучшего решения с помощью функторов

Функциональный способ обработки ошибок коренным образом отличается от других подходов, предлагающих меры борьбы с напастями, обнаруживаемыми в программном обеспечении. Хотя в его основу положен аналогичный принцип: создать безопасный контейнер, если угодно, вокруг потенциально опасного кода, как показано на рис. 5.1.

```
try {
    var student = findStudent('444-44-4444');
    ... остальной код
)
```

```
catch (e) {
     console.log('ERROR: Student not found!');

      // Обработка отсутствующего учащегося
I
```

Рис. 5.1. Конструкция try-catch образует невидимый безопасный контейнер вокруг функций, способных генерировать исключения. Этот воображаемый безопасный контейнер воплощается в конкретный контейнер

Такое обозначение контейнера, в который заключается опасный код, применяется и в функциональном программировании, но из него исключается блок операторов try-catch. А существенное отличие заключается в применении функциональных типов данных, ограждающих неизменяемость объектов первого класса в функциональном программировании. Рассмотрим сначала самый элементарный тип данных, а затем перейдем к более развитым типам.

5.2.1. Заключение ненадежных значений в оболочку

Заключение значений в оболочку или контейнер является основополагающим проектным шаблоном в функциональном программировании, поскольку оно защищает от прямого доступа к значениям, позволяя безопасно и неизменяемо манипулировать ими в прикладных программах. Это все равно что надеть доспехи перед сражением. Доступ к заключенному в контейнер значению возможен только путем *применения операции преобразования к его контейнеру.* Понятие *преобразования (map) еще* не раз будет обсуждаться в этой главе, но вам оно должно быть уже знакомо из главы 3, где демонстрировалось применение операции map к массивам, которые в каком-то смысле служили контейнерами для значений.

Оказывается, что операции преобразования можно применять не только к массивам. В функциональном языке JavaScript *преобразование* — это не более, чем *функция.* Это понятие происходит от ссылочной прозрачности, где функция должна всегда "приводить" к одному и тому же результату при тех же самых входных данных. Таким образом, операцию преобразования можно рассматривать как средство, позволяющее подключить лямбда-выражение с конкретным поведением для преобразования инкапсулированного значения. Так, к массивам операция map применяется для создания новых массивов с преобразованными значениями.

Продемонстрируем данное понятие на простом примере данных типа Wrapper, определяемого в листинге 5.1. И хотя это простой тип данных, в его основу положен весьма эффективный принцип, прокладывающий путь к материалу последующих разделов, и поэтому его очень важно понять.

Листинг 5.1. Функциональный тип данных для заключения значений в оболочку

```
class Wrapper {                    Простой тип данных для
    constructor (value) { «<- -  хранения единственного
        this._value = value; значения любого типа
    }
```

```
    // тар :: (A -> B) -> A -> B
    тар / f \ |                    __ _ | Применить функцию преобразования
       р                                 к данному типу аналогично массивам
         return f(this._value);
    };

    toStringO {
         return 'Wrapper (' + this._value + ')';
    } }

// wrap :: A -> Wrapper (A)                    | Вспомогательная функция, быстро
const wrap = (val) => new Wrapper (val); <<—J создающая оболочку вокруг значения
```

Такой объект-оболочку можно использовать для инкапсуляции потенциально ошибочного значения. А поскольку непосредственный доступ к нему нежелателен, то единственный способ извлечь его — воспользоваться функцией identity (), упоминавшейся в главе 4, “На пути к повторно используемому, модульному коду”, (следует, однако, иметь в виду, что явного метода get () у данного типа оболочки нет). Безусловно, BjavaScript предоставляются средства, упрощающие доступ к этому значению, но здесь важно понять, что как только значение будет заключено в оболочку, его нельзя извлечь или преобразовать непосредственно, поскольку оболочка служит для него виртуальным защитным барьером, как показано на рис. 5.2.

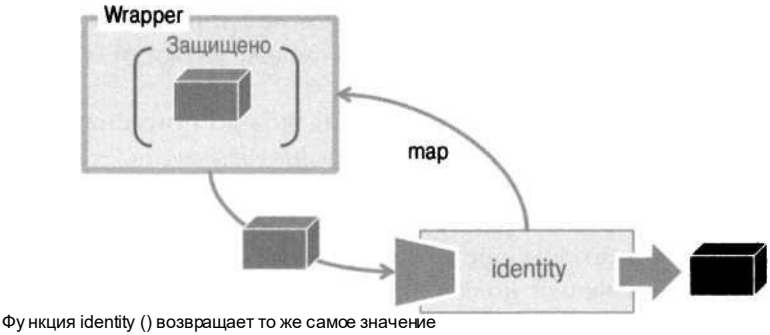


Рис. 5.2. Операция тар служит для безопасного доступа к объекту типа Wrapper и манипулирования заключенными в него значениями. В данном случае операция преобразования (тар) применяет функцию тождественности (identity) к оболочке, чтобы извлечь из нее значение как из контейнера

Ниже приведен конкретный пример использования достоверного значения.

```
const wrappedValue = wrap('Get Functional');
wrappedValue.map(R.identity); // -> 'Get Functional' < ..... Извлечь значение
```

К данной оболочке можно применить любую функцию преобразования, чтобы вывести значение на консоль или обработать нужным образом: `wrappedValue.map (console, log)` ; | Применить функцию преобразования к заключенному в оболочку значению

```
wrappedValue.map(R.toUpper); // -> 'GET FUNCTIONAL'
```

Преимущество столь простого подхода заключается в том, что любой код, написанный для подобных оболочек, должен получать доступ к их содержимому через метод Wrapper .тар (), чтобы воспользоваться защищенными в них значениями. Но если заключенное в оболочку значение окажется пустым (null) или неопределенным (undefined), как показано

ниже, то ответственность за его надлежащую обработку возлагается на вызывающий код, который может и не обработать его корректно. Далее будет представлена лучшая альтернатива.

```
const wrappedNull = wrap(null); |На функцию возлагается ответственность wrappedNul 1. map
(doWork) ;                      ——J за проверку наличия пустого значения
```

Как следует из данного примера, чтобы манипулировать защищенным значением, находящимся в контексте оболочке, необходимо применить к нему функцию, но вызвать ее непосредственно нельзя. В таком случае обработку ошибок можно поручить конкретным типам оболочек. Иными словами, проверку на пустое значение null, пустую символьную строку, отрицательное значение и т.д. можно произвести до вызова функции. Таким образом, семантика метода Wrapper .tap () определяется конкретной реализацией заключающего в оболочку типа.

Но не будем забегать вперед, поскольку нам нужно еще создать некоторый задел на будущее. А в качестве примера рассмотрим следующий, немного отличающийся вариант метода tap (), называемый f tap ():

```
// fmap :: (A -> B) -> Wrapper[A] -> Wrapper[B]
ftap (f) {                                          Заключить преобразованное значение
    return new Wrapper(f(this._value));        ---  в оболочку, прежде чем возвратить его
}                                                   вызывающему коду
```

Функции fmap () известно, как применять другие функции к значениям, заключенным в контекст оболочки. Сначала она открывает оболочку, а затем применяет заданную функцию к заключенному в нее значению и, наконец, заключает значение обратно, но уже в новую оболочку того же самого типа. Такого рода функция называется *функтором,*

5.2.2. Пояснение назначения функторов

По существу, функтор является не более, чем структурой данных, позволяющей применять функции преобразования с целью извлечь значения из оболочки, модифицировать их, а затем поместить их обратно в оболочку. Это проектный шаблон, в котором определяется семантика для нормальной работы функции fmap (). Ниже приведено общее определение этой функции.

```
fmap :: (A -> B) -> Wrapper(A) -> Wrapper (B) ----- 1 Объект «па Wrapper может служить
                                                      ! оболочкой для любого другого типа
```

Функции fmap () передается заданная функция преобразования (из A -> B) и функтор (контекст оболочки) Wrapper (A), а она возвращает новый функтор Wrapper (B), содержащий результат применения заданной функции преобразования к значению, снова заключенному в оболочку. На рис. 5.3 приведен краткий пример, в котором функция increment () применяется для преобразования из A -> B, за исключением того, что в данном случае типы A и B одинаковы.

Следует, однако, иметь в виду, что функция fmap (), по существу, возвращает новую копию оболочки при каждом ее вызове, действуя во многом подобно линзам, упоминавшимся в главе 2, “Сценарий высшего порядка”, и поэтому ее можно считать неизменяемой. Как показано на рис. 5.3, в результате применения функции преобразования increment () к объекту-оболочке Wrapper (1) возвращается совершенно новый объект-

оболочка Wrapper (2). Рассмотрим простой пример, прежде чем приступить к решению более практических задач с помощью функторов. В частности, реализуем простое сложение 2 + 3 = 5, используя функторы. Чтобы создать функцию plus3 (), достаточно выполнить карринг функции add () следующим образом:

```
const plus = R.curry((a, b) => a + b) ;
const plus3 = plus(3);
```

Рис. 5.3. Значение 1 содержится в объекте-оболочке типа Wrapper. Функтор вызывается с оболочкой и функцией increment (), преобразующей значение внутренним образом и снова заключающей его в оболочку

А теперь число 2 можно сохранить в функторе типа Wrapper таким образом: `const two = wrap(2);`

В результате вызова функции fmap (), применяющей функцию преобразования plus3 () к содержимому оболочки, выполняется операция сложения, как показано ниже.

```
const five = two.fmap(plus3); // -> Wrapper (5)
five.map(R.identity); // -> 5
```

Возвратил значение в контексте новой оболочки

В результате выполнения функции f map () получается другой контекст оболочки того же самого типа, к которой можно применить функцию преобразования R. identity (), чтобы извлечь из нее значение. Следует, однако, иметь в виду, что значение вообще не покидает свою оболочку, и поэтому к нему можно применять сколько угодно функций преобразования на каждой стадии процесса обработки, как показано ниже.

```
two.fmap(plus3).fmap(plus10); // -> Wrapper(15)
```

Чтобы это стало понятнее, на рис. 5.4 показано, каким образом функция fmap () взаимодействует с функцией преобразования plus3 ().

Функция fmap () возвращает тот же самый тип оболочки, в которую заключается полученный результат, для того, чтобы можно было продолжить связывание операций в цепочку. В листинге 5.2 демонстрируется еще один пример, в котором функция преобразования plus3 () применяется к заключенному в оболочку значению и протоколирует полученный результат.

Листинг 5.2. Связывание функторов в цепочку с целью придать дополнительное поведение заданному контексту

```
const two = wrap(2);
two.fmap(plus3).fmap(R.tap(infoLogger)); // ->Wrapper(5)
```

Рис. 5.4. Значение 2 прибавляется к значению в объекте-оболочке типа Wrapper. А функтор служит для манипулирования этим значением, извлекая его из контекста оболочки, применяя к нему заданную функцию преобразования и снова заключая его в контекст, но уже новой оболочки

В результате выполнения исходного кода из листинга 5.2 на консоль выводится следующее протокольное сообщение:

```
InfoLogger [INFO] 5
```

Не кажется ли вам такой шаблон связывания функций в цепочку очень знакомым? И это неслучайно, ведь вы постоянно пользовались функторами в приведенных ранее примерах, даже не подозревая об этом. Именно таким образом и выполнялись операции шар и filter над массивами (можете еще разубедиться в этом, вернувшись к разделам 3.3.2 и 3.3.4). Ниже приведено их общее определение с учетом функторов.

```
тар :: (A -> B) -> Array(A) -> Array(B)
filter :: (A -> Boolean) -> Array(A) -> Array(A)
```

По существу, операции тар и filter являются функторами, сохраняющими тип данных. Именно таким образом и активизируется шаблон связывания в цепочку. Атеперь рассмотрим еще один функтор compose, применявшийся в приведенных ранее примерах. Как пояснялось в главе 4, “На пути к повторно используемому, модульному коду”, он выполняет преобразование одних функций в другие, сохраняя также тип данных. Ниже приведено его общее определение.

```
compose :: (B -> C) -> (A -> B) -> (A -> C)
```

Как и любые другие элементы функционального программирования, функторы обладают следующими важными свойствами.

- **Они должны быть свободны от побочных эффектов.** Чтобы получить то же самое значение из контекста, достаточно применить к нему функцию преобразования R. identity (). Это лишний раз доказывает, что функторы не вызывают побочные эффекты, сохраняя структуру значения, заключенного в оболочку.

  ```
  wrap('Get Functional').fmap(R.identity);
      // -> Wrapper('Get Functional')
  ```

- **Они должны допускать композицию.** Это свойство означает, что композиция функции, применяемой к функции fmap (), должна выполняться точно так же, как и связывание этих же функций в цепочку. Таким образом, следующее выражение совершенно равнозначно программе из листинга 5.2:

  ```
  two.fmap(R.compose(plus3, R.tap(infoLogger))).map(R.identity);
      11 -> 5
  ```

Нет ничего удивительного в том, что функторы обладают подобными свойствами. Ведь именно поэтому им запрещается генерировать исключения, модифицировать элементы или изменять поведение функций. Их практическое назначение состоит в том, чтобы создать контекст или абстракцию для безопасного манипулирования значениями и выполнения над ними операций, никоим образом не изменяя исходные значения. Это вполне очевидно из того, как операция tap преобразует один массив в другой, не изменяя исходный массив. Данный принцип в равной степени применяется к любому типу оболочки.

Но сами функторы не особенно притягательны, поскольку они не обязаны знать, как поступать в тех случаях, когда встречаются пустые (null) данные. Например, нормальная работа функции R.compose () нарушится, если передать ей пустую (null) ссылку на другую функцию. И это не оплошность в проектировании, а сделано намеренно, поскольку *функторы преобразуют функции одного типа в другой.* Еще более специализированное поведение можно обнаружить в функциональных типах данных, называемых *монадами.* Среди прочего, монады позволяют рационализировать обработку ошибок в прикладном коде, допуская составление плавных композиций функций. Каким же образом они связаны с функциями? Монады служат оболочками или контейнерами, содержимое которых доступно для функторов.

Термин *монада* не должен отбивать у вас желание пользоваться им. Если вам приходилось писать код, пользуясь библиотекой jQuery, то монады должны быть вам уже знакомы. Если же отвлечься от всех сложных правил и теорий, положенных в основу монад, то их назначение состоит в том, чтобы предоставить абстракцию над некоторым ресурсом, будь то простое значение, элемент модели DOM, событие или AJAX-вызов, для безопасной обработки содержащихся в нем данных. В этом отношении следующий код jQuery можно отнести к категории монад DOM:

```
$('#student-info').fadein(3000).text(student.fullname());
```

Этот код ведет себя как монада, поскольку библиотека jQuery берет на себя обязанность безопасно выполнить преобразования с помощью методов fadeln () и text (). Если элемент student-info не существует, то указанные методы, применяемые к пустому o6beKTyjQuery, корректно выйдут из этой ситуации, а не сгенерируют исключения. Монады, предназначенные для обработки ошибок, обладают следующим полезным свойством: они безопасно транслируют ошибки, чтобы сделать приложение отказоустойчивым. Более подробно монады рассматриваются ниже.

5.3. Функциональный способ обработки ошибок с помощью монад

Монады решают все отмеченные ранее затруднения, возникающие при традиционной обработке ошибок, применительно к функциональным программам. Но прежде чем подробно обсуждать этот вопрос, выясним ограничение, накладываемое на применение функторов. Как пояснялось ранее, с помощью функторов можно применять функции к значениям безопасно и неизменяемо. По применяя функторы в прикладном коде, можно легко столкнуться с неудобной ситуацией. В качестве примера рассмотрим извлечение сначала записи об учащемся по его номеру социального страхования, а затем значения его

свойства адреса. Для решения этой задачи можно определить следующие две функции findStudent () и getAddress (), в которых применяются объекты функторов для создания безопасного контекста вокруг возвращаемых из них значений:

```
const findStudent = R. curry (function (db, ssn) {
    return wrap (find (db, ssn));
jj.

getAddress = function (student) {
    return wrap (student. fmap (R.prop ('address ') ) )
I
```

Заключить в оболочку извлекаемый объект для защиты на случай, если объект не будет найден

Применить функцию R. prop () из const библиотеки Ramda к объекту, чтобы извлечь из него сначала адрес, а затем заключить полученный результат в оболочку

Как и прежде, для выполнения данной программы обе приведенные выше функции составляются вместе следующим образом:

```
const studentAddress = R.compose(
    getAddress,
    findStudent(DB('student')) ) ;
```

И хотя в приведенном выше коде удалось вообще избежать обработки ошибок, результат его выполнения отличается от ожидаемого. В частности, вместо заключенного в оболочку объекта адреса возвращается дважды заключенный в оболочку объект адреса:

```
studentAddress('444-44-4444'); // -> Wrapper(Wrapper(address))
```

Чтобы извлечь это значение, придется дважды применить функцию R.identity():

```
studentAddress('444-44-4444').map(R.identity).map(R.identity);
```

Очевидно, что извлекать адрес подобным образом нежелательно. Стоит лишь представить, насколько это будет неудобно, если составить вместе три или четыре функции. Следовательно, требуется лучшее решение подобной задачи. И его позволяют найти монады.

5.3.1. Монады: от потока управления до потока данных

Монады подобны функторам, за исключением того что они способны делегировать специальную логику в особых случаях. Рассмотрим это понятие на кратком примере применения функции half : : Number -> Number к любому заключенному в оболочку значению, как показано ниже и на рис. 5.5.

```
Wrapper(2).fmap(half); // -> Wrapper(1)
Wrapper(3).fmap(half); // -> Wrapper(1.5)
```

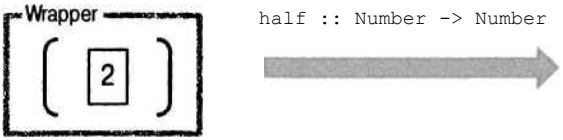


Рис. 5.5. Функторы применяют функцию к заключенному в оболочку значению. В данном случае заключенное в оболочку значение 2 делится пополам, а в итоге возвращается заключенное в оболочку значение 1

А теперь допустим, что требуется ограничить функцию half () только четными числами. Самому функтору известно лишь, как применять данную функцию и заключать полученный результат обратно в оболочку, а дополнительная логика у него отсутствует. Но что, если в качестве входного значения встретится нечетное число? В таком случае можно было бы возвратить пустое значение null или сгенерировать исключение. Но еще лучше сделать данную функцию более развитой логически, чтобы она вела себя корректно в любом случае, возвращая достоверное число, если ей задано правильное входное значение, а иначе — игнорируя его полностью.

Рассмотрим еще одну оболочку типа Empty, построенную по принципу оболочки типа Wrapper:

```
class Empty
map(f) { return
this;
  }

  toStringO { return
    'Empty ()';
  }
```

Пустая операция. В оболочке типа Empty значение не сохраняется. Она лишь представляет понятие "пустоты* или "отсутствия*Аналогично в результате применения функции преобразования к объекту-оболочке типа Empty операция над ним пропускается**

```
const empty = () => new Empty();
```

С учетом упомянутого выше нового требования функцию half () можно реализовать так, как показано ниже и на рис. 5.6.

```
const isEven = (n) => Number.isFinite(n) && (n % 2 == 0);^<
const half = (val) => isEven(val) ? wrap(val / 2)      empty ()

half(4); // -> Wrapper(2)
half(3); // -> Empty
```

Вспомогательная функция, различающая четные и нечетные числа

Функция half () обрабатывает лишь четные числа, возвращая иначе пустую оболочку

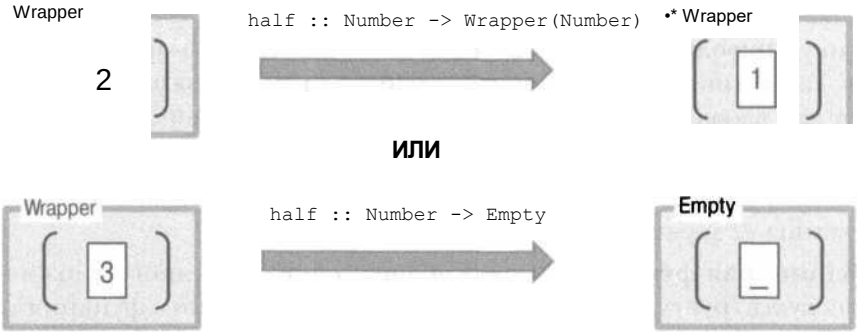


Рис. 5.6. Функция half () может возвратить заключенное в оболочку разделенное пополам значение или же пустую оболочку в зависимости от характера входных данных

Монада возникает при создании целого типа данных по принципу извлечения значений из оболочек и определения правил их вложенности. Подобно функторам, монады являются проектным шаблоном, применяемым для описания вычислений в виде последовательности стадий, где вообще неизвестно обрабатываемое значение. Функторы позволяют защитить значения, но именно монады дают возможность безопасно и без побочных эффектов управлять потоком данных, когда они применяются в композиции. В предыдущем примере при попытке разделить пополам нечетное число вместо пустого значения null возвращается объект-оболочка Empty. Это дает возможность выполнять операции над значениями, не опасаясь, что могут возникнуть ошибки, как показано ниже.

```
half(4).fmap(plus3); // -> Wrapper(5)
half(3).fmap(plus3); // -> Empty ◄
```

Неявно заданной оболочке известно, как применять функции преобразования даже к недостоверным входным данным

Монады могут быть направлены на решение самых разных задач. В этой главе рассматриваются монады, применяемые для объединения механизмов императивной обработки ошибок с целью преодолеть их сложность, а следовательно, позволить вам анализировать прикладной код более эффективно.

Теоретически монады зависят от системы типов в конкретном языке. В действительности многие считают, что их можно понять лишь в том случае, если имеются явные типы данных, как в языке Haskell. Но, как будет показано далее, даже в таком безтиповом языке, как JavaScript, монады нетрудно усвоить и без всех сложностей, присущих статической системе типов.

Для этого необходимо усвоить следующие важные понятия.

- **Монада.** Предоставляет абстрактный интерфейс для монадических операций.
- **Монадический тип.** Конкретная реализация данного интерфейса.

Монадические типы разделяют много общих принципов с объектом-оболочкой типа Wrapper, рассмотренным вначале этой главы. Но каждая монада своеобразна, и в зависимости от своего назначения она может определять разую семантику, приводящую ее в действие, т.е. порядок работы функции шар () или fmap (). Такие типы определяют, что именно означает связывание операций в цепочку или вложение функций данного типа, хотя

все они должны соблюдать следующий интерфейс.

- **Конструктор типа.** Создает монадические типы аналогично конструктору типа Wrapper.
- **Единичная функция.** Вводит значение определенного типа в монадическую структуру аналогично рассмотренным ранее функциям wrap () и empty (). Но если эта функция реализуется в монаде, то она называется of.
- **Функция связывания.** Связывает операции в цепочку (это функция fmap () конкретного функтора, иначе называемая f latMap). Здесь и далее она будет сокращенно называться tap. Между прочим, эта функция связывания не имеет никакого отношения к понятию связывания функций, упоминавшемуся в главе 4, "На пути к повторно используемому, модульному коду".
- **Операция соединения.** Сводит все уровни монадической структуры к одному. Это особенно важно при композиции нескольких функций, возвращающих монады.

Чтобы применить этот новый интерфейс к типу Wrapper, его можно реорганизовать в монаду, как демонстрируется в листинге 5.3.

Листинг 5.3. Монада типа Wrapper

```
class Wrapper {          <--------  Конструктор типа
  constructor(value) {
    this._value = value;
  }

  static of (a) {        <--------  Единичная функция
    return new Wrapper(a);
  }

  map (f) {              MI .....—■ — Функция связывания (функтор)
    return Wrapper.of(f(this._value));
  }

  j oin () {                    <- ------ Функция сведения вложенных уровней
    if(!(this._value instanceof Wrapper)) { return this;
    }
    return this._value.join();
  }

  get() {
    return this, value;
  }
                                  I Функция, возвращающая текстовое
  toSt ring () {          --- 1 представление данной структуры
    return 'Wrapper (${this._value})';
  }
]
```

В монаде типа Wrapper применяется функтор шар для заключения данных в оболочку, чтобы манипулировать ими без побочных эффектов, ограждая их от внешнего мира. И не

удивительно, что для проверки содержимого такой оболочки служит операция .identity:

```
Wrapper.of('Hello Monads!')
      .map(R.toUpper)
      .map(R.identity); //-> Wrapper('HELLO MONADS!')
```

Операция map считается *нейтральным функтором,* поскольку она всего лишь применяет заданную функцию преобразования, заключая полученный результат в оболочку. Как будет показано далее, другие монады вносят в операцию tap свои особенности. А функция joint) служит для сведения вложенных структур подобно очистке лука от шелухи. С ее помощью можно устранить недостатки, обнаруженные ранее в функторах, как показано в листинге 5.4.

Листинг 5.4. Сведение монадической структуры

```
// findObject :: DB -> String -> Wrapper const findObject = R.curryttdb, id) => {
return Wrapper.of(find(db, id));
});

// getAddress :: Student -> Wrapper const getAddress = student => {
     return Wrapper.of(student.map(R.prop('address'))); };

const studentAddress =
      R.compose(getAddress, findObject(DB('student')));

studentAddress('444-44-4444').join().get(); // Адрес
```

Из композиции функций в листинге 5.4 возвращается ряд вложенных оболочек, поэтому функция join () вызывается для выполнения операции сведения вложенной монадической структуры к одному уровню, как в следующем примере:

```
Wrapper.of(Wrapper.of(Wrapper.of('Get Functional'))).join();

// -> Wrapper('Get Functional')
```

Порядок выполнения операции сведения в функции j oin () наглядно показан на рис. 5.7.

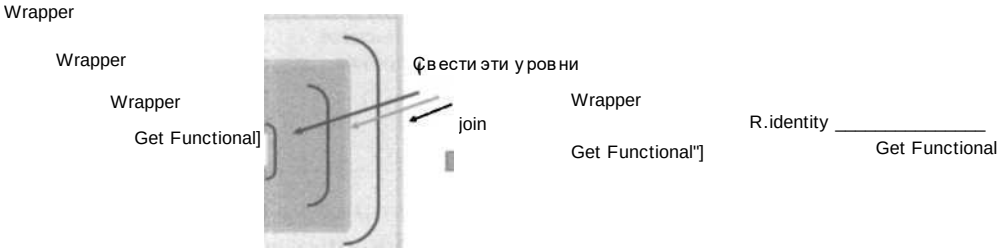


Рис. 5.7. Рекурсивное сведение вложенной монадической структуры с помощью функции join () подобно очистке лука от шелухи

Для выполнения аналогичной операции над массивами, которые также служат преобразуемыми контейнерами, вызывается функция R. flatten ():

```
R.flatten([1, 2, [3, 4], 5, [6, [7, 8, [9, [10, 11], 12]]]]);

// => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Как правило, монады состоят из намного большего количества операций, поддерживающих их особое поведение, а приведенный выше минимальный интерфейс является всего лишь подмножеством целого прикладного программного интерфейса API для монад. Но сами монады абстрактны и лишены всякого реального смысла. Их истинный потенциал начинает раскрываться только после того, как они будут реализованы в виде конкретного типа данных. Правда, большую часть функционального кода можно реализовать с помощью нескольких распространенных типов данных, и благодаря этому исключается немало стереотипного кода, но в то же время удается выполнить немалый объем работы. А теперь перейдем к рассмотрению некоторых полноценных монад: Maybe, Either и 10.

5.3.2. Обработка ошибок с помощью монад Maybe и Either

Помимо заключения достоверных значений в оболочку, монадические структуры могут также использоваться для моделирования ситуаций, когда значение отсутствует, т.е. оно пустое (null) или неопределенное (undefined). В функциональном программировании ошибки *овеществляются* с помощью типов Maybe и Either, позволяющих делать следующее.

- Ограждать неизменяемость.
- Объединять логику проверки наличия пустых (null) значений.
- Избегать генерирования исключений.
- Поддерживать композицию функций.
- Централизовать логику для предоставления значений по умолчанию.

Оба упомянутых выше типа монад обеспечивают эти преимущества по-своему. Начнем их рассмотрение с монады типа Maybe.

Объединение логики проверки наличия пустых значений с помощью монады типа Maybe

Действие монады типа Maybe, по существу, сосредоточено на объединении логики проверки наличия пустых (null) значений. Тип Maybe является пустым (т.е. маркерным) типом со следующими конкретными подтипами.

- Just (*значение)*. Представляет оболочку, в которую заключается определенное значение.
- Nothing (). Представляет оболочку без конкретного значения или сбой, не требующий дополнительной информации. Но и в этом случае можно по-прежнему применять функции к (несуществующему) значению данного подтипа.

Эти подтипы реализуют все рассмотренные ранее монадические свойства, а также некоторое дополнительное поведение, характерное для их назначения. Реализация типа Maybe приведена в листинге 5.5.

Листинг 5.5. Монада типа Maybe с подтипами Just и Nothing

```
class Maybe {                          i Тип оболочки
                                       I (родительский класс)
    static just(a) {
        return new Just(a);
    }

    static nothing() {
        return new Nothing();
    }

    static fromNullable(a) {
        return a !== null ? Maybe.just(a)


    static of(a) {
        return just(a) ; }

    get isNothingO { return false; }

    get isJustO { return false;
```

Создать тип Maybe из типа, допуска* ющего значение null (функции-конструктора). Если значение, заключаемое в монаду, оказывается пустым, получить экземпляр типа Nothing, а иначе — сохранить значение в оболочке подтипа Just, чтобы реализовать наличие значения

```
Maybe.nothing();    -- ----
```

```
class Just extends Maybe { // #C-<
    constructor(value) {
        super();
        this, value = value;


    get value() {
```

Подтип Just для обработки в случае, если значение присутствует

```
    return this, value;                          Применить к оболочке подтипа
}
                                  Just функцию, преобразующую ее значение, сохранив результат
map(f) {                                                    обратно в оболочке
        return Maybe.fromNullable(f(this._value)); // #D -< }

    getOrElseO {                                 Извлечь из структуры зна-
        return this._value; // #E }              чение или единичную опе-
                                                 рацию, предоставляемую
    filter(f) {                                  монаде по умолчанию
        Maybe.fromNullable(f(this._value) ? this._value : null); }

    chain(f) { return f(this._value); }
                                  I Возвратить текстовое представление toString () {  ◄
                                  J данной структуры
        return 'Maybe.Just(${this._value})'; }

                                                | Подтип Nothing для
class Nothing extends Maybe {                   защиты от отсутствия
                                                1 значения

    map(f) {                                                  При попытке извлечь значение из
        return this; }                                        оболочки типа Nothing сгенерировать
                                                              исключение, обозначающее
                                                              неверное применение монады (подробнее
    get value() {                                             об этом — далее), в против
        throw new TypeError (''Can  extract the               ном случае возвратить значение
                             value of  a Nothing.  : // ----------

                                  Проигнорировать
    getOrElse (other) {           данное значение и
        return other; ◄------возвратить другое
    }                                           Если значение имеется и совпадает с заданным
                                                предикатом, возвратить оболочку типа Just, filter ()
    {                                           описывающую данное значение, в противном случае
        return this, value; // #J ◄------ 1 возвратить оболочку типа Nothing
    }

    chain(f) {
        return f(this._value);
    }

                                  Возвратить текстовое
                                  представление данной
    toString() {                  структуры
        return 'Maybe.Nothing'; }
```

Монада типа Maybe явным образом абстрагирует обработку “обнуляемых” значений (null и undefined), освобождая разработчика для выполнения более важных обязанностей. Как видите, монада типа Maybe, по существу, служит абстрактным обобщенным объектом для конкретных монадических структур типа Just и Nothing, каждая из которых содержит свою реализацию монадических свойств. И, как упоминалось ранее, реализация поведения монадических операций в конечном итоге зависит от семантики, наделяемой конкретным типом. Например, операция tap ведет себя по-разному в зависимости от конкретного типа: Nothing или Just. Наглядное представление о возможности хранить объект, представляющий учащегося, в структуре монады типа Maybe дается ниже и на рис. 5.8.

```
// findStudent :: String -> Maybe(Student) function findStudent(ssn)
```

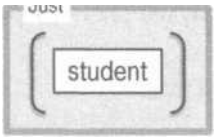


Рис. 5.8. Структура монады типа Maybe состоит из двух подтипов: Nothing и Just. В результате вызова функции findStudent () возвращается значение, заключенное в оболочку типа Just, или же состояние отсутствующего значения в оболочке типа Nothing

Данная монада нередко применяется при вызовах, содержащих неопределенность: запросах базы данных, поиске значений в коллекции, запросах данных с сервера и т.д. Продолжим рассмотрение начатого в листинге 5.4 примера извлечения значения свойства address объекта, представляющего учащегося и извлекаемого из локального хранилища. Запись об искомом учащемся может и не существовать, и поэтому извлекаемый результат заключается в оболочку типа Maybe, а соответствующие операции снабжаются префиксом safe, как показано ниже.

```
const safeFindObject = R.curry((db, id) => {
    return Maybe.fromNullable(find(db, id)); ) };

// safeFindStudent :: String -> Maybe(Student)
const safeFindStudent = safeFindObject(DB('student')); const address =

      safeFindStudent('444-44-4444').map(R.prop('address'));
address; // -> Just(Address (...)) или Nothing
```

Еще одно преимущество заключения получаемых результатов в оболочку с помощью монад состоит в том, что сигнатуры функции способствуют их самодокументированию и устраняют неопределенность возвращаемого значения. Вызов Maybe.fromNullable () удобен тем, что проверка на пустое значение (null) производится от имени того, кто делает этот вызов. В результате вызова функции safeFindStudent() возвращается результат типа Just (Address (...)), если встретится достоверное значение, а иначе — результат типа Nothing. Функция преобразования R. prop () применяется к монаде именно так, как и ожидалось.

Кроме того, она выполняет полезную работу по обнаружению программных ошибок или злоупотреблений обращениями к интерфейсу API. В частности, с ее помощью можно ввести в действие предусловия, обозначающие допустимость недостоверных параметров. Так, если при вызове Maybe. fromNullable () передается недостоверное значение, то в конечном счете получается результат типа Nothing, и поэтому при вызове метода get () с целью открыть контейнер будет сгенерировано следующее исключение:

```
TypeError: Can't extract the value of a Nothing.
(Ошибка соблюдения типов: Извлечь значение типа Nothing нельзя)
```

В монадах предполагается, что к ним должны применяться функции преобразования вместо непосредственного извлечения их содержимого. Еще одну полезную операцию над монадой типа Maybe выполняет метод getOrElse () в качестве альтернативы возврату устанавливаемых по умолчанию значений. Рассмотрим следующий пример установки значения в поле формы или стандартного значения по умолчанию, если входные данные не заданы:

```
const userName = findStudent('444-44-4444').map(R.prop('firstname'));

document.querySelector('#student-firstname').value =
    username.getOrElse('Enter first name');
```

Если операция извлечения завершится успешно, то отображается имя пользователя учащегося. В противном случае выполнение кода происходит по альтернативной ветви else, где выводится задаваемая по умолчанию символьная строка.

Монада типа Maybe под разной личиной

Монаду типа Maybe можно обнаружить в разных формах типов Optional или Option в таких языках, как Java 8 и Scala. Вместо типов Just и Nothing в этих языках объявляются типы Some и None, хотя семантически они служат той же самой цели.

А теперь вернемся к рассмотренному ранее антишаблону пессимистического исхода проверки на пустое значение (null), который нередко демонстрирует свою неприглядную сущность в объектно-ориентированном программном обеспечении. В качестве примера ниже приведено определение функции getCountry().

```
function getCountry(student) { let school = student.school(); if(school !== null)
    {
        let addr = school.address();
        if(addr !== null) {
            return addr.country();
        }
    }
    return 'Country does not exist!';
}
```

Если данная функция возвращает символьную строку с сообщением ' Country does not exist! ' (Страна отсутствует!), то непонятно, какой именно оператор привел к неудачному исходу? В таком коде трудно выявить в какой именно строке произошла ошибка. При написании подобного кода программисты не обращают особого внимания на его стиль и правильность. Они просто воссоздают защитный барьер вокруг вызовов функций. В отсутствие монадических характеристик проверки на пустое значение (null) приходится распределять по всему прикладному коду, чтобы предотвратить исключения типа TypeError. А в структуре монады типа Maybe такое поведение инкапсулируется для повторного использования. Рассмотрим следующий пример:

```
const country = R.compose(getCountry, safeFindStudent);
```

Функция safeFindStudent () возвращает заключенный в оболочку объект, представляющий учащегося. Это дает возможность избавиться от привычки к защитному программированию и без опаски распространять недостоверное значение. В качестве примера ниже приведен новый вариант метода getCountry().

```
const getCountry = (student) => student
    .map(R.prop('school'))
    .map(R.prop('address'))
    .map(R.prop('country'))
    .getOrElse('Country does not exist!');-<■
```

Если на любой из стадий вычислений получается результат типа Nothing, то все последующие операции будут пропущены

Если при обращении к любому из этих свойств возвращается пустое значение null, то данная ошибка распространяется по всем остальным уровням как результат типа Nothing, а все последующие операции будут корректно пропущены. Такой код оказывается не только декларативным и изящным, но и отказоустойчивым.

Поднятие функции

Внимательно проанализируйте следующую функцию:

```
const safeFindObject = R.curry((db, id) => { return
    Maybe.fromNullable(find(db, id));
) };
```

Обратите внимание на то, что в ее имени содержится префикс safe и в ней применяется монада с целью заключить в оболочку возвращаемое из нее значение.

Такая норма практики хороша тем, что она проясняет для вызывающего кода, что в функции имеется потенциально опасное значение. Означает ли это, что каждую функцию в программе следует непременно снабжать монадами? Нет, не означает. Методика под названием *поднятие функции (function lifting)* дает возможность

преобразовать любую обыкновенную функцию в такую форму, в которой она действует как контейнер и становится "безопасной", как показано ниже. И это удобно, поскольку избавляет от необходимости изменять уже существующие реализации.

