Исчерпывающее руководство по созданию простого программного кода для решения сложных задач

Программирование на F#

Символ®
O'REILLY®

Крис Смит

Programming F#

Chris Smith

O'REILLY®

Программирование на F#

Крис Смит

Символ®

Санкт-Петербург — Москва
2011

Крис Смит

Программирование на F#

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Научный редактор	*С. Тепляков*
Корректор	*О. Макарова*
Верстка	*К. Чубаров*

Смит К.

Программирование на F#. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 448 с., ил.

ISBN 978-5-93286-199-8

F# – это мультипарадигмальный язык программирования, который не только помогает повысить производительность труда за счет использования функционального стиля разработки, но и позволяет применять при создании приложений уже имеющиеся навыки объектно-ориентированного и императивного программирования. Книга «Программирование на F#» поможет открыть множество преимуществ этого языка, включая возможность доступа ко всем замечательным инструментам и библиотекам платформы .NET.

Это исчерпывающее руководство, написанное Крисом Смитом, одним из основных разработчиков F# компании Microsoft, знакомит с синтаксисом языка, реализацией асинхронных и параллельных вычислений, с расширенными концепциями языка F#, такими как цитируемые и вычислительные выражения.

От читателя не требуется знание конкретных технологий, хотя общий опыт программирования, безусловно, желателен. Единственное требование – это желание воспользоваться преимуществами функционального программирования при разработке своих проектов, будь то реализация численных алгоритмов, анализ данных или сценарии для личного использования. В этом случае издание послужит хорошей отправной точкой на пути изучения фундаментальных и расширенных концепций языка F#.

ISBN 978-5-93286-199-8
ISBN 978-0-596-15364-9 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 12.01.2011. Формат 70×100 1/16. Печать офсетная.
Объем 28 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Вступительное слово

Эта книга отмечает переломный момент в истории развития языка F# – переход от состояния исследовательского проекта подразделения Microsoft Research в Кембридже, уходящего корнями в такие языки программирования, как OCaml и Haskell, до стабильного, эффективного и удивительно производительного инструмента создания емких и лаконичных программ для платформы .NET. С выходом Visual Studio 2010 новое поколение программистов получило в свое распоряжение язык, который позволит им приближать будущее, разрабатывая программное обеспечение, будь то просто красивый программный код, революционные фреймворки, высокопроизводительные веб-сайты, новые методологии программирования, улучшенные алгоритмы математических вычислений, надежные механизмы тестирования, интеллектуальная параллелизация, улучшенное моделирование или любые другие проявления «грамотного программирования» на F#. Как создателю F# мне очень приятно, что именно Крис Смит (Chris Smith), один из основных разработчиков F#, представляет этот язык программирования широкой аудитории.

Язык F# представляет собой сочетание простоты и элегантности типизированного функционального программирования с широкими возможностями платформы .NET. Хотя для многих программистов типизированное функциональное программирование является относительно новой парадигмой и требует дополнительного изучения, оно существенно упрощает разработку программ. Программы на языке F# должны строиться из основных композиционных элементов, а возможность вывода типов делает их короче и понятнее. В этой книге Крис сначала представляет основные парадигмы языка F#: функциональное программирование, императивное программирование и объектно-ориентированное программирование, а затем демонстрирует, как они могут использоваться при создании программ. Особое внимание он уделяет простоте и ясности описания фундаментальных элементов языка, попутно подчеркивая, насколько приятным может быть процесс программирования вообще и программирования на языке F# в частности.

При грамотном подходе язык F# существенно упрощает разработку программ. Например, единицы измерения в языке F#, описываемые в этой книге и являющиеся важной вехой в истории развития языков программирования, ликвидируют сложности, связанные с использованием вещественных чисел в прикладных областях. Кроме того, разви-

тие языка F# в значительной степени определила ориентация на принципы асинхронной и параллельной обработки данных. В этой книге Крис представляет основы языка F# и платформы .NET, позволяющие вам освоиться в этом мире. Кроме того, язык F# обладает мощной поддержкой объектно-ориентированного программирования, и эта тема, так близкая сердцу Криса, также освещается в книге.

Прежде всего, F# – это практический язык программирования, и Крис ручается, что читатель получит здесь всю информацию, необходимую для успешного применения текущего поколения инструментов F# для создания программ. Крис описывает версию 1.0 языка F#, входящую в состав Visual Studio 2010, – первую официальную версию. Коллектив разработчиков F# и я чрезвычайно признательны Крису за его вклад в развитие языка и его инструментов. Я надеюсь, что вы получите такое же удовольствие, программируя на языке F#, как и мы с Крисом, разрабатывая F# в составе команды.

Дон Сайм (Don Syme),
ведущий разработчик и создатель языка F#,
Microsoft Research

Предисловие

В Visual Studio появилась поддержка нового языка программирования – F#. Он помогает писать более выразительные, простые в поддержке программы и дает возможность использовать весь богатейший потенциал платформы .NET. Знание языка F# не только позволит вам повысить свою производительность, но и обеспечит вам определенные преимущества перед другими программистами. Овладев приемами функционального программирования, представленными в F#, вы сможете применять их в программах, написанных на других языках, а также по-новому взглянуть на программирование в целом.

Введение в F#

Что же представляет собой язык F#? В двух словах, F# – это *мультипарадигмальный* язык программирования, основанный на .NET, то есть язык программирования, исходно поддерживающий различные *стили* программирования. Я избавлю вас от знакомства с полной историей развития этого языка и коротко отмечу лишь наиболее важные моменты:

- F# поддерживает функциональное программирование, то есть такой стиль программирования, когда описывается, *что* должна делать программа, а не *как* она должна работать.
- F# поддерживает объектно-ориентированное программирование. Язык F# позволяет заключать программный код в классы и объекты, что дает возможность упростить его.
- F# поддерживает императивное программирование. На языке F# можно реализовать изменение содержимого областей памяти, читать и записывать файлы, обмениваться данными по сети и так далее.
- F# является статически типизированным языком. Вследствие этого информация о типах доступна уже на этапе компиляции, что обеспечивает большую надежность программного кода – F# не позволит заткнуть круглое отверстие квадратной пробкой.
- F# – это язык для платформы .NET. Он работает на основе общей языковой инфраструктуры (Common Language Infrastructure, CLI), и поэтому при использовании этого языка доступны механизм автоматической сборки мусора (управления памятью) и мощные библиотеки классов. Кроме того, F# поддерживает все концепции, харак-

терные для .NET, такие как делегаты, перечисления, структуры, P/Invoke[1] и так далее.

Даже без понимания этих специальных терминов становится ясно, что F# – это мощный язык с широкими возможностями. Но не волнуйтесь, мы со всеми ними познакомимся шаг за шагом.

Кому адресована эта книга

Эта книга не является учебником по программированию и предполагает наличие у читателя знаний основных понятий, таких как циклы, функции и рекурсия. Однако она не требует наличия опыта функционального программирования или знакомства с платформой .NET.

Если вы имеете опыт использования C# или VB.NET, вы будете чувствовать себя совершенно комфортно. Несмотря на то, что в языке F# применяются иные подходы к программированию, тем не менее все имеющиеся у вас знания .NET вы сможете применять в программах на F#.

Если вы знакомы с такими языками программирования, как OCaml или Haskell, синтаксис F# покажется вам очень знакомым. Язык F# обладает большинством особенностей, присущих этим языкам, и добавляет множество новых возможностей, обеспечивающих интеграцию с платформой .NET.

Что необходимо для работы с этой книгой

Поддержка языка F# является составной частью Visual Studio 2010. В эту поддержку входят компилятор F# и система управления проектами, а также такие особенности, как подсветка синтаксиса и механизм автодополнения IntelliSense. При этом для создания программ на F# не обязательно иметь полную версию Visual Studio 2010. Поддержку F# можно обеспечить поверх Visual Studio 2008 Shell – бесплатной и ограниченной версии Visual Studio. Загрузить Visual Studio 2008 Shell (Integrated Mode) можно на странице:

http://www.microsoft.com/downloads/details.spx?FamilyID=40646580-97FA-4698-B65F-620D4B4B1ED7

После установки Visual Studio 2008 Shell вы можете установить последнюю версию F# Community Technology Preview, загрузив ее с сайта Microsoft F# Developer Center (*http://fsharp.net*).

Кроме того, имеется возможность создавать и распространять программы на языке F#, используя открытую (*http://open-source.org*) платформу Mono (*http://www.mono-project.com/*).