```
const lift = R.curry((f, value) => {
     return Maybe.fromNullable(value).map(f);
 }) ;
```

Вместо того чтобы применять монаду непосредственно в теле функции, последнюю можно сохранить в прежнем виде:

```
const findObject = R.curry((db, id) => { return find(db, id);
});
```

и воспользоваться функцией чтобы заключить данную функцию в кон тейнер:

```
const safeFindObject = R.compose(lift(console.log), findObject);
```

Поднятие пригодно для любой функции по какой угодно монаде!

Очевидно, что монада типа Maybe выгодно отличается тем, что проверки на недостоверность данных сосредоточены в одном месте, но в то же время она возвращает объект Nothing, если что-нибудь пойдет не так. Следовательно, требуется более дальновидное решение, позволяющее выяснить причину сбоя. И для этой цели лучше воспользоваться монадой типа Either.

Устранение сбоев с помощью монады типа Either

Эта монада несколько отличается от монады типа Maybe. Она являет собой структуру, представляющую логическое разделение двух значений а и Ь, которые вообще не должны возникать одновременно. В данном типе монады моделируются два следующих варианта.

- **Left (a).** Содержит возможное сообщение об ошибке или объект генерируемого исключения.
- Right (Ь). Содержит удачное значение.

Как правило, монада типа Either реализуется со смещением по правому операнду, а это означает, что применение функции преобразования к контейнеру всегда выполняется по подтипу Right (b). Это аналогично ветви Just в монаде типа Maybe.

Зачастую монада типа Either служит для хранения результатов вычисления, в ходе которых не удается предоставить дополнительные сведения о характере сбоя. В тех случаях, когда устранить сбой не удается, левый вариант может

содержать надлежащий объект генерируемого исключения. Реализация монады типа Either демонстрируется в листинге 5.6.

Листинг 5.6. Реализация монады типа Either с подклассами Left и Right

```
class Either {
constructor(value) {
this._value = value;
    }

    get value() { return this._value;
    }

    static left(a) {
        return new Left(a); }

    static right(a) {
        return new Right(a);
    }

      static fromNullable(val) { // #B
       return val !== null && val ? !== undefined :
                  Either.right(val) Either.left(val);
```

***---------- Функция-конструктор одного из двух типов. В этом объекте может содержаться либо информация об исключении, либо корректное значение (правое смещение)

Принять левый вариант типа Left, если передано недостоверное значением, а иначе — правый вариант типа Right

| Создать новый экземпляр, содержащий `static of (a) { // #C`

•<——J значение в структуре типа Right

```
    return Either.right(a); }
```

Преобразовать значение в структуре типа Right, применяя к ней функцию преобразования. Но `class Left extends Either {` ничего не делать со структурой типа Left

```
  map(_) {                ---
    return this; // пустая операция
  }
```

Извлечь значение из структуры типа Right, если оно существует, в противном случае сгенерировать исключение типа TypeError

```
  throw new TypeError(
      ''Can't extract the value of a Left (a).");
}
```

Извлечь значение из структуры типа Right. Если же оно не существует, то возвратить значение, `getOrElse (other) {` заданное по умолчанию

```
    return other; —
```

Применить заданную функцию к значению типа Left; ничего не делать со структурой типа Right

```
 orElse(f) {
    return f(this. value); // #G ◄F
```

. | Применить функцию к структуре типа Right и возвратить из нее значение; ничего не делать со структурой типа Left. Это первый случай возникновения цепочки, как поясняется далее

`cha i n (f) {`

```
        return this; }

    getOrElseThrow(a) {    <-------  Сгенерировать исключение со значением только из структуры типа Left, в противном случае проигнорировать
        throw new Error (a);         исключение и возвратить достоверное значение
    }

    filter(f) {  <-  Если значение присутствует и удовлетворяет заданному предика- — ту, возвратить структуру типа Right, описывающую данное зна- чение, а иначе возвратить пустую структуру типа Left
        return this;


    toString() {
        return 'Either.Left(${this._value})';
    }
}

                                                  Преобразовать значение в структуре типа Right, class Right
extends Either {                                  применив к ней соответствующую функцию;
    map(f) { // #D          <_ | ничего не делать со структурой типа Left
        return Either.of(f(this, value));


    getOrElse (other) {                  Извлечь значение из структуры типа Right,
        return this, value; // #F <- Если оно отсутствует, возвратить значение, } ~  заданное по умолчанию

                                  | Применить заданную функцию к значению типа 1 Lef t; ничего не делать со значением типа Right
    orElse()
        return this;
    }                                 Применить функцию к структуре типа Right и возвратить из нее значение; ничего не делать со структурой типа Left. Это первый случай возникновения цепочки, как поясняется далее
    chain(f) { // #H -<
                        return f(this. value);

                                      Сгенерировать исключение со значением только в структуре типа Left, в противном случае проигнорировать исключение и возвратить достоверное значение
    getOrElseThrow(_) {
        return this.value;
                                      Если значение присутствует и удовлетворяет заданному предикату, возвратить
filter(f) { // #J                     структуру типа Right, описывающую данное значение, в противном случае [1]
                                                  возвратить пустую структуру типа Lef t
        return                  Either.fromNullable(f(this._value) ? this, value :
               null);


    toString() {
      return 'Either.Right(${this._value})'; }
```

Обратите внимание на то, что некоторые операции в обеих монадах типа Maybe и Either оказываются пустыми. Они сделаны пустыми намеренно и слу

жат в качестве заполнителей, позволяющих функциям благополучно пропускать выполнение, если в конкретной монаде это окажется уместным.

А теперь применим монаду типа Either на практике. С ее помощью можно создать еще один вариант функции safeFindobject (), как показано ниже.

```
const safeFindObject = R.curry((db, id) => {
    const obj = find(db, id);
    if(obj) {
       return Either.of(obj); «<-
    }
    return Either.left('Object not found with ID: ${id}'); ◄ ---------------- 1
});
```

Чтобы абстрагировать весь условный оператор if-el ав,можно было бы также сделать вызов Either. fromNullable (). Но здесь сделано иначе для большей наглядности примера

Структура типа Left может также содержать значения I

Если операция доступа к данным завершится удачно, объект, представляющий учащегося, сохраняется справа, в противном случае выдается сообщение об ошибке слева (рис. 5.9).

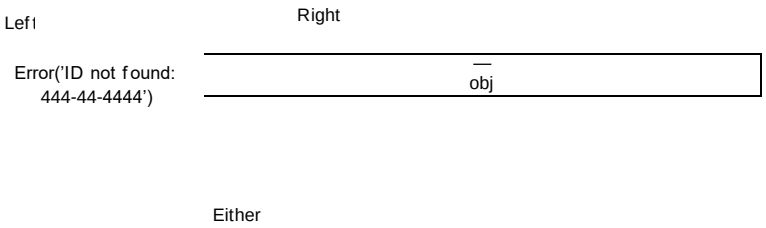


Рис. 5.9. В структуре типа Either можно хранить объект *(справа)* или ошибку типа Error с соответствующей информацией из трассировки стека *(слева)*. Это удобно, если требуется предоставить единственное возвращаемое значение, которое может также содержать сообщение об ошибке на случай сбоя

В связи с изложенным выше может возникнуть резонный вопрос: “Почему бы не воспользоваться двухэлементным кортежем (или типом Pai г; см. главу 4, “На пути к повторно используемому, модульному коду”), чтобы зафиксировать и объект, и сообщение?” Для этого имеется особое основание. В частности, кортежи представляют так называемый *тип произведения (product type),* подразумевающий логическое отношение И среди его операндов. А для обработки ошибок удобнее пользоваться взаимно исключающими типами, чтобы смоделировать случай, когда значение существует ИЛИ не существует. В случае обработки ошибок, оба типа не могут одновременно существовать.

Результат можно извлечь с помощью монады типа Either, вызвав метод getOrElse () и предоставив ему на всякий случай подходящее значение по умолчанию:

```
const findStudent = safeFindObject(DB('student')); findStudent('444-44-
4444').getOrElse(new Student());
    11 ->Right(Student)
```

В отличие от структуры типа Maybe. Nothing, структура типа Either .Left может содержать значения, к которым можно применять функции. Так, если

функция findStudent () не возвращает объект, к операнду типа Left можно применить метод orElse, чтобы вывести сообщение об ошибке:

```
const errorLogger = partial(logger, 'console[1]', 'basic',
                            'MyErrorLogger', 'ERROR');
findStudent('444-44-4444').orElse(errorLogger);
```

В итоге на консоль выводится следующее сообщение:

```
MyErrorLogger [ERROR] Student not found with ID: 444-44-4444
```

Структуру типа Either можно также использовать для защиты прикладного кода от непредсказуемых функций, способных генерировать исключения. Благодаря этому функции становятся более безопасными в отношении типов и свободными от побочных эффектов, предупреждая исключения на ранней стадии, а не распространяя их. В качестве примера рассмотрим доступную в JavaScript функцию decodeURIComponent (), способную выводить сообщение об ошибке, если заданный URL окажется недостоверным:

```
function decode(url) {

        const result = decodeURIComponent(url); return
        Either.of(result);
    }
    catch (uriError) {
        return Either.Left(uriError);
```

Может генерировать исключение типа URIError

Как показано в приведенном выше коде, в структуру типа Either. Left обычно помещается объект ошибки, содержащий сведения из трассировки стека, а также само сообщение об ошибке. Такой объект может быть сгенерирован, если потребуется известить о возникновении неустранимой ошибки. Допустим, требуется перейти по заданному URL, который сначала необходимо декодировать. В приведенном ниже примере функция вызывается как с достоверными, так и с недостоверными входными данными.

```
const parse = (url) => url.parseUrl(); ◄-----  ---------- 1 Эта функция была создана
                                                  в разделе 4.4.1
decode ('V') .map (parse) ; //-> Left(Error('URI malformed'))
decode('http%3A%2F%2Fexample.com').map(parse);
// -> Right(true)
```

Функциональное программирование позволяет вообще обойтись без генерирования исключений. Вместо этого можно пользоваться монадами для генерирования исключений по требованию, сохраняя объект соответствующего исключения в левой структуре монады типа Either. В таком случае исключение возникает только после распаковки левой структуры, как показано ниже.

```
catch (uriError) {
    return Either.Left(uriError); }
```

Итак, мы показали, как пользоваться монадами, чтобы сымитировать механизм генерирования и перехвата исключений в блоке операторов try-catch, содержащем потенциально опасные функции. Аналогичное понятие реализовано в языке Scala с помощью типа Try, являющегося функциональной альтернативой блоку операторов try-

catch. И хотя тип Try нельзя назвать полностью монадическим, тем не менее, он семантически равнозначен типу Either и включает в себя два класса, Success и Failure, для удачного и неудачного исхода операции соответственно.

Проекты ФП, заслуживающие изучения

Большинство принципов и понятий, рассмотренных в этой и предыдущей главе, в том числе частичное применение, кортежи, композиция, функторы и монады, а также те из них, которые еще предстоит рассмотреть в последующих главах, реализованы в виде модулей в формальной спецификации под названием Fantasy Land (https://github.com/fantasyland). Это эталонная реализация принципов ФП, определяющих порядок внедрения функциональной алгебры в JavaScript. И хотя в примерах из этой книги ради простоты применяются библиотеки Lodash и Ramda, Fantasy Land и функциональная библиотека Folktale (http://folktalejs.org/) заслуживают особого внимания и изучения теми, кто стремится более основательно разобраться в типах данных, носящих более функциональный характер.

Монады способны оказывать помощь в борьбе с неопределенностью и вероятными сбоями в реальном программном обеспечении. Но как организовать взаимодействие с внешним миром?

5.3.3. Взаимодействие с внешними ресурсами с помощью монады типа ю

Язык программирования Haskell считается единственным, сильно опирающимся на монады для выполнения операций ввода-вывода, включая чтение и запись данных в файлы, вывод информации на экран и т.д. А в коде JavaScript операции ввода-вывода можно представить следующим образом:

```
10.of('An unsafe operation').map(alert);
```

И хотя это довольно простой пример, он наглядно демонстрирует сложности внедрения ввода-вывода в монадические операции, передаваемые платформе на выполнение (в данном случае лишь для вывода предупреждающего сообщения). Но в коде JavaScript неизбежно требуется взаимодействовать с постоянно изменяющейся, коллективно используемой, сохраняющей состояние моделью DOM. В итоге любая операция, выполняемая над моделью DOM, будь то чтение или запись, вызывает побочные эффекты и нарушает ссылочную прозрачность. Итак, начнем рассмотрение данного вопроса со следующего примера самых элементарных операций ввода-вывода:

```
const read = (document, selector) => {t   read   =   (document,   selector)   =>   {
                                                    Последующие вызовы
                                                    могут привести к разным
                                                    результатам
    return document. queryselector (selector) . innerHTML;
                                                    Значение не возвращается,
                                                    что явно приводит к мо-
                                                    дификациям (т.е. к небез-
                                                    опасным операциям)
◄---------------------------------------------------- для чтения данных
const write = (document, selector, val) => {
    document.queryselector(selector).innerHTML = val;-< return val;
```

Если выполнять эти самостоятельные функции независимо, то нельзя гарантировать конечный результат. Ведь значение имеет не только порядок их выполнения, но и то обстоятельство, что модель DOM может быть изменена при вызове функции write () в промежутках между вызовами функции read (). Напомним, что главной причиной для отделения нечистого поведения от чистого кода, как это было показано на примере программы showStudent в главе 4, "На пути к повторно используемому, модульному коду", является стремление всегда гарантировать постоянный результат.

Избежать модификаций или устранить осложнения в связи с побочными эффектами при выполнении операций ввода-вывода нельзя. Но, по крайней мере, можно обращаться с операциями ввода-вывода так, как будто они неизменяемы с прикладной точки зрения, заключив их в монадические цепочки, а монаде предоставить возможность приводить в действие поток данных. И для этой цели служит монада типа 10, реализация которой приведена в листинге 5.7.

Листинг 5.7. Реализация монады типа Ю

```
class  10if (! . is Function (effect) ) {  DOM). Эта операция называется также функцией
                                           результата
            throw '10 Usage: function required';
        }

        this effect = effect;
{                                          Инициализировать конструктор типа Ю операцией чтения-
    constructor (effect) { ◄------ —■ записи (аналогично чтению или записи данных в модель

    static of(a) {              --
        return new 10( () => a );
                    ◄

    static from(fn) {
        return new IO(fn);
```

```
map(fn) {          ◄---- 1 преобразования
    let self = this;
    return new IO(() => fn(self.effect()));


chain(fn) {
    return fn(this.effect());
```

Единичные функции для заключения значений и функций в оболочку монады типа 10

I Функтор -J преобразования

```
    run () {                                  Запустить инициализи-
        return this, effect (); ^ --- руемую по требованию
    j                                         цепочку, чтобы выполнить
                                              операцию ввода-вывода
```

Эта монада действует иначе, чем другие, поскольку она заключает в оболочку функцию *результата,* а не значение. Напомним, что функцию можно рассматривать как *отложенное,* если угодно, *значение,* ожидающее своего вычисления. С помощью этой монады можно связать в цепочку любые операции, выполняемые над моделью DOM, как единую якобы ссылочно-прозрачную операцию, гарантируя при этом, что функции, приводящие к побочным эффектам, будут вызываться в строго установленном порядке, но не в промежутках между вызовами функций без побочных эффектов.

Прежде чем продемонстрировать эту монаду на практике, реорганизуем функции read () и write (), подвергнув их каррингу вручную:

```
const read = (document, selector) => { return () => {
        return document.queryselector(selector).innerHTML;
    };
};

const write = (document, selector) => {
    return (val) => {
        document.queryselector(selector).innerHTML = val;
        return val;
    };
};
```

А для того чтобы не передавать объект document по цепочке, можно упростить дело, применив его в качестве частичного аргумента этих функций:

```
const readDom = _.partial(read, document);
const writeDom = _.partial(write, document);
```

После этих изменений функции readDom () и writeDom () можно связывать в цепочку (и составлять в композицию) с отложенным выполнением. Это требуется для того, чтобы в дальнейшем связывать в цепочку операции ввода-вывода. Рассмотрим следующий простой пример, где имя учащегося сначала читается из элемента HTML-разметки, а затем изменяется, чтобы оно начиналось с прописной буквы:

```
<div id="student-name">alonzo church</div>

const changeToStartCase =
    10. from (readDom (' #student-name') )    i Здесь можно применить любую
    .map (_. startcase)                       «Г■——* операцию преобразования
    .map(writeDom('#student-name'));
```

Последняя в цепочке операция записи данных в модель DOM не является чистой. Какого же тогда результата следует ожидать от вызова функции changeToStartCase () ? Применение монад тем и примечательно, что в них сохраняются требования, предъявляемые чистыми функциями. Как и в любой другой монаде, результатом выполнения операции tap оказывается сама монада (в данном случае — экземпляр типа 10). Это означает, что на

данной стадии пока еще ничего не выполнено, но уже имеется декларативное описание операции ввода-вывода. И, наконец, выполним следующую строку кода:

```
changeToStartCase.run();
```

Проанализировав модель DOM, в ней можно обнаружить следующий элемент HTML-разметки:

```
<div id="student-name">Alonzo Church</div>
```

Таким образом, операции ввода-вывода выполняются ссылочно-прозрачно! Самое главное преимущество монады типа 10 заключается в том, что она явным образом отделяет чистую часть кода от нечистой. Как следует из определения функции changeToStartCase (), функции преобразования, применяемые к контейнеру ввода-вывода, полностью отделены от логики чтения и записи данных в модель DOM. Следовательно, преобразование содержимого элемента HTML-разметки может быть выполнено по мере надобности. А поскольку оно выполняется одномоментно, то в промежутках между операциями чтения и записи не произойдет ничего такого, что могло бы привести к непредсказуемым результатам.

Монады являются всего лишь соединяемыми в цепочку выражениями или вычислениями. Это дает возможность составлять последовательности для дополнительной обработки на каждой стадии процесса, как на ленте конвейерной линии сборки. Но связывание операций в цепочку является не единственным примером применения монад. Используя монадические контейнеры в качестве возвращаемых типов, можно формировать согласованные, безопасные по отношению к типу возвращаемые значения для функций, сохраняя при этом ссылочную прозрачность. Как упоминалось в главе 4, “На пути к повторно используемому, модульному коду”, этим удовлетворяется требование к составлению цепочек функций и композиций.

5.4. Монадические цепочки и композиции

Как видите, монады позволяют взять побочные эффекты под контроль, а следовательно, их можно применять в структурах, составляемых в композицию. Но в главе 4, “На нуги к повторно используемому, модульному коду”, нас не интересовала проверка недостоверности данных. Так, если бы функция findStudent () возвратила пустое значение null, то неудачно завершилась бы вся программа (рис. 5.10).

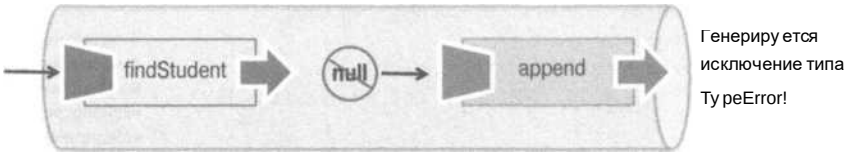


Рис. 5.10. Композиция функций findStudent () и append (). Если бы первая из них возвратила пустое значение null, то в отсутствие надлежащих проверок вторая функция завершилась бы неудачно, сгенерировав исключение типа TypeError

Правда, написав немного кода, можно и монады сделать составляемыми в композицию, чтобы извлечь выгоду из их плавного, выразительного механизма обработки ошибок при составлении безопасных композиций. Было бы совсем неплохо, если бы функции, организованные в конвейер, корректно обходили препятствия, связанные в появлением пустых значений, не так ли?

Как показано на рис. 5.11, на первой стадии необходимо убедиться, что первая выполняемая функция заключает свой результат в оболочку подходящей монады. И для этой цели подойдут обе монады типа Maybe и Either.

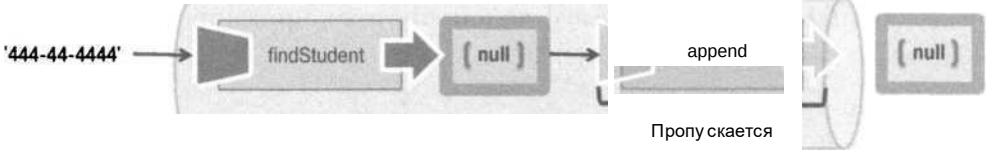


Рис. 5.11. Те же самые функции, что и на рис. 5.10, но на этот раз пустое значение null передается по конвейеру в монаде (Either или Maybe). В итоге остальные функции в конвейере корректно выходят из сбойной ситуации

Как известно, в ФП имеются две разновидности объединения функций: цепочка и композиция. Напомним, что версия программы showStudent, рассматривавшаяся в предыдущей главе, состояла из следующих частей.

1. Нормализация пользовательского ввода.
2. Поиск записи об учащемся.
3. Вывод сведений об учащемся на HTML-странице.

Если добавить к этому проверку достоверности входных данных, то дело усложнится еще больше. Таким образом, в данной программе обнаруживаются две точки сбоев: ошибка достоверности данных и неудачное завершение операции извлечения сведений об учащемся. Эти места программы можно реорганизовать, включив монаду типа Either, предоставляющую соответствующие сообщения об ошибках, как демонстрируется в листинге 5.8.

Листинг 5.8. Реорганизация функций с целью применить монаду типа Either

```
// validLength :: Number, String -> Boolean const validLength = (len, str) =>
str.length === len;
```

```
// checkLengthSsn :: String -> Either(String)
const checkLengthSsn = ssn =>                          ◄ ----
      validLength(9,ssn) ? Either.right(ssn):
              Either.left('Invalid SSN');

// safeFindObject :: Store, string -> Either(Object)
const safeFindObject = R.curry((db, id) => {          ----
    const val = find(db, id);
    return val ? Either.right(val) :
                 Either.left('Object not found with ID: ${id}'); });

// finStudent :: String -> Either(Student) const findStudent =
safeFindObject(DB('students'));

// csv :: Array => String const csv = arr
=> arr.join(',');-<
```

Вместо того чтобы заключать эти функции в оболочку монады типа Either, можно воспользоваться ею непосредственно, предоставив конкретные сообщения об ошибках в зависимости от характера ошибки

Реорганизованная функция csv () возвращает символьную строку из массива значений

Приведенные выше функции каррированы, и поэтому их можно вычислить частично, чтобы получить более простые функции, как это делалось раньше, а также добавить вспомогательные функции протоколирования:

```
const debugLog = partial(logger, 'console', 'basic',
                         'Monad Example', 'TRACE');

const errorLog = partial(logger, 'console', 'basic', 'Monad Example', 'ERROR');

const trace = R.curry((msg, val)=> debugLog(msg + ':' + val));
```

Вот, собственно, и все! Обо всем остальном позаботятся монадические операции, обеспечив перенос данных через вызовы функций без дополнительных издержек. В исходном коде из листинга 5.9 показано, как внедрить автоматическую обработку ошибок в программу showStudent, используя монады типа Either и Maybe.

Листинг 5.9. Применение монад в программе showStudent для автоматической обработки ошибок

```
const showStudent = (ssn) =>-<-
    Maybe.fromNullable(ssn)
        .map (cleaninput)
        .chain(checkLengthSsn)
        .chain(findStudent)

        .map (R.props(['ssn', 'firstname',   'lastname'])) ◄---
        .тap (csv)
        .тap (append('#student-info'));
```

С помощью методов map () и chain () можно преобразовать значение в монаде. Метод шар () возвращает монаду. Чтобы исключить вложение и потребность в сведении структуры, методы тар () и chain () связываются вместе, сохраняя единый уровень монады, проходящий через все вызовы

Извлечь выбранные свойства из объекта в массив

В листинге 5.9 демонстрируется применение метода chain () ради краткости, чтобы не прибегать к операции join после операции тар для сведения уровней, получающихся в результате объединения функций, возвращающих монады. Подобно методутар (), метод chain () применяет функцию к данным, не заключая полученный результат обратно в монадический тип.

Следует также заметить, что обе монады типа Either и Maybe плавно чередуются, поскольку они реализуют один и тот же монадический интерфейс. Так, если сделать следующий вызов:

```
showStudent('444-44-4444').orElse(errorLog);
```

то в конечном счете будут получены два результата. Если объект, представляющий учащегося, успешно найден, сведения об учащемся выводятся, как и предполагалось, на HTML-странице и возвращается следующий результат:

```
Monad Example [INFO] Either.Right('444-44-4444, Alonzo,Church')
```

В противном случае вся операция корректно пропускается и вызывается метод orElse () в альтернативной ветви, возвращая такой результат:

```
Monad Example [ERROR] Student not found with ID: 444444444
```

Связывание в цепочку — не единственный способ обработки ошибок. Логику обработки ошибок можно внедрить и с помощью композиции. С этой целью следует выполнить простой переход от ООП к ФП, как было показано ранее на примере преобразования монадных методов в функции, полиморфно обрабатывающие данные любого монадического типа в соответствии с принципом подстановки Барбары Л исков. В частности, можно создать обобщенные функции tap () и chain (), как демонстрируется в листинге 5.10.

Листинг 5.10. Обобщенные функции tap () и chain (), пригодные для работы с любым контейнером

```
// map :: (ObjectA -> ObjectB), Monad -> Monad[ObjectB] const map = R.curry((f,
container) => {
    return container.map(f);
}) ;

// chain :: (ObjectA -> ObjectB), Monad -> ObjectB const chain = R.curry((f,
container) => {
    return container.chain(f);});
```

Используя эти функции, можно внедрять монады в составляемое выражение. Выполнение исходного кода из листинга 5.11 дает такой же результат, как и выполнение исходного кода из листинга 5.9. Монады направляют поток данных из одного выражения в другое, и поэтому такой стиль безточечного программирования иначе называется *программируемыми запятыми*. В данном случае запятые служат для разделения выражений аналогично точкам с запятыми, по традиции разделяющими операторы в коде JavaScript. Кроме того, операторы trace позволяют отслеживать прохождение потока данных через операции, а операторы протоколирования полезны для отладки.

Листинг 5.11. Монады как программируемые запятые

```
const showStudent = R.compose(
    R.tap(trace('Student added to HTML page'))
    map(append('#student-info')),
    R.tap(trace('Student info converted to CSV')),
    map(csv),
    map(R.props(['ssn', 'firstname', 'lastname'])),
    R.tap(trace('Record fetched successfully!')),
    chain(findStudent),
```

```
    R.tap(trace('Input was valid')),
    chain(checkLengthSsn), lift(cleaninput));
```

В результате выполнения исходного кода из листинга 5.11 на консоль выводятся следующие сообщения:

```
Monad Example [TRACE] Input was valid:Either.Right(444444444)

Monad Example [TRACE] Record fetched successfully!:
      Either.Right(Person [firstname: Alonzo| lastname: Church])

Monad Example [TRACE] Student converted to CSV:
      Either.Right(444-44-4444,Alonzo, Church)

Monad Example [TRACE] Student added to HTML page: Either.Right(1)
```

Трассировка программ

В листинге 5.11 показано, насколько просто отслеживать функциональный код. Не углубляясь в тело функций, можно разметить всю программу операторами трассировки, выполняемыми до и после вызовов функций, что очень удобно для поиска неполадок и отладки. Если бы та же самая программа была написана в объектно-ориентированном стиле, добиться аналогичного результата не удалось бы, не внеся изменения в отдельные функции, а, возможно, и снабдив их элементами аспектно-ориентированного программирования, что само по себе дело непростое. А функциональное программирование предоставляет такую возможность бесплатно!

И, наконец, составим блок-схему всего потока выполнения рассматриваемой здесь программы, чтобы наглядно показать на рис. 5.12 все, что в ней происходит. А на рис. 5.13 показано поведение той же самой программы при неудачном завершении функции findStudent ().

Можно ли считать программу showStudent наконец-то завершенной? Не совсем. Как пояснялось ранее при обсуждении монады типа 10, исходный код, отвечающий за чтение и запись данных в модели DOM, может быть усовершенствован следующим образом:

```
map(append('#student-info')),
```

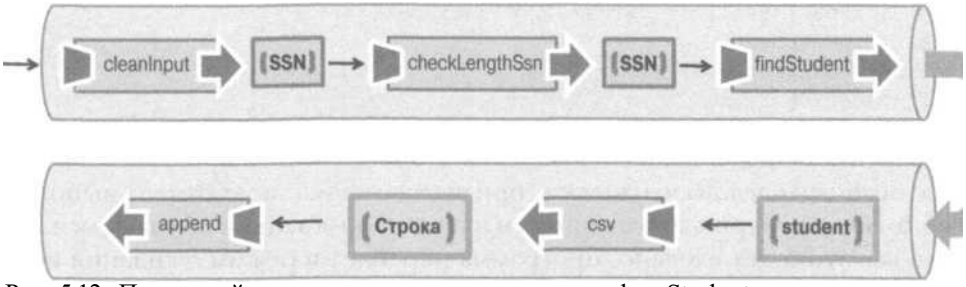


Рис. 5.12. Поэтапный поток выполнения программы showStudent в том случае, если функция findStudent () успешно найдет объект, представляющий учащегося, по заданному номеру социального страхования (SSN)

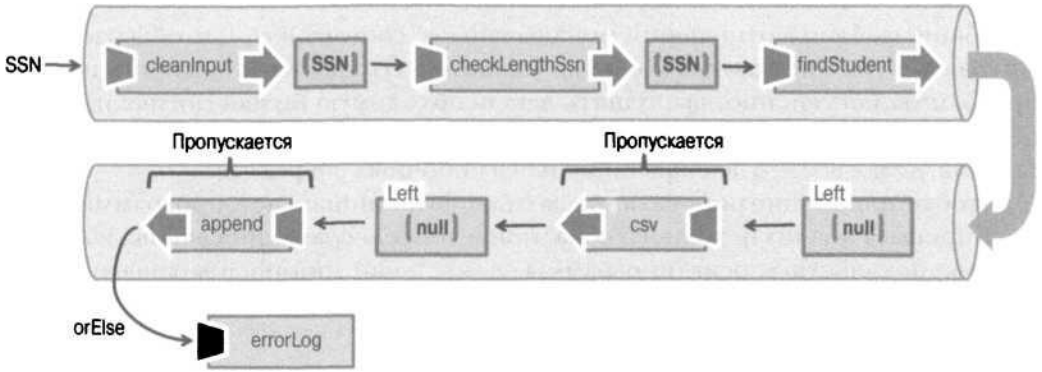


Рис. 5.13. Поэтапный поток выполнения той же самой программы при неудачном завершении функции findStudent O, что оказывает влияние на остальную часть композиции. Независимо от характера сбоя в любом месте конвейера, программа остается отказоустойчивой, корректно пропуская любые процедуры, зависящие отданных

Функция append () автоматически подвергается каррингу, и поэтому она вполне пригодна для оперирования монадой типа 10. Для этого достаточно поднять значение из функции csv (), извлечь его содержимое, применив функцию преобразования R. identity () к монаде типа 10 через вызов метода 10. of (), как показано ниже, а затем связать обе операции в цепочку.

```
const liftIO = function (val) { return 10.of(val);
};
```

В итоге рассматриваемая здесь программа приобретет окончательный вид, приведенный в листинге 5.12.

Листинг 5.12. Окончательный вариант программы showStudent

```
const showStudent = R.compose(
   map(append('#student-info')),
   liftIO,
   map (csv) ,

   map(R.props(['ssn', 'firstname', 'lastname'])),
   chain(findStudent),
   chain(checkLengthSsn), lift(cleaninput));
```

Внедрение монады типа 10 позволяет добиться весьма примечательных результатов. Как и следовало ожидать, при вызове showStudent (ssn) выполняется вся логика проверки достоверности и извлечения записи об учащемся. И как только все это будет сделано, программа перейдет в режим ожидания вывода этих данных на экран. А поскольку они заключены в монаду типа 10, то нужно вызвать ее метод run () для вывода на экран данных, которые по требованию содержатся в ней (т.е. в ее замыкании):

```
showStudent(studentld).run(); // -> 444-44-4444, Alonzo, Church
```

Общий шаблон в отношении монады типа 10 состоит в том, чтобы расположить нечистую операцию в конце композиции. Это дает возможность писать программы постепенно, выполнять всю необходимую бизнес-логику и, наконец, предоставлять монаде типа 10 данные как на блюдечке для завершения задания, делая все это декларативно и без побочных эффектов.

Чтобы продемонстрировать, насколько функциональное программирование упрощает анализ исходного кода, приведем для сравнения эквивалентную нефункциональную версию программы showStudent, принося заранее извинение за воскрешение не вполне приглядного кода:

```
function showStudent(ssn) {
    if(ssn != null) {
        ssn = ssn.replace(/^\s*|\-|\s*$/g, '');
        if(ssn.length !== 9) {
            throw new Error('Invalid Input');
        }
        let student = db.get(ssn);
        if (student) {
            document.queryselector('#${elementld}').innerHTML =
                                '${student.ssn},
                                    ${student.firstname},
                                    ${student.lastname}';
        }
        else {
            throw new Error('Student not found!');
        }
    }
    else {
        throw new Error('Invalid SSN!');
    }
}
```

Из-за побочных эффектов, отсутствия модульности и императивного способа обработки ошибок такую программу трудно тестировать и эксплуатировать. Если композиция управляет потоком выполнения программы, то монады управляют потоком данных. И это самые важные понятия в экосистеме функционального программирования.

Этой главой завершается часть II данной книги. Проработав материал этой части, вы смогли пополнить арсенал своих средств разработки всеми понятиями и принципами ФП, которые требуются для принятия практических решений.

Резюме

Из этой главы вы узнали следующее.

- Механизмы генерирования исключений в объектно-ориентированном коде становятся причиной появления нечистых функций, возлагающих на вызывающий код немалое бремя ответственности за предоставление подходящей логики генерирования и перехвата исключений в блоке операторов try-catch.
- Шаблон заключения значений в оболочку служит для получения свободного от побочных эффектов кода, где возможные модификации охвачены единым ссылочно прозрачным процессом.
- Функторы служат для применения функций преобразования к контейнерам (или оболочкам) с целью получить доступ и модифицировать объекты без побочных эффектов и неизменяемым способом.
- Монады служат в качестве проектного шаблона в ФП для уменьшения сложности приложений, организуя и направляя безопасный поток данных через функции.
- В безотказных и надежных композициях чередуются монадические типы вроде Maybe, Either и IO.

Расширение функциональных навыков

частях I, “Умение мыслить функционально”, и II, “Погружаемся в функцио- нальное программирование”, были представлены инструментальные средства функционального программирования, необходимые для решения практических задач. Из материала этих частей вы узнали о новых методиках и проектных шаблонах, предназначенных для устранения побочных эффектов и позволяющих сделать прикладной код модульным, расширяемым и легко понимаемым. А в части III будет показано, как воспользоваться полученными знаниями и навыками для решения задач модульного тестирования приложений HaJavaScript, оптимизации прикладного кода по принципам ФП и преодоления трудностей обработки асинхронных событий и данных.

В главе 6, “Отказоустойчивость прикладного кода”, основное внимание уделяется модульному тестированию императивных приложений и раскрываются причины, по которым функциональные приложения оказываются менее сложными и более удобными для тестирования. Достижение ссылочной прозрачности также приводит к методике автоматического тестирования, называемой *тестированием на основе свойств (property-based testing)*.

В главе 7, “Оптимизация функционального кода”, исследуется внутренний механизм работы контекста функций BjavaScript, а также обсуждаются вопросы производительности, которые приходится принимать во внимание, применяя глубоко вложенные замыкания функций и рекурсию. Здесь представлены методики отложенного вычисления, запоминания и оптимизации хвостовых вызовов.

И, наконец, в главе 8, “Обработка асинхронных событий и данных”, рассматриваются монадические проектные шаблоны, предназначенные для борьбы с увеличением сложности приложений. В частности, здесь обсуждается решение двух часто встречающихся BjavaScript задач: асинхронного извлечения данных из сервера или базы данных с помощью обязательств (обещаний), а также

сокращения традиционных обратных вызовов функций в управляемых событиями программах с помощью реактивного подхода и библиотеки RxJS.

Проработав весь материал этой книги, вы будете вооружены достаточными знаниями, чтобы успешно применять методики ФП в своей профессиональной деятельности.

Отказоустойчивость прикладного кода

в этой главе...

- Влияние функционального программирования на тестиро- вание программ
- Выявление трудностей в тестировании императивного кода
- Тестирование функционального кода средствами QUnit
- Исследование возможностей тестирования на основе свойств средствами JSCheck
- Количественная оценка сложности программ средствами Blanket

Сосед хорош, когда забор хороший.

Из стихотворения "Mending Wall" (Починка стены)
Роберта Фроста (Robert Frost)

Приглашаем к чтению части III этой книги. Читая части I и II, вы, вероятно, обратили внимание на центральную тему, проходящую красной нитью через всю книгу: функциональное программирование упрощает понимание, чтение и сопровождение прикладного кода. Можно даже сказать, что декларативный характер прикладного кода делает его самодокументирующимся.

Итак, написав функциональный код, необходимо убедиться в его работоспособности. Но как добиться, чтобы он удовлетворял техническим требованиям заказчиков? Единственный способ — написать код, который можно протестировать на соответствие результирующего поведения ожидаемому. Умение мыслить функционально оказывает глубокое влияние на прикладной код и непосредственно на порядок разработки его тестов.

Модульные тесты создаются для того, чтобы проверяемый код удовлетворял постановке решаемой задачи. Они строят ограды вокруг всех возможных граничных условий, которые могут вызвать его сбой. Если у вас имеется опыт написания модульных тестов, то вам, вероятно, известно, что тестирование императивных программ — дело непростое, особенно в крупных кодовых базах. Из-за побочных эффектов императивный код подвержен ошибкам, возникающим из ложных предположений относительно глобального состояния системы. Аналогично одни тесты нельзя выполнять независимо от других, как это обычно требуется, что затрудняет возможность гарантировать постоянные результаты независимо от порядка, в котором выполняются тесты. И в этом состоит, к сожалению, главная причина, по которой тестирование нередко откладывается до самого конца разработки, а зачастую и вообще пропускается.

В этой главе рассматриваются причины, по которым функциональный код по определению оказывается тестируемым, тогда как в большинстве других парадигм программирования прикладной код приходится намеренно делать удобным для тестирования. Большая часть норм передовой практики, связанных с надлежащим тестированием, в том числе устранение внешних зависимостей, достижение прогнозируемости функций и прочие, составляют базовые принципы, положенные в основу функционального проектирования. Чистые, ссылочно-прозрачные функции обладают этим внутренним качеством по определению и пригодны для более развитой методики тестирования, например, на основе свойств. Но прежде чем перейти к особенностям тестирования функционального кода, остановимся немного на разъяснении того влияния, которое ФП оказывает на разные виды тестов и, в частности, на модульные тесты как наиболее производительные.

6.1. Влияние функционального программирования на модульные тесты

В общем, имеются следующие виды тестов: модульные, комплексные и приемочные. На рис. 6.1 приведена пирамида видов тестирования, наглядно демонстрирующая рост влияния ФП на прикладной код по мере перехода от приемочных тестов *(сверху) к* модульным *(снизу)*. И это вполне очевидное влияние, поскольку ФП — это такая парадигма разработки программного обеспечения, где основное внимание уделяется проектированию функций и модулей, а также интеграции их составных частей.

Несмотря на всю важность тестирования по критериям приемки пользователей в отношении внешнего вида, практичности и возможностей навигации в веб-приложении оно все-таки выполняется отстраненно от кода, а следовательно, мало или совсем никак не связано с функциональным или императивным стилем написания программы. Эта задача больше подходит для сред автоматизации тестирования. Что же касается комплексных тестов, то, как было показано в главе 4, “На пути к повторно используемому, модульному коду”, ФП пере-

дает композиции управление разными компонентами приложения. Поэтому часть времени, затрачиваемого на комплексные тесты, возмещается только за счет принятия парадигмы ФП.

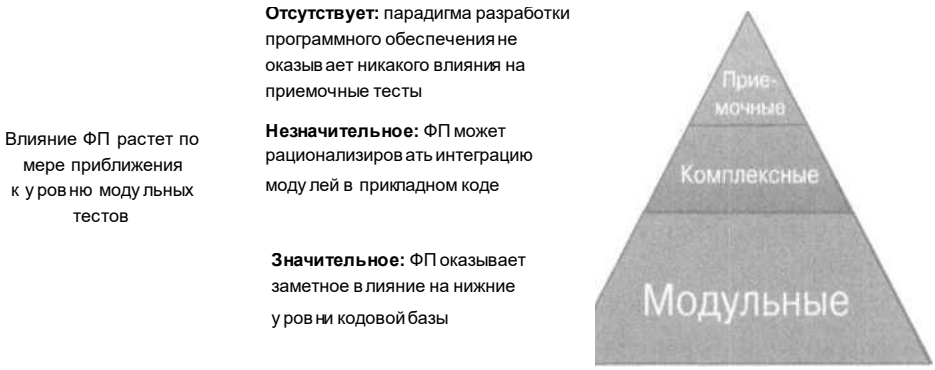


Рис. 6.1. ФП — это такая парадигма разработки программного обеспечения, где основное внимание уделяется разрабатываемому коду, и поэтому она оказывает влияние в основном на модульные тесты, в малой степени на комплексные тесты и почти никак на приемочные тесты

Основное внимание в ФП уделяется непосредственно функциям — единицам модульности прикладного кода, а также их взаимодействию. Для выполнения тестов в примерах из этой книги выбрана распространенная библиотека QUnit. Установка библиотек тестирования здесь не рассматривается. Если вам приходилось раньше устанавливать любую библиотеку модульного тестирования, то установить и привести в действие библиотеку QUnit не составит для вас особого труда. Подробнее об этом см. в приложении.

Ниже приведена элементарная структура одиночного модульного теста.