---

[1] Механизм вызова неуправляемого программного кода. – *Прим. перев.*

Структура книги

Книга делится на две части. В первой части основное внимание уделяется раскрытию мультипарадигмальной природы языка F#. Первые главы посвящены отдельным парадигмам программирования на F#, а последующие помогут вам понять основные особенности языка. К концу первой части вы освоите язык F# и стиль программирования на нем.

Во второй части представлено несколько дополнительных концепций, но основное внимание уделяется применению F# в специализированных областях. К концу второй части вы будете знать, как использовать F# для программирования параллельных вычислений и создания предметно-ориентированных языков программирования.

Часть I «Мультипарадигмальное программирование»

Глава 1 «Введение в F#» представляет язык F# и интегрированную среду разработки Visual Studio 2010. Я рекомендую прочитать эту главу даже тем, кто уже знаком с Visual Studio, потому что использование языка F# привносит свои особенности в создание и управление проектами.

Глава 2 «Основы» знакомит с основными типами и понятиями, которые являются базой для всех остальных глав.

Глава 3 «Функциональное программирование» знакомит с функциональным программированием и особенностями создания программного кода F# с использованием этого стиля.

Глава 4 «Императивное программирование» описывает, как изменять значения и состояние программы при использовании императивного стиля.

Глава 5 «Объектно-ориентированное программирование» описывает особенности объектно-ориентированного программирования, начиная с определения простых типов и заканчивая обсуждением таких понятий, как наследование и полиморфизм.

Глава 6 «Программирование для .NET» рассматривает некоторые концепции, не зависящие от используемого стиля программирования и обусловленные особенностями платформы .NET Framework и общей языковой инфраструктуры (Common Language Infrastructure, CLI).

Глава 7 «Прикладное функциональное программирование» охватывает дополнительные темы, относящиеся к функциональному программированию, такие как концевая рекурсия[1] (tail recursion) и функциональные шаблоны проектирования.

[1] Иногда ее называют «хвостовая рекурсия». – *Прим. перев.*

Глава 8 «Прикладное объектно-ориентированное программирование» описывает приемы создания и использования богатой системы типов. Особое внимание в этой главе уделяется использованию функциональных аспектов языка F# применительно к объектно-ориентированному программному коду.

Часть II «Программирование на F#»

Глава 9 «Сценарии» рассматривает F# как язык сценариев и описывает особенности создания сценариев на языке F#.

Глава 10 «Вычисляемые выражения» представляет дополнительную особенность языка F#, которая позволяет устранить избыточность программного кода и расширить ядро языка F# новыми возможностями.

Глава 11 «Асинхронное и параллельное программирование» рассказывает о том, как в программах на языке F# можно использовать преимущества многоядерных процессоров, а также о средствах параллельного программирования, имеющихся в языке F# и в библиотеках .NET.

Глава 12 «Рефлексия» рассматривает библиотеку, имеющуюся в .NET и позволяющую выполнять рефлексию, и порядок ее использования для создания декларативных программ.

Глава 13 «Цитирование» представляет цитируемые выражения в языке F# и рассказывает, как они могут использоваться в метапрограммировании и для выполнения программного кода F# на других вычислительных платформах.

Приложения

В этой книге также имеется пара приложений, в которых подробно рассматриваются некоторые дополнительные понятия, которые могут вам пригодиться.

В приложении A кратко рассматриваются существующие технологии, доступные на платформе .NET, и способы их использования в программах на языке F#.

В приложении B описываются способы организации взаимодействия программ на языке F# с существующими библиотеками и неуправляемым программным кодом с применением механизмов P/Invoke и COM-interop.

Типографские соглашения

В книге приняты следующие соглашения:

Курсив

Применяется для выделения терминов, когда они упоминаются впервые.

`Моноширинный шрифт`

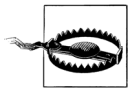
Применяется для представления программного кода и ключевых слов языка F#.

`Моноширинный жирный`

Используется для выделения фрагментов программного кода.

Особое внимание следует уделять фрагментам текста, выделенным, как показано ниже.

Примечания, выделенные таким способом, содержат дополнительную информацию для вдумчивых читателей.



Предупреждения, выделенные таким способом, призваны помочь избежать распространенных ошибок.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный)
707-829-0104 (факс)

Мы тщательно проверили всю информацию, которая приводится в этой книге, тем не менее вы можете столкнуться с некоторыми изменениями в языке F#, появившимися уже после выхода книги (или даже с ошибками в тексте!). Книга имеет собственную веб-страницу, где приводится список ошибок и рассказывается о наших планах по выходу следующих изданий. Адрес веб-страницы:

http://oreilly.com/catalog/9780596153649

Вы можете также отправлять свои сообщения по электронной почте. Чтобы отправить вопрос в список рассылки или запросить каталог, отправляйте сообщения по адресу:

info@oreilly.com

Свои отзывы о книге отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию об этой и других книгах, а также технические статьи о языке F# и его обсуждение вы найдете на веб-сайте издательства O'Reilly:

http://www.oreilly.com

или на сайте O'Reilly .NET DevCenter:

http://www.oreillynet.com/dotnet

или на портале Microsoft Developer Network:

http://www.msdn.com/fsharp

Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить значительные по объему части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «Programming F#, by Chris Smith. Copyright 2010 Chris Smith, 978-0-596-15364-9».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online – это виртуальная библиотека, которая позволяет легко и быстро находить ответы на вопросы среди более чем 7500 технических и справочных изданий и видеороликов.

Подписавшись на услугу, вы сможете загружать любые страницы из книг и просматривать любые видеоролики из нашей библиотеки. Читать книги на своих мобильных устройствах и сотовых телефонах. Получать доступ к новинкам еще до того, как они выйдут из печати.

Читать рукописи, находящиеся в работе, и посылать свои отзывы авторам. Копировать и вставлять отрывки программного кода, определять свои предпочтения, загружать отдельные главы, устанавливать закладки на ключевые разделы, оставлять примечания, печатать страницы и пользоваться массой других преимуществ, позволяющих экономить ваше время.

Благодаря усилиям O'Reilly Media данная книга также доступна через услугу Safari Books Online. Чтобы получить полный доступ к электронной версии этой книги, а также книг с похожими темами издательства O'Reilly и других издательств, подпишитесь бесплатно по адресу *http://my.safaribooksonline.com*.

Благодарности

Я получил все лавры за создание этой книги, однако основной объем работ по созданию языка F# был выполнен коллективом проекта F# в корпорации Microsoft в Редмонде и в подразделении Microsoft Research в Кембридже (Англия). Я получил огромное удовольствие, работая с удивительно талантливыми людьми, без которых язык F# так и остался бы простой пометкой в списке дел, приклеенном к монитору Дона Сайма (Don Syme).

Ниже я перечислил сотрудников Microsoft, участвовавших в создании этого языка:

Анар Алимов (Anar Alimov)
Эндрю Кеннеди (Andrew Kennedy)
Баоф Фенг (Baofa Feng)
Брайан Макнамара (Brian McNamara)
Дэниел Квирк (Daniel Quirk)
Дмитрий Ломов (Dmitry Lomov)
Доминик Куни (Dominic Cooney)
Дон Сайм (Don Syme)
Гордон Хогенсон (Gordon Hogenson)
Грег Неверов (Greg Neverov)
Джеймс Маргетсон (James Margetson)
Джо Пэймер (Joe Pamer)
Йомо Фишер (Jomo Fisher)
Кайл Росс (Kyle Ross)
Лоран ле Блюн (Laurant Le Blun)
Люк Хобан (Luke Hoban)
Матео Тавеггиа (Matteo Taveggia)
Ральф Хербрич (Ralf Herbrich)
Сантош Захария (Santosh Zachariah)
Шон Ванг (Sean Wang)
Тимоти Ын (Timothy Ng)
Томаш Петричек (Tomas Petricek)

Кроме того, эта книга не появилась бы на свет без помощи замечательных редакторов издательства O'Reilly, членов сообщества F# и многих других: Джона Осборна (John Osborn), Лаурель Рума (Laurel Ruma), Брайана Макдональда (Brian MacDonald), Мэтта Эллиса (Matt Ellis), Андре ван Мейлбрука (André van Meulebrouck), Стефена Тоуба (Stephen Toub), Джареда Парсонса (Jared Parsons), «почетного члена» Дастина Кэмпбелла (Dustin «honorary member» Campbell), Майкла де ла Маза (Michael de la Maza), Алекса Пика (Alex Peake), Райана Каванафа (Ryan

Cavanaugh), Брайана Пика (Brian Peek), Мэтью Подвысоцки (Matthew Podwysocki), Ричарда Минерича (Richard Minerich), Кейт Мур (Kate Moore) и наконец, Стюарта Боуэрса (Stuart Bowers) – вы настоящие мастера 9-го дана в F#.