```
QUnit.test('Test Find Person', function(assert) { const ssn = '444-44-4444';
    const p = findPerson(ssn);
    assert.equal(p.ssn, ssn);
});
```

Тестовый код обычно находится в файле JavaScript, не входящем в состав основного кода приложения, но импортирующем все тестируемые функции. Выполнить модульное тестирование императивной программы крайне трудно из-за побочных эффектов и модификаций. Рассмотрим некоторые уязвимые стороны тестирования императивного кода.

6.2. Трудности тестирования императивных программ

Императивные тесты страдают теми же недостатками, что и императивный код. А поскольку императивный код зависит от глобального состояния и мутаций программы, а не от содержащихся в нем потоках данных и связанных вычислениях, то его тестирование представляет настоящую трудность. При проектировании модульных тестов следует придерживаться одного из главных принципов — *изоляции*. Это означает, что модульный

тест должен выполняться как в вакууме независимо от любых других данных или окружающих его тестов. По побочные эффекты в коде серьезно ограничивают пределы, до которых можно тестировать функции.

Таким образом, императивный код:

- с трудом поддается выявлению и разбиению на простые задачи;
- зависит от общих ресурсов, которые делают результаты тестов непостоянными;
- вынужден выполняться в предопределенном порядке.

Рассмотрим некоторые из перечисленных выше трудностей тестирования императивного кода более подробно.

6.2.1. Трудность выявления и разбиения на простые задачи

Модульные тесты предназначены для тестирования наименьших частей приложения. В процедурных программах намного труднее выявить *единицы модульности* из-за отсутствия интуитивного способа разбиения на части единой монолитной программы, не предназначенной изначально для этой цели. В данном случае такими единицами являются *функции,* инкапсулирующие бизнес-логику. В качестве примера можно вспомнить императивную версию программы showStudent, которая рассматривается на протяжении всей этой книги. Удачная попытка разбить эту программу на составные части приведена на рис. 6.2.

Как видите, данная программа состоит из тесно связанной вместе бизнес- логики, отвечающей за разные аспекты программы и объединенной в единую монолитную функцию. Но нет никаких реальных причин связывать вместе проверку достоверности данных с извлечением записей об учащихся и добавлением элементов в модель DOM. Они должны быть разделены на тестируемые единицы бизнес-логики, соединяемые посредством композиции. И, как пояснялось в главе 5, “Проектные шаблоны и сложность”, логику обработки ошибок следует вынести в отдельные монады, способные справиться с ошибками.

Монады и обработка ошибок

В главе 5 был представлен ряд проектных шаблонов, которые можно применять для объединения и удаления кода обработки ошибок из основных функций, сохраняя в то же время их отказоустойчивость. Так, используя монады типа Maybe и Either, можно написать в бесточечном стиле код, в котором известно, как распространять ошибки через компоненты программы, поддерживая в то же время ее способность реагировать на действия пользователя.

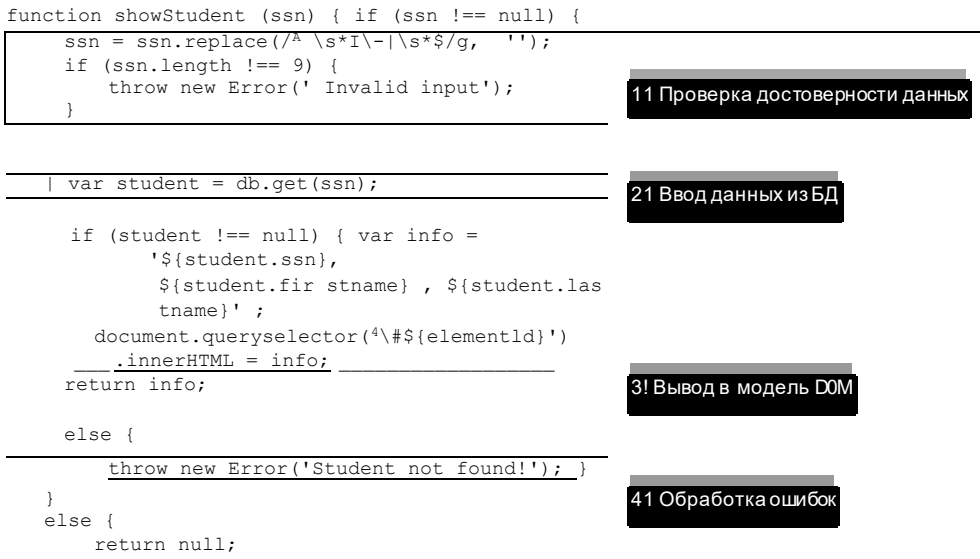


Рис. 6.2. Функциональные части монолитной программы на основе функции showStudent (). Чтобы упростить написание тестов, эти части должны быть разбиты на отдельные функции, отвечающие за проверку достоверности данных, ввод-вывод и обработку ошибок

Чтобы расширить рамки тестирования данной программы, придется найти способы ее разбиения на слабо связанные составляющие, отделяющие чистый код от нечистого. Нечистый код с трудом поддается тестированию из-за побочных эффектов, которые могут возникать при чтении и записи данных в таких внешних ресурсах, как модель DOM или внешнее хранилище.

6.2.2. Зависимость от общих ресурсов, приводящая к непостоянным результатам

Как пояснялось в главе 2, “Сценарий высшего порядка”, язык JavaScript предоставляет ничем не связанную свободу доступа к глобальным общим данным. Поэтому для тестирования программ с побочными эффектами требуется особая осторожность и дисциплина, поскольку разработчик отвечает за управление состоянием, связанным с тестируемой функцией. Слишком часто приходится наблюдать, как добавление нового теста в рабочий тестовый набор приводит к неумышленному непрохождению других, не связанных с ним тестов. И объясняется это тем, что тесты должны быть самодостаточными или независимыми от всего остального, чтобы стать надежными. Это означает, что каждый модульный тест, по существу, выполняется в своей “песочнице”, оставляя систему в том же состоянии, в каком она находилась до его выполнения. Тесты, нарушающие это правило, никогда не смогут давать одинаковые результаты.

Обратимся для иллюстрации к простому примеру упоминавшейся ранее императивной версии функции increment (): `var counter = 0; // (глобальная переменная)`

```
function increment() { return ++counter;
}
```

Чтобы убедиться в том, что число инкрементируется от 0 до 1, можно написать простой модульный тест. Полученный результат должен оставаться неизменным, сколько бы раз (1 или 100) ни выполнялся тест. Но этого не происходит, поскольку тестируемая функция читает и модифицирует внешние данные (рис. 6.3).

Повторение jfIHi одного и того И же теста

```
QUnit. test("Increment with zero", function (assert) { assert
     .equal (increment (), 1)
}>; </
```

```
QUnit .te st ("Increment with zero (again)", function (assert) {
     assert .equal (increment (), 1)                          к/
));                                                           Д
```

Рис. 6.3. Повторение модульного теста императивной версии функции increment () невозможно из-за того, что она зависит от внешней переменной counter

Тест не проходит уже на второй итерации, поскольку на первой итерации значение внешней переменной counter было изменено на 1, вытеснив глобальный контекст для второго выполнения того же самого кода, а следовательно, заданное в нем утверждение не подтверждается. Аналогично функции с побочными эффектами подвержены ошибкам, обусловленным порядком их вычисления. Подробнее об этом речь пойдет далее.

6.2.3. Предопределенный порядок вычисления

Аналогично постоянству результатов, модульные тесты должны быть *коммутативными*. Это означает, что изменение порядка их выполнения не должно оказывать влияния на получаемые результаты. По тем же самым причинам, что и прежде, данный принцип не соблюдается в нечистых функциях. В качестве обходного приема в таких библиотеках модульного тестирования, как QUnit, предоставляются готовые механизмы для установки и очистки глобальной среды проведения тестов. Но исходные данные для одного теста могут совершенно отличаться от другого, и поэтому вначале каждого теста приходится задавать соответствующие предусловия. Это подразумевает также вашу ответственность за выявление всех побочных эффектов (внешних зависимостей) проверяемого кода в каждом тесте.

В качестве примера рассмотрим создание простых тестов, охватывающих функцию increment (), чтобы проверить с их помощью ее поведение относительно отрицательных, нулевых и положительных числовых значений (рис. 6.4). При первой попытке *(слева)* все тесты проходят. Но достаточно изменить порядок их выполнения *(справа),* не внося никаких дополнительных изменений, чтобы второй тест не прошел. И объясняется это тем, что тесты с побочными эффектами выполняются на основании того, что окружающее их состояние устанавливается соответствующим образом.

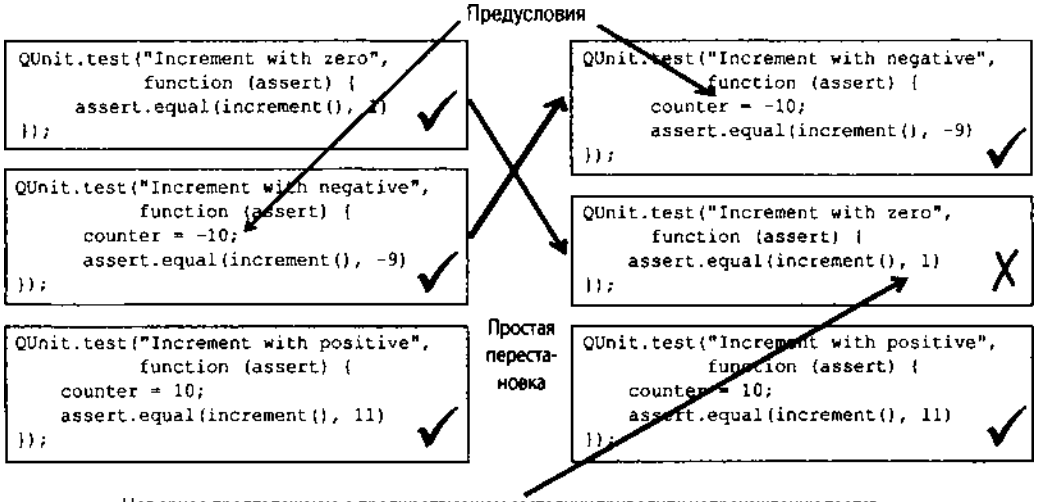


Рис. 6.4. Ложное предположение о глобальном состоянии системы приводит к тому, что даже простой тест не проходит. Как показано на рисунке *слева,* все тесты выполняются идеально, поскольку перед выполнением каждого из них соответственно подготовлено окружающее его состояние. Но стоит изменить порядок выполнения тестов *(справа),* и все предположения об окружающем их состоянии будут нарушены

Как следует даже из такого простого примера, если попытаться успешно выполнить несколько модульных тестов конкретной функции, манипулируя глобальным контекстом в каждом тесте, то нет никакой гарантии, что они пройдут' после изменения порядка их выполнения. Достаточно хотя бы немного изменить этот порядок, чтобы все утверждения в тестах оказались недостоверными.

Умение мыслить функционально помогает и при создании надежных тестовых наборов. И если прикладной код написан в функциональном стиле, то подобная надежность достигается автоматически. Вместо того чтобы безнадежно подгонять принципы ФП под тестовый код, не лучше ли написать его в функциональном стиле изначально, возместив затраченное время на стадии тестирования? Рассмотрим далее преимущества функционального кода в отношении тестирования.

6.3. Тестирование функционального кода

Независимо от того, тестируется ли императивный или функциональный код, многие нормы передовой практики разработки модульных тестов, в том числе изоляция, прогнозируемость и повторяемость, распространяются и на ФП. А поскольку в каждой функции ясно определяются все ее входные параметры, то совсем не трудно задать различные граничные условия для тщательной проверки всех путей выполнения прикладного кода. Что же касается побочных эффектов, то напомним, что все функции определяются ясно и просто, а весь чистый код может быть надежно заключен в монады.

Кроме того, нечистота, обнаруживающаяся в конструкциях ручной организации циклов, устраняется путем передачи управления операциям более высокого порядка вроде map, reduce, filter и рекурсии, а также библиотекам, свободным от побочных эффектов. Такие

методики и проектные шаблоны позволяют эффективно абстрагировать сложность прикладного кода, чтобы тестировать его более продуктивно, заботясь лишь о самых главных частях биз- нес-логики. В этом разделе обсуждаются преимущества тестирования функционального кода, в том числе следующие.

- Интерпретация функции как "черного ящика".
- Сосредоточение основного внимания на бизнес-логике, а не на потоке управления программой.
- Отделение чистого кода от нечистого путем монадического изолирования.
- Имитация внешних зависимостей.

6.3.1. Интерпретация функции как "черного ящика"

Функциональное программирование побуждает к написанию независимых функций, где известно, как обрабатывать входные данные в слабо связанной манере и независимо от остальной части приложения. Такие функции не вызывают побочных эффектов и соблюдают ссылочную прозрачность, и поэтому их тесты получаются вполне прогнозируемыми, а результаты — одинаковыми, сколько бы раз и в каком бы порядке они ни выполнялись. Это дает возможность интерпретировать функцию как "черный ящик", уделяя основное внимание только входным данным, от которых зависят соответствующие выходные данные. Например, для тестирования функции showStudent () требуется приложить столько же усилий, сколько и для тестирования функциональной версии функции increment (), приведенной на рис. 6.5.

Как упоминалось в главе 1, "Основы функционального программирования", явное объявление всех параметров функции в сигнатуре делает ее более удобной для настройки. Благодаря этому значительно упрощается тестирование функции. Ведь в момент предоставления надлежащих аргументов ничего не скрывается от вызывающего кода, где нетрудно предположить, чего следует ожидать от вызываемой функции. Как правило, в простых функциях объявляются один или два параметра, а затем из них путем композиции составляются более сложные функции.

```
QUnit.test("Increment with zero",
function (assert) { assert .equal
(increment (0), 1)
}); v
```

```
    QUnit.test("Increment with zero
     (again)", function (assert) {
    assert.equal(increment(0), 1)
});
```

```
QUnit.test("Increment with ten",
          function (assert) {
    assert.equal(increment(10), 11)
}); ▼
```

```
QUnit.test("Increment with ten",
function (assert) { assert .equal
(increment (10), 11) }); v
```

```
QUnit.test("Increment with zero",
           function (assert) {
    assert.equal(increment(0), 1)
));
```

```
QUnit.test("Increment with negative
one", function (assert) { assert .equal
(increment (-1), 0) }); ~
```

Рис. 6.5. Тесты функциональной версии функции increment() можно повторять и выполнять в любом порядке, неизменно получая одинаковые результаты

6.3.2. Сосредоточение основного внимания на бизнес-логике, а не на потоке управления программой

Тема декомпозиции (или разбиения) задач на простые функции проходит красной нитью через всю эту книгу. Как упоминалось в главе 1, при написании функционального кода большую часть времени приходится уделять разбиению решаемой задачи на более мелкие части, что само по себе непросто, а остальную часть времени — на их соединение вместе. Правда, такие библиотеки, как Lodash и Ramda, восполняют функциональные пробелы BjavaScript, предоставляя точки соединения с помощью функций вроде curry () и compose (). Благодаря комбинаторам функций, упоминавшимся в разделе 4.6, время, предварительно затраченное на проектирование и декомпозицию решаемой задачи, возмещается на стадии тестирования. А на разработчика возлагается единственная ответственность — протестировать отдельные функции, составляющие основную логику разрабатываемой программы. В качестве примера начнем с написания ряда тестов для функциональной версии программы computeAverageGrade. Ее исходный код еще раз приводится для напоминания в листинге 6.1.

Листинг 6.1. Тестирование программы computeAverageGrade

```
const fork = (join, fund, func2) => { return (val) => {
```

```
        return join (fund (val), func2(val)) };
};

const toLetterGrade = (grade) => { if (grade >= 90) return 'A'; if (grade >= 80)
    return 'B'; if (grade >= 70) return 'C; if (grade >= 60) return 'D'; return
    'F';


const computeAverageGrade = R.compose(toLetterGrade,
                                fork (R.divide, R.sum, R.length));
QUnit.test('Compute Average
Grade', function(assert) {

    assert.equal(computeAverageGrade([80, 90, 100]), 'A');
});
```

В данной программе применяется немало простых функций, в том числе функции R. divide (), R. sum () и R. length () из библиотеки Ramda в сочетании со специальным комбинатором f or к (), результат выполнения которого объединяется с функцией toLetterGrade (). Функции, предоставляемые в библиотеке Ramda, уже тщательно проверены и не требуют дополнительного тестирования. Остается лишь написать следующий модульный тест для функции toLetterGrade():

```
QUnit.test('Compute Average Grade:
            toLetterGrade', function (assert) {

    assert.equal(toLetterGrade(90), 'A');
    assert.equal(toLetterGrade(200),'A');
    assert.equal(toLetterGrade(80), 'B');
    assert.equal(toLetterGrade(89), 'B');
    assert.equal(toLetterGrade(70), 'C');
    assert.equal(toLetterGrade(60), 'D');
    assert.equal(toLetterGrade(59), ' F' ) ;
    assert.equal(toLetterGrade(-10),'F');
});
```

Функция toLetterGrade () является чистой, и поэтому ее можно выполнять несколько раз для разных входных данных с целью тестирования многих ее граничных условий. Кроме того, данная функция является ссылочно прозрачной, что дает возможность изменить порядок выполнения контрольных примеров в рассматриваемом здесь тесте, не изменяя его результат. Далее будет продемонстрирован автоматизированный способ формирования надлежащих образцовых входных данных, а до тех пор это будет делаться вручную, чтобы выяснить, правильно ли функция обрабатывает полный набор входных данных. Итак, протестировав отдельные части данной программы, можно с уве

ренностью предположить, что программа в целом работоспособна, поскольку в ее основе лежат композиция и комбинаторы функций.

А что же комбинатор f or к () ? Комбинаторы функций не требуют особого тестирования, поскольку они не содержат никакой бизнес-логики, кроме организации вызовов функций в протоке управления приложением. Как упоминалось в разделе 4.6, комбинаторы функций удобны для подстановки стандартных элементов управляющей логики вроде условных операторов if-else (смена порядка выполнения) и операторов циклов (последовательность выполнения).

В некоторых библиотеках реализованы готовые комбинаторы вроде R. tap (). Но если применяются специальные комбинаторы наподобие fork (), то их можно протестировать независимо от остальной части приложения и отдельно от бизнес-логики. Ради полноты изложения материала ниже приведен краткий тест комбинатора f or к (), наглядно демонстрирующий еще один удачный пример применения функции R. identity ().

```
QUnit.test('Functional Combinator: fork', function (assert) {
    const timesTwo = fork((x) => x + x, R.identity, R.identity)
assert.equal(timesTwo(1),      2) ;
assert.equal(timesTwo(2), });  4) ;
```

Следует еще раз подчеркнуть, что комбинаторы достаточно протестировать с помощью простой функции, поскольку они совершенно не зависят от предоставляемых аргументов. Благодаря применению функциональных библиотек, композиции и комбинаторов функций разработка и тестирование программ заметно упрощается. Но дело может существенно усложниться из-за нечистого поведения.

6.3.3. Отделение чистого кода от нечистого путем монадического изолирования

Как пояснялось в предыдущих главах, большинство программ состоит из чистых и нечистых частей. И это особенно справедливо для клиентских приложений HaJavaScript, поскольку этот язык предназначен для организации взаимодействия с моделью DOM. Требования для серверных приложений несколько отличаются и включают в себя, например, чтение из базы данных или файла. Ранее пояснялось, как пользоваться композицией для объединения чистых и нечистых функций, образующих программы. Но и после этого они остаются нечистыми. Чтобы добиться большей их чистоты в отношении ссылочной прозрачности, декларативности и простоты понимания в приведенных ранее примерах применялась монада типа 10. Помимо монады типа 10, для обеспечения безопасного способа выполнения программ, по-прежнему реагирующих на сбойное событие, в этих примерах применялись также монады типа Maybe и Either. Все эти методики и средства ФП позволяют контролировать большинство побочных эффектов. Но если в прикладном коде JavaScript потребуется прочитать и записать данные в модели DOM, то как гарантировать, что тесты останутся по-прежнему изолированными и повторяемыми?

Напомним, что в императивной версии программы showStudent не предпринимается никаких попыток отделить ее нечистые части. Все они перемешаны, и поэтому она будет выполняться как единое целое в каждом очередном тесте. Но ведь это совершенно

неэффективно и непродуктивно, поскольку программу приходится каждый раз выполнять полностью, даже если требуется проверить, например, что вызов функции db.get (ssn) правильно выполняется с разными сочетаниями номеров социального страхования. Еще один недостаток такого подхода заключается в том, что программу нельзя тщательно протестировать, поскольку все ее операторы тесно связаны. Например, первый блок кода может завершиться преждевременным выходом из функции с генерацией исключения, что не даст возможность протестировать вызов функции db. get (ssn) с недостоверными входными данными.

С другой стороны, функциональное программирование нацелено на сведение операций (например, ввода-вывода), приводящих к побочным эффектам, к простым функциям (например, чтения и записи). Это дает возможность расширить рамки тестирования логики приложения, ясно разграничив свою и чужую ответственность за тестирование операций ввода-вывода. Вернемся снова к функциональной версии программы showStudent:

```
const showStudent = R.compose(
   map(append('#student-info') ),
   liftIO,
   map(csv),
   map(R.props(['ssn', 'firstname', 'lastname'])),
   chain(findStudent),
   chain(checkLengthSsn), lift(cleaninput));
```

Внимательно проанализировав обе версии данной программы, можно обнаружить, что ее императивная версия разделяется на части и затем соединяется в функциональной ее версии с помощью композиции и монад. В конечном итоге это дает возможность существенно расширить рамки тестирования данной программы, ясно понять и отделить чистые функции от нечистых (рис. 6.6).

Проанализируем тестируемость отдельных компонентов программы showStudent. Из пяти составляющих ее функций лишь три могут быть протестированы надежно: cleaninput(), checkLengthSsn () и csv(). И хотя функция findStudent () вызывает побочные эффекты при чтении данных из внешних ресурсов, далее будут показаны способы обойти данное препятствие. Оставшаяся функция append () не содержит никакой бизнес-логики, поскольку все ее обязанности сведены к простому добавлению в модель DOM передаваемых данных. Тестировать интерфейсы DOM API малоинтересно да и не нужно, поскольку бремя этой ответственности можно возложить на производителей браузеров. Таким образом, пользуясь средствами ФП, можно взять с трудом

поддающуюся тестированию программу и разделить ее на вполне поддающиеся тестированию части.

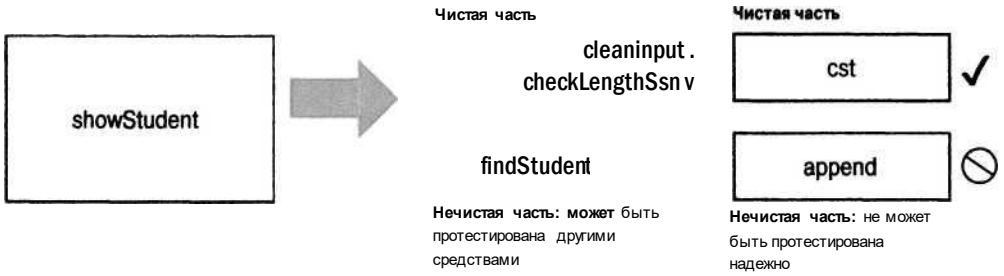


Рис. 6.6. Выяснение тестируемых участков в программе на основе функции showStudent (). Те ее компоненты, которые выполняют операции ввода-вывода, не могут быть протестированы надежно, поскольку они содержат побочные эффекты. Если не принимать во внимание нечистые части программы, то вся она вполне поддается тестированию

А теперь сравним этот код с императивным, тесно связанным кодом из листинга 6.2. В функциональной версии можно надежно протестировать около 90% рассматриваемой здесь программы, тогда как императивную ее версию постигнет та же участь, что и упоминавшуюся ранее функцию increment (), — ее тестирование завершится неудачно при последующих или выполняемых не по порядку проходах.

Листинг 6.2. Модульное тестирование чистых компонентов программы showstudent

```
QUnit.test('showStudent: cleaninput', function (assert) {

    const input =       '-44-44-', '44444', ' 4 ', ' 4-4
    const assertions = ['', '4444', '44444', '4', '44'];

    assert.expect(input.length);
    input.forEach(function (val, key) {
        assert.equal(cleaninput(val), assertions[key]);
    }) ;


QUnit.test('showStudent: checkLengthSsn', function (assert) {

    assert.ok(checkLengthSsn('444444444*).isRight);
    assert.ok(checkLengthSsn('').isLeft);
    assert.ok(checkLengthSsn (' 44444444 ' ) .isLeft) ;

    assert.equal(checkLengthSsn('444444444').chain(R.length), 9);
});

QUnit. test ('showStudent: csv', function (assert) {
    assert.equal(csv(['Alonzo']), 'Alonzo');
    assert.equal(csv(['Alonzo', 'Church']), 'Alonzo,Church');
    assert.equal(csv(['Alonzo', 'Church'])., 'Alonzo,,Church,'); });
```

Используем входные данные переменной длины с пробелами

Вызвать метод Either.isLeft() или Either .isRight O, чтобы проверить утверждения о содержимом монад `assert.equal(csv(['']), '');`

Все функции в данной версии программы обособлены и могут быть тщательно проверены по отдельности (далее будет продемонстрирован автоматизированный механизм

формирования входных данных). Поэтому их можно благополучно реорганизовать, не боясь что-нибудь нарушить в других местах программы.

Осталось лишь протестировать последнюю функцию f indStudent (), в основу которой положена нечистая функция safeFindObject (), запрашивающая поиск записей об учащихся во внешнем хранилище объектов. Но побочные эффекты в данной функции поддаются контролю с помощью методики, называемой *имитирующими объектами (mock objects)*.

6.3.4. Имитация внешних зависимостей

Имитация (mocking) является весьма распространенной методикой тестирования, предназначенной для моделирования поведения внешних зависимостей функции под полным контролем и путем утверждений, и поэтому она вполне пригодна для обращения с некоторыми видами побочных эффектов. Имитирующие объекты приведут к непрохождению теста, если утверждаемые в нем ожидания не оправдаются. Они подобны программируемым пустым методам (или заглушкам), с помощью которых можно заранее определить ожидаемое поведение объекта, взаимодействующего с функциями. В данном случае имитация обращения к объекту типа DB дает полный контроль над этим внешним ресурсом, чтобы создавать более предсказуемые и согласованные тесты. Для решения этой задачи будет использован модуль имитации Sinon JS, подключаемый к библиотеке QUnit (подробнее о том, как его устанавливать, см. в приложении).

Подключаемый модуль SinonJS дополняет среду тестирования объектом sinon, применяемым для создания имитирующих версий любого объекта, доступных в имитационном контексте. В данном случае этот контекст заполняется объектом типа DB, который послужит в качестве действующей заглушки для подобной зависимости:

```
const studentstore = DB('students'); const mockContext = sinon.mock(studentstore);
```

Используя этот имитационный контекст, можно установить многие ожидания разного поведения имитируемого объекта, чтобы, в частности, проверить, сколько раз он вызывается, какие аргументы ему передаются и каким должно быть возвращаемое им значение. Чтобы проверить поведение монады типа Either, в которую заключается значение, возвращаемое функцией safeFindObject (), создадим два модульных теста: один — для проверки структуры типа Either. Right, другой — для инициирования структуры типа Either.Left. Воспользуемся для этого каррированным характером функции

findStudent (), позволяющим легко внедрять любую реализацию хранилища, в котором можно искать информацию, подобно тому, как это делалось в главе 4, "На пути к повторно используемому, модульному коду" с помощью шаблона "Фабричный метод". Как демонстрировалось в приведенных ранее листингах, эта функция вызывает метод get () для объекта хранилища. Имея полный контроль над этим объектом через имитационный контекст, можно без особого труда контролировать возврат желаемого значения, как показано в листинге 6.3.

Листинг 6.3. Имитация внешней зависимости функции findStudent ()

```
var studentstore, mockContext;

QUnit.module('CH06', {                    | Подготовить имитационный контекст
    beforeEach: functionO {       -----   | для всех модульных тестов
      studentDb = DB('students'); mockContext =
      sinon.mock(studentDb); Очистить после y          каждого теста
      afterEach: functionO {      -----  --  ---  -
        mockContext. verify () ;  ---  ---  Проверить утверждения, установленные
            ,2,.     .    .    „.                в имитационной конфигурации
        mockContext.restore();
      }
    })'

    QUnit. test (showStudent: findStudent returning null',
      function (assert) {

        mockContext.expects (' get ') . once () . returns (null); ■<

        const findStudent = safefetchRecord(studentstore);

        assert.ok(findStudent('xxx-xx-xxxx').isLeft); });

    QUnit.test('showStudent: findStudent returning valid
      object', function (assert) {

        mockContext.expects('get').once().returns( ◄------
            new Student('Alonzo', 'Church', 'Princeton').
                setSsn('444-44-4444'));

        const findStudent = safefetchRecord(studentstore); Утверждение достовер-
        assert .ok (findStudent (' 444-44-4444 ') .isRight) ;
```

В первом модульном тесте имитирующий объект моделирует (только раз) вызов своего метода запроса, возвращающего пустое

Утверждение возвращаемого значения заключается в оболочку структуры Either.Left

Во втором модульном тесте имитирующий объект моделирует вызов своего метода запроса с достоверным результатом

. ~.ного ответа заключается

структуры Either.Left

На рис. 6.7 представлены результаты выполнения приведенных выше тестов средствами QUnit и Sinon.JS для проверки тестируемых частей программы на основе функции showStudent ().

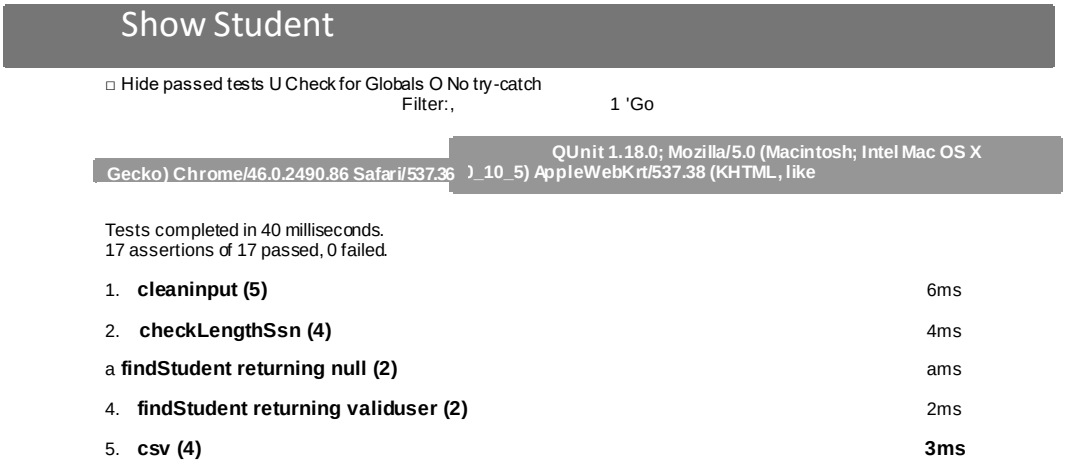


Рис. 6.7. Результаты выполнения всех модульных тестов программы showStudent. Тесты 3 и 4 проведены средствами QUnit и Sinon JS, и поэтому для них требуются имитируемые зависимости, чтобы смоделировать функциональные возможности для извлечения записи об учащемся

То обстоятельство, что функциональный код поддается тестированию в гораздо большей степени, чем императивный код, может быть сведено к единственному принципу ссылочной прозрачности. Поэтому сущность утверждения состоит в проверке, всегда ли соблюдается эта ссылочная прозрачность: `assert.equal(computeAverageGrade([80, 90, 100]), 'A');`

Но ссылочная прозрачность оказывается намного более широким принципом, чем кажется на первый взгляд. Этот принцип может быть распространен и на другие области разработки программного обеспечения, включая и спецификации программ. Ведь в конечном счете тесты предназначены лишь для того, чтобы проверять систему на соответствие ее спецификациям.

6.4. Фиксация спецификаций при тестировании на основе свойств

Модульные тесты могут использоваться как средства для документирования и фиксации спецификации функции в среде ее выполнения. Так, с помощью следующего теста функции computeAverageGrade ():