Об авторе

Крис Смит работает в Microsoft в группе разработки языка F#. Его должность инженера-программиста в отделе тестирования позволила ему в совершенстве овладеть языком F#. Крис имеет степень магистра информатики, полученную в Вашингтонском университете, и испытывает настоящую страсть к любым вкусным напиткам, в бокалы с которыми добавляется зонтик. Вы можете встретиться с ним в Интернете, в его персональном блоге «Chris Smith's Completely Unique View» (совершенно уникальный взгляд Криса Смита) по адресу: *http://blogs.msdn.com/chrsmith/*.

I

Мультипарадигмальное программирование

1

Введение в F#

F# – это мощный язык программирования, поддерживающий несколько парадигм программирования. Эта глава представляет собой краткое введение в основы F# – компилятор, инструменты и место языка в Visual Studio 2010.

В этой главе вы напишете пару простых программ на языке F#, а затем я познакомлю вас с ключевыми особенностями Visual Studio, имеющими отношение к разработке приложений на этом языке. Я не буду здесь подробно описывать Visual Studio, поэтому для расширения своих представлений об этой интегрированной среде разработки рекомендую вам заняться самостоятельными исследованиями или обратиться к электронной документации по адресу *http://msdn.microsoft.com/en-us/vstudio/default.aspx*[1].

Даже если вы уже знакомы с Visual Studio, вам все равно стоит прочитать эту главу. Создание и отладка проектов на языке F# выполняется точно так же, как и проектов на языке C# или VB.NET, однако проекты на F#, состоящие из нескольких файлов, имеют некоторые уникальные особенности. Кроме того, при работе с этим языком поддерживается такая возможность, как *F# Interactive*, которая позволит вам резко поднять свою производительность. Не пропустите!

Знакомство с F#

Во многих книгах по программированию принято писать программы «Hello, World», и я не буду отказываться от этой традиции. Откройте текстовый редактор Блокнот (Notepad) или любой другой текстовый ре-

[1] На русском языке: *http://msdn.microsoft.com/ru-ru/vstudio/default.aspx* – *Прим. перев.*

дактор по своему выбору и создайте новый файл с именем *HelloWorld.fs* со следующим содержимым:

```
// HelloWorld.fs
printfn "Hello, World"
```

Отлично! Вы только что написали свою первую программу на языке F#. Чтобы скомпилировать эту программу, необходимо воспользоваться компилятором F#, `fsc.exe`, находящимся в папке *Program Files\Microsoft F#\v4.0*. (Или, если вы пользуетесь Mono, в каталоге, выбранном при установке F#.) В следующем фрагменте показано использование компилятора F# из командной строки для сборки и запуска приложения:

```
C:\Program Files\Microsoft F#\v4.0>fsc HelloWorld.fs
Microsoft F# Compiler, (c) Microsoft Corporation, All Rights Reserved
F# Version 1.9.8.0, compiling for .NET Framework Version v4.0.21017

C:\Program Files\Microsoft F#\v4.0>HelloWorld.exe
Hello, World!
```

Visual Studio 2010

Инструментальные средства наполняют жизненной силой любой язык программирования, и F# не является исключением. Хотя у вас все получится, даже если вы будете писать программы на F# в своем любимом текстовом редакторе и вызывать компилятор исключительно из командной строки, тем не менее использование инструментальных средств позволит вам добиться более высокой производительности. Подобно языкам C# и VB.NET, F# имеет полноценную поддержку в Visual Studio. Для языка F# в среде Visual Studio поддерживаются все известные возможности, такие как отладчик, IntelliSense (механизм автодополнения), шаблоны проектов и другие.

Чтобы создать новый проект на языке F#, откройте среду разработки Visual Studio и выберите пункт меню File → New Project (Файл → Создать → Проект)[1], чтобы открыть диалог New Project (Создать проект), как показано на рис. 1.1. В левой панели выберите пункт Visual F#, а в правой панели выберите пункт F# Application (Приложение F#) и щелкните на кнопке OK.

После щелчка на кнопке OK в диалоге New Project (Создать проект) вы увидите пустое окно редактора – чистый холст, готовый к созданию вашего шедевра на F#.

Вернемся еще раз к нашей программе «Hello, World». Введите следующий фрагмент в окне редактора программного кода F#:

```
printfn "Hello, World"
```

[1] Здесь и далее перевод интерфейсных элементов будет даваться в соответствии с русифицированной версией Visual Studio 2010. – *Прим. перев.*

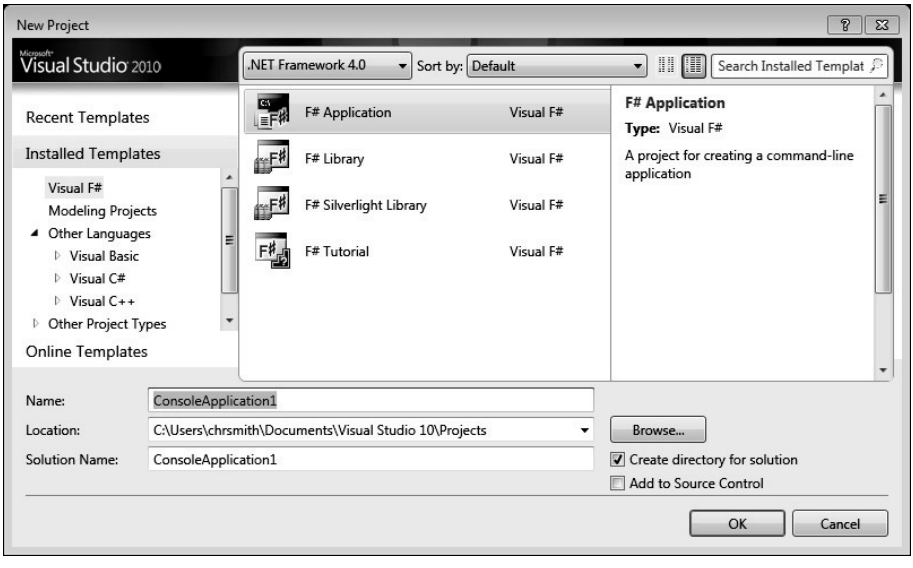


Рис. 1.1. *Выберите пункт F# Application, чтобы создать новый проект F#*

Теперь нажмите комбинацию клавиш Control + F5, чтобы запустить приложение. Когда приложение запустится, на экране появится окно консоли, в котором будет выведен вполне ожидаемый текст, как показано на рис. 1.2.

Рис. 1.2. *Программа «Hello, World» на языке F#*

Ваша вторая программа на F#

Кому-то может показаться странной программа, в которой отсутствует явный метод Main. Почему такое возможно, вы узнаете в следующей главе, а пока давайте создадим чуть более осмысленную, хотя такую же простую программу, чтобы почувствовать особенности синтаксиса языка F#.

В примере 1.1 приводится исходный текст программы, которая принимает два параметра командной строки и выводит их в консоли. Кроме того, она выводит текущее время.

Пример 1.1. Mega Hello World

```
(*
Mega Hello World:
Принимает два параметра командной строки и выводит их
вместе со значением текущего времени в окно консоли.
*)

open System

[<EntryPoint>]
let main (args : string[]) =

    if args.Length <> 2 then
        failwith "Error: Expected arguments <greeting> and <thing>"

    let greeting, thing = args.[0], args.[1]
    let timeOfDay = DateTime.Now.ToString("hh:mm tt")

    printfn "%s, %s at %s" greeting thing timeOfDay

    // Код завершения программы
    0
```

Теперь, когда у вас имеется реальный код на языке F#, вам, вероятно, будет интересно узнать, что в нем происходит. Рассмотрим эту программу строчка за строчкой, чтобы понять, как она работает.

Значения

В примере 1.1 содержатся три значения с именами `greeting`, `thing` и `timeOfDay`:

```
let greeting, thing = args.[0], args.[1]
let timeOfDay = DateTime.Now.ToString("hh:mm tt")
```

Обратите внимание на ключевое слово `let`, которое связывает *имя* со *значением*. Важно отметить, что, в отличие от большинства других языков программирования, значения в F# являются *неизменяемыми* по умолчанию, то есть их нельзя изменить после того, как они будут инициализированы. Почему значения являются неизменяемыми, мы узнаем в главе 3, а пока достаточно будет отметить, что это одна из особенностей функционального программирования.

Кроме того, язык F# отличает символы верхнего и нижнего регистров, поэтому два значения, имена которых отличаются только регистром символов, считаются разными:

```
let number = 1
let Number = 2
let NUMBER = 3
```

Имя значения может содержать буквы, цифры, символы подчеркивания _ и апострофы ‘ в любых комбинациях. Единственное ограничение: имя должно начинаться с буквы или с символа подчеркивания.

Допускается заключать имена значений в пары обратных апострофов. В этом случае имена могут содержать любые символы, за исключением символов табуляции и перевода строки. Благодаря этому вы можете ссылаться на значения и функции в библиотеках, написанных на других языках .NET, имена которых могут конфликтовать с ключевыми словами языка F#:

```
let ``this.Isn't %A% good value Name$!@#`` = 5
```

Пробельные символы имеют значение

В других языках программирования, таких как C#, для обозначения окончания инструкций и выделения блоков программного кода используются точка с запятой и фигурные скобки. При этом программисты для повышения удобочитаемости обычно оформляют программный код, используя отступы, поэтому эти дополнительные символы часто лишь загромождают программный код.

В языке F# пробельные символы – пробелы и символы перевода строки – имеют важное значение. Компилятор F# позволяет использовать пробельные символы для разграничения блоков кода. Например, любой код, имеющий отступ больше, чем ключевое слово `if`, считается телом инструкции `if`. Поскольку символы табуляции могут соответствовать различному числу пробелов, они запрещены к использованию в коде F#.

Вы можете настроить редактор Visual Studio так, что он будет автоматически преобразовывать символы табуляции в пробелы, для чего следует изменить соответствующий параметр в диалоге Tools → Options → Text Editor → F# (Сервис → Параметры → Текстовый редактор → F#).

Возвращаясь к примеру 1.1, обратите внимание, что тело метода `main` имеет отступ в четыре пробела и отступ в теле инструкции `if` увеличен еще на четыре пробела:

```
let main (args : string[]) =

    if args.Length <> 2 then
        failwith “Error: Expected arguments <greeting> and <thing>”

    let greeting, thing = args.[0], args.[1]
```

```
    let timeOfDay = DateTime.Now.ToString("hh:mm tt")

    printfn "%s, %s at %s" greeting thing timeOfDay

    // Код завершения программы
    0
```

Если бы тело `failwith` инструкции `if` не имело дополнительного отступа в четыре пробела и располагалось бы на том же расстоянии от начала строки, что и ключевое слово `if`, компилятор F# выдал бы предупреждение. Проблема состоит в том, что в этом случае компилятор не смог бы распознать `failwith` как тело инструкции `if`:

```
[<EntryPoint>]
let main (args : string[]) =

    if args.Length <> 2 then
    failwith "Error: Expected arguments <greeting> and <thing>"
```

```
Warning FS0058: possible incorrect indentation: this token is offside of
context started at position (25:5). Try indenting this token further or
using standard formatting conventions
```

(Предупреждение: возможно, неправильные отступы: этот маркер находится вне контекста, начиная с позиции (25:5). Попробуйте увеличить отступ маркера или использовать стандартные соглашения о форматировании.)

Основное правило заключается в следующем: все, что относится к методу или к инструкции, должно иметь дополнительный отступ относительно ключевого слова, с которого начинается метод или инструкция. Так, в примере 1.1 весь код в теле метода `main` имеет отступы относительно первого ключевого слова `let`, а код в инструкции `if` имеет отступы относительно ключевого слова `if`. По мере обретения навыков работы с программным кодом F# вы быстро убедитесь, что отсутствие точек с запятой и фигурных скобок существенно упрощает написание кода и его чтение.

.NET Interop

В примере 1.1 также демонстрируется, как программы на языке F# взаимодействуют с существующими библиотеками .NET:

```
open System

// ...

let timeOfDay = DateTime.Now.ToString("hh:mm tt")
```

Платформа .NET Framework содержит множество библиотек, упрощающие создание различных приложений, начиная от приложений с графическим интерфейсом и работы с базами данных и заканчивая веб-при-

ложениями. Программы на языке F# могут без всяких ограничений использовать любые библиотеки .NET и обращаться к ним непосредственно. Свойство `DateTime.Now`, используемое в примере 1.1, находится в пространстве имен `System`, в сборке `mscorlib.dll`. Код на языке F#, в свою очередь, также может использоваться в программах, написанных на других языках, поддерживаемых платформой .NET.

За дополнительной информацией о библиотеках .NET вы можете обратиться к приложению А, где приводится краткий обзор доступных библиотек.

Комментарии

Подобно любым языкам программирования, F# позволяет добавлять комментарии в программный код. Однострочные комментарии начинаются с двух символов слеша // – все, что следует за ними до конца строки, будет игнорироваться компилятором:

```
// Код завершения программы
```

Для добавления более объемных описаний, занимающих несколько строк, можно использовать многострочные комментарии, которые заключаются в пары символов (* и *):

```
(*
Mega Hello World:
Принимает два параметра командной строки и выводит их
вместе со значением текущего времени в окно консоли.
*)
```

При создании приложений на языке F# в среде Visual Studio можно использовать еще одну разновидность комментариев: *комментарии XML-документирования*. Если строка комментария начинается с трех символов слеша /// и находится непосредственно перед идентификатором, Visual Studio будет отображать текст комментария при наведении указателя мыши на этот идентификатор.

На рис. 1.3 приводится пример XML-документирования и соответствующая ему всплывающая подсказка.

F# Interactive

К настоящему моменту мы уже написали некоторый код на языке F# и выполнили его. Далее в книге вы встретите еще множество примеров. При опробовании примеров из книги вы *могли бы* каждый раз создавать новые проекты, однако в среде Visual Studio имеется очень удобный инструмент, который называется F# Interactive, или FSI. Используя окно FSI, вы поймете, что этот инструмент не только упрощает работу с примерами из этой книги, но и помогает разрабатывать приложения.

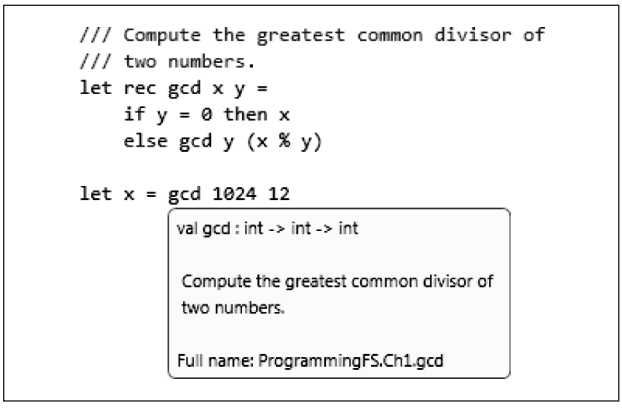


Рис. 1.3. *Комментарии документирования XML*

F# Interactive (Интерактивный сеанс F#) относится к классу инструментов, известных под названием *REPL* (Read-Evaluate-Print Loop – цикл чтения-вычисления-вывода). Этот инструмент принимает код F#, компилирует его, выполняет и выводит результаты. Это позволяет легко и быстро экспериментировать с программным кодом F#, не создавая новые проекты или полноценные приложения только ради того, чтобы увидеть результаты работы фрагмента из пяти строк.

При использовании языков C# и VB.NET вы должны скомпилировать и затем запустить свое приложение, чтобы увидеть результаты, что существенно усложняет процесс работы с небольшими фрагментами программного кода. Конечно, во время отладки в среде Visual Studio можно использовать Immediate Window (Окно интерпретации), но оно может использоваться только для вычисления выражений и не позволяет писать программный код, такой как определение новых функций или классов.

В Visual Studio с типичными настройками окно F# Interactive можно открыть, нажав комбинацию клавиш Control+Alt+F. Как только окно FSI будет открыто, в него можно вводить код F#, пока вы не введете последовательность `;;` и символ перевода строки. Введенный код будет скомпилирован и выполнен, как показано на рис. 1.4.