```
QUnit.test('Compute Average Grade', function (assert) {
    assert.equal(computeAverageGrade([80, 90, 100]),'A');
    assert.equal(computeAverageGrade([80, 85, 89]), 'B');
    assert.equal(computeAverageGrade([70, 75, 79]), 'C');
    assert.equal(computeAverageGrade([60, 65, 69]), 'D');
    assert.equal(computeAverageGrade([50, 55, 59]), *F');
    assert.equal(computeAverageGrade([-10]), 'F'); });
```

можно получить простой документ, в котором утверждается следующее:

- если средняя оценка учащего не меньше 90 баллов, ему присваивается степень A;
- если средняя оценка учащего в пределах от 80 до 89 баллов, ему присваивается степень B, и т.д.

Естественный язык нередко используется как средство для фиксации требований к системе, но естественные языки выражают назначение в определенном контексте, зачастую известном не всем сторонам, а это приводит к неопределенности при попытке воплотить требования в коде. Именно поэтому приходится постоянно требовать от владельцев продуктов или руководителей команд разработчиков прояснить неоднозначности, присутствующие в спецификациях решаемых задач. Одной из главных причин неоднозначности служит результат принятия императивного стиля документирования с применением вариантов выбора вроде следующего: если выбран вариант A, то система должна выполнить операцию B. Недостаток такого подхода заключается в том, что он не описывает задачу в целом с учетом всех граничных условий. Что, если выбор варианта A так и не произойдет? Каких действий в таком случае следует ожидать от системы?

Удачные спецификации не должны зависеть от вариантов выбора. Они должны быть обобщенными и универсальными. Проанализируем незначительные отличия в формулировках следующих двух утверждений:

- если средняя оценка учащего не меньше 90 баллов, ему присваивается степень A;
- только при средней оценке не менее 90 баллов учащему будет присвоена степень A.

Второе утверждение выглядит намного более полным, поскольку в нем устранены императивные фразы, допускающие варианты выбора. Это дает возможность не только выразить, что же произойдет, если средняя оценка учащегося достигнет 90 баллов или выше, но и наложить ограничение на то, что ни один другой числовой диапазон не приведет к присваиванию степени A. Из второго утверждения можно сделать вывод, что, по крайней мере, любая другая вычисляемая оценка не приведет к присваиванию степени A. Но такой вывод нельзя сделать интуитивно из первого утверждения.

Работать с универсальными требованиями намного проще, поскольку они не зависят от состояния системы в любой момент времени. Именно поэтому удачные спецификации не вызывают, как и модульные тесты, побочные эффекты и предположения относительно окружающего их контекста.

Ссылочно-прозрачные спецификации упрощают понимание назначения функций и дают ясное представление о входных условиях, которым они должны удовлетворять. А поскольку ссылочно-прозрачные функции единообразны и имеют ясно определенные входные параметры, то они легко поддаются тестированию с помощью автоматизированных механизмов, способных дойти до предела их возможностей. И это приводит к более привлекательной методике тестирования, называемой *тестированием на основе свойств*. В тесте на основе свойств делается утверждение о том, каким должен быть результат выполнения функции при определенных входных данных. Канонической или эталонной формой реализации подобной методики служит библиотека QuickCheck языка Haskell.

Библиотека QuickCheck языка Haskell — воплощение методики тестирования на основе свойств

Библиотека QuickCheck языка Haskell служит для рандомизированного тестирования на основе свойств конкретной спецификации или свойств программы. В этом случае спецификация чистой программы проектируется в форме свойств, которым должна удовлетворять данная программа. А библиотека QuickCheck формирует крупное сочетание контрольных примеров для программы и выдает соответствующий отчет. Подробнее об этой библиотеке см. по адресу https: // hackage.haskell.org/package/QuickCheck.

Возможности библиотеки QuickCheck эмулируются BjavaScript средствами библиотеки JSCheck (подробнее о том, как она устанавливается, см. в приложении), разработанной Дугласом Крокфордом — автором книги *JavaScript: The Good Parts* (издательство O'Reilly, 2008 г.; в русском переводе книга вышла под названием *JavaScript: Сильные стороны* в издательстве "Питер", 2012 г.) [VIII]. Средствами библиотеки JSCheck можно составить формальный ответ на соответствие ссылочно-прозрачной спецификации функции или программы. Следовательно, для подтверждения свойств функции формируется большое количество произвольных контрольных примеров, предназначенных для строгой проверки всех возможных путей выполнения функции.

Кроме того, тесты на основе свойств позволяют контролировать и вести развитие программы по мере реорганизации ее исходного кода, чтобы не внести неумышленно программные ошибки в систему вместе с новым кодом. Главное преимущество, которое дает применение инструментальных средств вроде JSCheck, заключается в их алгоритмах, формирующих отклоняющиеся от нормы массивы данных для применения в тестах. Ряд крайних случаев, которые воспроизводит библиотекаJSCheck, чаще всего упускаются из виду, если пытаться воссоздавать их вручную.

Модуль JSCheck изящно инкапсулируется в глобальный объект JSC следующим образом:

```
JSC.claim(name, predicate, specifiers, classifier)
```

В основу этой библиотеки положено создание *заявок* и *вердиктов*. Заявка состоит из следующих частей.

Как следует из листинга 6.4, для фиксации свойств данной программы можно воспользоваться следующими декларативными спецификаторами.

■ **JSC. array ().** Обозначает, что функция ожидает входные данные типа Array.

- **Имя.** Описание заявки (аналогично описанию теста в QUnit).
- **Предикат.** Функция, возвращающая истинный (true) вердикт, если заявка удовлетворена, а иначе — ложный (false) вердикт.
- **Спецификаторы.** Массив, описывающий тип входных параметров и спецификацию, с помощью которой формируются произвольные массивы данных.
- **Необязательный классификатор.** Функция, связанная с каждым контрольным примером и предназначенная для отклонения неприменимых контрольных примеров.

Заявки передаются функции JSCheck. check () для выполнения произвольных контрольных примеров. Создаваемая в этой библиотеке заявка заключается в оболочку и передается интерпретатору в одном вызове функции JSCheck. test (). Именно этот сокращенный способ будет использован в последующих примерах тестов. Так, в листинге 6.4 приведен пример написания в JSCheck простой спецификации для функции computeAverageGrade (), где утверждается следующее: "Только при средней оценке не меньше 90 баллов учащему будет присвоена степень А".

Листинг 6.4. Тестирование функции computeAverageGrade () на основе свойств

```
                                    Чтобы заново инициализировать контекст тестирования, лучше начать с вызова
JSC.clearO

                                        => console.log(str));


        >► JSC.array(JSC.integer(20), JSC.number(90,100)),
            'A'
    JSC.clearO; ◄-----  ], function
    JSC.on_report((str)  (grades,        Функция классификатора, выполняемая в каждом
                                         тесте. С ее помощью можно присоединить данные
                                         — к отчету

        JSC.test(                                  Передать предикатной функции
                                                   объект вердикта, где определяется
Имя заявки ┌ 'Compute Average Grade', function     — проверяемое условие
            (verdict, grades, grade)
                return verdict(computeAverageGrade(grades) === grade);
            grade) {
                return 'Testing for an ' + grade + ' on grades: ' + grades; }
```

Массив сигнатур или спецификаторов, описывающий контракт на формирование средних оценок, заслуживающих степени А

- JSC. integer (20). Обозначает, что для данной функции ожидается максимальная длина. В данном случае она произвольна, и поэтому подойдет любое число от 1 до 20.
- JSC. number (90, 100). Описывает типы элементов во входном массиве. В данном случае они числовые (как целочисленные, так и с плавающей запятой) в пределах от 90 до 100.

Понять предикатную функцию немного сложнее. Она возвращает истинный (true) вердикт, если заявка удовлетворяется. Но все, что происходит в теле предикатной функции, определяется разработчиком в зависимости от конкретной программы и того, что требуется проверить. Помимо функции вердикта, применяемой для объявления результата выполнения контрольного примера, разработчику предоставляются произвольно

сформированные и ожидаемые выходные данные. В данном случае требуется объявить результат проверки, возвращает ли функция computeAverageGrade () ожидаемую степень А. Здесь рассмотрено лишь несколько спецификаторов, а об остальных можно прочитать на веб-сайте по адресу https: //github. com/douglascrockford/JSCheck или же создать собственный спецификатор.

Итак, пояснив основные части рассматриваемой здесь программы, перейдем к ее выполнению. Формируемый ею отчет может оказаться длинным, поскольку библиотека JSCheck по умолчанию генерирует 100 произвольных контрольных примеров на основании предоставленной спецификации. Поэтому в приведенном ниже примере этот отчет усечен, но так, чтобы из него было ясно, что происходит на самом деле.

```
Compute Average Grade: 100 classifications,
        100 cases tested, 100 pass
(Вычислить среднюю оценку: 100 классификаций, 100 проверенных контрольных
примеров, 100 пройдено)

Testing for an A on grades:
(Проверка оценок на соответствие степени А)
    90.042,98.828,99.359,90.309,99.175,95.569,97.101,92.24
    pass 1
    (пройдено 1)
Testing for an A on grades: 90.084,93.199, pass 1 // and so on 98 more times //
(и т.д. еще 98 раз)

Total pass 100, fail 0
(Итого: пройдено 100, не пройдено 0)
```

Программы JSCheck являются самодокументирующимися. В них можно без особого труда описать контракт на входные и выходные данные функции вплоть до такого уровня, на котором этого нельзя сделать в модульных тестах. А в отчете, формируемом средствами JSCheck, наблюдается значительный уровень детализации. Кроме того, программы JSCheck можно выполнять как самостоятельные сценарии или встраивать их в модульные тесты QUnit. В последнем случае их можно включать в тестовые наборы. Порядок взаимодействия этих библиотек наглядно показан на рис. 6.8.

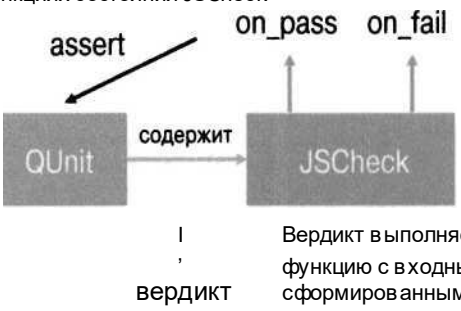


Рис. 6.8. Интеграция главных компонентов JSCheck и QUnit для тестирования на основе свойств. Тест QUnit инкапсулирует спецификацию теста JSCheck. Спецификация и тестируемая функция предоставляются функции verdict (), которая интерпретируется в JSCheck для выполнения обратных вызовов, когда тесты проходят или не проходят. Эти обратные вызовы могут быть использованы для инициирования утверждений в модульных тестах QUnit

В следующем примере мы воспользуемся библиотекой JSCheck для тестирования программы checkLengthSsn, имеющей перечисленную ниже спецификацию.

- Достоверный номер социального страхования может удовлетворять следующим условиям.
  - Не содержит пробелы.
  - Не содержит тире.
  - Имеет длину девять символов.
  - Соответствует формату, описанному по адресу https:/ /www. ssa. gov, и состоит из трех частей.

1. Первая группа из трех цифр, обозначающих номер региона.
2. Вторая группа из двух цифр, обозначающих групповой номер.
3. И последняя группа из четырех цифр, обозначающих серийный номер.
4. Исходный код данной программы приведен листинге 6.5, а отдельные ее части поясняются ниже.

Листинг 6.5. Тест JSCheck программы checkLengthSsn

```
QUnit.test('JSCheck Custom Specifier for SSN', function (assert) { JSC.clear();

    JSC.on_report((report) => trace('Report'+ str));
    JSC.on_pass((object) => assert.ok(object.pass));
                                    I Вызвать метод JSC. on_f ail (), чтобы убедиться в том, что
    JSC.onfail ( (object) => ◄----- * при длине аргументов не равной 9, тест не проходит
        assert.ok(object.pass I I object.args.length === 9,
                  'Test failed for: ' + object.args));

    JSC.test(                                  Данная функция относится к логиче-
        'Check Length SSN', function           скому типу, и поэтому результат про-
        (verdict, ssn) {                       верки можно передать для вердикта
            return verdict(checkLengthSsn(ssn));


            JSC.SSN(JSC.integer(100, 999), JSC.integer(10, 99),
            JSC.integer(1000,9999))          Воспользоваться специальным спецификатором
        ], function (ssn) {                  JSC. SSN (), определенным в тексте и состоящим из
            return 'Testing Custom SSN:      спецификаторов JSC. integer (). Каждый
                                             спецификатор JSC. integer () выби- ssn'  рвет
                                             произвольное число в указанном
                                                        диапазоне
```

В данной программе функциональные возможности библиотек JSCheck и QUnit соединяются через вызов функций JSC. on_fail () и JSC. on pass (), которые сообщают библиотеке QUnit о любых утверждениях, в которых выполняется или нарушается предоставляемая спецификация. В следующем спецификаторе:

```
JSC.SSN(JSC.integer(100, 999), JSC.integer(10, 99),
        JSC.integer(1000,9999))
```

описывается контракт на достоверные номера социального страхования, и поэтому в данной программе предполагается всегда выводить правильные результаты для любого сочетания номера социального страхования в форме `XXX-XX- XXXX`, как показано ниже.

```
Check Length SSN:
100 classifications,  100 cases tested, 100 pass

Testing Custom SSN: 121-76-4808 pass 1
Testing Custom SSN: 122-87-7833 pass 1
Testing Custom SSN: 134-44-6044 pass 1
Testing Custom SSN: 139-47-6224 pass 1


Testing Custom SSN: 992-52-3288 pass 1
Testing Custom SSN: 995-12-1487 pass 1
Testing Custom SSN: 998-46-2523 pass 1

Total pass 100
```

В приведенном выше отчете нет ничего удивительно. Но спецификацию можно откорректировать таким образом, чтобы включить в нее недостоверные входные данные с групповым номером из трех цифр и посмотреть, как при этом поведет себя программа:

```
JSC.SSN(JSC.integer(100, 999), JSC.integer(10, 999),
        JSC.integer(1000,9999))
```

Тест QUnit с признаками JSCheck не проходит, как и предполагалось. На рис. 6.9 ради краткости приведен результат одного непрохождения теста.

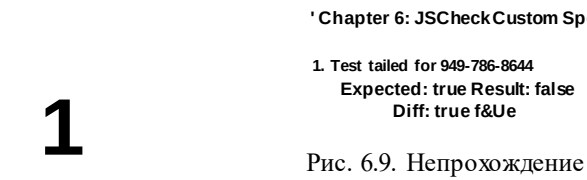


Рис. 6.9. Непрохождение теста обнаруживается в результате неверной проверки свойств средствами QUnit. Если выполнить рандомизацию входных данных, чтобы включить в них недостоверные данные, то алгоритм JSCheck обладает достаточной энтропией, чтобы 89 тестов из 90 не прошли

Откуда же берется спецификатор JSC. SSN () ? Спецификаторы JSCheck ведут себя как комбинаторы функций, которые могут быть составлены для создания более конкретизированных спецификаторов. В данном случае используется специальный спецификатор JSC. SSN (), состоящий из определенного сочетания спецификаторов JSC. integer (), описывающих свойства каждой группы цифр в номере социального страхования (см. листинг 6.6).

Листинг 6.6. специальный спецификатор JSC . SSN ()

```
* Формирует символьную строку с достоверным номером
* социального страхования (с тире)
* @param paraml Area Number -> JSC.integer(100, 999)
* @param param2 Group Number -> JSC.integer(10, 99)
* @param parат3 Serial Number -> JSC.integer(1000,9999)
* ^returns {Функция} функция спецификатора
*/
JSC.SSN = function (paraml, param2, parат3) { >4 -------
      return function generator() { const
    parti = typeof paraml === ? paraml(): 'function'
                                paraml;

     const part2 = typeof param2 ===      'function'
          ? param2(): param2;

        const part3 = typeof parат3 === 'function'
              ? parат3(): parат3;
```

I Добавить как часть объекта JSC, чтобы сделать код согласованным

Каждая часть номера социального страхования состоит из константы или функции, используемой в JSCheck для внедрения произвольных входных данных

```
        return [parti , part2, part3].join◄----------
    };
};
```

Все три точки ввода данных объединяются в достоверный синтаксис номера социального страхования

БиблиотекаJSCheck оперирует лишь чистыми программами, а это означает, что программу showStudent нельзя проверить полностью. Но в то же время этой библиотекой можно воспользоваться для тестирования каждого компонента данной программы в отдельности, что оставляется вам для самостоятельного упражнения. Тестирование на основе свойств привлекательно тем, что оно позволяет проверять функции до предела. Его наилучшим качеством является возможность проверить, действительно ли тестируемый код является ссылочно-прозрачным. Ведь при этом предполагается, что он должен действовать согласованно по тому же самому контракту и вердикту. Но зачем подвергать прикладной код столь суровому испытанию? Ответ прост: это необходимо для того, чтобы сделать тесты эффективными.

6.5. Количественная оценка эффективности тестов через покрытие ими кода

Количественная оценка эффективности модульного теста — непростая задача, если она не решается с помощью подходящих инструментальных средств, поскольку она включает в себя изучение покрытия проверяемого кода тестами через тестируемые функции. Для получения сведений о покрытии проверяемого кода тестами требуется обойти все особые пути, относящиеся к потоку управления программой. И этого можно, в частности, достичь, изучая поток выполнения кода по граничным условиям функции.

Очевидно, что одно лишь покрытие тестами не является достаточным показателем качества, хотя оно и описывает степень, до которой можно протестировать функции, что несомненно связано с улучшением качества. Кому нужен код, который так и не дошел до стадии промышленной эксплуатации? Я думаю, что никому!

В ходе анализа покрытия проверяемого кода тестами можно обнаружить в нем участки, которые не были протестированы, что дает возможность создавать дополнительные тесты для их выявления. Как правило, это код для обработки ошибок, который был в свое время проигнорирован и затем забыт. Покрытие кода тестами позволяет количественно оценить долю в процентах тех строк кода, которые выполняются при вызове программы через модульные тесты. Чтобы получить такую информацию, можно воспользоваться библиотекой Blanket.js в качестве инструментального средства для покрытия тестами проверяемого кода JavaScript. Она служит для того, чтобы дополнить существующие модульные тесты JavaScript статистическими данными о покрытии проверяемого кода этими тестами. В процессе работы эта библиотека проходит три стадии, как поясняется ниже.

1. Загрузка исходных файлов.
2. Оснащение проверяемого кода строками отслеживания.
3. Соединение перехватчиков в исполнителе тестов с выводимыми подробностями о покрытии тестами.

Библиотека Blanket.js собирает сведения на второй стадии оснащения, в течение которой она фиксирует метаданные, касающиеся выполнения операторов. Эти данные можно изящно отобразить в отчете QUnit. Подробнее о том, как устанавливать библиотеку Blanket.js, см. в приложении. Любой модуль или программу JavaScript можно снабдить специальным атрибутом data-covered в строке сценария include. Анализируя долю в процентах покрытия отдельных операторов тестами, можно обнаружить, что функциональный код в большей степени поддается тестированию, чем императивный.

6.5.1. Количественная оценка эффективности тестирования функционального кода

В этой главе представлены примеры функциональных программ, которые в большей степени поддаются тестированию благодаря той простоте, с которой задачи могут быть разделены на атомарные, проверяемые единицы. Но, не беря эти слова на веру, попробуем количественно оценить эффективность тестирования программы showStudent, произведя анализ покрытия тестами каждого ее оператора. Рассмотрим сначала положительный тест как простейший контрольный пример.

Количественная оценка эффективности императивного и функционального кода при достоверных входных данных

Итак, проанализируем статистические данные покрытия проверяемого кода тестами при удачном выполнении императивной версии программы showStudent, исходный код которой представлен в листинге 6.2. Оснастим эту программу средствами Blanket и QUnit, пометив ее следующим образом:

```
<script src="imperative-show-student-program.j s'' data-cover></script>
```

Если теперь выполнить приведенный ниже тест, то в конечном итоге будет получено 80%-ное покрытие проверяемых операторов этим тестом, как следует из результата, выводимого в QUnit/Blanket на экран и представленного на рис. 6.10.

```
QUnit.test('Imperative showStudent with valid user', function (assert) { const
result = showStudent('444-44-4444'); assert.equal(result, '444-44-4444, Alonzo,
Church'); }) ;
```

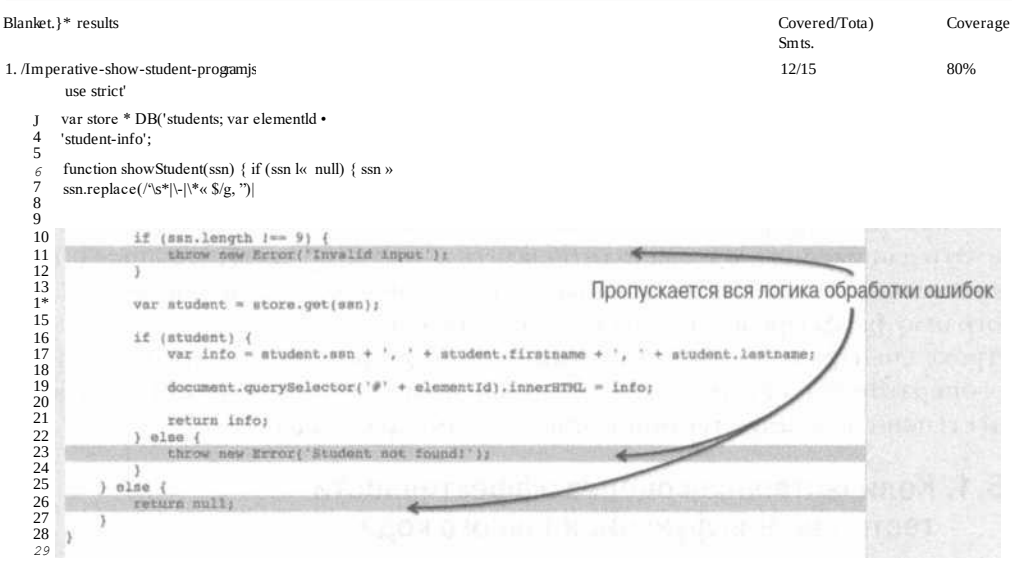


Рис. 6.10. Выводимый в QUnit/Blanket результат выполнения императивной версии программы showStudent с достоверными входными данными. В специально выделенных строках кода представлены операторы, которые вообще не были выполнены. В итоге из 15 строк кода было выполнено только 12, а следовательно, общее их покрытие тестами составило лишь 80%

Такой результат неудивителен, поскольку в данном случае был пропущен весь код обработки ошибок. Показатель 75-80%-ного покрытия проверяемого кода тестами считается довольно приличным для императивных программ. Из данного примера можно сделать вывод, что 80%-ное покрытие — это наилучший показатель, который можно получить при однократном выполнении модульного теста. А теперь оснастим средствами Blanket и QUnit функциональную версию той же самой программы, чтобы выполнить ее положительный тест:

```
<script src="functional-show-student-program.js" data-cover> </script>
```

И в этом случае тестирование программы было выполнено с достоверным номером социального страхования, т.е. по “удачному пути”. Но на этот раз покрытие тестами достигло показателя 100% (рис. 6.11)!

Рис. 6.11. В результате выполнения положительного модульного теста функциональной версии программы showStudent достигнуто 100%-ное покрытие тестами ее исходного кода. Это означает, что была выполнена буквально каждая строка кода проверяемой бизнес-логики!

Но постойте: ведь если входные данные были достоверными, то почему не был пропущен код обработки ошибок? Это результат действия монад в проверяемом функциональном коде, где понятие пустого значения или пустоты (в форме структуры Either. Left или Maybe .Nothing) плавно распространяется по всей программе. Таким образом, в данной версии программы выполняется каждая функция, хотя при этом пропускается логика, инкапсулированная в функциях преобразования.

Просто удивительно, насколько надежным и гибким оказывается функциональный код. А теперь выполним отрицательный тест с недостоверными входными данными.

Количественная оценка эффективности императивного и функционального кода при недостоверных входных данных

Оценим количественно эффективность обеих версий программы при их выполнении с недостоверными входными данными, например, когда передаются пустые (null) значения. Как следует из отчета о тестировании императивного кода, приведенного ниже и на рис. 6.12, покрытие тестами этого кода оказывается умеренным, что неудивительно.

```
QUnit.test('Imperative Show Student with null', function (assert) { const result
    = showStudent(null);
    assert.equal(result, null);
}) ;
```

Blanketje results

Ccvered/Total Smts.

Coverage (%)

i /^mperative-show student-prograrTi.ja — 40%

```
1   'use strict';
2
J var store " DB('students'; elementld • 'student-info*;

          showStudent(ssn) { if san 1- null
        ssn * ssn.replace(/*\»*|\ᵘ|\в*$/д♦ '

        if (ssn.length !ᵒᵉ 9) {
         throw new Error('invalid input *); >

        var student • store.get(ssn);

        if (student) {
        'var info • student.sen +       * + student.firstname + *ₜ * ♦ student.last name;.

             document.quer/Selector(♦ elementld).innerHTML < info;

        21    return inf
            else
        23    throw new Error('Student not found!');
24          }
25      } else {
26      return null; }
2B }
```

Пропускается вся логика об- ошибок и бизнес- логика

Рис. 6.12. При тестировании императивной версии программы showStudent пропускается по- ложительный путь ее выполнения. В итоге выполняется лишь несколько строк кода, а его по- крытие тестами составляет всего 40%

Такой результат объясняется наличием блоков условных операторов if-else, где создается поток управления, разветвляющийся в разных направлениях. И, как будет показано ниже, это приводит также к появлению сложных функций.

И напротив, в функциональной версии данной программы пустые (null) входные данные обрабатываются корректно, поскольку в этом случае пропускается лишь логика для непосредственного манипулирования недостоверными (теперь пустыми) входными данными. Но вся структура программы, включая взаимодействие функций, остается на месте и успешно вызывается и тестируется от начала и до конца. Напомним, что вследствие ошибки функциональный код выдает результат типа Nothing. Вместо того чтобы проверять выходные данные на наличие пустого значения (null), достаточно выполнить следующий тест:

```
QUnit.test('Functional Show Student with null', function (assert) {
    const result = showStudent(null).run();
    assert.ok(result.isNothing);
```

Участки, которые остались незатронутыми тестированием из-за пропущенной логики, приведены на рис. 6.13.

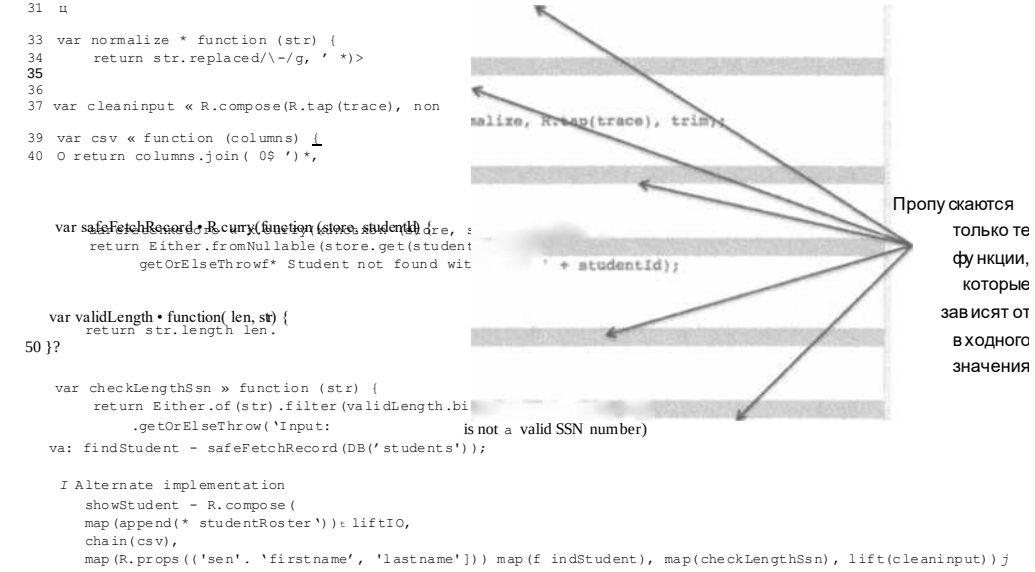


Рис. 6.13. При тестировании функциональной версии программы showStudent пропускаются
строки кода, связанные только с манипулированием входными данными, которые в противном
случае происходили бы из достоверного источника

Даже при наличии недостоверных данных целые участки кода в функциональной версии данной программы не просто пропускаются, а корректно и благополучно распространяют недостоверное условие в монадах. В итоге достигается приличный показатель 80%-ного

покрытия тестами, в два раза превышающий аналогичный показатель для императивной версии данной программы (рис. 6.14).

Blanket.js results	Covered/Total Smts.	Coverage (%)
1. /functkxial-show-student-prugram.js	23/29	79.31 %

Рис. 6.14. Показатель покрытия тестами функциональной версии программы showStudent по-прежнему остается высоким даже при недостоверных входных данных

Благодаря тому что функциональный код в намного большей степени поддается тестированию, он должен вызывать ощущение безопасности и удобства его развертывания в производственных системах — даже если неизменяемость и устранение побочных эффектов не достигли своей цели. Как упоминалось ранее, наличие блоков условных операторов и циклов в императивном коде не только затрудняет его тестирование и понимание, но и дополнительно увеличивает сложность рассматриваемой функции. Как же оценить сложность кода количественно?

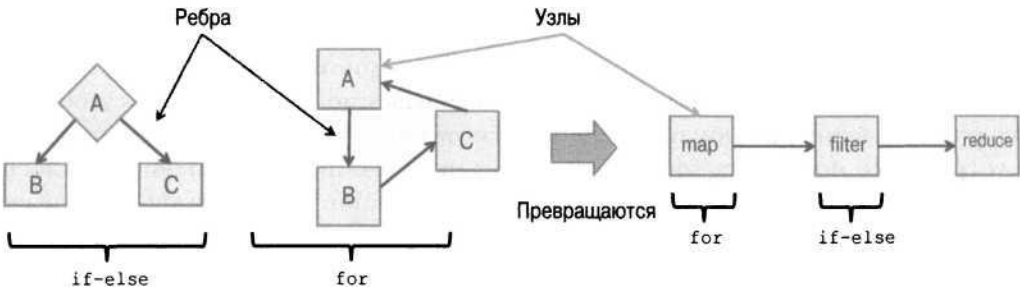
6.5.2. Количественная оценка сложности функционального кода

Чтобы количественно оценить сложность программы, необходимо тщательно исследовать ее поток управления. На первый взгляд, для этого достаточно определить, что блок кода сложен, если его трудно проследить визуально. В этом отношении функциональное программирование дает ясное декларативное представление о коде, которое делает его визуально привлекательным. Это равнозначно уменьшению сложности кода с точки зрения разработчика. В этом разделе будет показано, что функциональный код оказывается менее сложным и с алгоритмической точки зрения.

Сложность кода определяется многими факторами, включая блоки условных операторов и циклы, которые могут быть также вложенными в другие структуры. Например, логика условных переходов является взаимно исключающей, разделяя поток управления программой на две независимые ветви в соответствии с логическим условием. Если в прикладном коде имеется немало блоков условных операторов if-else, то логику такого кода трудно проследить. Но сделать это еще труднее, когда условия основываются на внешних факторах — побочных эффектах, навязывающих путь, по которому должно пойти выполнение кода. Чем больше количество обычных и вложенных блоков условных операторов, тем труднее тестировать функции. Именно поэтому так важно сохранять функции как можно более простыми. Это положение глубоко укоренено в концепцию ФП — нужно сводить все функции к простым лямбда- выражениям везде, где это только возможно, объединяя их с помощью композиции и монад.

Цикломатическая сложность—это количественный показатель программного обеспечения, применяемый для оценки количества линейно независимых путей, которыми может пойти выполнение функции. Из этого понятия возникает идея проверять граничные условия функции, чтобы протестировать все возможные пути ее выполнения. Для этой цели служит простая теория графов, состоящих из узлов и ребер, как показано на рис. 6.15, где

- узлы обозначают неразделяемые блоки кода;



- направленные ребра соединяют два блока кода, если второй блок может быть выполнен после первого.

В главе 3, "Меньше структур данных и больше операций", исследовалось отличие графа потока управления императивным кодом от аналогичного графа для функционального кода, а также пояснялось, каким образом в функциональном коде вся логика итерации и условных переходов передается операциям высшего порядка вроде tap и filter.

Что же входит в понятие цикломатической сложности? С математической точки зрения сложность любой программы может быть определена по формуле *M*= *E*- N+ P, где

- *E*— количество ребер в потоке;
- *N*— количество узлов или блоков;

Императивный код ----------Функциональный код----------

Рис. 6.15. Императивные блоки условных операторов if-else и циклы for в императивном коде превращаются в операции map, filter и reduce в функциональном коде

- *P*— количество узлов, имеющих точки выхода.

Все управляющие структуры вносят свой вклад в цикломатическую сложность, и чем меньше ее величина, тем лучше. В частности, блок условных операторов оказывает влияние на сложность кода, главным образом, потому, что он разветвляет поток управления программой на два линейно независимых пути. И естественно, что чем больше подобных управляющих структур в про

грамме, тем выше количественный показатель цикломатической сложности, а следовательно, тем труднее тестировать программу.

Вернемся к потоку управления императивной версией программы на основе функции showStudent (). Чтобы упростить обозначение такого потока, операторы превращены в узлы графа, приведенного на рис. 6.16. Если обозначить математически этот граф, состоящий из 10 узлов, 11 ребер и 3 точек выхода, то в конечном итоге получится следующая формула: $M = E$- N+ P= 11-10 + 3 = 4.

```
function showStudent(ssn) {
A if(ssn !== null) {
    ssn = ssn.replace(/^\s*I\-I\s*$/g, '');
  C if(ssn.length !== 9) {
        throw new Error('Invalid input'); D }
    var student = db.get(ssn); E
    if (student !== null) { F
       '${student.ssn},
          ${student.fir stname} ,
          ${student.las tname}';
    document.queryselector('\#${elementld}
G .innerHTML = info;
   return info;
    }
    else {
        throw new Error(1 Student not H found!');
    }
  }
  else {                    I
      return null;
```

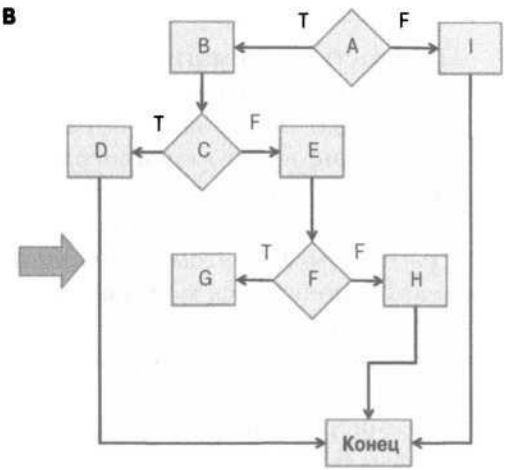


Рис. 6.16. Потенциальные узлы в императивной версии программы showStudent. Указанные метки были превращены в граф, состоящий из узлов и ребер и наглядно показывающий количество различных линейно независимых путей выполнения кода, обусловленных наличием
условных операторов

С другой стороны, оценить количественно цикломатическую сложность в функциональных программах намного проще, потому что стиль ФП предполагает исключение в как можно большей степени циклов, условных операторов и других управляющих структур в пользу функций высшего порядка, комбинаторов функций и прочих абстракций. И все это приводит к сокращению узлов и ребер графа, а также всех линейно независимых путей выполнения функций. Таким образом, функциональные программы обычно стремятся к показателю цикломатической сложности, близкому к 1. Именно это и происходит с функциональной версией программы showStudent, поскольку она состоит из многих функций, графы которых содержат лишь единственную точку выхода, но не содержат узлов и ребер, что в конечном итоге приводит к следующему показателю цикломатической сложности: M= E-N+ P= 0 - 0 + 1 = 1. Но что касается сложности, то из обеих версий рассматриваемой здесь программы можно вывести ряд других количественных показателей, заслуживающих упоминания и перечисленных в табл. 6.1.

Таблица 6.1. Другие важные статические количественные показатели сложности кода в сравнении с его императивной и функциональной версиями

Императивная версия	Функциональная версия
Цикломатическая сложность: 4 Плотность цикломатической сложности: 29%	Цикломатическая сложность: 1 Плотность цикломатической сложности: 3%
Индекс эксплуатационной надежности: 100	Индекс эксплуатационной надежности: 148

Плотность цикломатической сложности иначе выражает первоначальную величину цикломатической сложности в процентах, исходя из количества строк императивного кода, которых значительно меньше в функциональном коде. Степень, до которой программа поддается тестированию, прямо пропорциональна качеству разработки программы. Проще говоря, чем более модульным оказывается прикладной код, тем легче его тестировать. Функциональные программы имеют в этом отношении заметное преимущество, поскольку в них выгодно используется модульность ее составных единиц, которые сами являются функциями.

В функциональное программирование глубоко укоренено исключение организуемых вручную циклов в пользу функций высшего порядка, композиции вместо императивного стиля программирования, последовательного вычисления кода и более высоких уровней абстракции с помощью карринга. Поэтому есть веские основания считать, что все это может повлиять на производительность. А можно ли сделать так, чтобы и волки были сыты, и овцы целы? Об этом и пойдет речь в следующей главе.

Резюме

Из этой главы вы узнали следующее.

- Программы, в которых используются абстракции для соединения очень простых функций, являются модульными.
- Модульный код, основанный на чистых функциях, легко поддается тестированию и в большей степени пригоден для применения строгих методик проверки вроде тестирования на основе свойств.
- Тестируемый код должен иметь простой проток управления.
- Простой поток управления уменьшает сложность программы в целом. Это можно оценить количественно с помощью соответствующих показателей цикломатической сложности.
- Уменьшение сложности программ облегчает их понимание.

Оптимизация функз^ионалъного кода

в этой главе...

- Выявление мест, где функциональный код более производителен
- Исследование внутреннего механизма выполнения функций BjavaScript
- Последствия вложения контекстов функций и рекурсии
- Оптимизация выполнения функций посредством отложенного вычисления
- Ускорение выполнения программ благодаря запоминанию
- Сворачивание вызовов хвостово-рекурсивных функций

В 97 из 100 случаев мы должны забывать о небольшом повышении эффективности, ведь весь корень зла — в преждевременной оптимизации. Но очень важно не упускать такую возможность в остальных 3 случаях.

Из книги *Искусство программирования* (The Art of Computer Programming) *Дональда Кнута (Donald Knuth)*

Как рекомендуют знающие люди, оптимизировать код следует в последнюю очередь. В предыдущих главах пояснялось, как писать и тестировать функциональный код, а теперь, ближе к завершению этого увлекательного экскурса в область ФП, мы рассмотрим способы оптимизации подобного кода. Ни одна парадигма программирования не является идеальной и имеет свои компромиссы, например, между производительностью и абстракцией. Функциональное программирование предоставляет уровни абстракции для достижения высо

кой степени текучести и декларативности прикладного кода. В связи с необходимостью применять внутренний карринг, рекурсию, заключение в оболочку монад и композицию для решения даже самых простых задач может возникнуть следующий вопрос: оказывается ли функциональный код таким же производительным, как и императивный?

В самом деле, сокращение времени выполнения даже самых современных веб-приложений, кроме игровых, мало что дает. Быстродействие современных компьютеров и технологическое качество компиляторов возросло настолько, что неизменно гарантирует высокую производительность правильно написанного кода. В этом отношении функциональный код не менее производителен, чем императивный, хотя его особенности проявляются иначе.

Было бы неразумно перейти к применению на практике новой парадигмы программирования, не уяснив последствия этого шага для среды, в которой должен выполняться написанный код. Именно поэтому в этой главе поясняются особенности функционального кода HaJavaScript, которые следует иметь в виду, особенно при обработке больших массивов данных. По ходу изложения материала этой главы будут упоминаться такие основные языковые средства JavaScript, как замыкания, поэтому прочитайте главу 2, “Сценарий высшего порядка”, чтобы ясно понимать, о чем здесь идет речь. В этой главе обсуждаются также интересные методики оптимизации, в том числе отложенное вычисление, запоминание и оптимизация вызовов рекурсивных функций.

Функциональное программирование не ускоряет время вычисления отдельных функций, а скорее служит стратегией для исключения дублирующихся вызовов функций и откладывания вызовов кода до тех пор, пока в этом не возникнет абсолютная необходимость. Это дает возможность ускорить работу всего приложения. В языках исключительно функционального программирования подобные виды оптимизации встроены в саму платформу и могут использоваться вообще без вмешательства со стороны программиста. А BjavaScript эти виды оптимизации приходится внедрять вручную через специальный код или функциональные библиотеки. Но прежде чем обсуждать этот вопрос подробно, рассмотрим вкратце трудности, возникающие на пути применения JavaScript в ФП, а также веские основания для оптимизации функционального кода.

7.1. Внутренний механизм выполнения функций

Поскольку в основу ФП положено вычисление функций для всего, что делается в программе, поэтому приступая к изучению вопросов производительности и оптимизации, очень важно уяснить, что же происходит при вызове любой функции BjavaScript. При вызове функции в стеке ее контекста интерпретатор JavaScript создает специальную запись (или фрейм).

Стек контекста функции является компонентом принятой В javaScript модели программирования и отвечает за организацию выполнения функции и переменные, которые она замыкает (за дополнительными разъяснениями об

ращайтесь к разделу 2.4). Такой стек всегда начинается с фрейма глобального контекста выполнения, где содержатся все глобальные переменные (рис. 7.1).

Примечание

Стек является основной структурой, содержащей объекты, которые можно размещать и извлекать из стека по принципу "последним пришел, первым обслужен" (LIFO). В качестве аналогии можно привести стопку тарелок, т.е. все операции в стеке выполняются на его вершине.

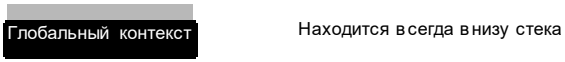


Рис. 7.1. Стек контекста выполнения в JavaScript после его инициализации. В глобальном контексте можно отслеживать немало переменных и функций, хотя это зависит и от количества сценариев, загружаемых на странице

Фрейм глобального контекста выполнения всегда располагается внизу стека. Каждый фрейм контекста функции занимает определенный объем памяти
в зависимости от количества локальных переменных, которые в нем находятся. А в отсутствие каких-либо локальных переменных пустой фрейм занимает около 48 байт. Для хранения каждой локальной переменной или параметра (числового и логического) требуется около 8 байт. Естественно, что чем больше переменных объявлено в теле функции, тем крупнее становится фрейм стека. В каждом фрейме содержится приблизительно следующая информация в структуре данных[IX]:

```
executiorLContexttota =
     scopeChain,
     variableobject,
     this
```

тельского контекста выполнения

В этом контексте имеется доступ к свойству variableobject данной функции, а также к свойству variableobject из роди-
\ Содержит аргументы, внутренние переменные и объявления данной функции

Ссылка на функциональный объект (напомним, что каждая функция в JavaScript считается объектом)

Из этой структуры данных можно сделать ряд важных выводов. Во-первых, именно свойство variableobject в основном определяет размер фрейма стека, поскольку в нем содержатся ссылки на аргументы функции, конкретный объект arguments в виде массива (см. главу 2, “Сценарий высшего порядка”), а также все локальные переменные и функции. Во-вторых, именно цепочка областей видимости функций связывает контекст данной функции

[IX] **Эти сведения взяты из отличной публикации “What Is the Execution Context & Stack in JavaScript?” (Назначение контекста выполнения и его стека BjavaScript) Дэвида Шериффа (David Shariff) от 19 июня 2012 г. в блоге по адресу** http: //davidshariff. com/blog/hat-is-the-execution-context-in-javascript/.

с его родительским контекстом выполнения, как поясняется ниже. Прямо или косвенно, но цепочка областей видимости каждой функции в конечном счете связывает ее с глобальным контекстом.

Примечание

Цепочка областей видимости отдельной функции отличается от цепочки прототипов объекта в JavaScript. И хотя обе цепочки ведут себя сходным образом, цепочка прототипов обозначает ссылку, устанавливаемую при наследовании объектов через свойство prototype. А цепочка областей видимости обозначает, в частности, порядок доступа внутренней функции к замыканию ее внешней функции.

Поведение стека определяется следующими важными правилами.

- JavaScript — однопоточный язык, а следовательно, в нем поддерживается синхронное выполнение кода.
- Имеется один и только один глобальный контекст, общий для всех контекстов функций.
- Допускается неограниченное количество контекстов функций (различные браузеры могут накладывать свои ограничения на выполнение клиентского кода).
- При каждом вызове функции создается новый контекст ее выполнения, даже если она вызывает сама себя рекурсивно.

Как известно, в функциональном программировании функции применяются в максимальной степени. Это побуждает разбивать решаемые задачи на как можно больше функций, а также подвергать их каррингу для дополнительного удобства и повторного использования. Но применение большого количества кар- рированных функций оказывает влияние на размер стека контекста функций.

7.1.1. Карринг и стек контекста функций

Я являюсь большим поклонником карринга, и мне бы очень хотелось, чтобы все вычисления функций выполнялись BjavaScript с автоматическим каррин- гом. Но такой дополнительный уровень абстракции может привести к некоторым контекстным издержкам по сравнению с обычным выполнением функции. Чтобы это положение стало понятнее, выясним, что же в действительности происходит внутри механизма вызова каррированной функции BjavaScript.

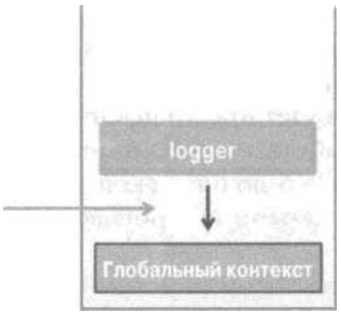
Как упоминалось в главе 4, “На пути к повторно используемому, модульному коду”, когда производится карринг функции, интерпретатор JavaScript заменяет ее одноразовый вызов со всеми параметрами на несколько последовательных внутренних вызовов. Иными словами, в результате карринга функции logger () из главы 4:

```
const logger = function (appender, layout, name, level, message)
```

получается следующая вложенная структура:

```
const logger =
    function (appender) {
        return function (layout) {
            return function (name) {
                return function (level) {
                    return function (message) {
```

В такой вложенной структуре стек используется более интенсивно, чем при прямом вызове. Поясним сначала порядок выполнения некаррированной версии функции logger (). Вследствие синхронного характера выполнения кода HaJavaScript вызов функции logger () приводит к приостановке работы глобального контекста и подготовке к запуску этой функции. При этом создается новый активный контекст, а в нем указывается ссылка на глобальный контекст для доступа к переменным (рис. 7.2).



Ссылка `scopeChain` на глобальный

"Г Становится активным контекстом -Г (и ссылается на глобальный контекст) ""I Приостанавливается (и всегда J находится внизу стека)

Рис. 7.2. При вызове любой функции (в данном случае — logger ()) однопоточный интерпретатор JavaScript приостанавливает работу текущего глобального контекста и активизирует контекст выполнения новой функции. В этот момент между глобальным контекстом и контекстом функции устанавливается связь, доступная по ссылке scopeChain. А после возврата из функции logger () контекст ее выполнения удаляется из стека и восстанавливается глобальный контекст

Во внутреннем механизме выполнения самой функции logger () вызываются остальные операции из библиотеки Log4js, в результате чего создаются новые контексты функций, размещаемые в стеке (подробнее о библиотеке Log4js см. в приложении). Благодаря действию замыканий в JavaScript контексты функций, возникающие в результате внутренних вызовов функций, накапливаются в стеке один над другим и занимают выделенную для них память. Ссылка на старый контекст указывается в элементе scopeChain (рис. 7.3).

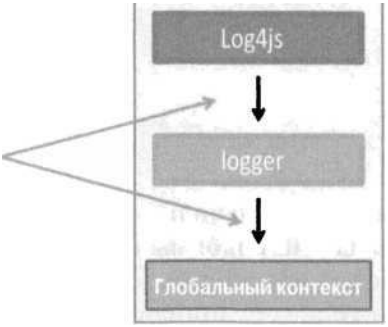
И, наконец, по завершении кода вложенных операций из библиотеки Log4js текущий контекст удаляется из стека, и за ним активизируется контекст выполнения функции logger (). После завершения работы и этой функции интерпретатор JavaScript возвращает все в исходное состояние, в котором действует только глобальный контекст (см. рис. 7.1). Вот, собственно, и вся тайна, скрывающаяся за действием замыканий BjavaScript.

Рис. 7.3. Расширение контекста функции при выполнении вложенных функций. При вызове каждой из этих функций создается новый фрейм в стеке, и поэтому размер стека растет пропорционально уровню вложенности функций. В процессе карринга и рекурсии используются вызовы вложенных функций

Ссылка scopeChain на родительский контекст функции

Становится активным контекстом

Приостанавливается (и всегда находится внизу стека)



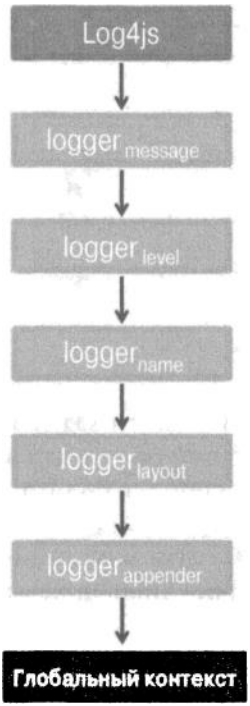
Несмотря на всю эффективность такого подхода, глубоко вложенные функции могут потребовать немало оперативной памяти. В главе 8, “Обработка асинхронных событий и данных”, будет представлена функциональная библиотека RxJS, предназначенная для работы с асинхронным кодом. Весь исходный код самой последней ее версии RxJS 5 полностью переписан по сравнению с предыдущей версией с главным акцентом на производительность и сокращение количества замыканий.

А теперь рассмотрим рис. 7.4, на котором показано выполнение каррированной версии функции logger().

На первый взгляд карринг всех функций может показаться неплохой идеей, но злоупотребление им способно привести к тому, что программы будут занимать большие участки пространства стека и выполняться значительно медленнее. Чтобы убедиться в этом, запустите следующую простую программу оценки производительности:

```
const add = function (a, b) { return a + b;
};

const c_add = curry2(add);

const input = _.range(80000);

addAll(input, add); 11 ->511993600000000
addAll(input, c_add); // -> браузер останавливается

function addAll(arr, fn) {
    let result= 0;
    for(let i = 0; i < arr.length; i++) {
        for(let j = 0; j < arr.length; j++) {
```

```
            result += fn(arr[i], arr[j]);
        }

    }
    return result;
```



Становится активным контекстом

Приостанавливается. Каждый новый контекст функции представляет ее каррированный вызов, для чего требуется целый новый стек

Приостанавливается (и всегда находится внизу стека)

Рис. 7.4. При карринге каждый параметр каррированной функции преобразуется внутренним образом во вложенный вызов. Такое удобство в предоставлении параметров по очереди достигается за счет дополнительных фреймов, занимающих место в стеке

В этой программе создается массив из 80000 чисел и сравнивается некар- рированная версия функции с каррированной. Правильный результат возвращается из некаррированной версии данной функции через несколько секунд, тогда как выполнение ее каррированной версии приводит к зависанию браузера. Без сомнения, за карринг приходится платить определенную цену, хотя обработка столь крупных массивов данных в большинстве приложений JavaScript маловероятна.

И это не единственный случай, когда размер стека может заметно вырасти. Неэффективные и неверные рекурсивные решения также приводят к переполнению стека.

7.1.2. Трудности, связанные с рекурсивным кодом

Новые контексты функций создаются даже в тех случаях, когда функции вызывают сами себя. Неверный рекурсивный вызов, когда основная ветка программы так и не достигается, может легко привести к переполнению стека. Правда, рекурсия относится к тем приемам,

которые либо срабатывают, либо не срабатывают, и в последнем случае рекурсия никак не дает о себе знать. Если вам когда-нибудь приходилось получать сообщения об ошибке вроде ''Range Error: Maximum Call Stack Exceeded'' (Ошибка при проверке границ: превышен максимальный размер стека вызовов) или ''Too much recursion" (Слишком много рекурсии), то вы знаете, что здесь имеется в виду. С помощью следующего простого сценария можно выполнить эталонное тестирование браузера, чтобы получить приблизительный размер стека вызовов функций:

```
function increment(i) {
    console.log(i);
    increment(++i);
} increment(1);
```

В различных браузерах выявление ошибок переполнения стека реализовано по-разному. Так, на моем компьютере браузер Chrome генерирует исключение приблизительно через 17500 итераций в данном сценарии, тогда как браузер Firefox — выполняет только 7600 итераций. Но не следует полагаться на эти цифры как на верхние границы стека вызовов, приступая к написанию собственных рекурсивных функций! Эти граничные цифры лишь показывают те пределы, которые не следует превышать. При написании собственных рекурсивных функций вы должны придерживаться намного более низких значений счетчика рекурсии. Если же в составленной вами программе счетчик рекурсий постоянно растет, либо имеет очень высокие значения, то, вероятнее всего, в программу вкралась ошибка.

Если же вам придется обрабатывать необычно крупный массив данных, используя рекурсию, имейте в виду, что размер стека будет расти пропорционально размеру массива данных. В качестве примера рассмотрим следующую рекурсивную функцию для обнаружения самой длинной символьной строки в массиве:

```
function longest(str, arr) {
    if(R.isEmpty(arr)) { return str;
    else {
        let currentstr = R.head(arr).length >= str.length
                         ? R.head(arr): str;
        return longest(currentstr, R.tail(arr));
    } }
```

Вызвать функцию longest () для списка всех 192 стран в мире несложно, но поиск с ее помощью города с самым длинным названием среди 2,5 млн городов

может привести к неудачному завершению данной программы, как показано на рис. 7.5. (На самом деле этот конкретный алгоритм не приведет к сбоям при обработке крупных массивов в стандарте ES6 языка JavaScript, как поясняется далее.)

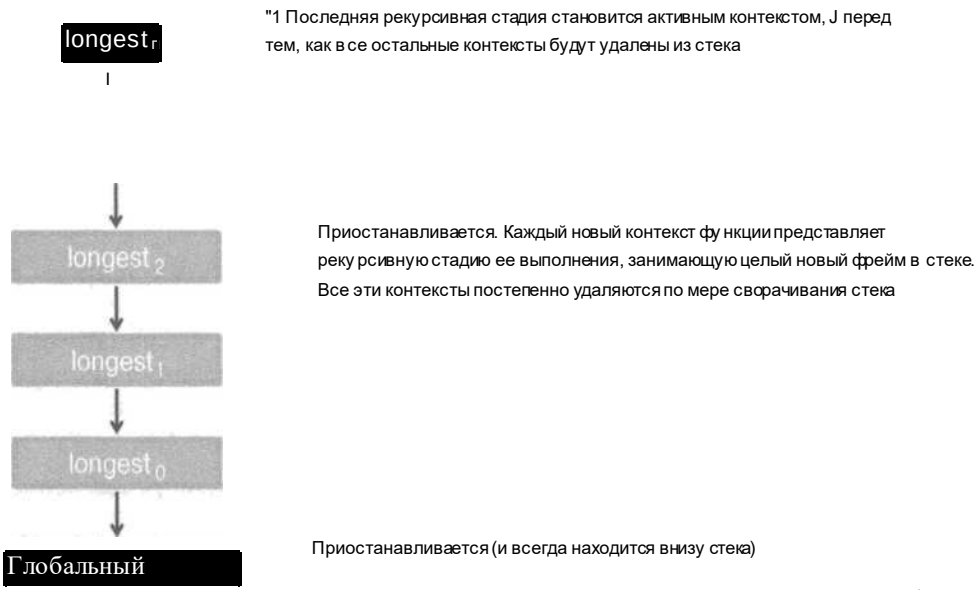


Рис. 7.5. При поиске самой длинной символьной строки в массиве из *n* элементов функции longest () приходится наращивать размер стека пропорционально размерности входных данных, вводя *n* фреймов в стек контекста ее выполнения

В качестве альтернативы такому способу обхода списков, особенно с необычайно крупными массивами, применяют функции высшего порядка, упоминавшиеся в главе 3, “Меньше структур данных и больше операций”, и реализующие такие операции, как map, filter и reduce. В этом случае не формируются вызовы вложенных функций, а размер стека остается постоянным на каждой итерации.

Несмотря на то что карринг и рекурсия приводят к тому, что функции занимают больше места в оперативной памяти, чем их императивные аналоги, следует также иметь в виду те преимущества, которые приносит карринг в отношении гибкости и повторного использования кода, а также корректность, присущую рекурсивным решениям. Эти преимущества стоят требований, предъявляемых к дополнительному расходу оперативной памяти.

Впрочем, нет худа без добра. Ведь функциональное программирование предоставляет такие возможности для оптимизации прикладного кода, которые отсутствуют у других парадигм. Размещение большого количества вызовов

функций в стеке может привести к увеличению объема оперативной памяти, потребляемой программой. Так почему бы не исключить полностью вызовы некоторых функций?

7.2. Отсрочка выполнения с помощью отложенного вычисления

Исключив ненужные вызовы функций и крупные входные данные, когда достаточно их подмножества, можно добиться существенного улучшения производительности программы. В таких языках ФП, как Haskell, в каждое функциональное выражение встроено *отложенное вычисление* функций. Имеются разные алгоритмы отложенного вычисления функций, но все они преследуют одну и ту же цель: задержать выполнение функции как можно дольше или хотя бы до вызова зависящего от нее выражения.

Но для вычисления функций в JavaScript применяется более широко распространенная стратегия энергичного вычисления. В этом случае выражение вычисляется, как только оно будет привязано к переменной, независимо от потребности в результате его выполнения. Такой способ вычисления иногда еще называют *энергичным (или жадным)*. В качестве примера рассмотрим извлечение подмножества элементов из массива, приведенное на рис. 7.6.

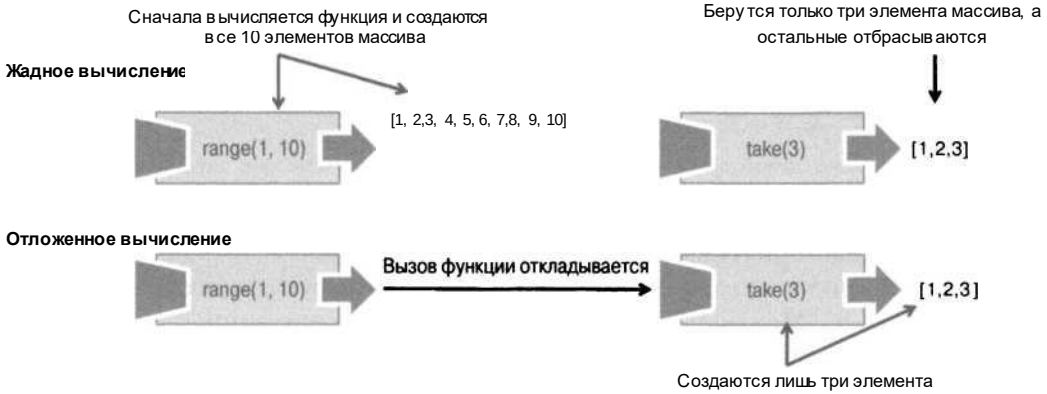


Рис. 7.6. Композиция функции range (), возвращающей список чисел от начала и до конца, с функцией take (), читающей первые *n* элементов из этого списка. При энергичном вычислении функция range () выполняется полностью, передавая результат функции take(). А при отложенном вычислении функция range () не запускается до тех пор, пока не будет вызвана зависящая от нее операция take

Как видите, в алгоритме энергичного вычисления сначала выполняется функция range (), а затем ее результат передается функции take (), которой требуется лишь часть этого результата, тогда как остальная его часть просто отбрасывается. Только представьте, насколько это было бы расточительно, если бы пришлось формировать крупный массив чисел. А при отложенном вычислении выполнение функции range () откладывается до тех пор, пока ее результат не потребуется зависящей от нее операции, реализуемой функцией take (). Если заранее известно назначение функции range (), то из нее можно получить лишь нужное количество элементов. Рассмотрим еще один пример *с* использованием монады типа Maybe:

```
Maybe.of(student).getOrElse(createNeStudent());
```

На первый взгляд, применение монады типа Maybe может навести на мысль, что данное выражение ведет себя следующим образом:

```
if(!student) {
    return createNeStudent();
}
else {
    return student;
}
```

Но благодаря принятому в JavaScript алгоритму энергичного вычисления функция createNeStudent () будет выполнена в данном коде независимо от того, окажется ли пустым (null) объект student. А при отложенном вычислении данное выражение повело бы себя таким же образом, как и приведенный ранее фрагмент кода, вообще не вызывая функцию createNeStudent (), если объект student окажется недостоверным. Так как же воспользоваться преимуществами отложенного вычисления? В этом разделе рассматриваются следующие рекомендации относительно его применения.

- Исключение ненужных вычислений.
- Применение сокращенного слияния в функциональных библиотеках.

7.2.1. Исключение вычислений с помощью комбинатора чередования функций

Нет ничего удивительного в том, чтобы предпринять определенные действия для эмуляции отложенного вычисления и в то же время извлечь ряд преимуществ из языков исключительно функционального программирования. В простейшем случае ненужных вычислений можно избежать, передав функции по ссылке (или по имени) и вызвав ту или иную функцию по заданному условию. В главе 4 был представлен комбинатор функций alt (), в котором выгодно используется логическая операция || (ИЛИ) для вычисления в первую очередь функции fund и вызова функции f unc2 лишь в том случае, если функция fund возвратит значение false, null или undefined, как демонстрируется в следующем примере:

```
const alt = R.curry ( (fund, func2, val) => fund (val) || func2(val));

const shoStudent = R.compose(append('#student-info'), alt (findStudent,
      createNeStudent) ) -------------------- Функции не требуется вызывать преждевременно, поскольку они передаются комбиниатору по ссылке для согласования их вызовов
shoStudent (*444-44 — 4444* ) ;
```

Комбинатор берет на себя обязанность согласовывать вызовы функций, и поэтому приведенный выше пример кода равнозначен следующей условной логике, написанной в императивном стиле:

```
var student = findStudent('444-44-4444');
if(student !== null) {
    append('#student-info', student);
}
else {
    append('#student-info', createNeStudent('444-44-4444'));
```

```
}
```

Это очень простой пример исключения ненужных вычислений функций с намного меньшим дублированием. Более эффективная стратегия будет продемонстрирована далее в этой главе, когда речь пойдет о *запоминании (memoization)*. А с другой стороны, определение всей программы еще до ее выполнения позволяет осуществить в функциональных библиотеках оптимизацию, называемую *сокращенным слиянием (shortcut fusion)*.

7.2.2. Использование преимуществ сокращенного слияния

В главе 3, "Меньше структур данных и больше операций", была представлена функция, реализующая операцию chain в библиотеке Lodash. Данная функция позволяет заключить в оболочку всю последовательность функций, инициировав ее выполнение через конечную функцию value (). Это дает возможность не только отделить описание программы от ее выполнения, но и выявить средствами Lodash места оптимизации кода, объединив, например, выполнение некоторых функций для более эффективного использования памяти. Так, в следующем примере получается список стран, отсортированный по численности населения:

```
_.chain([p1, p2, p3, p4, p5, p6, p7])
    .filter(isValid)
    .map(_.property('address.country')).reduce(gatherStats, {})
    .values()
    .sortBy('count')
    .reverse()
    .first()
    .value()
```

Такой декларативный стиль программирования означает, что беспокоиться следует не о том, как работают функции, а том, что для этого нужно сделать, заранее определив требующиеся действия. Иногда это дает библиотеке Lodash возможность оптимизировать внутренним образом выполнение программы, используя сокращенное слияние. Это вид оптимизации на уровне функции, допускающий слияние процесса выполнения некоторых функций в одно уплотненное количество внутренних структур данных, предназначенных для вычисления промежуточных результатов. Благодаря созданию меньшего количества

структур данных сокращается чрезмерное расходование оперативной памяти, требующейся для обработки крупных коллекций.

Такое слияние оказывается возможным благодаря принятым в функциональном программировании строгим правилам соблюдения ссылочной прозрачности, которая придает ему особую математическую или алгебраическую точность. Например, вызов функции compose (map (f) , map (g)) можно заменить выражением map (compose (f, g)), не меняя ее назначение. Аналогично вызов функции compose (filter (pl) , filter (p2)) равнозначен выражению filter ((x) => pl (x) && p2(x)). Именно это и происходит в паре операций filter и tap, с которых начинается цепочка в предыдущем примере. Опять же манипулирование последовательностью операций подобным математическим способом оказывается возможным лишь благодаря чистым функциям. Это наглядно демонстрирует еще один пример из листинга 7.1.

Листинг 7.1. Реализация отложенного вычисления и сокращенного слияния средствами библиотеки Lodash

```
const square = (x) => Math.pow(x, 2);
const isEven = (x) => x % 2 === 0;
                                        Сформировать массив из
const numbers = range(200); -«f -----1 чиеелот1до20О

const result =
      .chain(numbers)                   Обработать только три первых
      .map(square)                      числа, которые удовлетворяют
      .filter(isEven)                   критериям, накладываемых
      .take(3) <•—                      операциями filter или шар
      .value();       //->
[0,4,16] result.length; // -> 5
```

В исходном коде из листинга 7.1 выполнены два вида оптимизации. Во- первых, при вызове функции take(3) библиотеке Lodash предписывается принять во внимание лишь три первых значения, которые удовлетворяют критериям преобразования и фильтрации. Это позволяет не расходовать напрасно драгоценные циклы ЦП на остальные 195 элементов массива. *И* во-вторых, сокращенное слияние позволяет соединить последующие вызовы функций tap () HfilterO в выражение compose (filter (isEven) , tap (square)). В этом нетрудно убедиться, снабдив функции square () и isEven () операторами трассировки в системный журнал (для целей протоколирования в данной композиции, по существу, применяется комбинатор tap () из библиотеки Ramda), как показано ниже.

```
square = R.compose(R.tap(() => trace('Mapping'))л square);
isEven= R.compose(R.tap(() => trace('then filtering')), isEven);
```

В итоге на консоли появится следующая пара сообщений, повторяемых пять раз:

```
Mapping
then filtering
```

чем подтверждается слияние операций tap и filter. Применение функциональных библиотек не только упрощает написание тестов, но и улучшает время выполнения прикладного кода. Сокращенное слияние приносит выгоду и в других функциях из библиотеки Lodash, реализующих следующие операции: _.drop, _.dropRight, dropRightWhile, dropWhile, first, initial, _.last, pluck, reject, _.rest, reverse, slice, _.takeRight, takeRightWhile, takeWhile и here.

Аналогично для исключения вычислений до тех пор, пока они не потребуются, имеется еще одно эффективное средство оптимизации функциональных программ, называемое *запоминанием (memoization)*.

7.3. Реализация стратегии вызовов по требованию

Чтобы ускорить выполнение приложений, можно, в частности, исключить повторное вычисление значений, особенно если это вычисление обходится дорого. В традиционных объектно-ориентированных системах это достигается организацией кеша на уровне посредника, который проверяется перед вызовом функции. После возврата из функции результату ее выполнения присваивается уникальный ключ для ссылки на него, и полученная в итоге пара "ключ- значение" сохраняется в кеше. В качестве *кеша* может служить промежуточное хранилище или оперативная память, запрашиваемая перед выполнением затратной операции. В веб-приложениях кеш применяется для временного хранения изображений, документов, скомпилированного кода, HTML-страниц, результатов запроса и т.д. В качестве примера рассмотрим следующий фрагмент кода, где реализуется уровень кеширования произвольной функции:

Составить из имени функции и ее аргументов ключ для идентификации результата ее выполнения

Проверить сначала в кеше, вызывалась ли данная функция раньше

```
function cachedFn (cache, fn, args) { let key =
    fn.name + JSON.stringify(args);-<------------
    if(contains(cache, key)) {          ◄ ------
        return get(cache, key);                  ◄
    }
```

Если функция вызывалась, то возвращается значение, найденное в кеше

```
    else {
        let result = fn.apply(this, args); -<■
        put(cache, key, result); ◄ ------
        return  result;
    }
```

В противном случае (т.е. при отсутствии в кеше) функция выполняется

Результат выполнения функции сохраняется в кеше

Функцией cachedFn (), определенной в приведенном выше коде, можно воспользоваться, чтобы заключить в ее оболочку выполнение функции findStudent (), как показано в следующем примере:

```
var cache = {};
cachedFn(cache, findStudent, '444-44-4444');
cachedFn (cache, findStudent, ' 444-44-4444'); •■<
```

При первом вызове результат отсутствует в кеше, и поэтому функция findStudent () выполняется

При втором вызове вычисленное значение извлекается непосредственно из кеша

Функция cachedFn () действует в качестве посредника между вызовом функции и ее

результатом, с целью исключить ненужные повторные вызовы. Но написать код, где такая функция служила бы оболочкой для вызова каждой функции, было бы затруднительно, а сам код оказался бы неудобочитаемым. Хуже того, данная функция вызывает побочный эффект, поскольку она зависит от глобального объекта общего кеша. Поэтому требуется универсальное решение, позволяющее выгодно воспользоваться преимуществами кеширования, сохранив независимость прикладного кода и тестов от данного механизма. В языках ФП такой механизм называется *запоминанием.*

7.3.1. Общее представление о запоминании

Алгоритм кеширования, положенный в основу запоминания, реализуется аналогично приведенному выше коду, поэтому аргументы функции используются в нем для формирования уникального ключа, по которому сохраняется результат выполнения функции. Таким образом, при последующих вызовах функции с теми же самыми аргументами сохраненный результат может быть возвращен немедленно. Соотнесение результата выполнения функции с входными данными, иными словами, — сравнение вычисленного функцией значения со входным значением, достигается благодаря определенному принципу ФП. Нетрудно догадаться, что этим принципом является ссылочная прозрачность. Прежде всего рассмотрим преимущества запоминания на примере простого вызова функции.

7.3.2. Запоминание функций, требующих интенсивных вычислений

В одних языках исключительно функционального программирования алгоритм запоминания реализуется автоматически, а в других языках вроде JavaScript и Python программирующим предоставляется возможность выбрать момент для запоминания функции. Естественно, что в функциях, требующих интенсивных вычислений, можно выгодно воспользоваться чередованием уровня кеширования. Рассмотрим в качестве примера вычисление функции rot 13 (), где символьные строки кодируются по алгоритму ROT13 (путем циклического сдвига 26 букв английского алфавита на 13 позиций в коде ASCII). И хотя это слабый для шифрования алгоритм, он применяется на практике в веб-приложениях для сокрытия решений загадок и кодов скидок, исключения оскорбительных материалов и т.д.: `var discountcode = 'functional_js_50_off';`

```
rot!3(discountcode); // -> shapgvbany_f_50_bss
```

Ниже показано, каким образом реализуется алгоритм ROT 13.

```
var rotl3 = s =>
    s.replace(/[a-zA-Z]/g, c =>
        String.fromCharCode((c <= 'Z' ? 90 : 122)
            >= (c = c.charCodeAt(0) + 13) ? c : c - 26));
            (c = c.charCodeAt(0) +13) ? c : c - 26) ;
    });
};
```

Понимание этого алгоритма необязательно для обсуждения данного вопроса. Тем не менее важно знать, что для одной и той же входной символьной строки (со ссылочно прозрачной функцией) по этому алгоритму всегда вычисляется одно и то же сообщение. Это означает, что запоминание позволяет добиться необычайного повышения производительности. Но прежде чем рассматривать реализацию запоминания в исходном коде функции memoize (), отметим, что ее можно применять следующими способами.

- Вызывая метод для функционального объекта: `var rotl3 = rot13.memoize();`
- Заключая в оболочку определение функции, как было показано ранее: `var rotl3 = (s =>`

```
        s.replace(/[a-zA-Z]/д, c =>
           String.fromCharCode((c <= 'Z' ? 90 : 122)
              >= (c = c.charCodeAt(0) + 13) ? c : c - 26))).memoize();
```

Благодаря запоминанию можно ожидать, что при последующем вызове функции с одними и теми же входными данными будет инициирована проверка внутреннего кеша и немедленный возврат значения. Чтобы продемонстрировать это, воспользуемся интерфейсом High Resolution Hine API HaJavaScript, иначе называемым Performance API, чтобы получить более точные отметки времени, чем с помощью традиционных функций JavaScript вроде Date.no () и console. time (). В конечном счете это позволит нам точно измерить время, истекшее после вызова функции. Для внедрения операторов фиксации времени до и после тестируемой функции в данном примере будет использована монада типа 10. Исходный код требующихся для этой цели функций приведен в листинге 7.2, а в последующих примерах кода он будет опущен ради краткости.

Листинг 7.2. Измерение временных характеристик производительности вызываемых функций с помощью комбинатора tap ()

```
const start - o -> now();          Вызвать функции
const runs = [];                   start o и end o для
const end = function (start) {  измерения времени let end = now();
    let result = (end - start).toFixed(3); < ....   Воспользоваться средствами Performance
    runs.push(result);                              API для измерения времени в миллисекундах
    return result;                                  с точностью до трех десятичных цифр
```

```
const test = function (fn, input) {
    return () => fn(input); };

const testRotl3 = 10. from
(start)
     .map(R.tap(test(rotl3, -functional js 50 off') )) ^-1
     . map (end);

testRotl3.run(); // первый раз: 0.733 мс testRotl3.run(); // второй раз: 0.021 мс
assert.ok(runs[0] >= runs[l]);
```

Воспользоваться комбинатором tap (), чтобы распространить сведения о начальном моменте времени через мо- делается потому, что .--- важен не результат функции, а время ее выполнения)

Как видите, в результате второго вызова функции rot 13 () та же самая символьная строка возвращается в мгновение ока. И хотя автоматическое запоминание не встроено BjavaScript, его можно внедрить вручную, расширив объект типа Function, как демонстрируется в листинге 7.3.

Листинг 7.3. Внедрение запоминания в вызовы функций

```
Function.prototype.memoized = function () {
    let key = JSON.stringify(arguments);
    this, cache = this, cache I I {};
    this._cache[key] = this._cache[key] ||
                       this.apply(this, arguments);
    return this._cache[key];
};

Function.prototype.memoize = function () {
    let fn = this;
    if (fn.length === 0 I I fn.length > 1) { return fn-
    return function () {
        return fn.memoized.apply (fn, arguments); };
```

Внутренний вспомогательный метод, отвечающий за выполнение логики кеширования данного конкретного экземпляра функции

Преобразование ряда входных данных в символьную строку для получения идентификатора данной функции. Этот процесс можно сделать более надежным, выявив тип входных данных и применив к ним соответствующий алгоритм формирования ключей. Но в данном и других примерах можно обойтись и без этого

Создать внутренний локальный кеш для данного экземпляра функции Попытаться прочитать сначала кеш, чтобы выяснить, обработаны ли входные данные. Если в кеше обнаружено значение, пропустить вызов данной функции и возвратить прежний ее результат, а иначе выполнить ее

Разрешить запоминание данной функции

Попытаться запомнить только унарные функции

Заключить экземпляр функции в оболочку запомненной функции

В

Благодаря расширению объекта типа Function запоминание в данной реализации становится универсальным, устраняя любые наблюдаемые побочные эффекты доступа к глобальному общему кешу. Кроме того, абстрагирование внутреннего механизма кеширования в функции делает его полностью независимым от тестов. Это означает, что вы освобождаетесь от обязанности снабжать свой код целым рядом операторов или тестировать функциональные

возможные кеширования, уделив основное внимание только тому, что должна делать сама функция.

Для прояснения общей картины на рис. 7.7 схематически представлена последовательность запоминания функции rotl3(). При первом вызове этой функции значения в кеше еще нет, поэтому вычисляется сообщение в формате ROT 13. А по завершении результат вычисления сохраняется по ключу, сформированному из входных аргументов, чтобы повторно воспользоваться им, пропустив все вычисления при последующем вызове.

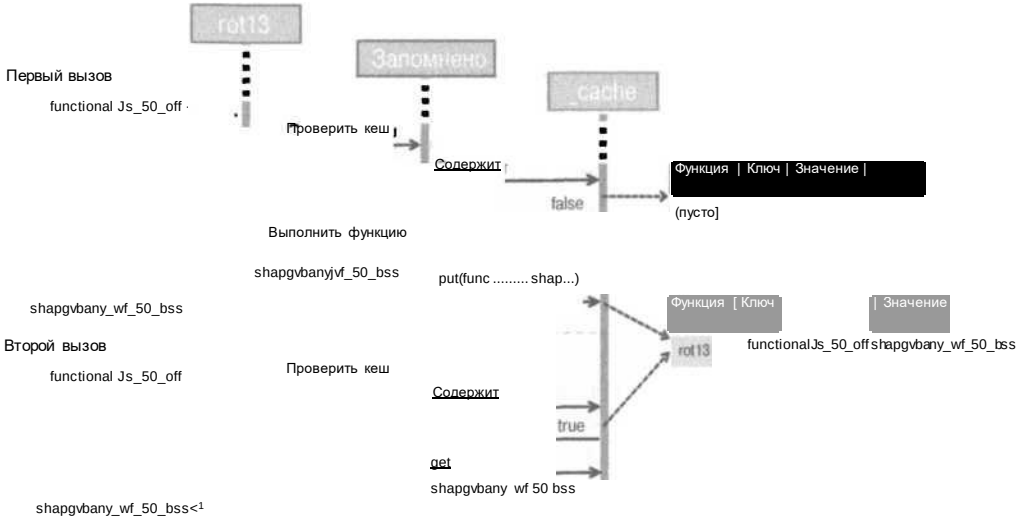


Рис. 7.7. Подробное представление двух вызовов функции rotl3() с сообщением "functional_js_50_off". В первый раз, когда кеш еще пуст, предоставляемый код скидки преобразуется по алгоритму ROT13. Полученный результат далее сохраняется во внутреннем кеше по ключу из этих входных данных. А во второй раз происходит попадание в кеш, в результате чего из него сразу же возвращается значение без повторного вычисления хеш-значения

Примечание

В примерах из этой книги запоминанию подвергаются функции с одним аргументом. Но как оперировать функциями с несколькими аргументами? Этот вопрос в данной книге не рассматривается, но оставляется вам для исследования в качестве упражнения. Для этого вы можете воспользоваться следующим двумя стратегиями: создать многомерный кеш (массив массивов) или однозначный ключ, состоящий из символьного представления аргументов.

Если внимательно проанализировать исходный код из листинга 7.3, то можно обнаружить, что запоминание ограничивается унарными функциями. Это сделано именно так с целью упростить этап формирования ключа в логике кеширования. Если же требуется подвергнуть запоминанию функции, которым

передаются несколько аргументов, соответствующая логика для формирования надлежащего ключа кеша может оказаться более сложной и затратной. Но иногда карринг может оказать помощь в разрешении данного вопроса.

7.3.3. Выгодное применение карринга и запоминания

Более сложные функции или те, которым передается несколько аргументов, труднее кешировать, даже если они чистые. Это объясняется увеличением сложности формирования ключа — операции, которая должна выполняться просто и быстро, чтобы исключить дополнительные издержки на уровне кеширования. Несколько упростить дело можно с помощью карринга. Как пояснялось в главе 4, "На пути к повторно используемому, модульному коду", карринг применяется для преобразования многомерной функции в унарную. Карринг позволяет запоминать функцию аналогично функции saf eFindOb j ect () через функцию findStudent (), как показано ниже.