После выполнения каждого фрагмента в окне FSI все имена, созданные в нем, будут выводиться в виде `val <имя>`. Если для выражения не было указано имя, ему автоматически будет присвоено имя `it`. Вслед за именем идентификатора будет выводиться символ двоеточия, *тип* результата и фактическое значение. Например, на рис. 1.4 видно, что было создано значение типа `int` с именем `x` и со значением, равным `42`.

Если вы не используете Visual Studio, можно воспользоваться консольной версией F# Interactive – утилитой `fsi.exe`, которая находится в том же каталоге, что и `fsc.exe`.

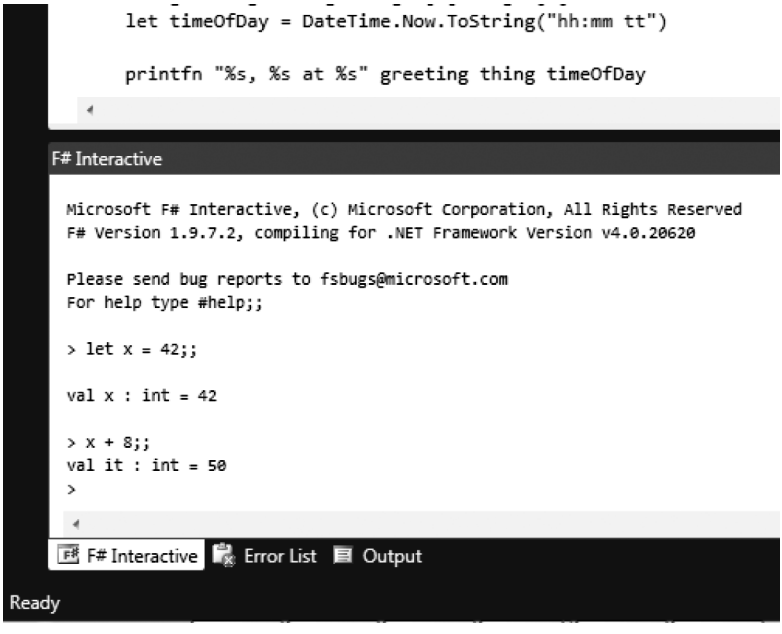


Рис. 1.4. *Окно F# Interactive*

Попробуйте выполнить следующие фрагменты в окне FSI. Обратите внимание, что каждый фрагмент завершается символами ;;:

```
> 2 + 2;;
val it : int = 4
> // Создаем два значения
let x = 1
let y = 2.3;;

val x : int = 1
val y : float = 2.3

> float x + y;;
val it : float = 3.3
> let cube x = x * x * x;;

val cube : int -> int

> cube 4;;
val it : int = 64
```

FSI существенно упрощает тестирование и отладку приложений благодаря тому, что в Visual Studio вы можете переслать F# код из своего текущего проекта в окно FSI, выделив его и нажав комбинацию клавиш Alt+Enter.

Если в окне редактора Visual Studio выделить весь код примера 1.1 и нажать Alt+Enter, в окне FSI вы увидите следующее:

```
>

val main : string []-> int
```

Это позволяет писать код в редакторе Visual Studio, пользуясь такими возможностями, как подсветка синтаксиса и IntelliSense, а тестировать его – с помощью окна FSI. Вы могли бы проверить работу метода `main`, скопировав его в окно `FSI` и просто вызвав:

```
> main [| "Hello"; "World" |];;
Hello, World at 10:52 AM
val it : int = 0
```

Большинство примеров в этой книге было скопировано непосредственно из окна FSI. Я также рекомендую вам использовать интерактивный сеанс при изучении синтаксиса языка F#.

Управление файлами с исходными кодами F#

Когда вы только начинаете изучать программирование на языке F#, большинство ваших программ будут существовать только в окне FSI или, может быть, в виде одного файла. Однако со временем проекты F# будут быстро расти в размерах, и программный код будет распределен по нескольким файлам и, в конце концов, не по одному проекту.

Управление проектами, содержащими несколько файлов на языке F#, имеет свои особенности. В языке F# имеет большое значение порядок, в котором выполняется компиляция файлов.

Вы можете использовать только те функции и классы, которые были определены выше в этом же файле или в другом файле, скомпилированном перед тем файлом, в котором используется эта функция или класс. Если изменить порядок компиляции исходных файлов, программа может перестать собираться!

Причина, по которой порядок компиляции имеет большое значение, связана с *выводом типов* (type inference) – темой, которая будет рассматриваться в следующей главе.

Исходные файлы F# компилируются в том же порядке, в каком они отображаются в окне Solution Explorer (Обозреватель решений), сверху вниз. Всякий раз, когда создается новый файл, он добавляется в конец списка, однако если необходимо изменить порядок компиляции файлов, можно щелкнуть правой кнопкой мыши на требуемом файле и в контекстном меню выбрать пункт Move Up (Вверх) или Move Down (Вниз), как показано на

рис. 1.5. Кроме того, для переупорядочения файлов можно использовать горячие комбинации клавиш Alt+Стрелка вверх и Alt+Стрелка вниз.

Теперь, когда вы получили представление о том, как компилировать приложения на языке F#, далее в книге мы сосредоточимся исключительно на изучении синтаксиса и семантики этого языка программирования.

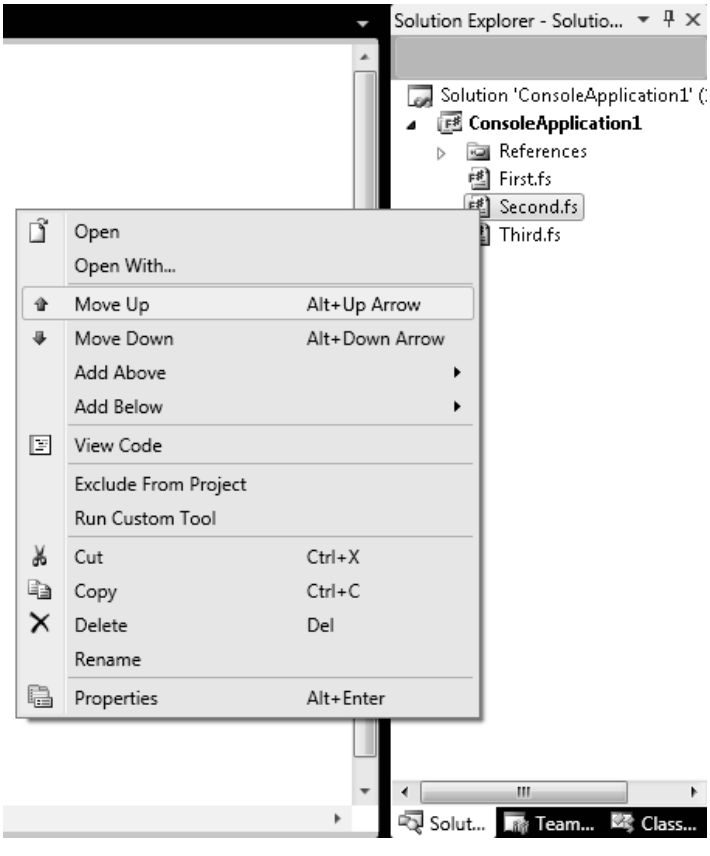


Рис. 1.5. *Переупорядочение файлов в проекте F#*

2

Основы

В главе 1 вы написали свою первую программу на языке F#. Я подробно рассмотрел ее, чтобы вы могли получить представление о том, что она делает, но большая часть кода в ней пока еще остается для вас загадкой. В этой главе я познакомлю вас с фундаментальными понятиями, необходимыми для полного понимания уже написанного кода, но, что еще более важно, я представлю еще несколько примеров, с помощью которых вы сможете усвоить основы F#, прежде чем мы перейдем к изучению более сложных возможностей языка.

В первом разделе этой главы рассматриваются элементарные типы, такие как `int` и `string`, которые являются строительными блоками любых программ на языке F#. Затем я расскажу о функциях, с помощью которых вы сможете манипулировать данными.

В четвертом разделе детально рассматриваются такие фундаментальные типы, как `list`, `option` и `unit`. Понимание принципов использования этих типов позволит вам перейти к изучению объектно-ориентированного и функционального стилей программирования на F#, представленных в последующих главах.

К концу этой главы вы научитесь писать на этом языке простые программы, выполняющие обработку данных. В последующих главах вы узнаете, как расширять возможности программного кода и увеличивать его выразительность, а пока начнем с начала.