```
safeFindObject = R. curry (function (db, ssn) {◄---------  Эта функция не является ссылочно-прозрачной,но
    // затратная поисковая операция ввода-вывода          на практике она специально кеширует результаты затратных операций поиска и полученных HTTP-запросов
jч.

const findStudent = safeFindObject(DB('students')).memoize(); findStudent('444-44-4444');
```

Такой прием оказывается работоспособным потому, что объект DB использ уется лишь для доступа к данным и никак не связан с процессом формирования уникальн ого ключа для параметров функции findStudent (), к которому будут привязаны данные об учащемся, найденные по его уникальному идентификатору. Следует особо подчеркнуть, что преобразование функции в унарную требуется не только для того, чтобы упростить обращение с ней и ее композицию, но и для того, чтобы выгодно воспользоваться при запоминании декомпозицией (или разбиением) программы на мелкие части и реализовать кеширование по этим частям. И об этом речь пойдет далее.

7.3.4. Декомпозиция для максимального запоминания

Взаимосвязь запоминания с декомпозицией легче понять, если прибегнуть к простой аналогии из школьного курса химии. Как известно, любой раствор состоит из растворяемого вещества и растворителя. Скорость растворения вещества определяется многими факторами, в том числе *площадью поверхности*. Так, если приготовить два раствора сахара (кускового и песочного) в воде, то в процессе растворения сахар вступает в контакт с водой. И чем больше площадь его поверхности, тем быстрее он растворяется.

По этой же аналогии задачи могут быть разбиты на мелкие запоминаемые функции. Чем более мелкой становится структура прикладного кода, тем больше выгод можно извлечь из запоминания. Внутренний механизм кеширования каждой функции в отдельности вносит свой вклад в ускорение процесса выполнения программы, увеличивая, если угодно, поверхностный контакт.

Так, если обратиться снова к примеру программы shoStudent, то зачем вы полнять проверку достоверности некоторых входных данных, если это уже было сделано раньше? И

если объекты, представляющие учащихся, были извлечены по номерам их социального страхования из локальной базы данных, с помощью cookie-файлов или даже путем обращения к удаленному серверу, а их изменение не предполагается, то зачем тратить драгоценное время на повторный поиск? Примечательно, что если речь идет о функции findStudent (), то запоминание может служить в качестве небольшого кеша запросов, сохраняя уже извлеченные объекты для ускорения доступа к ним. Запоминание дает еще одну удобную возможность рассматривать функции как обыкновенные значения, вычисляемые по требованию. В качестве примера рассмотрим замену некоторых функций в программе shoStudent их запоминаемыми аналогами (где ради удобства имена запоминаемых функций снабжаются префиксом т_, хотя такое обозначение не является общепринятым). Приведенная ниже функция разбита на мелкие части, и благодаря этому выполнение всей программы ускоряется во втором случае на 75%!

```
const m_cleanlnput = cleaninput.memoize();
const m_checkLengthSsn = checkLengthSsn.memoize(); const m_findStudent = findStudent.memoize();

const shoStudent = R.compose(
   map(append('#student-info')),
   liftIO,
   chain(csv),
   map(R.props(['ssn', 'firstname', 'lastname'])),
   map(m_findStudent),
   map(m_checkLengthSsn),
   lift(m_cleanlnput));

shoStudent('444-44-4444').run();
    // -> в среднем 9.2 мс (без запоминания) shoStudent('444-44-4444').run();
    // -> в среднем 2.5 мс (с запоминанием)
```

Еще одной разновидностью декомпозиции является рекурсия, где программа разбивается на самоподобные, мелкие, запоминаемые подзадачи. Аналогично запоминание позволяет заметно ускорить медленно действующий рекурсивный алгоритм.

7.3.5. Применение запоминания к рекурсивным вызовам

Рекурсия способна привести к зависанию браузера или генерированию малоприятных исключений. Обычно это происходит в том случае, когда размер стека выходит из-под контроля, например, при обработке очень крупного массива входных данных. Но иногда запоминание может оказать помощь в решении данного вопроса. Как пояснялось в главе 3, “Меньше структур данных и больше операций”, рекурсия — это механизм разбиения задачи на более мелкие варианты ее самой. Как правило, при рекурсивном вызове решается одна и та же задача или подмножество более крупной задачи, и делается это неоднократно до тех пор, пока не будет достигнут основной вариант, когда наконец-то начинается сворачивание стека вызовов и возврат получаемого результата. Если бы можно было кешировать результаты решения подзадач, то удалось бы повысить производительность при вызове одной и той же функции с более крупными входными данными.

В качестве примера рассмотрим простую функцию, вычисляющую факториал числа n,

обозначаемый как *n!*. Он вычисляется как произведение всех положительных значений, меньших и равных n, по следующей формуле:

```
n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1
```

Например:

Обратите внимание на то, что числа факториала могут быть также определены рекурсивно в виде более мелких - факториалов,например4» «4x3?

Программу для решения этой задачи можно изящно выразить в виде следующего запоминаемого рекурсивного решения:

Вычислить подряд произведения всех |

```
const factorial = ( (n) => (n === 0) ? 1
```

чисел 100 x **99** x **98** x... x **3** x **2** x **1**

```
                  : (n * factorial(n - 1))).memoize();

factorial(100); // -> В первый раз выполняется за 0,299 мс, ◄------ -------- J
factorial(101); // -> а во второй раз - за 0,021 мс ◄-------
```

Воспользоваться кешированным ранее значением, чтобы сократить вычисление, остановившись на произведении 101 x 100!

При запоминании применяются математические принципы вычисления факториалов, и поэтому при втором вызове приведенной выше функции достигается значительное повышение производительности, когда функция "вспоминает" формулу 101! =101x100! и может повторно воспользоваться значением, возвращаемым из первого вызова factorial (100). В итоге вычисление всего алгоритма резко сокращается и результат возвращается мгновенно. Такой подход имеет и другие преимущества с точки зрения управления фреймами стека вызовов, исключая его засорение (рис. 7.8).

Как видите, алгоритм вычисления факториала выполняется полностью при первом вызове функции factorial (100). В итоге создается 100 фреймов в стеке вызовов данной функции. И в этом проявляется следующий недостаток некоторых рекурсивных решений: в них нерационально используется свободное пространство стека, особенно в таких случаях, как при вызове функции factorial (), где количество создаваемых в стеке фреймов пропорционально получаемым входным данным. Но благодаря запоминанию количество фреймов, требующихся в стеке для вычисления следующего числа, значительно сокращается.

Запоминание — не единственный способ оптимизации рекурсивных вызовов. Имеются и другие способы повышения производительности с помощью средств компилятора.

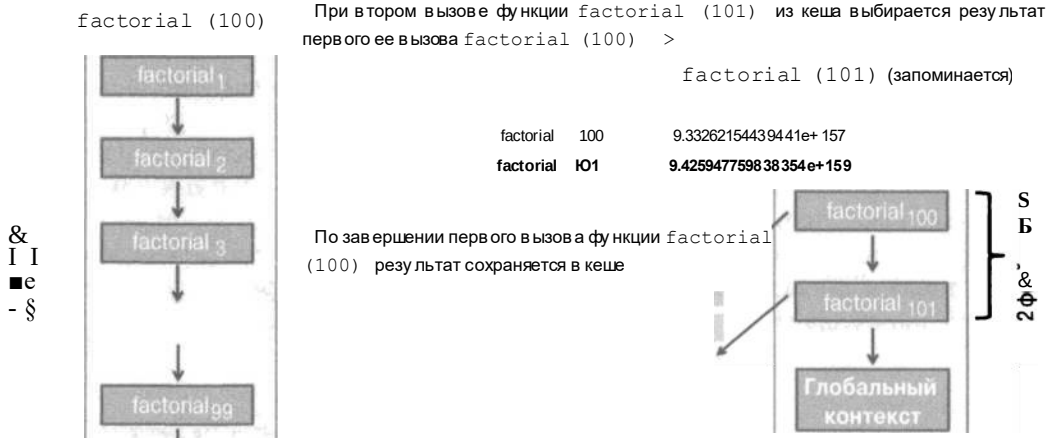


Рис. 7.8. При вызове запоминаемой функции factorial (100) в первый раз создается 100 фреймов стека, поскольку требуется вычислить факториал числа 100!, умножая все числа подряд. А при вызове функции factorial (101) во второй раз повторно используется результат первого вызова, и при этом создаются только два фрейма стека

7.4. Рекурсия и оптимизация хвостовых вызовов

Как было показано ранее, в программах с рекурсией стек используется намного более интенсивно, чем в программах без рекурсии. В некоторых языках ФП даже отсутствуют встроенные механизмы организации циклов, а основной акцент делается на рекурсию и запоминание для реализации эффективной итерации. Но иногда не особенно помогает даже запоминание, например, когда характер входных данных функции постоянно меняется. И в этом случае мало что дает наличие внутреннего уровня кеширования. Можно ли оптимизировать рекурсию, чтобы она выполнялась так же эффективно, как и циклы в подобных случаях? Оказывается, что рекурсивные алгоритмы можно составить таким образом, чтобы компиляторы помогали достичь этого с помощью *оптимизации хвостовых вызовов*[10] *(tail-call optimization, TCO)*. В этом разделе поясняется, что приведенная ниже версия рекурсивной функции factorial ():

```
const factorial = (n, current = 1) => (n === 1) ? current : factorial(n - 1, n *
```

несколько отличается от предыдущей версии, поскольку рекурсивная стадия размещается в ней на хвостовой позиции. Поэтому она выполняется также быстро, как и приведенная ниже императивная версия функции factorial ().

```
var factorial = function (n) {
    let result = 1;
    for(let x = n; x > 1; x—) { result *= x;
    }
                                                current); <
```

Рекурсивная стадия в этой версии функции теперь выполняется в последнем операторе, находящемся на так называемой хвостовой позиции

[10] **Иногда этот термин переводят как** *оптимизация хвостовой рекурсии. — Примеч. ред.*

```
    return result;
}
```

Оптимизация хвостовых вызовов, является усовершенствованием, внедренным в компилятор стандарта ES6 с целью свести выполнение всех вызовов рекурсивной функции в едином фрейме. Но это может произойти только на последней стадии решения рекурсивной задачи, когда вызывается другая (как правило, та же самая) функция. И этот последний вызов находится в самом *хвосте* функции, откуда и пошло название данного способа оптимизации рекурсивных вызовов.

В чем же состоит подобная оптимизация? Вызывая функцию в самом конце рекурсивной функции, можно дать интерпретатору JavaScript понять, что он может больше не хранить текущий фрейм в стеке, поскольку он больше не понадобиться и может быть удален. Как правило, это достигается передачей всего необходимого состояния из контекста одной функции в другой как части ее аргументов, что и демонстрировалось выше на примере рекурсивной функции factorial (). Таким образом, рекурсивная итерация будет происходить всякий раз в новом фрейме, в качестве которого будет использоваться фрейм из предыдущего вызова, вместо нагромождения фреймов друг на друга в стеке. Приведенная выше рекурсивная функция factorial () определена в хвостовой форме, и поэтому выполнение вызова factorial (4) переходит от типичной пирамиды рекурсивных вызовов:

```
factorial(4)
   4 * factorial(3)
     4 * 3 * factorial(2)
       4 * 3 * 2 * factorial(1)
         4*3*2*1* factorial(0)
            4*3*2*1*1
       4 * 3 * 2 * 1
     4*3*2
   4*6
return 24
```

к плоской структуре с учетом стека контекстов выполнения, как показано ниже.

```
factorial (4)
   factorial (3, 4)
   factorial (2, 12)
   factorial (1, 24)
   factorial (0, 24)
   return 24 return 24
```

Как видите, эта плоская структура позволяет более эффективно пользоваться стеком, который больше не придется сворачивать на *n* фреймов. На рис. 7.9 поэтапно демонстрируется процесс преобразования нехвостовой функции factorial () в ее хвостово-рекурсивную версию.

Аннулировать текущий фрейм и заменить его новым, чтобы вычислить следующее значение

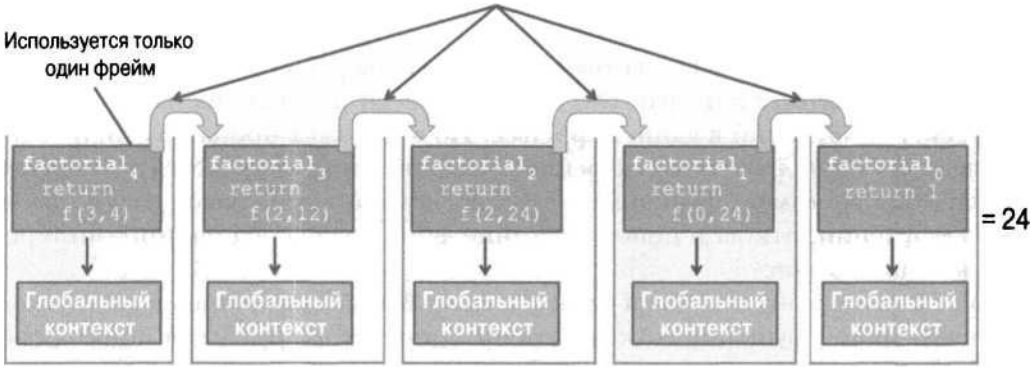


Рис. 7.9. Иллюстрация вызовов хвостово-рекурсивной функции factorial (4). Как видите, в этой функции используется единственный фрейм. Оптимизация хвостовых вызовов заключается в аннулировании фрейма текущей функции в стеке и распределении на его месте нового фрейма, как будто бы функция factorial () вычисляется в цикле

7.4.1. Преобразование нехвостовых вызовов в хвостовые

Оптимизируем функцию factorial (), чтобы выгодно воспользоваться механизмом оптимизации хвостовых вызовов BjavaScript. В приведенной ниже первоначальной реализации: `const factorial = (n) => (n === 1) ? 1 : (n * factorial(n - 1));`

рекурсивная функция factorial () не находилась на хвостовой позиции, поскольку в последнем возвращаемом выражении многократно умножается значение, вычисляемое на рекурсивной стадии:

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

Напомним, что для оптимизации хвостовых вызовов последняя стадия должна быть рекурсивной. Ведь именно это дает интерпретатору возможность преобразовать функцию factorial () в цикл. Это делается в следующие два этапа.

1. Перемещение операции умножения на место дополнительного параметра функции, чтобы отслеживать текущее состояние данной операции.
2. Применение новшества ES6 — стандартного значения параметров функции, определяющих предварительное значение данного аргумента. Такой аргумент можно было бы определить и иначе, но при использовании стандартного значения параметров функции это будет выглядеть намного яснее:

```
const factorial = (n, current = 1) =>
                  (n === 1) ? current :
                  factorial(n - 1, n * current);
```

Теперь данная функция вычисления факториала будет выполняться так, как будто она была реализована путем организации стандартного цикла, не требуя создания дополнительных фреймов в стеке, но в то же время сохраняя в какой-то степени свой первоначальный декларативный и математический вид. Подобное преобразование возможно потому, что свойства хвостово-рекурсивная функции в большой степени совпадают со свойствами стандартного цикла, как показано на рис. 7.10.

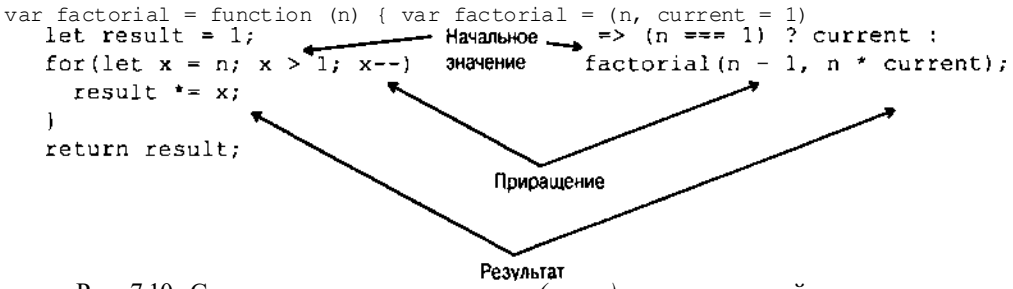


Рис. 7.10. Сходство стандартного цикла *(слева)* с эквивалентной ему хвостово-рекурсивной функцией *(справа)*. В обоих примерах кода можно легко выявить начальное значение, приращение цикла и накопление результата в аккумуляторе

Рассмотрим еще один пример. В главе 3, "Меньше структур данных и больше операций", демонстрировалось следующее небольшое рекурсивное решение задачи суммирования всех элементов в массиве: `function sum(arr) { if(_.isEmpty(arr)) { return 0; } return _.first(arr) + sum(_.rest(arr)); }`

И в этом случае можно заметить, что последнее действие в данной функции, т.е. _. first (arr) + sum(_. rest (arr)), выполняется не в хвостовой форме. Поэтому реорганизуем ее код, чтобы оптимизировать его в отношении расхода оперативной памяти. Любые данные, которые должны быть переданы последующим вызовам данной функции, в приведенной ниже оптимизированной версии теперь введены в состав ее аргументов.

```
function sum(arr, ace = 0) { if(_.isEmpty(arr)) { return 0;
    }
    return sum(_.rest(arr), acc + _.first(arr));
}
```

Хвостовая рекурсия приближает производительность рекурсивного цикла к циклу, организуемому вручную. Таким образом, в тех языках, где такая рекурсия внедрена (например, в стандарте ES6 языка JavaScript), ею можно заменить организуемые вручную циклы, когда первостепенное значение приобретает производительность, и в то же время требуется сохранить правильность применяемого алгоритма и контролировать мутации. Но применение оптимизации хвостовых вызовов не ограничивается только рекурсией. Ее можно, например, применить к любой функции, в последнем операторе которой вызывается другая функция, что нередко происходит в приложениях HaJavaScript. Следует заметить, что оптимизация хвостовых вызовов, являющая новинкой стандарта ES6, была разработана в черновом варианте еще в стандарте ES4, но пока широко не используется в браузерах. По сути, на момент написания данной книги оптимизация хвостовых вызовов еще не была толком реализована ни в одном из популярных браузеров. Именно поэтому приходится пользоваться транспи- лятором Babel, чтобы привести исходный код стандарта ES6 к прежней версии.

Эмуляция оптимизации хвостовых вызовов в стандарте ES5

В наиболее распространенном в настоящее время стандарте ES5 языка JavaScript оптимизация хвостовых вызовов не поддерживается. Она была введена в стандарт ES6

этого языка в качестве предложения под названием *Специфика хвостовых вызовов (proper tail calls)* (см. раздел 14.6 спецификации ECMA-262). Как упоминалось в главе *2*, "Сценарий высшего порядка" работоспособность примеров, представленных в данной книге, обеспечивается с помощью транспи- лятора Babel, выполняющего роль компилятора исходного кода из нового стандарта в прежний, что очень удобно для тестирования перспективных языковых средств.

В качестве обходного приема можно прибегнуть к средству, метафорически называемому *прыжками на батуте (trampolining)*. Это средство позволяет смоделировать хвостовую рекурсию итеративным способом, что идеально подходит для контроля над ростом стека вызовов функций в таких языках со стековой организацией, как JavaScript.

Батутом здесь служит комбинатор функций, которому передается другая функция в качестве входных данных, а он повторно вызывает ее (т.е. раскачивает ее как на батуте) до тех пор, пока выполняется заданное условие. Повторно вызываемая функция инкапсулирована в структуру, которая называется *переходником*[11] *(thunk)* и представляет собой оболочку для функции, служащую вспомогательным средством для вызова другой функции. В контексте функционального характера языка JavaScript в переходники помещают выражение в виде анонимной функции без параметров, которое используется в качестве аргументов и позволяет отложить тем самым свое вычисление до тех пор, пока принимающая функция не вызовет анонимную функцию.

Подробное рассмотрение батутов и переходников выходит за рамки данной книги, но если вы отчаянно ищите пути оптимизации своих рекурсивных функций, начните поиск с более основательного изучения этих средств. А проверить совместимость оптимизации хвостовых вызовов с другими языковыми средствами в стандарте ES6 можно на веб-сайте по адресу https://kangax.github. io/compat-table/es6/.

Если требуется организовать непрерывный цикл визуализации графики или обработку крупных массивов данных за короткий период времени, то на первый план выходит производительность. В подобных случаях приходится идти на вынужденные компромиссы, жертвуя изящностью кода в пользу быстрого выполнения поставленной задачи. Для этой цели рекомендуется придерживаться стандартных циклов, хотя для большинства прикладных потребностей функциональное программирование по-прежнему остается весьма производительной методикой написания кода. Оптимизацию прикладного кода следует всегда выполнять в последнюю очередь, а в некоторых крайних случаях, когда требуется повысить производительность, сэкономив хотя бы немного миллисекунд, можно всегда воспользоваться методиками, представленными в этой главе.

Всякому решению при разработке программного обеспечения противоборствует равнозначная сила. Но при разработке большинства приложений жертвование эффективностью в пользу удобства сопровождения считается вполне допустимым компромиссом. Ведь лучше написать код, который удобно читать и отлаживать, даже если он и не самый быстродействующий. Как сказал Дональд Кнут: “В 97 случаях из 100 экономия нескольких миллисекунд при оптимизации написанного кода не имеет особого значения, особенно в сравнении с той ценностью, которую приносит написание удобного для сопровождения кода”.

---

[11] **В простонародье** — *санками. — Примеч. ред.*

Парадигма функционального программирования имеет вполне законченный характер. Она обеспечивает богатый уровень абстракции и переадресации, намечая интересные пути для достижения эффективности. До сих пор пояснялось, как создавать функциональные программы с линейными потоками данных посредством связывания в цепочку и композиции. Но, как известно, программы HaJavaScript сочетают в себе немало нелинейного или асинхронного поведения, например, при обработке вводимых пользователем данных или составлении удаленных HTTP-запросов. Разрешению подобных затруднений посвящена глава 8, "Обработка асинхронных событий и данных", в которой рассматривается также парадигма реактивного программирования, построенная на принципах функционального программирования.

Резюме

Из этой главы вы узнали следующее.

- В некоторых случаях функциональный код может работать медленнее или потреблять больше оперативной памяти, чем равнозначный ему императивный код.
- Реализуя стратегию отложенного вычисления, можно выгодно воспользоваться комбинатором чередования функций и соответствующей поддержкой в функциональных библиотеках вроде Lodash.
- Запоминание может быть использовано как стратегия внутреннего кеширования на уровне функций с целью исключить дублирование вычисления потенциально затратных функций.
- Декомпозиция (или разбиение) программ на простые функции позволяет не только создавать расширяемый код, но и повышать его эффективность через запоминание.
- Декомпозиция распространяется также на рекурсию как методика решения сложной задачи путем ее разбиения на более простые самоподобные задачи, где возможности запоминания используются в полной мере с целью оптимизировать применение стека контекста выполнения.
- Преобразуя функции в хвостово-рекурсивную форму, можно выгодно воспользоваться исключением хвостовых вызовов как одним из средств оптимизации, встроенной в компилятор.

Обработка асинхронных событий_и_данных

В этой главе...

- Выяснение трудностей, возникающих при написании асинхронного кода
- Применение методик ФП для исключения вложенных обратных вызовов
- Рационализация асинхронного кода с помощью обязательств
- Генерирование данных по требованию с помощью генераторов
- Введение в реактивное программирование
- Написание управляемого событиями кода средствами реактивного программирования

> **Программирующие функционально извлекают из этого стиля немало материальных выгод, считая его на порядок более продуктивным, чем обычный стиль, поскольку функциональные программы на порядок короче обычных.**
>
> Из статьи *"О важности функционального программирования"* (Why Functional Programming Matters) *Джона Хьюза (John Hughes)*[1]

До сих пор пояснялось, как мыслить функционально и пользоваться методиками ФП для написания, тестирования и оптимизации прикладного кода HaJavaScript. Все эти методики предназначены для борьбы с трудностями,

[1] **Из издания** *Research Topics in Functional Programming* **под редакцией Д. Тернера (D. Turner), стр. 17-42 (издательство Addison-Wesley, 1990 г.); см. также по адресу** `https://www.es.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf`.

присущими веб-приложениям средних и крупных масштабов и способными существенно затруднить их сопровождение. Много лет назад взаимодействие с веб-приложениями ограничивалось передачей серверу данных из больших веб-форм и выводом в ответ целых веб-страниц. Но по мере развития вебприложений повышались и требования к ним. Ныне все пользователи ожидают, что страницы будут вести себя в большей степени подобно платформенноориентированным приложениям, реагирующим на действия пользователей в реальном масштабе времени.

В клиентском коде HaJavaScript количество затруднений, которые приходится преодолевать разработчикам, оказывается еще больше, чем в любой другой среде. На это непосредственное влияние оказывает появление объемного клиентского кода, в котором не только должны быть учтены особенности межплатформенного взаимодействия с программным обеспечением веб-сервера, но и эффективная обработка вводимых пользователем данных, взаимодействие с удаленными сайтами по технологии AJAX и одновременное отображение данные на экране. В данной книге предлагается решение, выработанное средствами ФП, что идеально подходит для систем, где требуется поддерживать высокий уровень целостности, несмотря на все эти трудности.

В этой главе возможности ФП применяются для разрешения реальных трудностей программирования HaJavaScript, связанных с асинхронными потоками данных, где код не является линейным с точки зрения выполнения программы. В некоторых из рассматриваемых здесь примеров применяется браузерная технология вроде AJAX и обрабатываются запросы из локального хранилища. При этом преследуется главная цель — продемонстрировать применение ФП вместе с обязательствами, появившимися в стандарте ES6, а также возможности реактивного программирования. Все это позволяет превратить запутанный код, состоящий из вложенных обратных вызовов в изящные и удобочитаемые выражения. Реактивное программирование может показаться вам знакомым, поскольку оно представляет собой способ осмысления решаемых задач, близко наследующий то, что принято в функциональном программировании.

Добиться асинхронного поведения нелегко. В отличие от обычных функций, асинхронные функции не могут просто возвратить данные вызывающему коду. Вместо этого в них используется бесславный шаблон обратных вызовов, извещающий о завершении длительных вычислений, операций извлечения информации из базы данных или обработки удаленных вызовов по сетевому протоколу HTTP. Обратные вызовы применяются также для обработки в браузерах событий от щелчков кнопками мыши, нажатий клавиш и жестов на сенсорных экранах мобильных устройств в ответ на действия пользователя. Для реагирования на подобные события после запуска программы на выполнение необходимо написать код, вызывающий немало трудностей для функционального проектирования, отличный от кода, где поступление данных ожидается в предсказуемые моменты времени. В конечном счете нужно найти ответ на следующий вопрос: как составлять или связывать в цепочку функции, призванные обеспечить то, что должно произойти в будущем?

8.1. Трудности, возникающие при написании асинхронного кода

Современные программы HaJavaScript редко выполняют загрузку данных, получаемых

от сервера в ответ на одиночный запрос. Зачастую данные загружаются на страницу по мере необходимости по многим асинхронным запросам в ответ на потребности пользователей. Простым тому примером служит клиентское приложение для чтения электронной почты. И хотя во входящей почте могут быть тысячи длинных сообщений, клиентам как правило нужно просмотреть и обработать только самые свежие из них. Очевидно, что нет никакого смысла ожидать несколько секунд (или даже минут) пока загрузится вся входящая почта. Разработчикам веб-приложений на JavaScript приходится часто разрешать затруднения подобного рода, и для этого обычно требуется реализовывать в той или иной форме неблокирующие асинхронные вызовы, в которых могут возникать следующие трудности.

- Создание временных зависимостей между функциями.
- Неизбежное обращение к пирамиде обратных вызовов.
- Несовместимое сочетание синхронного и асинхронного кода.

8.1.1. Создание временных зависимостей между функциями

Рассмотрим в качестве примера функцию get JSON (), предназначенную для выполнения AJAX-запроса по извлечению из сервера списка объектов, представляющих учащихся. Как показано на рис. 8.1, это асинхронная функция, и поэтому она возвратит управление программе, как только запрос будет отправлен, после чего вызывается функция showStudents (). Но в этот момент объект students по-прежнему остается пустым (null) из-за того, что удаленный запрос еще не обработан. В таком случае единственный способ обеспечить наступление событий в правильном порядке — создать *временную зависимость* между асинхронным кодом и следующим предпринимаемым действием. Чтобы

```
var students = null;
getJSON('/students', function(studentObjs) { students = studentObjs; к.
   },
   function (errorObj) {
      console.log(errorObj.message);
   }
);

showStudents(students);
```

Этот поток временно прерывается ?> из-за того, что объект `student` не инициализируется вовремя

Рис. 8.1. Этот код обладает существенным недостатком. Дело в том, что данные требуется извлечь асинхронно, и поэтому объект students так и не будет содержать нужных данных в момент отображения таблицы со списком учащихся выполнить функцию showStudents () вовремя, ее придется включить в состав функции обратного вызова.

Временное связывание или *временное сцепление* возникает при выполнении определенных, логически сгруппированных функций. Это делается в том случае, когда функциям приходится ожидать, когда данные окажутся доступными или будут выполнены другие функции. Зависимость от данных или времени, как в рассматриваемом здесь случае, может вызвать побочные эффекты.

Удаленные операции ввода-вывода выполняются заметно медленнее, чем остальная часть кода, и поэтому их выполнение поручается неблокирующим процессам, способным запросить данные и “ожидать” возврата надлежащего ответа. Как только данные будут

получены, вызывается предоставляемая пользователем функция обратного вызова. Именно это и делает функция get JSON (), как подробно демонстрируется в листинге 8.1.

Листинг 8.1. Функция getJSON (), в которой применяется платформенно-ориентированный объект типа XMLHttpRequest

```
const getJSON = function (url, success, error) {
    let req = new XMLHttpRequest();
    req.responseType = 'json';
    req.open('GET', url);
    req. onload = functionO {
       if(req.status -= 200) {
           let data = JSON.parse(req.responseText);
           success(data);
       }
       else {
           req.onerror();
       }
    }
    req.onerror = function () {
        if(error) {
            error(new Error(req.statusText));
        }
    };
    req.send();
```

В

Функции обратного вызова нередко применяются в коде JavaScript. Но они с трудом поддаются масштабированию, когда требуется последовательно загрузить больше данных, что в конечном итоге приводит к использованию распространенного шаблона обратных вызовов.

8.1.2. Неизбежное обращение к пирамиде обратных вызовов

Обратные вызовы применяются в основном для того, чтобы исключить блокировку пользовательского интерфейса при ожидании завершения длительных процессов. Функции, которым передается функция обратного вызова, вместо возврата значения, реализуют следующую форму *инверсии управления'*.

"Не звони мне, я сам тебе позвоню". Как только наступает событие, например, когда данные становятся доступными или же пользователь щелкает на экранной кнопке, вызывается функция обратного вызова с запрашиваемыми данными, что дает возможность выполнить синхронный код:

```
var students = null;
getJSON('/students',
     function(students) {
          showStudents(students);
     },
     function (error) {
          console.log(error.message);
     }
) ;
```

Если возникнет ошибка, то вызывается соответствующая функция обратного вызова, предоставляющая возможность выдать сообщение об ошибке и устранить ее. Но такая инверсия управления противоречит замыслу функциональных программ, где функции должны быть независимы друг от друга и должны немедленно возвращать значения в вызывающий код. Как утверждалось ранее, данная ситуация ухудшается, если во вложенные уже обратные вызовы требуется ввести логику, носящую более асинхронный характер.

Чтобы продемонстрировать это, рассмотрим несколько более сложный случай. Допустим, что после извлечения списка учащихся из сервера требуется также извлечь оценки, но только тех учащихся, которые проживают в Соединенных Штатах. Затем эти данные сортируются по номеру социального страхования и отображаются на HTML-странице, как показано в листинге 8.2.

Листинг 8.2. Реализация вложенных вызовов функции get JSON (), причем у каждого из них имеются свои функции обратного вызова для удачного завершения и для обработки ошибки

```
getJSON('/students', function (students) {
                                        ◄ ----
          students.sort(function(a, b){
               if(a.ssn < b.ssn) return -1; if(a.ssn > b.ssn) return 1; return 0;

          for (let i = 0; i < students.length; i++) { student = students[i];
               let if              (student.address.country === 'US') {
                    getJSON('/students/?{student.ssn}/grades',
                         function (grades) { --------◄
                         showStudents(student, average(grades));
                    }, function (error) {--- <
                         console.log(error.message);
                    }
               } г
                    function (error) {
                         console.    log    (error    .message)    ;
     I
);
```

Первый уровень вложения при первом AJAX-запросе; указана функция обратного вызова, которая вызывается при удачном исходе

После получения списка учащихся отобразим успеваемость тех из них, кто проживает в США. Для этого нам нужно изменить данную функцию так, чтобы она отображала таблицу успеваемости на веб-странице построчно '

Второй уровень вложения для извлечения данных об успеваемости каждого учащгося; указаны собственные обратные вызовы для обработки удачного и неудачного исхода обработки запроса

Первый уровень вложения при первом AJAX-запросе; указана функция обратного вызова, которая вызывается в случае возникновения ошибки

До чтения этой книги приведенный выше исходный код, вероятнее всего, выглядел для вас вполне приемлемым, но теперь, когда вы усвоили принципы ФП, он должен выглядеть беспорядочным и запутанным (полностью функциональный вариант этого кода будет представлен далее). То же самое происходит и при обработке ошибок. Так, в исходном коде программы из листинга 8.3 AJAX-вызовы чередуются с обработкой вводимых пользователем данных. При этом обрабатываются события от щелчков кнопками и перемещения мыши, извлекается несколько фрагментов данных из сервера, а затем обработанные данные воспроизводятся в модели DOM.

Листинг 8.3. Программа для извлечения записей об учащихся из сервера по номеру социального страхования

```
var _selector = document.queryselector;
_selector('#search-button').addEventListener('click',
    function (event) {
      event.preventDefault();

      let ssn = _selector('#student-ssn').value;
      if(!ssn) {
         console.log('WARN: Valid SSN needed!');
         return;
      }
      else {
        getJSON('/students/${ssn}', function (info) {
           -Selector('#student-info').innerHTML = info;
           _selector('#student-info').addEventListener('mouseover', functionO {
                  getJSON('/students/${info.ssn}/grades', function (grades) {
                      // обработать список оценок учащихся
                      });
              }) ;
        })
        .fail(function() {
            console.log('Error occurred!');
        }) ;
      }
});
```

И этот исходный код трудно понять. Как видите, вложение последовательности обратных вызовов быстро превращает его в код, напоминающий пирамиду, приведенную на рис. 8.2. Иногда данная ситуация называется “ужасом обратных вызовов” или “рождественской елкой судного дня” и характеризует программы, в которых выполняется немало асинхронного кода и происходит взаимодействие пользователя с моделью DOM.

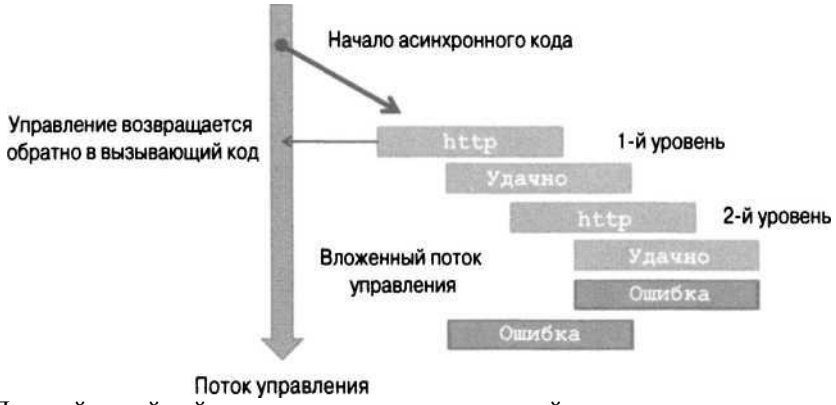


Рис. 8.2. Простой линейный поток управления программой сводится к удаленному вызову и вырождается в водопад вызовов вложенных функций, т.е. в пирамиду, расширяющуюся по горизонтали подобно "рождественской елке судного дня"

Когда программа начинает принимать подобную форму, приходится полагаться на отступы и синтаксическую организацию, например, логичное группирование операторов, чтобы улучшить удобочитаемость исходного кода. А теперь выясним, каким образом умение мыслить функционально может помочь в подобной ситуации.

8.1.3. Применение стиля передачи продолжений

В листинге 8.3 приведен еще один пример программы, которая не была подвергнута надлежащей декомпозиции. Вложенные функции обратного вызова не только с трудом поддаются пониманию, но и создают замыкания, охватывающие их собственную область видимости, а также область видимости переменных в тех функциях, в которые они вложены. Единственная причина для вложения одной функции в другую возникает в том случае, если ей требуется непосредственный доступ к внешним переменным, чтобы выполнить свое назначение. Но в данном случае внутренняя функция обратного вызова, обрабатывающая все оценки, по-прежнему ссылается на ненужные внешние данные. Чтобы сделать рассматриваемый здесь исходный код более удобным для чтения и понимания, можно, в частности, прибегнуть к стилю *передачи продолжений,* Именно в этом стиле исходный код программы из листинга 8.3 реорганизован в код из листинга 8.4.

Листинг 8.4. Версия программы для извлечения записей об учащихся, реорганизованная в стиле передачи продолжений

```
var _selector = document.querySelector;

  selector('#search-button').addEventListener(
            'click', handleMouseMovement);

var processGrades = function (grades) {
     // обработать список оценок данного учащегося };

var handleMouseMovement = () =>
     getJSON('/students/${info.ssn}/grades', processGrades);
```

```
var showStudent = function (info) {
    -Selector('#student-info*).innerHTML = info;
    _selector('#student-info').addEventListener(
              'mouseover', handleMouseMovement); };

var handleError = error =>
    console.log('Error occurred' + error.message);

var handleClickEvent = function (event) { event.preventDefault();

    let ssn = _selector('#student-ssn').value; if(!ssn) {
       alert('Valid SSN needed!');
       return;
    }
    else {
       getJSON('/students/${ssn}', showStudent).fail(handleError);
    } B
```

Единственно, что было сделано в этой версии программы, — внутренние обратные вызовы выделены в отдельные функции или лямбда-выражения. Передача продолжений — это стиль программирования, применяемый для написания неблокирующих программ, побуждающий к разделению программы на отдельные составляющие. Именно по этой причине данный стиль является промежуточной формой для перехода к функциональному программированию. В данном случае функции обратного вызова называются *текущими продолжениями,* которые предоставляются вызывающими их фрагментами кода по возвращаемому значению. Важное преимущество стиля передачи продолжений заключается в его эффективности с точки зрения стека контекста исполнения (подробнее о разновидностях стеков BjavaScript см. в главе 7, "Оптимизация функционального кода"). Если программа написана полностью в стиле передачи продолжений, как, например, в листинге 8.4, то продолжение в других функциях очистит контекст текущей функции и подготовит новый контекст для поддержки функции, продолжающей поток выполнения программы, причем каждая функция, по существу, представлена в хвостовой форме.

Применяя продолжения, можно также устранить затруднение в исходном коде из листинга 8.2, возникающее при чередовании синхронного поведения с асинхронным. Трудным участком кода в данном случае является вложенный цикл, в котором формируются AJAX-запросы, чтобы извлечь оценки каждого учащегося и вычислить среднюю оценку: `for (let i = 0; i < students.length; i++) { let student = students[i]; if (student.address.country === 'US') {`

```
        getJSON('/students/${student.ssn}/grades',
            function (grades) {
                showStudents(student, average(grades));
            },
            function (error) {
                console.log(error.message);
            }
        );
    }
```

```
}
```

На первый взгляд, приведенный выше код вполне работоспособен и выводит имена учащихся Alonzo Church и Haskell Curry с соответствующими сведениями о них (в данном коде применяется HTML-таблица для вывода всех данных о каждом учащемся, но на ее месте мог быть файл или база данных). Тем не менее выполнение данного кода дает результат, приведенный на рис. 8.3.

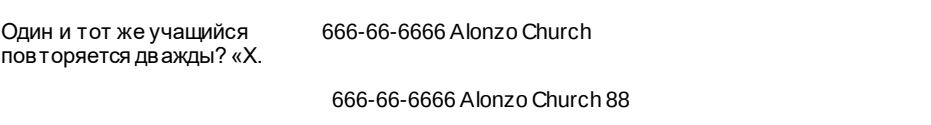


Рис. 8.3. Результат выполнения ошибочного императивного кода, где асинхронные функции сочетаются с синхронным циклом. И хотя в нем извлекаются удаленные данные, при вызове функции всегда происходит обращение к последней итеративной записи об учащемся (в ее замыкании), которая выводится несколько раз

Очевидно, что это не тот результат, который предполагалось получить. В частности, почему сведения об одном и том же учащемся выводятся дважды? Ошибка обусловлена применением синхронного языкового средства (в данном случае — цикла), чтобы выполнить асинхронную функцию get JSON (). В цикле неизвестно, что приходится ожидать завершения функции getJSON (). Независимо от применения ключевого слова let для определения переменных с блочной областью видимости, при всех внутренних вызовах функции showStudents (student, average (grades)) в ее замыкании доступна ссылка на объект, представляющий учащегося, и поэтому отображается одна и та же запись об учащемся. Как пояснялось в главе 2, “Сценарий высшего по

рядка", при обсуждении недостатков цикла, связанных с неоднозначностью, это лишний раз свидетельствует о том, что замыкание является не копией ее объемлющего окружения, а фактической ссылкой на нее. Обратите, однако, внимание на то, что столбец с оценкой по-прежнему воспроизводится правильно. Объясняется это тем, что извлекаемое значение надлежащим образом передается функции обратного вызова путем связывания правильного значения с параметром данной функции.

Как следует из главы 2, чтобы разрешить данное затруднение, следует ввести надлежащим образом объект student в область видимости той функции, где составляется AJAX-запрос. Воспользоваться для этой цели стилем передачи продолжений, как и прежде, не так-то просто, поскольку вложенная функция обратного вызова для обработки оценок зависит и от самого объекта student. Напомним, что это побочный эффект. Чтобы восстановить продолжение, придется призвать на помощь карринг, упоминавшийся в главе 4, "На пути к повторно используемому, модульному коду". С его помощью можно связать входные и выходные данные функции, как показано ниже.

```
const showStudentsGrades = R.curry(function (student, grades) {
    (student, average (grades) ) ; < ----------
});
```

Карринг позволяет преобразовать данную функцию в унарную

Функция appendData () добавляет строки к HTML-таблице

```
const handleError = error => console.log(error.message);

const processstudent = function (student) {
    ] : (student.address.country === 'US') {
        getJSON('/students/${student.ssn}/grades'
         showStudentsGrades(student), handleError)
};

for (let i = 0; i < students.length; i++) {
    processstudent(students[i]);
```

Каррированная функция showStudentsGrades(student) в конечном итоге вызывается обратно с данными оценок

Объект student, по существу, фиксируется в своем замыкании, когда он передается функции для обработки в цикле

В этой новой версии рассматриваемого здесь примера кода получаются правильные результаты (рис. 8.4).

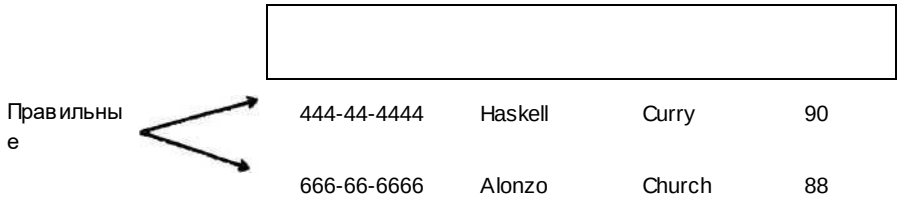


Рис. 8.4. Когда текущий объект student передается функции в качестве параметра, надлежащим образом устанавливается ее замыкание и разрешается неоднозначность, возникающая в результате выполнения удаленных вызовов в цикле

Принятие стиля передачи продолжений помогает устранить временную зависимость в прикладном коде, а также превратить асинхронный поток в линейно вычисляемую последовательность функций — и то и другое хорошо. Но если кто-нибудь посторонний попытается прочитать приведенный выше исходный код, не зная его особенностей, то у него может возникнуть недоуменный вопрос: почему функции не выполняются

своевременно? Для этого столь длительные операции необходимо превратить в объекты первого класса в своих программах.

8.2. Достижение асинхронного поведения объектов первого класса с помощью обязательств

В предыдущем примере исходный код был заметно усовершенствован по сравнению с императивными примерами асинхронных программ, демонстрировавшимися в начале этой главы. Тем не менее этот код еще далек от того, чтобы считать его функциональным. В любой функциональной программе можно, среди прочего, обнаружить следующие качества.

- Применение композиции и приемов бесточечного программирования.
- Сведение вложенной структуры к более линейному потоку.
- Абстрагирование понятия временной связи до такой степени, чтобы оно не доставляло больше никаких хлопот.
- Объединение обработки ошибок в единой функции вместо многих обратных вызовов по ошибке, чтобы исключить подобный стиль написания кода.

Всякий раз, когда речь заходит о сведении структур, композиции и объединении поведения, на ум должен приходить подходящий проектный шаблон. Похоже, что в данном случае лучше всего подходит монада, поэтому рассмотрим монаду типа Promise. Чтобы стало понятнее назначение этой монады в общих чертах, достаточно представить, что она заключает в оболочку длительное вычисление (это не интерфейс Promise, но близкая к нему аналогия):

```
Promise. of (<длительное вычислением .tap (функцияj) .tap (функция J ;
    // -> Promise(результат)
```

В отличие от других монад, рассмотренных ранее в этой книге, обязательствам известно, что нужно "подождать" завершения длительной операции, прежде чем выполнить преобразуемые функции. Подобным образом в этом типе данных непосредственно разрешается затруднение, связанное с запаздыванием в асинхронных вызовах. Подобно документным функциям в монадах типа Maybe и Either с неопределенными возвращаемыми значениями, обязательства делают ясным и прозрачным представление об ожидании данных. Они обладают также тем преимуществом, что предоставляют более простую альтернативу выполнению, композиции и организации асинхронных операций по сравнению с традиционными подходами, основанными на обратных вызовах.

Используя обязательства, можно заключить обрабатываемое значение или функцию в оболочку будущего действия (для программирующих Hajava это аналогично объекту типа Future<V>). Длительная операция может потребовать сложных вычислений, извлечения записей из базы данных или информации из сервера, чтения содержимого файла и т.д. А на случай сбоя обязательства позволяют объединить всю логику обработки ошибок, применяя такие же подходы, как и в монадах типа Maybe и Either. Аналогичным образом обязательство может предоставить сведения о состоянии выполняемого задания, отвечая на

следующие вопросы: были ли данные извлечены успешно и возникли ли какие- нибудь ошибки во время данной операции?

Как показано на рис. 8.5, в любой момент времени обязательство может находиться в одном из следующих состояний: ждущее, выполненное, отклоненное или разрешенное. Вначале обязательство находится в *ждущем* состоянии, иначе называемом *неразрешенным.* В зависимости от результата выполнения длительной операции обязательство может перейти в *выполненное* состояние, если вызвана функция resolve (), или же в *отклоненное* состояние, если вызвана функция reject (). И как только обязательство будет выполнено, оно может известить другие объекты (продолжения или обратные вызовы) о том, что ожидаемые данные поступили, а если возникнут ошибки — вызвать любую зарегистрированную в нем функцию обратного вызова. В этот момент обязательство переходит в *разрешенное* состояние.

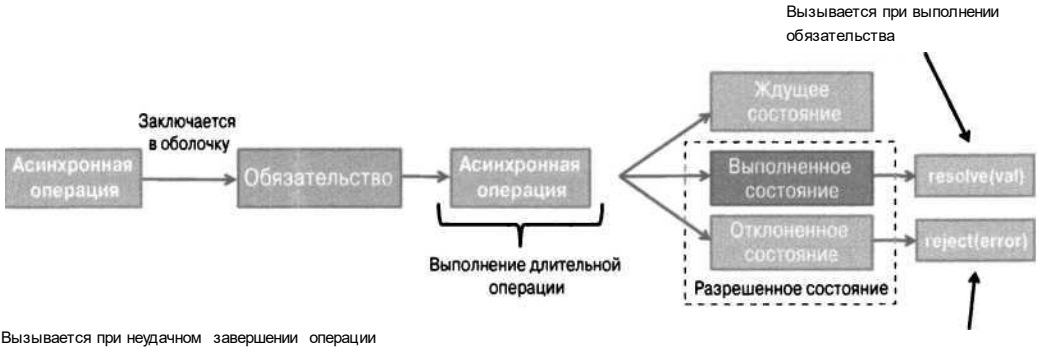


Рис. 8.5. Порядок заключения операции в оболочку монады типа Promise и предоставления двух обратных вызовов: одного — для выполнения функции resolve (), а другого — для выполнения функции reject (). Вначале обязательство находится в ждущем состоянии, а затем переходит в выполненное или отклоненное состояние. Для этого вызываются функции resolve () или reject () соответственно, после чего обязательство переходит в разрешенное состояние

Обязательства дают возможность более эффективно осмысливать программы, распутывая гордиевы узлы тесно связанных обратных вызовов. Подобно тому как монада типа Maybe применялась для исключения целого ряда вложенных условных операторов if-else при проверках наличия пустых (null) значений в прикладном коде, монада типа Promise может быть использована для преобразования целого ряда вложенных обратных вызовов функций в последовательность действий аналогично монадному функтору tap ().

В ES6 был принят открытый стандарт Promises/A+ на протокол взаимодействия обязательств в коде JavaScript с браузерами разных производителей. Справочный документ по этому стандарту можно найти по адресу https:// promisesaplus. com. Настоятельно рекомендуется прочитать его, чтобы ознакомиться с особенностями данного протокола и соответствующей терминологией. На самом элементарном уровне объект типа Promise можно построить следующим образом:

```
var fetchData = new Promise(function (resolve, reject) {

    // извлечь данные или выполнить длительное вычисление
```

```
    if (<success>) {
        resolve(result);
    }
    else {
        reject(new Error('Error performing this operation!'));
    } });
```

Конструктору объектов обязательств передается в качестве параметра единственная функция, называемая функцией *действия* и заключающая асинхронную операцию в оболочку. Этой функции, в свою очередь, передаются две функции обратного вызова resolve () и reject (), которые можно рассматривать как своего рода продолжения. Эти функции вызываются в тех случаях, когда обязательство выполнено или отклонено соответственно. Ниже приведен краткий пример применения обязательств в сочетании с простым объектом типа Scheduler из главы 4, “На пути к повторно используемому, модульному коду”. Подобно монадному функтору тар (), обязательства предоставляют механизм для преобразования значения, которое пока еще не существует, но предполагается в будущем.

```
var Scheduler = (function () {
    let delayedFn = _.bind(setTimeout, undefined, _, _);

    return {
        delay5:    partial(delayedFn, _, 5000),
        delay10:   partial(delayedFn, _, 10000),
        delay: _.partial(delayedFn, _, _)
```

Запланировать задержанный вызов функции, чтобы сымитировать var

```
promiseDemo = new Promise (function (resolve, reject) {
    Scheduler.delay5(function () {
        resolve (' Done ! ') ;
```

длительную операцию

Разрешить обязательство

Обязательство разрешается через5секунд

```
promiseDemo. then (function (status) {
    console. log ('After 5 seconds, the status is: ' + status);
```

8.2.1. Цепочки будущих методов

В объекте типа Promise определяется метод then() аналогично методу fmap () функтора. Он определяет операцию для значения, возвращаемого в обязательстве, заключая его обратно в объект типа Promise. Аналогично методу Maybe.map (f), метод Promise. then (f) может быть использован для того, чтобы связывать в цепочку операции преобразования данных, а также соединять функции во времени, абстрагируя употребление временного связывания функций. Подобным образом можно связать в линейную цепочку несколько уровней зависимого асинхронного поведения, не создавая новые вложенные уровни (рис. 8.6).

Каждая операция вызывается синхронно

Методу then () передается два дополнительных параметра: обратный вызов при удачном

Рис. 8.6. Последовательность обязательств, связанных в цепочку через метод then ().

Каждый вызов метода then() производится последовательно с одним обещанным значением после другого, как только будет выполнено каждое обязательство

завершении операции, а также при возникновении ошибки. Идеальным для подробного извещения об ошибках считается предоставление обратных вызовов для обработки ошибок в каждом блоке с методом then (). Но в то же время можно воспользоваться последовательностью обратных вызов при удачном завершении, перенеся всю логику обработки ошибок в единый метод catch (), вызываемый в самом конце. Но, прежде чем приступить к связыванию обязательств в цепочку, реорганизуем исходный код функции get JSON (), чтобы выгодно воспользоваться объектом типа Promise и тем самым реализовать так называемое *исполнение обещания функции* (см. листинг 8.5).

Листинг 8.5. Исполнение обещания функции getJSON ()

```
var getJSON = function (url) {
    return new Promise(function(resolve, reject) {
```

```
let req = new XMLHttpRequest() ;
req.responseType = 'json';
req.open('GET', url);
req.onload = functionO {
    if(req.status == 200) {
        let data = JSON.parse(req.responseText);
        resolve(data);
    } else {
        reject (new Error (req. statusText) ) ;
    }
};
req.onerror = function () {
    if(reject) {
        reject (new Error ('IO Error' ));
    }
};

req.send();
```

Вызвать при возврате из функции обработки AJAX-запроса

Разрешить обязательство при удачном ответе (коде ответа 200)

Отклонить обязательство, если получен код ответа, отличающийся от 200, или возникла ошибка при установлении соединения

Отправить удаленный запрос

Исполнение обещаний прикладных интерфейсов API считается подходящей нормой практики. Оно намного упрощает работу с прикладным кодом по сравнению с традиционными обратными вызовами. А поскольку обязательства предназначены для заключения в оболочку любой длительной операции, а не только извлечения данных, то их можно применять вместе с любым объектом, *реализующим* метод then (). Вскоре обязательства будут внедрены в функции из всех библиотекДауаЗспрС

Поддержка обязательств в библиотеке jQuery

Если вы пользуетесь библиотекой jQuery, то, вероятнее всего, уже имели дело с обязательствами. Так, в результате выполнения операции $.getJSON (и любой разновидности операции $.ajax) из библиотеки JQuery возвращается ее собственный объект типа Deferred (нестандартный вариант объекта типа Promise), реализующий интерфейс Promise и метод then(). Следовательно, чтобы интерпретировать этот объект как объект типа Promise, можно воспользоваться методом Promise. resolve () следующим образом:

```
Promise.resolve($.getJSON('/students')).then(function () ...);
```

Теперь этот объект реализует метод then () и может быть использован как любой другой исполненный обещанием объект. Чтобы продемонстрировать процесс реорганизации обращения к интерфейсу API для применения обязательств, в листинге 8.5 была представлена авторская версия операции getJSON.

Рассмотрим сначала простой пример, в котором данные об учащемся извлекаются из сервера с помощью приведенной ниже новой версии функции getJSON (), созданной на основе обязательств, а затем внедрим его вызов в код извлечения оценок, чтобы продемонстрировать связывание обязательств в цепочку.

```
getJSON('/students').then(
    function(students) {
        console.log(R.map(student => student.name, students));
    },
    function (error) {
        console.log(error.message);
    } ) ;
```

А теперь реорганизуем исходный код из листинга 8.2, заменив в нем передачу

продолжений более совершенным решением на основе обязательств. Исходный код листинга 8.2 приводится ниже для справки еще раз.

```
getJSON('/students',
    function (students) {
        students.sort(function(a, b) {
            if(a.ssn < b.ssn) return -1;
            if(a.ssn > b.ssn) return 1;
            return 0;
        });
        for (let i = 0; i < students.length; i++) { let student = students[i];
            if (student.address.country === 'US') {
                   getJSON('/students/${student.ssn}/grades',
                     function (grades) {
                     showStudents(student, average(grades));
                     },
                     function (error) {
                           console.log(error.message);
                     });
            }

        }
    },
    function (error) {
        console.log(error.message);
    } ) ;
```

В исходный код из листинга 8.6, реализующий функциональный подход, внесены следующие изменения.

- Вместо вложения асинхронные вызовы связываются в цепочку с помощью метода then (), а для абстрагирования асинхронных частей кода применяется монада типа Promise.
- Все объявления переменных и модификации заменены лямбда-выражениями.

- Каррированные функции из библиотеки Ramda выгодно использованы для выполнения лаконичных операций обработки данных вроде сортировки, фильтрации и преобразования.
- Логика обработки ошибок сосредоточена в конечной функции перехвата вызовов.
- Данные заключены в оболочку монады типа 10, чтобы их можно было записывать в модели DOM без побочных эффектов.

Листинг 8.6. Извлечение записи об учащемся и данных о его оценках с помощью асинхронных вызовов

Скрыть счетчик. Данная функция не возвращает значение, и поэтому значение, заключаемое в оболочку обязательства, передается далее

Удалить учащихся, не проживающих в США

Отсортировать оставшиеся объекты по номеру социального страхования

Преобразовать очередной запрос типа get JSON для извлечения оценок каждого учащегося. Отдельные объекты обязательств отвечают за результат для каждого извлеченного объекта учащегося

Использовать комбинаторы функций вместе с функциями из библиотеки Ramda для вычисления средних оценок

Использовать монаду типа 10 для добавления данных об учащемся и его оценках к модели DOM

```
getJSON('/students')
    .then(hide('spinner'))
    .then(R.filter(s =>
s.address.country == 'US'))
    .then(R.sortBy(R.prop('ssn')))
    . then (R.map (student => {
        return getJSON('/grades?ssn=' + student.ssn)
    }) .then(R.compose(Math.ceil,
            fork(R.divide, R.sum, R.length)))
        .then(grade =>
            10.of(R.merge(student, {'grade':
            grade})) .map(R.props(['ssn',
            'firstname', 'lastname', 'grade']))
            .map(csv)
                .map(append('#student-info')).run())

    .catch(function(error) {
        console.log('Error occurred: ' + error.message);
```

Поскольку обязательства скрывают подробности обработки асинхронных вызовов можно создавать программы, которые внешне ведут себя так, как будто функции выполняются друг за другом. При этом не тратится время на ожидание их выполнения и вы даже не будете подозревать, что данные запрашиваются из внешнего сервера. И хотя обязательства скрывают асинхронный поток, в них делается акцент на представлении о времени с помощью метода then (). Иными словами, вызов функции get JSON (url) можно легко заменить вызовом к локальному хранилищу, например get JSON (db), в котором исполняются обещания функции, и прикладной код будет действовать точно так же. Такая степень гибкости называется *прозрачностью местоположения.* Обратите также внимание на то, что рассматриваемый здесь код написан в бесточечном стиле. Его поведение наглядно показано на рис. 8.7.

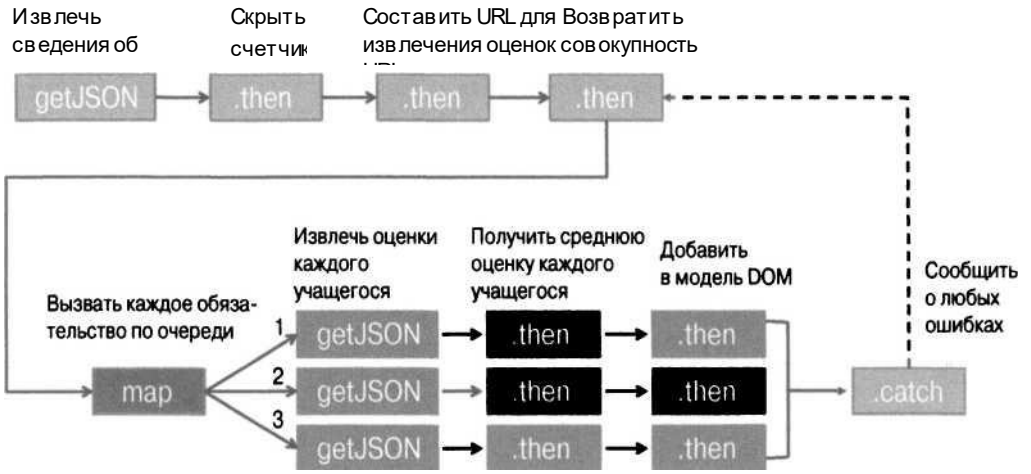


Рис. 8.7. Продвижение поведения через соединяемые в цепочку обязательства. Каждый блок с методом then () содержит функцию, преобразующую проходящие через него данные. И хотя данный код не содержит программных ошибок и обладает всеми функциональными качествами, он неэффективен, поскольку в нем применяется водопадная последовательность запросов типа getJSON для извлечения оценок каждого учащегося

В исходном коде из листинга 8.6 сведения о каждом учащемся извлекаются и добавляются в модель DOM по очереди. Но поскольку операции для извлечения оценок выполняются по очереди, то в конечном итоге теряется какая-то часть драгоценного времени. У монады типа Promise имеется также возможность выгодно пользоваться возможностью браузера устанавливать несколько подключений к серверу, чтобы извлекать несколько элементов одновременно. В качестве примера рассмотрим немного иную задачу. Допустим, требуется вычислить среднюю оценку в одной и той же группе учащихся. В этом случае не имеет значения, в каком именно порядке извлекаются данные или поступают запросы. Следовательно, операции можно выполнять параллельно, воспользовавшись методом Promise. all (), приведенным в листинге 8.7.

Листинг 8.7. Одновременное извлечение нескольких элементов с помощью метода Promise. all ()

```
const average = R.compose(Math.ceil,                    I Выделить вычисление средней оценки в отдельную I
    fork(R.divide, R.sum, R.length));                   функцию, поскольку она используется неоднократно

getJSON (' / students' )                          Загрузить сведения об учащихся
    . then (hide (' spinner' ) )                  параллельно по всем URL
    .then(R.map(student => '/grades?ssn=' + student.ssn))
    .then(gradeUrls =>                                                  Воспользоваться монадой типа 10, чтобы
          Promise.all(R.map(getJSON, gradeUrls))) <--------             вывести полученные значения на консоль
    . then (R. map (average) ) «0 ■■■ ----- Вычислить общую среднюю оценку в классе
    . then (average) <--------- Вычислить среднюю оценку каждого учащегося
    .then(grade => 10.of(grade).map(console.log).run())                 <--
    .catch(error => console.log('Error occurred: ' + error.message));
```

В методе Promise.all () выгодно используется способность браузера загружать несколько элементов одновременно. Получающееся в итоге обязательство разрешается, как только будут разрешены все обязательства в итерируемом аргументе. В исходном коде из листинга 8.7 сочетаются вместе две основные составляющие функционального кода: разбиение программы на простые функции и последующая их композиция через монадический тип данных, управляющий поведением всей программы. На рис. 8.8 наглядно показано, что при этом происходит.

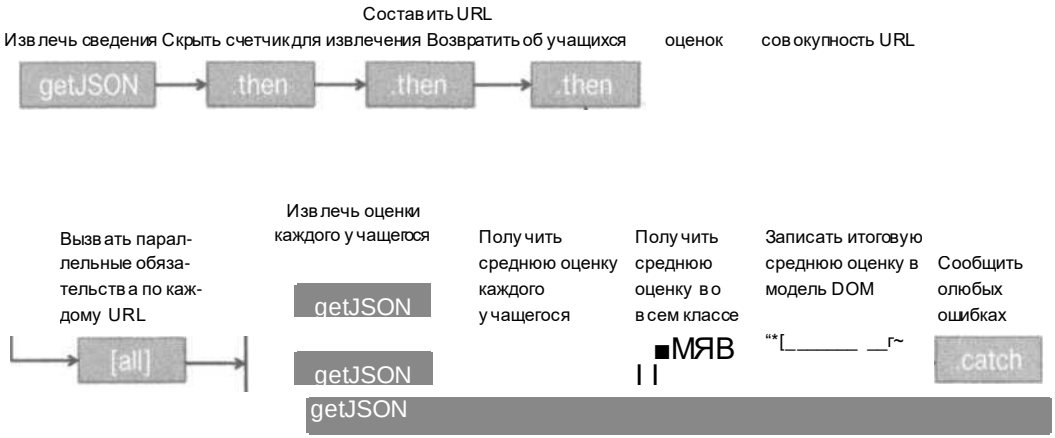


Рис. 8.8. Продвижение поведения через линейно соединяемые и параллельно используемые обязательства с помощью метода Promise. all (). В каждом блоке с методом then () содержится функция, преобразующая передаваемые ей данные. Эффективность данного кода объясняется тем, что в нем порождается несколько параллельных соединений для одновременного извлечения всех данных

Но монады оказываются эффективными не только для составления методов в цепочки. Как пояснялось в предыдущих главах, их эффективность проявляется и в композициях.

8.2.2. Композиция синхронного и асинхронного поведения

Если подумать о том, каким образом входные и выходные данные составляемых в композицию функций связываются вместе, то интуиция подсказывает, что такие функции должны выполняться линейно одна за другой. Но применяя обязательства, можно выполнять функции, разделяемые во времени, сохраняя в то же время внешний вид синхронной в остальном программы, состоящей из композиции функций. Чтобы такой подход к композиции синхронного и асинхронного поведения программы стал понятнее, поясним его на конкретном примере.

В примерах кода, приведенных ранее в данной книге, демонстрировалась синхронная версия функции findfdb, ssn) для реализации функции showStudent (). Чтобы упростить дело, в этих примерах функция find () считалась синхронной. А теперь реализуем ее асинхронную версию с помощью API типа IndexedDB, предназначенный для локального хранения объектов в браузере по определенному ключу (номеру социального страхования). Если этот

интерфейс вам еще незнаком, то и неважно. Ведь для реализации асинхронной версии функции find() будут использованы обязательства, как демонстрируется в листинге 8.8, где самое главное — понять, что если объект student существует, то обязательство будет разрешено с этим объектом, а иначе оно будет отклонено.

Листинг 8.8. Реализация асинхронного взаимодействия функции find O с локальным хранилищем в браузере

```
// find :: DB, String -> Promise(Student) const find = function (db, ssn) {
    let trans = db.transaction(['students'], 'readonly');
    const store = trans.objectstore('students;
    return new Promise (function (resolve, reject) {    i Заключить результат операции из-
        let request = store. get (ssn) ;          ◄----------1 влечения в оболочку обязательства
        request. onerror = functionO {
            if (reject) {                                                  Если найти нужный объект
                reject(new Error ('Student not found!'));<— в хранилище не удастся,
            }                                                              отклонить обязательство
        };
        request .onsuccess = functionO {  Если нужный объект найден, resolve (request.
            result) ;                  ◄ --- разрешить обязательство и
        у.                                   передать найденный объект
    }j.'                                     учащегося
```

В

В исходном коде из листинга 8.8 опущены подробности установки объекта db, поскольку они не имеют никакого отношения к обсуждаемому здесь вопросу. А о том, как инициализировать и применять интерфейс API типа IndexedDB, можно узнать из документации, доступной по адресу https://developer. mozilla.org/ru/docs/IndexedDB. Из этой документации следует, что все интерфейсы для чтения и записи данных в хранилище являются асинхронными, причем в них используется передача функций обратного вызова. Но как в таком случае составить композицию из функций, выполняющихся в разные моменты времени? До сих пор функция f ind () была синхронной. Правда, обязательства абстрагируют выполнение асинхронного кода до такой степени, что композиция функций с обязательствами, по существу, означает композицию функций в будущем с незначительными изменениями в коде. Но, прежде чем реализовать такой код, создадим ряд вспомогательных функций:

```
// fetchStudentDBAsync :: DB -> String -> Promise(Student) const
fetchStudentDBAsync = R.curry(function (db, ssn) {
    return find(db, ssn);
```

```
});

// findStudentAsync :: String -> Promise
const findStudentAsync = fetchStudentDBAsync(db)
```

Произвести карринг объекта хранилища данных, чтобы включить данную функцию композицию

В приведенной выше программе раскрывается истинный

```
// then :: f -> Thenable -> Thenable
const then = R.curry(function (f, thenable)
    return thenable.then(f);
});
```

Активизировать связывание в цепочку операций над такими типами объектов, реализующих метод then (), как Promise

```
// catchP :: f -> Promise -> Promise
const catchP = R.curry(function (f, promise) {
    return promise.catch(f);
}) ;
```

Предоставить логику обработки ошибок для объекта типа Promise

```
// errorLog :: Error -> void
    const errorLog = partial(logger, 'console', 'basic',
                             'ShowStudentAsync', 'ERROR');
```

| Создать консольный регистраторошибок

Сочетая приведенные выше вспомогательные функции с функцией R. compose (), можно получить исходный код, приведенный в листинге 8.9.

Листинг 8.9. Асинхронная версия программы showStudent

```
const ShowStudentAsync = R.compose(
    catchP(errorLog),                    <---
    then(append('#student-info')),       <-
    then(csv), then(R.props(['ssn',
    'firstname',                         'lastname'])),
    chain(findStudentAsync),             ---
    map(checkLengthSsn),
    lift(cleaninput));
```

Универсальный вызов функции на случай ошибок

1 Применение метода then () равнознач- I но монадной функции тар ()

Стык, где синхронный код соединяется в цепочку с асинхронным кодом, как поясняется далее

потенциал композиции с обязательствами. Как показано на рис. 8.9, при выполнении функции f indStudentAsync () рассматриваемая здесь программа ожидает возврата данных из асинхронной функции в вызывающий код, чтобы продолжить выполнение остальных функций. Обязательство в данном случае действует в качестве шлюза в асинхронную часть программы. Примечательно также, что в данной программе ни коим образом не раскрывается внутренний характер асинхронного поведения функции или применение обратных вызовов. Таким образом, функцию compose () можно по-прежнему использовать для организации программ в бесточечном стиле, чтобы соединять вместе функции, которые выполняются не в одно и то же время, а в будущем, в чем проявляется ее истинная сущность как комбинатора функций.

Данная программа дополнена логикой обработки ошибок. Так, если вызвать ее с существующим номером социального страхования:

```
ShowStudentAsync('444-44-4444')
```

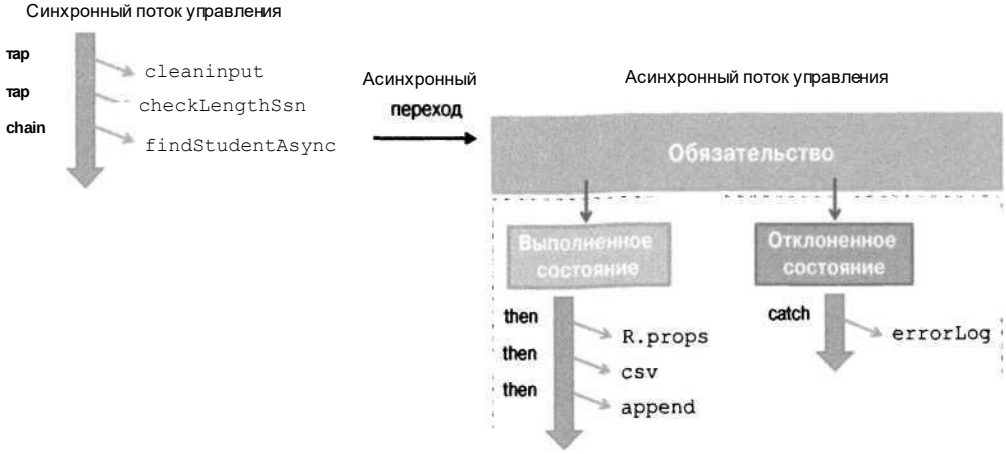


Рис. 8.9. При композиции синхронного поведения кода с асинхронным в программе возникает стык, где код переходит в ограниченную во времени соседнюю последовательность событий, происходящих в пределах типа обязательства

то запись об учащемся будет успешно выведена на веб-страницу. А если обязательство отвергнуто, то ошибка благополучно распространится по всей программе, достигнув вызова функции catch (), в которой будет выведено следующее сообщение об ошибке:

```
[ERROR] Error: Student not found!
([ОШИБКА] Ошибка: Учащийся не найден!)
```

Безусловно, данная программа сложна. Тем не менее нам удалось сохранить функциональный стиль ее написания, соединив вместе многие понятия, рассмотренные в этой книге, в том числе композицию, функции высшего порядка, монады, заключение в оболочку, преобразование, связывание в цепочку и пр. Более того, представление о программе, как ожидающей или выдающей данные, чтобы они стали доступными для последующей обработки, настолько привлекательно, что оно было внедрено в стандарт ES6 языка JavaScript в виде элемента первого класса, как поясняется далее.

8.3. Генерирование данных по требованию

К числу самых эффективных языковых средств, внедренных в стандарт ES6, относится способность функций взаимодействовать с вызвавшим их кодом. При этом выполнение функции приостанавливается до получения очередного запроса на выдачу данных. Это позволяет предоставлять данные в вызвавший функцию код без завершения работы самой функции. Данное средство открывает практически безграничные возможности для того, чтобы функции генерировали данные лишь по требованию, а не обрабатывали сразу все крупные массивы данных.

С одной стороны, в своем распоряжении можно иметь крупные коллекции объектов, преобразуемые по установленным бизнес-правилам, как это делалось в приведенных ранее примерах с помощью операций map, filter, reduce и др., а с другой стороны — задать правила для генерирования данных в нужном порядке. Например, в математическом смысле

функция x => x * x означает не что иное, как определение квадратов всех чисел (1, 4, 9, 16, 25 и т.д.). Если же выразить эту функцию с помощью какого-нибудь особого синтаксиса, то ее можно назвать *генератором.*

Функция-генератор — это языковое средство, обозначаемое как function* (т.е. ключевым словом function со знаком "звездочки"). Этот новый тип функции обладает особым качеством, позволяющим временно выйти из функции с помощью ключевого слова yield, а в дальнейшем снова войти в нее вместе с ее контекстом (т.е. с сохранением значений всех локальных переменных), сохраняемым в промежутках между последовательными выходами и вхождениями в функцию. (Подробнее о контексте выполнения функций см. в главе 7, "Оптимизация функционального кода"). В отличие от типичных вызовов функций, в функцию-генератор можно входить повторно, поскольку контекст ее исполнения может быть временно приостановлен и затем возобновлен по мере необходимости.

В языках с поддержкой отложенных вычислений можно формировать списки произвольных размеров по требованию в программе. Если бы такая возможность имелась и BjavaScript, то теоретически можно было бы написать код, аналогичный следующему:

```
R.ranged, Infinity).take(1); // -> [1]
R.ranged, Infinity).take(3); // -> [1,2,3]
```

Этот код выражает лишь самый общий принцип работы генераторов. Ведь как пояснялось в главе 7, BjavaScript для вычисления функций принята энергичная стратегия, и поэтому вызов R. range (1, Infinity) никогда не сможет завершиться, что в конечном счете приведет к переполнению стека вызовов функций в браузере. Генераторы обеспечивают отложенное поведение с помощью внутреннего объекта iterator, который создается при вызове функции-генератора. Этот объект предоставляет данные вызывающему коду по требованию при каждом обращении по ключевому слову yield, как показано на рис. 8.10.

Рассмотрим следующий быстрый пример, где из списка выбираются только первые три числа и не предпринимается попытка создать бесконечный ряд чисел:

```
function *range(start = 0, finish = Number.POSITIVE-INFINITY) { for(let i = start;
     i < finish; i++) {
         yield i,
     )
}
```

Вернуться обратно в вызывающий код, но запомнить состояние всех локальных переменных

```
const num = range(1);
```

```
num.next().value; // -> 1
num.next().value; // -> 2
num.next().value; // -> 3
```

Генератор относится к итерируемому типу, а это означает, что его можно использовать в цикле подобно массиву (подробнее об этом — далее). В стандарте ES6 внедрена новая конструкция for.. of для организации циклов с генераторами

```
// или

for (let n of range(1)) {
    console.log(n);
    if(n === threshold) {
        break;
    }
} // -> 1,2,3,...
```

Проверить граничное значение, чтобы цикл не выполнялся в программе бесконечно

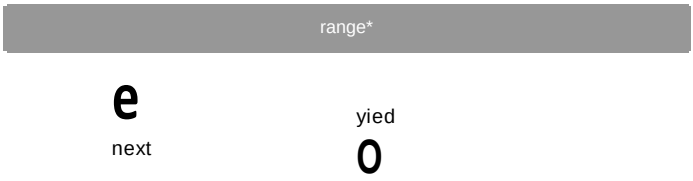


Рис. 8.10. Выполнение генератора range в цикле for. .of. На каждом шаге цикла генератор будет приостанавливать свое выполнение и выдавать новые данные. Следовательно, семантически генераторы подобны итераторам

С помощью генераторов можно реализовать программу с отложенными вычислениями на основе следующей функции take (), которая выбирает заданное количество элементов из бесконечного множества:

```
function take(amount, generator) { let result = []; for (let n of generator) {
result.push(n); if(n === amount) { break;
      }
  } return result; } take(3, ranged, Infinity)); // -> [1, 2, 3]
```

Если не учитывать некоторые ограничения, то генераторы ведут себя во многом так же, как и стандартно вызываемые функции. В частности, им можно передавать аргументы, в том числе и функции, чтобы определять характер генерируемых значений, как показано ниже. Функции-генераторы отличаются еще и тем, что их можно применять рекурсивно.