Элементарные типы

Тип – это некоторая абстракция, которая в первую очередь имеет отношение к обеспечению целостности. Типы являются своего рода гарантией выполнения необходимых преобразований. Некоторые типы являются достаточно простыми, например тип целых чисел, тогда как другие являются более абстрактными, например функции. Язык F#

относится к языкам со статической типизацией – в том смысле, что проверка типов выполняется на этапе компиляции. Например, если функция принимает целое число в виде параметра, вы получите ошибку на этапе компиляции, если попытаетесь передать ей строку.

Подобно C# и VB.NET, язык F# поддерживает полный комплект *элементарных типов .NET* (которые одинаковы в большинстве языков программирования). Они встроены в F# и отделены от *пользовательских типов*, которые вы определяете сами.

Чтобы создать значение, просто воспользуйтесь операцией *связывания* `let`.

Операция связывания имеет и более широкие возможности, однако мы отложим их рассмотрение до главы 3. А пока достаточно будет знать, что операцию связывания (с использованием ключевого слова `let`) можно использовать для создания нового идентификатора. Например, ниже определяется новое значение `x` в окне сеансе FSI:

```
> let x = 1;;

val x : int = 1
```

Элементарные числовые типы

Элементарные числовые типы делятся на две группы: используемые для представления целых чисел и вещественных чисел. Целочисленные типы могут иметь разные размеры, поэтому значения некоторых типов занимают меньше памяти и могут представлять меньший диапазон чисел. Кроме того, целые числа могут быть знаковыми или беззнаковыми, что определяет, могут ли они иметь отрицательные значения или нет.

Вещественные типы также могут иметь различные размеры – в обмен на увеличение объема занимаемой памяти они могут обеспечивать более высокую точность представления значений.

Для определения новых числовых значений используется оператор связывания `let`, за которым следует целочисленный или вещественный литерал с необязательным суффиксом. Суффикс определяет тип целого или вещественного числа. Полный перечень элементарных числовых типов и их суффиксов приводится в табл. 2.1.

```
> let answerToEverything = 42UL;;

val answerToEverything : uint64 = 42UL

> let pi = 3.1415926M;;

val pi : decimal = 3.1415926M

> let avogadro = 6.022e23;;

val avogadro : float = 6.022e+23
```

Таблица 2.1. Элементарные числовые типы в языке F#

Тип	Суффикс	Тип .NET	Диапазон
byte	uy	System.Byte	от 0 до 255
sbyte	y	System.SByte	от –128 до 127
int16	s	System.Int16	от –32768 до 32767
uint16	us	System.UInt16	от 0 до 65535
int, int32		System.Int32	от -2^{31} до $2^{31}-1$
uint32	u	System.UInt32	от 0 до $2^{32}-1$
int64	L	System.Int64	от -2^{63} до $2^{63}-1$
uint64	UL	System.UInt64	от 0 до $2^{64}-1$
float		System.Double	Вещественное двойной точности согласно стандарту **IEEE64. Представляет значения, состоящие примерно из 15 значащих цифр.**
float32	f	System.Single	Вещественное одинарной точности согласно стандарту **IEEE32. Представляет значения, состоящие примерно из 7 значащих цифр.**
decimal	M	System.Decimal	Вещественный тип фиксированной точности хранит точно 28 цифр после запятой.

Кроме того, в языке F# допускается указывать значения в шестнадцатеричной (по основанию 16), восьмеричной (по основанию 8) и в двоичной (по основанию 2) системах счисления, используя префиксы `0x`, `0o` и `0b` соответственно:

```
> let hex = 0xFCAF;;

val hex : int = 64687

> let oct = 0o7771L;;

val oct : int64 = 4089L

> let bin = 0b00101010y;;

val bin : sbyte = 42y

> (hex, oct, bin);;

val it : int * int64 * sbyte = (64687, 4089L, 42y)
```

Если вы знакомы со стандартами IEEE32 и IEEE64, для представления вещественных чисел вы можете также использовать шестнадца-

теричную, восьмеричную и двоичную формы записи. Компилятор F# автоматически преобразует двоичные значения в вещественные числа. При использовании других систем счисления для представления вещественных чисел используйте суффикс `LF` для преобразования в тип `float` и `lf` – для преобразования в тип `float32`:

```
> 0x401E000000000000LF;;
val it : float = 7.5

> 0x00000000lf;;
val it : float32 = 0.0f
```

Арифметика

К значениям элементарных числовых типов можно применять стандартные арифметические операторы. В табл. 2.2 перечислены все поддерживаемые операторы. Как и в других языках среды CLR (Common Language Runtime – общеязыковая среда исполнения), при делении целых чисел результат округляется до ближайшего меньшего целого, а остаток отбрасывается.

Таблица 2.2. Арифметические операторы

Оператор	Описание	Пример	Результат
`+`	Сложение	`1 + 2`	`3`
`-`	Вычитание	`1 - 2`	`-1`
`*`	Умножение	`2 * 3`	`6`
`/`	Деление	`8L / 3L`	`2L`
`**`	Возведение в степень[a]	`2.0 ** 8.0`	`256.0`
`%`	Деление по модулю (остаток от деления)	`7 % 3`	`1`

[a] Оператор `**` возведения в степень может применяться только к значениям типов `float` и `float32`. Чтобы возвести в степень целочисленное значение, его необходимо либо преобразовать в вещественное число, либо использовать функцию `pown`.

По умолчанию арифметические операторы не выполняют проверку на переполнение, поэтому если, например, при выполнении операции сложения целых чисел произойдет переполнение, в результате получится отрицательное значение. (Аналогично при выполнении операции вычитания, если результат окажется слишком маленьким, чтобы поместиться в целое число, будет возвращено положительное значение.)

```
> 32767s + 1s;;
val it : int16 = -32768s
```

```
> -32768s + -1s;;
val it : int16 = 32767s
```

Если переполнение целых чисел может вызывать проблемы, вам следует подумать о возможности использования более емких типов или применять арифметические операции с проверкой, как описывается в главе 7.

В языке F# поддерживаются все стандартные математические функции, которые можно было бы ожидать; полный их перечень приводится в табл. 2.3.

Таблица 2.3. Стандартные математические функции

Функция	Описание	Пример	Результат
abs	Абсолютное значение числа	abs -1.0	1.0
ceil	Округление вверх до ближайшего целого	ceil 9.1	10
exp	Возведение e в степень	exp 1.0	2.718
floor	Округление вниз до ближайшего целого	floor 9.9	9.0
sign	Знак числа	sign -5	-1
log	Натуральный логарифм	log 2.71828	1.0
log10	Логарифм по основанию 10	log10 1000.0	3
sqrt	Корень квадратный	sqrt 4.0	2.0
cos	Косинус	cos 0.0	1.0
sin	Синус	sin 0.0	0.0
tan	Тангенс	tan 1.0	1.557
pown	Возведение целого числа в степень	pown 2L 10	1024L

Функции преобразования

Один из принципов, которым следует язык F#, заключается в отсутствии каких-либо неявных преобразований. Это означает, что компилятор не будет выполнять неявные автоматические преобразования элементарных типов, такие как преобразование значения типа int16 в значение типа int64. Это устраняет трудноуловимые ошибки, вызванные неявными преобразованиями. Для преобразования значений элементарных типов следует явно использовать функции преобразования, перечисленные в табл. 2.4. Все стандартные функции преобразования принимают значения любых элементарных типов, включая строки и символы.

Таблица 2.4. Функции преобразования элементарных числовых типов

Функция	Описание	Пример	Результат
sbyte	Преобразует значение в тип sbyte	sbyte -5	-5y
byte	Преобразует значение в тип byte	byte "42"	42uy
int16	Преобразует значение в тип int16	int16 'a'	97s
uint16	Преобразует значение в тип uint16	uint16 5	5us
int32, int	Преобразует значение в тип int	int 2.5	2
uint32	Преобразует значение в тип uint32	uint32 0xFF	255
int64	Преобразует значение в тип int64	int64 -8	-8L
uint64	Преобразует значение в тип uint64	uint64 "0xFF"	255UL
float	Преобразует значение в тип float	float 3.1415M	3.1415
float32	Преобразует значение в тип float32	float32 8y	8.0f
decimal	Преобразует значение в тип decimal	decimal 1.23	1.23M

Если эти функции принимают строку, то она разбирается с помощью семейства методов System.Convert, которые в случае недопустимых значений генерируют исключение System.FormatException.

Тип bigint

Если вам приходится иметь дело со значениями, превышающими 2^{64}, можно воспользоваться типом данных bigint, имеющимся в языке F# и позволяющим представлять целые значения произвольной величины. Хотя тип bigint – это всего лишь псевдоним для типа System.Numerics.BigInteger, стоит отметить, что ни в C#, ни в VB.NET нет специального синтаксиса для поддержки целых чисел произвольной величины.

Литералы типа bigint должны оканчиваться символом I. В примере 2.1 определяются значения типа bigint.

Пример 2.1. Тип bigint используется для представления целочисленных значений произвольной величины

```
> open System.Numerics

// Единицы измерения объема данных
let megabyte = 1024I * 1024I
let gigabyte = megabyte * 1024I
let terabyte = gigabyte * 1024I
let petabyte = terabyte * 1024I
let exabyte = petabyte * 1024I
let zettabyte = exabyte * 1024I;;
```

```
val megabyte : BigInteger = 1048576
val gigabyte : BigInteger = 1073741824
val terabyte : BigInteger = 1099511627776
val petabyte : BigInteger = 1125899906842624
val exabyte : BigInteger = 1152921504606846976
val zettabyte : BigInteger = 118059162071741130342 4
```

Несмотря на то, что операции со значениями типа `bigint` чрезвычайно оптимизированы, тем не менее они выполняются существенно медленнее, чем операции над значениями элементарных целочисленных типов.

Битовые операции

Элементарные целочисленные типы поддерживают битовые операции, позволяющие манипулировать отдельными битами. Битовые операторы обычно используются при чтении и записи двоичных данных в файлы. Перечень битовых операций приводится в табл. 2.5.

Таблица 2.5. Битовые операторы

Оператор	Описание	Пример	Результат
`&&&`	Битовое «И»	`0b1111 &&& 0b0011`	`0b0011`
`|||`	Битовое «ИЛИ»	`0xFF00 ||| 0x00FF`	`0xFFFF`
`^^^`	Битовое «Исключающее ИЛИ»	`0b0011 ^^^ 0b0101`	`0b0110`
`<<<`	Сдвиг влево	`0b0001 <<< 3`	`0b1000`
`>>>`	Сдвиг вправо	`0b1000 >>> 3`	`0b0001`

Символы

Платформа .NET основана на использовании Юникода, поэтому символы в ней представляются 2-байтными символами в кодировке UTF-16. Чтобы определить литерал символа, можно просто указать символ Юникода в одиночных кавычках. Символы могут также задаваться в виде шестнадцатеричных кодов символов.

В следующем фрагменте определяется список гласных символов и выводится результат задания символа в виде шестнадцатеричного кода:

```
> let vowels = ['a'; 'e'; 'i'; 'o'; 'u'];;

val vowels : char list = ['a'; 'e'; 'i'; 'o'; 'u']

> printfn "Hex u0061 = '%c'" '\u0061';;
Hex u0061 = 'a'
val it : unit = ()
```

Для представления специальных управляющих символов следует использовать escape-последовательности, перечисленные в табл. 2.6. Экранированные последовательности начинаются с символа обратного слеша, за которым следуют специальные символы.

Таблица 2.6. Экранированные последовательности управляющих символов

Символ	Значение
\'	Апостроф
\"	Кавычка
\\	Обратный слеш
\b	Забой (Backspace)
\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция

Если вам потребуется получить числовое представление символа Юникода в .NET, вы можете передать этот символ любой из функций преобразования, перечисленных в табл. 2.4. Как вариант, чтобы получить байтовое значение литерала символа, можно добавить в литерал суффикс B:

```
> // Преобразовать символ 'C' в целое число
int 'C';;
val it : int = 67
> // Преобразовать символ 'C' в байт
'C'B;;
val it : byte = 67uy
```

Строки

Строковые литералы определяются как последовательности символов, заключенные в кавычки, при этом строковые литералы могут занимать несколько строк. Для доступа к отдельным символам в строке используется синтаксис индексатора .[ ], где в квадратных скобках указывается индекс требуемого символа. Нумерация позиций символов начинается с нуля:

```
> let password = "abracadabra";;

val password : string = "abracadabra"

> let multiline = "This string
takes up
multiple lines";;
```

```
val multiline : string = “This string
takes up
multiple lines”

> multiline.[0];;
val it : char = ‘T’
> multiline.[1];;
val it : char = ‘h’
> multiline.[2];;
val it : char = ‘i’
> multiline.[3];;
val it : char = ‘s’
```

Если необходимо указать длинную строку, ее можно разбить на несколько строк, используя символ обратного слеша \. Если последним символом в строке является символ обратного слеша, то строковый литерал будет продолжен со следующей строки после удаления всех лидирующих пробелов:

```
> let longString = “abc-\
                   def-\
                 ghi”;;

> longString;;
val longString : string = “abc-def-ghi”
```

В строковых литералах допускается использовать escape-последовательности управляющих символов, такие как \t или \\, но это затрудняет определение путей в файловой системе и ключей реестра. Чтобы определить строковый литерал, который не должен подвергаться дополнительным преобразованиям, перед текстом, заключенным в кавычки, следует использовать символ @:

```
> let normalString = “Normal.\n.\n.\t.\t.String”;;

val normalString : string = “Normal.
.
.       .       .String

> let verbatimString = @”Verbatim.\n.\n.\t.\t.String”;;

val verbatimString : string = “Verbatim.\n.\n.\t.\t.String”
```

Аналогично тому как добавление суффикса B к символу позволяет получить байтовое представление символа, добавление B в конец строки позволяет получить представление символов строки в виде массива байтов. (Массивы будут рассматриваться в главе 4.)

```
> let hello = “Hello”B;;

val hello : byte []= [|72uy; 101uy; 108uy; 108uy; 111uy|]
```

Булевы значения

Для работы с величинами, которые могут иметь только значения `true` или `false`, в языке F# имеется тип `bool` (`System.Boolean`), а также несколько стандартных булевых операторов, которые перечислены в табл. 2.7.

Таблица 2.7. Булевы операторы

Оператор	Описание	Пример	Результат
`&&`	Логическое «И»	`true && false`	`false`
`||`	Логическое «ИЛИ»	`true || false`	`true`
`not`	Логическое «НЕ»	`not false`	`true`

В примере 2.2 создаются и выводятся таблицы истинности для булевых функций. В нем определяется функция `printTruthTable`, которая принимает функцию с именем `f` в виде параметра. Эта функция вызывается для определения значения в каждой ячейке таблицы истинности и полученный результат выводится на экран. Затем в функцию `printTruthTable` передаются операторы `&&` и `||`.

Пример 2.2. Вывод таблиц истинности

```
> // Выводит таблицу истинности для заданной функции
let printTruthTable f =
    printfn "       |true   | false |"
    printfn "       +-------+-------+"
    printfn " true  | %5b | %5b |" (f true true) (f true false)
    printfn " false | %5b | %5b |" (f false true) (f false false)
    printfn "       +-------+-------+"
    printfn ""
     ();;

val printTruthTable : (bool -> bool -> bool) -> unit

> printTruthTable (&&);;
       |true   | false |
       +-------+-------+
 true  | true  | false |
 false | false | false |
+-------+-------+

val it : unit = ()
> printTruthTable (||);;
       |true   | false |
       +-------+-------+
 true  | true  | true  |
 false | true  | false |
       +-------+-------+

val it : unit = ()
```

В языке F# используется сокращенная схема вычисления булевых выражений (short circuit), то есть если результат может быть определен после вычисления первого выражения из двух, второе значение не вычисляется. Например, выражение

```
true || f()
```

вернет `true`, при этом функция `f` вызываться не будет. Точно так же выражение

```
false && g()
```

вернет `false`, функция `g` также вызываться не будет.

Сравнение и равенство

Для сравнения числовых значений в языке F# используются стандартные операторы «больше», «меньше» и «равно», перечисленные в табл. 2.8. Все операторы сравнения возвращают логическое значение, тогда как функция `compare` возвращает значение –1, 0 или 1 в зависимости от того, является первый параметр меньше, равным или больше второго параметра.