```
function *range(specification, start = 0, finish = Number.POSITIVE-INFINITY) {
for (let i = start; i < finish; i++) {
        yield specif ication (i);
    j
r

for (let n of range(x => x * x, 1, 4)) {
    console.log(n);
} // -> 1,4,9,16
```

Вызвать функцию specification для каждого генерируемого значения

Генератор ведет себя как любая функция высшего порядка, которой могут передаваться аргументы для реализации специального поведения. В данном случае генератору предписывается возвращать квадраты чисел

8.3.1. Генераторы и рекурсия

Как и любые стандартно вызываемые функции, одни генераторы можно вызывать из других. Это удобно в тех случаях, когда требуется создать сведенное в таблицу представление вложенного ряда объектов, что идеально подходит для обхода древовидных структур. А поскольку генераторы можно применять в цикле for. . of, делегирование одного генератора другому подобно слиянию двух коллекций и перебору всего массива данных. Напомним, что в главе 3, "Меньше структур данных и больше операций", рассматривалась древовидная структура отношений ученичества между выдающимися математиками, еще раз представленная на рис. 8.11.

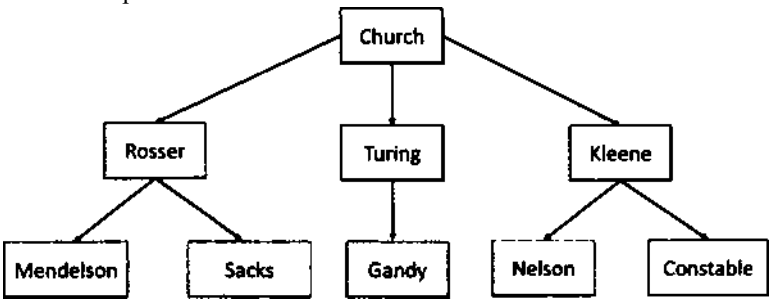


Рис. 8.11. Древовидная структура отношений ученичества из главы 3, где каждый узел представляет объект student, а каждая линия — отношение ученичества

Используя простые генераторы, подобные приведенным ниже, можно легко смоделировать данные в ветвях этого дерева. Получаемый в итоге результат будет представлен далее.

```
function* AllStudentsGenerator(){ yield 'Church';

    yield 'Rosser';
    yield* RosserStudentGenerator (); I Делегировать один генератор
                                      I другому по ссылке yield*

    yield ' Turing';                  < I A3TM14® из этого генератора можно чередовать
    yield* TunngStudentGenerator ();  Л I сданными из другого генератора
```

```
    yield 'Kleene';
    yield* KleeneStudentGenerator();


function* RosserStudentGenerator(){
    yield 'Mendelson';
    yield 'Sacks';


function* TuringStudentGenerator(){
    yield 'Gandy';
    yield 'Sacks';


function* KleeneStudentGenerator(){
    yield 'Nelson';
    yield 'Constable';


for(let student of AllStudentsGenerator()){
    console.log(student);
```

Циклический механизм выполняет обход, как будто это один крупный генератор

Рекурсия является неотъемлемой частью функционального программирования, и поэтому следует также продемонстрировать, что генераторы, несмотря на свою особую семантику, ведут себя во многом так же, как стандартно вызываемые функции, делегируя себя друг другу. Ниже приведен простой пример обхода того же самого дерева, в узлах которого содержатся объекты типа Person, но на этот раз — с помощью рекурсии.

```
function* TreeTraversal(node) {
    yield node.value;
    if (node.hasChildren()) {
       for(let child of node.children) {
          yield* TreeTraversal(child);


var root = node(new Person('Alonzo', 'Church', '111-11-1231'));

for(let     person     of     TreeTraversal(root))     {
    console.log(person.lastname);
```

Делегировать генератор обратно самому себе по ссылке yield*

Напомним, что корень дерева из главы 3 начинается с узла, содержащего объект Church

Выполнение приведенного выше примера кода приводит к тому же самому результату: Church, Rosser, Mendelson, Sacks, Turing, Gandy, Kleene, Nelson, Constable. Как видите, управление передается сначала другим генераторам, а затем, как только их выполнение завершится, оно возвращается в то же самое место, откуда было предано из вызывающего кода. Но с точки зрения цикла for. .of происходит лишь вызов внутреннего *итератора* до тех пор, пока не исчерпаются все данные. При этом никто и не подозревает, что на самом деле задействуется рекурсия.

8.3.2. Протокол итератора

С генераторами тесно связаны *итераторы* — еще одно языковое средство, внедренное в стандарт ES6. Благодаря именно этому средству генераторы можно обходить в цикле подобно любым другим структурам данных вроде массивов. Функция-генератор возвращает внутренним образом объект типа Generator, соответствующий протоколу итератора. Это означает, что в нем реализован метод next (), возвращающий значение, указанное после ключевого слова yield. У этого объекта имеются следующие свойства:

- done — флаг, принимающий логическое значение true, если итератор достиг конца своей последовательности, а иначе — логическое значение false, которое означает, что итератору удалось извлечь очередное значение из данной последовательности.
- value — значение, возвращаемое итератором.

Этого должно быть достаточно, чтобы понять внутренний механизм работы генераторов. Обратимся снова к примеру генератора range, реализованного на этот раз в первоначальной форме:

```
function range(start, end) { return {
        [Symbol. iterator] () { Pt-
            return this;
        },
        next() {
            if(start < end) {
                return { value: start++, done:false
            }
            return { done: true, value:end
        }
```

Указать на то, что возвращаемый объект является итерируемым (реализует протокол итератора)

Реализовать основную логику данного генератора. Если для генерирования имеются какие-нибудь дополнительные данные, возвратить объект с произведенным значением и установить в свойстве done логическое значение false, в противном случае логическое значение true

С помощью такой реализации можно создавать генераторы, производящие любые данные по определенному шаблону или спецификации. В качестве примера ниже приведен генератор squares квадратов чисел: `function squares() { let n = 1; return { [Symbol.iterator] () { return this;`

```
        },
        next() {
            return { value: n * n++ };
        } };
}
```

Подробнее с особенностями работы итераторов и итерируемых объектов можно ознакомиться в документации, доступной но следующему адресу: https://developer.mozilia.org/ru/docs/Web/JavaScript/Reference/ Iteration protocols. Благодаря внутреннему свойству @@iterator многие элементы языка JavaScript можно рассматривать как итерируемые объекты. В частности, можно предположить, что массивы способны действовать следующим образом:

```
var iter = ['S', 't', 'r', 'e', 'a', 'm'][Symbol.iterator](); iter.next().value;
// S iter.next().value; // t
```

Но даже символьные строки можно обходить приведенным ниже образом.

```
var iter = ' Stream'[Symbol. iterator] Obiter .next () .value; // -> S
iter.next().value; // -> t
```

Следует особо подчеркнуть, что данные можно рассматривать как потоки, способные при обращении к ним производить дискретные последовательности событий или значений. Как не раз демонстрировалось ранее, эти значения вливаются потоком в последовательность чистых функций высшего порядка и преобразуются в требуемый выходной поток. Такое представление о данных имеет решающее значение для перехода к еще одной, основанной на ФП парадигме, называемой *реактивным программированием.*

8.4. Функциональное и реактивное программирование средствами RxJS

Как уже упоминалось, характер веб-приложений значительно изменился, главным образом, под влиянием революционной технологии AJAX. По мере расширения возможностей веб-приложений ожидания пользователей постепенно становятся более требовательными не только к данным, но и к взаимодействию с ними. Приложения должны быть способны обрабатывать вводимые пользователем данные, поступающие из разных источников вроде нажимаемых кнопок, текстовых полей, перемещений мыши, жестов пальцами, голосовых команд и прочих, и поэтому очень важно, чтобы приложения были способны согласованно взаимодействовать со всеми этими источниками данных.

В этом разделе представлена реактивная библиотека под названием Reactive Extensions for JavaScript (RxJS — реактивные расширения JavaScript), позволяющая изящно сочетать асинхронные и управляемые событиями программы (подробнее об установке этой библиотеки см. в приложении). Библиотека RxJS во многом действует аналогично функциональным, основанным на обязательствах примерам, рассматривавшимся ранее в этой главе, но в то же время она обеспечивает более высокую степень абстракции и намного более эффективные операции. Но, прежде чем приступить к ее рассмотрению, необходимо объяснить понятие *наблюдаемые объекты.*

8.4.1. Данные как наблюдаемые объекты

Наблюдаемым (observable) называется любой информационный объект, на который можно *подписаться.* Приложения могут подписываться на асинхронные события, наступающие в результате чтения содержимого файла, вызовы веб служб, запросы к базе данных, извещение системных уведомлений, обработку вводимых пользователем данных, обход элементов коллекций или даже синтаксический анализ простой символьной строки. Реактивное программирование унифицирует все эти поставщики данных в единое понятие, называемое *наблюдаемым потоком,* используя объект типа Rx. Observable. Поток является *последовательностью упорядоченных событий, происходящих во времени.* Чтобы извлечь из такого объекта значение, необходимо подписаться на него. Итак, рассмотрим следующий пример применения данного объекта:

```
Rx.Observable.range(1, 3) .subscribe( --------
          x => console.log('Next: ${x}'), err =>
          console.log('Error: ${err}'), () =>
          console.log('Completed')
```

Методу subscribe () нужно передать три функции обратного вызова для обработки каждого элемента в последовательности, исключительного и корректного завершения операции

В результате выполнения приведенного выше примера кода создается наблюдаемая последовательность из диапазона чисел, генерируя ряд значений 1, 2, 3. И, наконец, в данном коде устанавливается признак окончания потока, как показано ниже.

```
Next: 1 Next: 2 Next: 3 Completed
```

Рассмотрим еще один пример применения представленной ранее функции- генератора squares () для заполнения потока значениями. С этой целью она дополнена параметром для генерирования конечного ряда квадратов чисел: `const squares = Rx.Observable.wrap(function* (n) { for(let i = 1; i <= n; i++) { return yield Observable.just(i * i);`

```
    } }) ;

squares(3).subscribe(x => console.log('Next: ${x}'));


Next: 1
Next: 4
Next: 9
```

Как следует из приведенных выше примеров, любыми типами данных можно оперировать одинаковым образом, используя объект типа Rx.Observable, поскольку он преобразует эти данные в поток. Объект типа Rx.Observable заключает в оболочку любой наблюдаемый объект, чтобы применить разные функции для преобразования наблюдаемых значений в требуемый выходной поток. Отсюда и возникает монада.

8.4.2. Функциональное и реактивное программирование

Объект типа Rx. Observable объединяет парадигмы функционального (ФП) и реактивного программирования (РП). Он реализует эквивалент минимального монадического интерфейса, представленного в главе 5, “Проектные шаблоны и сложность”, (операции map, of и j oin), а также методы, характерные для манипулирования потоком.

Ниже приведен характерный тому пример.

```
Rx.Observable.of(1,2,3, 4,5)
     .filter(x => x%2 !== 0)
     .map(x => x * x)              ----- — Отсеять четные числа
     .subscribe(x => console.log('Next: ${x}'));

// -> Next: 1
       Next: 9
       Next: 25
```

Чтобы продемонстрировать внутренний механизм действия наблюдаемого объекта, на рис. 8.12 показан процесс преобразования данных.

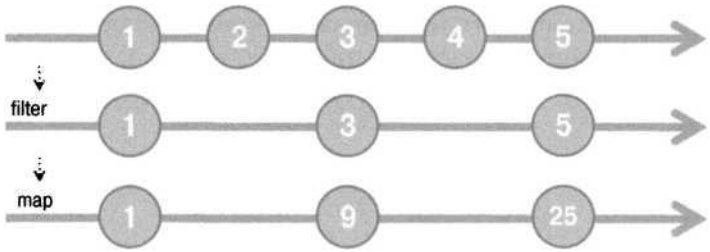


Рис. 8.12. Процесс применения функций, реализующих операции filter и тар, из наблюдаемой последовательности чисел

Если вам не приходилось раньше читать книги по функциональному программированию, то у вас может сложиться впечатление, будто самое трудное в реактивном программировании — научиться мыслить "реактивно". Но реактивное мышление ничем не отличается от функционального, только для этого применяются другие инструментальные средства, а это означает, что полдела уже сделано. На самом деле у обеих рассматриваемых здесь парадигм программирования так много общего, что большая часть литературы по реактивному программированию, которую можно найти в Интернете, начинается с обучения методикам функционального программирования. Потоки придают декларативность прикладному коду, снабжая его выполняемыми по цепочке вычислениями. Следовательно, реактивное программирование похоже на функциональное, из чего возникает термин *функционально-реактивное программирование* (ФРП).

Рекомендуемая литература

Истоки реактивного программирования прослеживаются с 2013 года, и поэтому большая часть литературы на эту тему связана не только с реактивным, но и с функционально-реактивным программированием. Здесь же преследуется цель не обучить реактивному программированию, а продемонстрировать, что оно на самом деле является функциональным программированием, применяемым для решения асинхронных и основанных на событиях задач.

Если же вы желаете узнать больше о парадигмах РП и ФРП, рекомендуем прочитать книгу *Functional Reactive Programming* Стивена Блекхита и Энтони Джонса (Stephen Blackheath & Anthony Jones, издательство Manning, 2016 г.; см. также https://www.manning.com/books/functional-reactive-prog ramming). А если вас интересует применение библиотеки RxJS в ФП, то для ее изучения рекомендуется книга *RxJS in Action* Пола Дэниелса и Луиса Атенсио (Paul Daniels & Luis Atencio; издательство

Manning, 2017 г.; см. также https ://www. manning. com/books/rx j s-in-action).

Итак, рассмотрев наблюдаемые объекты, перейдем к применению библиотеки RxJS для обработки вводимым пользователем данных. Если требуется взаимодействовать с событиями из самых разных источников и фиксировать их, можно легко прийти к запутанному и с трудом читаемому коду. Рассмотрим следующий простой пример чтения и проверки достоверности поля ввода номера социального страхования:

```
document.queryselector('#student-ssn')
    .addEventListener('change', function (event) { let value =
        event.target.value;
        value = value.replace(/^\s*I\-I\s*$/g, '');
        console.log(value.length == 9 ? 'Invalid' : 'Valid'));
    }) ;
// -> 444 Invalid
// -> 444-44-4444 Valid
```

Событие типа change происходит асинхронно, и поэтому всю бизнес-логику приходится размещать в одной функции обратного вызова. Но, как демонстрировалось ранее в этой главе, такой подход не способствует масштабированию, если продолжать и дальше нагромождать код обработки событий для каждой экранной кнопки, текстового поля или ссылки на странице. Единственная возможность для повторного использования кода — реорганизовать его, вынеся основную логику из функции обратного вызова. Но как масштабировать такой код, чтобы его сложность не увеличивалась пропорционально внедрению дополнительной логики?

Как и при написании асинхронного кода, методики ФП не удается изящно совмещать с традиционными, основанными на событиях функциями, поскольку это совершенно разные парадигмы программирования. Подобно тому, как потеря соответствия между чистыми и асинхронными функциями разрешается с помощью обязательств, так и для наведения моста между областями обработ-

ки событий и ФП требуется некоторый уровень абстракции, предоставляемый объектом типа Rx. Observable. В частности, исходный код из приведенного выше примера, где прослушиваются события типа change, наступающие в то время, когда пользователь обновляет поле ввода номера социального страхования учащегося, можно смоделировать в виде потока, как показано на рис. 8.13.

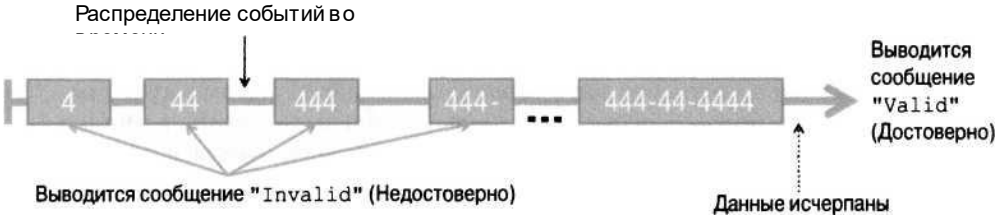


Рис. 8.13. Наглядное представление событий, наступающих в связи с обновлением номера социального страхования учащегося, в виде наблюдаемого потока, создаваемого в результате подписки на событие типа change в поле ввода этого

Принимая во внимание все сказанное выше, можно реорганизовать императивный, основанный на событиях исходный код из предыдущего примера, воспользовавшись ФРП. Это, по существу, означает подписку на событие и применение чистых функций для реализации всей бизнес-логики, как показано ниже.

```
Rx.Observable.fromEvent(document.queryselector('#student-ssn'),
                        ' change ') -«I--  Подписаться на событие типа change
    .map(x => x.target.value) -<          Извлечь значение при наступлении события
    .map(cleaninput)                      Применить функции из предыдущих глав,
                                          чтобы удалить пробелы и очистить номер
                                          социального страхования
    .map(checkLengthSsn) *0 .             Проверить, является ли результатом проверки
                                          структура Either. Right или Either. Left, чтобы
                                          определить его достоверность
    .subscribe(ssn => ssn.isRight ? console.log('Valid') :
              console.log('Invalid'));
```

В приведенном выше примере кода применяются функции из предыдущих глав, и поэтому значение, передаваемое функции subscribe (), заключается в оболочку монады типа Either, содержащей значение Right (SSN), если входные данные достоверны, а иначе — значение Left (null). Библиотека RxJS отличается тем, что с ее помощью можно изменить линейные асинхронные потоки данных для обработки событий, но это еще не все. В ее эффективные интерфейсы API внедрены обязательства, что дает возможность использовать единую модель для программирования всех асинхронных действий. Именно об этом и пойдет речь далее.

8.4.3. Библиотека RxJS и обязательства

Средствами библиотеки RxJS любой объект, соответствующий стандарту Promises/A+, можно преобразовать в наблюдаемую последовательность.

Это означает, что длительную операцию, реализуемую в функции getJSON (), можно заключить в оболочку обязательства, чтобы после его разрешения полученное значение было превращено в поток. В качестве примера ниже демонстрируется получение отсортированного списка учащихся, проживающих в Соединенных Штатах:

```
Отсортировать по имени, ио без учета регистра все объекты учащихся
Rx.Observable. fromPromise (getJSON (' /students ') )
     .map(R.sortBy(R.compose(R.toLower, R.prop('firstname'))))
     .flatMapLatest(student => Rx.Observable.from(student))    ц___ Преобразовать един
     .filter(R.pathEq(['address', 'country']/ 'US'))       --      ственный массив
     .subscribe(                                                  объектов учащихся в
          student => console, log (student. fullname), err =>     наблюдаемую по-
          console.log(err)                                        следовательность
      .                                                           учащихся
                                  Вывести результаты i
                                                          Отсеять учащихся, не
// -> Alonzo Church                                       проживающих в США
       Haskell Curry
```

Как видите, в приведенном выше примере кода реализована большая часть того, на что способны обязательства. Обратите внимание на то, что вся логика обработки ошибок сосредоточена в функции subscribe (). Так, если обязательство нельзя выполнить из-за того, что запрашиваемая веб-служба недоступна, то соответствующая ошибка распространяется вплоть до обратного вызова функции, выводящей следующее сообщение об ошибке, что вполне подходит для монад:

```
Error: 10 Error
```

В противном случае список объектов, представляющих учащихся, сортируется (в данном случае — по имени) и передается функции flatMapLatest (), преобразующей объект получаемого ответа в наблюдаемый массив учащихся. И, наконец, учащиеся, не проживающие в Соединенных Штатах, отсеиваются из потока и далее выводятся полученные результаты. В библиотеке RxJS предоставляется немало других инструментальных средств, а здесь была затронута лишь незначительная их часть. Более подробно с возможностями этой библиотеки можно ознакомиться по адресу https://xgrommx.github.io/rx-book.

В данной книге обсуждались решения самых разных задач средствами функционального программирования HaJavaScript, включая обработку коллекций, AJAX-запросов, обращений к базе данных, событий и пр. Можно надеяться, что, усвоив теоретические основы функционального программирования, а также примеры программ, демонстрирующие практическое применение методик ФП, вы научились мыслить функционально и вскоре станете делать это интуитивно.

Резюме

Из этой главы вы узнали следующее.

- Обязательства обеспечивают функциональное решение задач проектирования, традиционно ориентированных на обратные вызовы, которые долгое время считались настоящим бедствием для программ HaJavaScript.
- Обязательства дают возможность связывать функции в цепочку и составлять из них

композицию “на будущее”, абстрагируясь от низкоуровневых сложностей, обусловленных временными зависимостями в коде.

- В генераторах принят иной подход к асинхронному коду, для чего предоставляются специальные средства программирования, которые при поддержке отложенных итераторов позволяют выдавать данные по мере их доступности.
- Функционально-реактивное программирование повышает уровень абстракции прикладных программ до такой степени, которая позволяет уделить основное внимание обработке событий как логически независимых единиц. Это дает возможность сосредоточиться на решаемой задаче вместо того, чтобы бороться со сложными подробностями реализации.

Приложение А Библиотеки JavaScript, упоминаемые в книге

В этом приложение вкратце описываются библиотеки JavaScript, упоминаемые в данной книге.

Функциональные библиотеки JavaScript

JavaScript не является языком исключительно функционального программирования, и поэтому приходится полагаться на помощь сторонних библиотек, которые можно загружать в свой проект для эмулирования таких средств ФП, как карринг, композиция, запоминание, отложенное вычисление, неизменяемость и т.д., которые относятся к основным в языках исключительно функционального программирования вроде Haskell. Библиотеки избавляют от необходимости реализовывать собственные средства и позволяют уделить основное внимание написанию функций, реализующих требующуюся бизнес-логику, поручая организацию функционального кода этим библиотекам. В этом разделе перечислены функциональные библиотеки, упоминаемые в данной книге и предназначенные для выполнения следующих действий.

- **Восполнение любых пробелов в стандартных средах JavaScript с предоставлением дополнительных языковых конструкций и служебных функций высокого уровня, побуждающих писать код на основе простых функций.**
- **Обеспечение совместимости функциональных возможностей клиентских приложений HaJavaScript с разными браузерами.**
- **Согласованное абстрагирование от внутренних особенностей реализации таких методик функционального программирования, как карринг, композиция, частичное и отложенное вычисление и пр.**

В краткое описание каждой библиотеки включены инструкции по установке как в средах браузеров, так и в средах серверов (на платформе Node.js).

Библиотека Lodash

Эта служебная библиотека является ответвлением от библиотеки Underscore.js (`http://underscorejs.org/`), **широко принятой раньше в среде функционально программирующих HaJavaScript, имея зависимость от таких важных для JavaScript библиотек и каркасов, как Backbone.js. В библиотеке Lodash прослеживается тесная связь с ин**

терфейсами API из библиотеки Underscore.js, и ее внутренний движок был полностью переписан для повышения производительности. В примерах из данной книги библиотека Lodash применяется, главным образом, для создания модульных цепочек функций. Ниже перечислены основные сведения о библиотеке Lodash.

- **Версия: 3.10.1**
- **Начальная страница:** `https: / /lodash. com/`
- **Установка:**
  - **Браузер:** `<script src="lodash. js"x/script`
  - **Узел:** `$npm i —save lodash`

Библиотека Ramda

Эта служебная библиотека предназначена для функционального программирования и позволяет упростить создание конвейеров функций. Все функции из библиотеки Ramda неизменяемы и свободны от побочных эффектов. Кроме того, они автоматически каррированы, а их параметры специально организованы для удобства карринга и композиции. В состав библиотеки Ramda входят также линзы свойств, применяемые в примерах из данной книги для чтения, и записи свойств объектов неизменяемым способом.

Ниже перечислены основные сведения о библиотеке Ramda.

- **Версия: 0.18.0**
- **Начальная страница:** `http: / /ramda js. com/`
- **Установка:**
  - **Браузер:** `<script src="ramda. j s"x/script>`
  - **Узел:** `$npm install ramda`

Библиотека RxJS

В библиотеке Reactive Extensions for JavaScript (RxJS) реализуется парадигма так называемого *реактивного программирования,* сочетающая в себе наилучшие идеи из шаблонов "Наблюдатель" и "Итератор", а также функционального программирования, чтобы упростить написание программ, асинхронных и ориентированных на события.

Ниже перечислены основные сведения о библиотеке RxJS.

- **Версия: 4.0.7**
- **Начальная страница родительского проекта:** `https : //github.com/Reactive-Extensions/RxJS.`
- **Установка:**
  - **Браузер: загрузить нужные пакеты из хранилища JavaScript, например, по адресу** `http: //www. j sdelivr. com/?query=rxjs.` **Для выполнения примеров из данной книги потребуются следующие пакеты:** `rx-async, rx-dom` **и** `rx-binding.`
  - **Узел:** `$npm install rx-node`

Другие применяемые библиотеки

В примерах из данной книги применяются также нефункциональные библиотеки для

демонстрации дополнительных средств разработки программного обеспечения вроде протоколирования, тестирования и статического анализа кода.

Библиотека Log4js

Библиотека Log4JavaScript является каркасом для протоколирования на стороне клиента и сокращенно называется Log4js аналогично пакетам на других языках, например, Log4j Hajava, log4php на PHP и прочим. Эта библиотека зачастую применяется для организации протоколирования в масштабах предприятия, обладая намного большими возможностями, чем типичная функция `console. log ().`

Ниже перечислены основные сведения о библиотеке Log4j.

- **Версия: 1.0.0**
- **Начальная страница:** `http://stritti.github.io/log4j s/`
- **Установка:**
  - **Браузер:** `<soript src="log4 . j s"x/script>`
  - **Узел:** `$npm install log4js`

Библиотека QUnit

Эта библиотека является эффективным, стройным и простым каркасом для модульного тестирования кода HaJavaScript. Она применяется в таких широко распространенных проектах, как jQuery, и пригодна для тестирования любого типичного кода на JavaScript.

- **Версия: 1.20.0**
- **Начальная страница:** `https : //qunit j s. com/`
- **Установка:**
  - **Браузер:** `<script src="qunit-1. 20. O. js”x/script>`
  - **Узел:** `$npm install —save-dev qunitjs`

Библиотека Sinon

Библиотека Sinon JS служит каркасом для организации заглушек и средой для имитации в коде JavaScript. В примерах из данной книги она применяется в сочетании с библиотекой QUnit для усиления среды тестирования контекстом имитации и интерфейсом API.

- **Версия: 1.17.2**
- **Начальная страница:** `http: //sinonj s . org/`
- **Установка:**
  - **Браузер:** `<script src="sinon-1. 17.2 . j s"x/script> <script src=”sinon-qunit-1. 0.0. js"x/script>`
  - **Узел:** `$npm install sinon`
    `$npm install sinon-qunit`

Библиотека Blanket

Библиотека Blanket.js является инструментальным средством для покрытия тестами кода

JavaScript. Она предназначена в качестве дополнения существующих модульных тестов HaJavaScript (а по существу, тестов QUnit) статическими данными о покрытии проверяемого кода этими тестами.

Покрытие проверяемого кода тестами определяется долей в процентах тех строк кода, которые выполняются за один проход модульного теста. Этот процесс происходит в течение следующих этапов.

1. **Загрузка исходных файлов.**
2. **Оснащение проверяемого кода строками отслеживания.**
3. **Соединение перехватчиков в исполнителе тестов с выводимыми подробностями о покрытии тестами.**

- **Версия: 1.1.5**
- **Начальная страница:** `http: //blanket js.org/`
- **Установка:**
  - **Браузер:** `<script src=”blanket. js”x/script>`
  - **Узел:** `$npm install blanket`

Библиотека JSCheck

Это библиотека для тестирования кода JavaScript на основе спецификации (или свойств), написанная Дугласом Крокфордом под влиянием проекта QuickCheck на языке Haskell. Она позволяет сгенерировать из описания свойств функции случайные контрольные примеры, в которых предпринимается попытка подтвердить эти свойства.

- **Начальная страница: www.** `jscheck.org/`
- **Установка:**
  - **Браузер:** `<script src=" jscheck. js"x/script>`
  - **Узел:** `$npm install jscheck`

Предметный указатель

П

p

c

т

ф

ц

ч

э

я

СЕКРЕТЫ JAVASCRIPT НИНДЗЯ *2-е издание*

Джон Резиг
Безр Бибо
Иосип Марас

В этой книге раскрываются секреты мастерства разработки веб-приложений на JavaScript с учетом нововведений в стандартах ES6 и ES7. Начиная с пояснения таких основных понятий, как функции, объекты, замыкания, прототипы, регулярные выражения, обещания, события и таймеры, авторы постепенно проводят читателя по пути обучения от ученика до мастера, раскрывая немало секретов и специальных приемов программирования на конкретных примерах кода JavaScript. В книге уделяется немало внимания вопросам написания кросс-браузерного кода и преодолению связанных с этим типичных затруднений, что может принести немалую пользу всем, кто занимается разработкой веб-приложений.

Книга рассчитана на подготовленных читателей, стремящихся повысить свой уровень мастерства в программировании на JavaScript в частности и разработке веб-приложений вообще.

www.dialektika.com

ISBN 978-5-9908911-8-0 **в продаже**

ИЗУЧАЕМ JAVASCRIPT. РУКОВОДСТВО ПО СОЗДАНИЮ СОВРЕМЕННЫХ ВЕБ-САЙТОВ *3-е издание*

Этан Браун

Этан Браун

www.dialektika.com

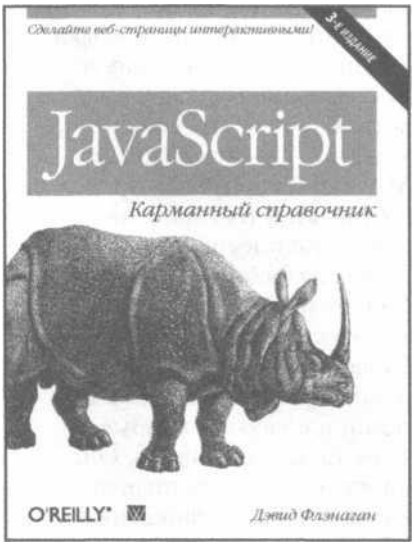
Эта книга — хорошо написанное сжатое введение в язык *JavaScript*, включая стандарт *ECMAScript 6*. Она знакомит программистов (любителей и профессионалов) со спецификацией *ES6* наряду с некоторыми связанными с ней инструментальными средствами и методиками на сугубо практической основе.

Эта книга предназначена, прежде всего, для читателей, уже обладающих некоторым опытом программирования (освоивших хотя бы вводный курс программирования). Опытные программисты найдут здесь практически полное описание важнейших концепций *JavaScript*, введенных в стандарт *ES6*. Программистам, переходящим на *JavaScript* с другого языка, содержимое этой книги также должно понравиться.

ISBN 978-5-9908463-9-5 в продаже

JAVASCRIPT
КАРМАННЫЙ СПРАВОЧНИК 3-Е ИЗДАНИЕ

Дэвид Флэнаган



www.williamspublishing.com

JavaScript — популярнейший язык программирования, который уже более 15 лет применяется для написания сценариев интерактивных веб-страниц. В книге представлены наиболее важные сведения о синтаксисе языка и показаны примеры его практического применения. Несмотря на малый объем карманного издания, в нем содержится все, что необходимо знать для разработки профессиональных веб-приложений.

Главы 1-9 посвящены описанию синтаксиса последней версии языка (спецификация ECMAScript 5).

- Типы данных, значения и переменные
- Инструкции, операторы и выражения
- Объекты и массивы
- Классы и функции
- Регулярные выражения В главах 10-14 рассматриваются функциональные возможности языка наряду с моделью DOM и средствами поддержки HTML5.

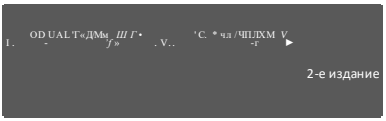
ISBN 978-5-8459-1830-7

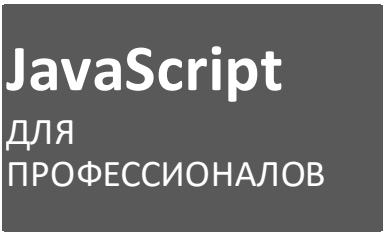
в продаже

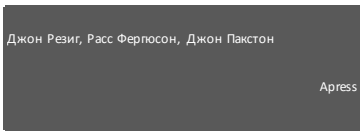
JavaScript для профессионалов

Второе издание

Джон Резиг,
Расс Фергюсон,
Джон Пакстон







www.williamspublishing.com

Эта книга является незаменимым пособием для профессиональных разработчиков современных веб-приложений на JavaScript. Читатель найдет в ней все, что требуется знать о современном состоянии языка JavaScript, его достоинствах и недостатках, новых языковых средствах, внедренных в последних версиях стандарта ECMAScript, передовых приемах отладки и тестирования кода, а также инструментальных средствах разработки. Книга изобилует многочисленными практическими и подробно разбираемыми примерами кода, повторно используемых функций и классов, экономящих время разработчиков. Она помогает им овладеть практическими навыками написания динамических веб-приложений на высоком профессиональном уровне, а также повысить свою квалификацию.

Книга рассчитана на тех, кто интересуется разработкой веб-приложений и имеет опыт программирования HaJavaScript.

ISBN 978-5-8459-2054-6 **в продаже**

Функциональное программирование на JavaScript

Луис Атенсио

В сложных веб-приложениях низкоуровневые детали JavaScript-кода могут затруднить анализ программы и повлиять на работоспособность системы в целом. Функциональное программирование (ФП) как стиль написания кода способствует слабо связанным отношениям между отдельными компонентами приложений и позволяет составить общее представление о проекте, упростить его разработку, общение с заказчиками и сопровождение.

В этой книге поясняются методики усовершенствования веб-приложений, влияющие в том числе на их расширяемость, модульность, повторное использование и тестируемость, а также производительность. В удобной для чтения форме на конкретных примерах и доходчивых пояснениях демонстрируется, как пользоваться методиками ФП на практике. Начинающие осваивать ФП по достоинству оценят немало удачных примеров сравнения ФП с императивным и объектно-ориентированным программированием, что позволяет лучше понять особенности функционального проектирования. Прочитав эту книгу, читатель научится осмысливать свои проекты функционально, а возможно, дорастет и до понимания монад!

Основные темы книги

- Применение ценных методик ФП на практике и там, где это наиболее целесообразно
- Отделение логики системы от подробностей ее реализации
- Обработка ошибок, тестирование и отладка прикладного кода в стиле ФП
- Демонстрация и обсуждение всех примеров кода на JavaScript, написанных по стандарту ES6 (ES 2015)

Книга адресована разработчикам, твердо усвоившим основы программирования на JavaScript и обладающим достаточным опытом проектирования веб-приложений.

Об а вторе

Луис Атенсио — инженер-разработчик и архитектор приложений масштаба предприятия на языках Java, PHP и JavaScript.

```
Rx.Observable.fromEvent(document.queryselector('#s
tudent-ssn') , 'keyup')
   .map(input => input.srcElement.value)
   .filter(ssn => ssn !== null && ssn.length !== 0)
```

fl MANNING

^АЦДЛЕКШЦКА

www.williamspublishing.com

```
   .map(ssn => ssn.replace(/^\s*I\s*$I\-/g, ''))
```

ОТЗЫВЫ О

"Эта книга должна входить в состав знаменитой книги *Секреты JavaScript ниндзя* в виде отдельной части".

Благодарный читатель

"Эта книга коренным образом изменила мои представления о написании кода на JavaScript".

Эндрю Мередит, компания Intrinsitech Corporation

"Удобный справочник с практическими примерами".

Эми Тенг, компания Dell

"Теперь именно так нужно писать код на JavaScript".

Уильям Е. Уиллер, компания West *Corporation*

"Прочитав эту книгу, я переосмыслил свой подход к написанию кода и даже сумел перестроить свое мышление на применение более совершенных приемов и методик".

Кристофер Хаупт,

Категория: программирование
Предмет рассмотрения: методики функционального программирования на JavaScript

```
   .skipWhile(ssn => ssn.length !== 9)
   .subscribe(
      validSsn => console.log('Valid SSN
${validSsn}')
```

[1] **Дуглас Крокфорд — известный программист, автор книг и докладов, неизменно принимающий активное участие в развитии языка JavaScript, популяризации формата JSON и создании таких библиотек JavaScript, KaKjSLint, JSMin и JSCheck.**