Таблица 2.8. Операторы сравнения

Оператор	Описание	Пример	Результат
`<`	Меньше	`1 < 2`	`true`
`<=`	Меньше или равно	`4.0 <= 4.0`	`true`
`>`	Больше	`1.4e3 > 1.0e2`	`true`
`>=`	Больше или равно	`0I >= 2I`	`false`
`=`	Равно	`"abc" = "abc"`	`true`
`<>`	Не равно	`'a' <> 'b'`	`true`
`compare`	Сравнивает два значения	`compare 31 31`	`0`

Равенство в .NET является сложной темой. Существует *равенство значений* и *равенство ссылок*. Применительно к значимым типам понятие равенства подразумевает идентичность значений, например `1 = 1`. Однако применительно к ссылочным типам равенство определяется путем переопределения метода `Equals`, объявленного в классе `System.Object`. За дополнительной информацией обращайтесь к разделу «Равенство объектов» в главе 5.

Функции

Теперь, когда мы получили представление обо всех элементарных типах в языке F#, давайте определим функции, которые будут манипулировать ими.

Функции определяются точно так же, как значения, с той лишь разницей, что все, что следует за именем функции, является ее параметрами. Ниже приводится пример определения функции с именем `square`, которая принимает целочисленный аргумент `x` и возвращает его квадрат:

```
> let square x = x * x;;

val square : int -> int

> square 4;;
val it : int = 16
```

В отличие от языка C#, в F# отсутствует ключевое слово `return`. Поэтому в качестве возвращаемого значения функции используется результат вычисления последнего выражения.

Напишем еще одну функцию, которая добавляет 1 к значению аргумента:

```
> let addOne x = x + 1;;

val addOne : int -> int
```

Вывод в окне FSI показывает, что функция имеет тип `int -> int`, который читается так: «функция принимает целое число и возвращает целое число». При наличии нескольких входных параметров сигнатура функции приобретает более сложный вид:

```
> let add x y = x + y;;

val add : int -> int -> int
```

Говоря техническим языком, тип `int -> int -> int` читается так: «функция, принимающая целое число, возвращает функцию, которая принимает целое число и возвращает целое число». Не стоит сразу пугаться фразы «функция возвращает функцию». Единственное, что вам сейчас нужно запомнить, – это то, что для вызова функции ей следует передать параметры, разделенные пробелами:

```
> add 1 2;;

val it : 3
```

Вывод типов

Язык F# относится к языкам со статической типизацией, поэтому при попытке вызвать только что созданный нами метод `add` с вещественным значением компилятор выдаст сообщение об ошибке:

```
> add 1.0 2.0;;

  add 1.0 2.0;;
  ----^^^

stdin(3,5): error FS0001: This expression was expected to have type
```

```
        int
but here has type
        float.
```

(В данном выражении требовалось наличие типа int, но получен тип float.)[1]

Здесь вполне резонно возникает вопрос: почему компилятор решил, что функция может принимать только целые числа? Оператор + может применяться и к вещественным числам!

Причина заключается в механизме *вывода типов (type inference)*. В отличие от C#, компилятор F# не требует от программиста, чтобы он явно указывал типы всех параметров функции. Компилятор сам определяет их типы исходя из использования этой функции.

Будьте внимательны и не путайте вывод типов с *динамической типизацией*. Хотя язык F# позволяет опускать объявления типов, это не означает, что не будет выполняться проверка типов на этапе компиляции.

Оператор + может применяться к различным типам, таким как `byte`, `int` и `decimal`, поэтому при отсутствии дополнительной информации о типах компилятор по умолчанию просто предполагает, что аргументы имеют тип `int`.

Ниже приводится сеанс в окне FSI, где объявляется функция, которая возвращает произведение двух значений. Так же как и в случае использования оператора +, компилятор делает вывод, что ей будут передаваться целочисленные значения, потому что никакой дополнительной информации об ее использовании ему не предоставляется:

```
> // Нет никакой дополнительной информации для вывода типов
  // на основе использования этой функции
let mult x y = x * y;;

val mult : int -> int -> int
```

Если в интерактивном сеансе помимо определения функции `mult` добавить ее вызов с двумя вещественными значениями, сигнатура функции будет выведена как `float -> float -> float`:

```
> // Механизм выведения типов в действии
let mult x y = x * y
let result = mult 4.0 5.5;;

val mult : float -> float -> float
val result : float = 22.0
```

[1] Напоминаю, что перевод сообщений об ошибках взят из русифицированной версии Visual Studio. – *Прим. перев.*

При этом имеется возможность явно добавить *аннотацию типов (type annotation)*, или подсказку для компилятора F# о типах аргументов. Чтобы добавить аннотацию типов, достаточно просто заменить параметр функции следующим образом:

```
ident -> (ident : type)
```

где *type* – это тип параметра. Чтобы указать, что первый аргумент нашей функции `add` имеет тип `float`, достаточно просто переопределить функцию, как показано ниже:

```
> let add (x : float) y = x + y;;

val add : float -> float -> float
```

Обратите внимание, как из-за добавления аннотации типа для `x` тип функции изменился на `float -> float -> float`. Дело в том, что единственная перегруженная версия оператора `+`, принимающего вещественное число в первом операнде, имеет тип `float -> float -> float`, поэтому компилятор F# выводит тип аргумента `y` как `float`.

Механизм вывода типов существенно уменьшает захламленность программного кода, вынуждая компилятор автоматически определять, какие типы использовать. Однако в некоторых случаях аннотации типов просто необходимы, а иногда служат для улучшения читабельности кода.

Обобщенные функции

Можно написать функцию, которая будет работать с параметрами любых типов. Например, следующая функция просто возвращает входной аргумент:

```
> let ident x = x;;

val ident : 'a -> 'a

> ident "a string";;
val it : string = "a string"
> ident 1234L;;
val it : int64 = 1234L
```

Поскольку в данной ситуации механизм вывода типов не смог определить конкретный тип аргумента `x` функции `ident`, тип аргумента стал обобщенным. Если параметр становится *обобщенным (generic)*, это означает, что он может быть значением любого типа, таким как целое число, строка или вещественное число.

Имеется возможность явно объявлять обобщенные параметры, указывая в качестве имени типа любой допустимый идентификатор, перед которым стоит апостроф. Обычно для этих целей используются алфа-

витные символы, начиная с «a». В следующем фрагменте приводится определение функции ident с использованием аннотации типа, которая явно объявляет параметр x как обобщенный:

```
> let ident2 (x : 'a) = x;;

val ident2 : 'a -> 'a
```

Написание обобщенного кода имеет большое значение для его повторного использования. Мы продолжим обсуждение вопросов, связанных с механизмом вывода типов и обобщенными функциями, далее в этой книге, поэтому пока не следует стремиться вникнуть во все подробности. Просто имейте в виду, что всякий раз, когда вы увидите тип 'a, это может быть тип int, float, string, пользовательский тип и так далее.

Область видимости

Каждое значение, объявленное в коде F#, имеет определенную область видимости, которая определяет границы, в которых это значение может использоваться. (Говоря более формальным языком, это называется *областью действия объявления (declaration space)*.) По умолчанию областью видимости значений является модуль, то есть они могут использоваться везде после их объявления. Однако значения, объявленные внутри функции, доступны только внутри этой функции. То есть внутри функции можно использовать любые значения, объявленные ранее снаружи функции, но снаружи нельзя сослаться на значения, объявленные внутри.

В следующем примере значение moduleValue объявлено в области видимости модуля и используется внутри функции. Другое значение functionValue объявлено внутри функции, и любые попытки обратиться к нему из-за пределов его области видимости приводят к ошибке:

```
> // Область видимости
let moduleValue = 10
let functionA x =
    x + moduleValue;;

val moduleValue : int = 10
val functionA : int -> int

> // Вариант с ошибкой
let functionB x =
    let functionValue = 20
    x + functionValue

// Обращение к значению 'functionValue' за пределами его области видимости
functionValue;;

    functionValue;;
    ^^^^^^^^^^^^^
```

```
error FS0039:  The value or constructor 'functionValue' is not defined.
```

(Значение или конструктор 'functionValue' не определено.)

Изучение подробных правил областей видимости значений на первый взгляд может показаться не особенно важным, тем не менее эти подробности имеют большое значение, потому что в языке F# допускается определять вложенные функции. В F# вы можете объявить новую функцию в теле другой функции. Вложенные функции имеют доступ к любым значениям, объявленным в объемлющих областях видимости, например в родительской функции или в модуле, а также к значениям, объявленным в самой вложенной функции.

В следующем фрагменте приводится пример вложенной функции. Обратите внимание, что внутри функции `g` имеется возможность использовать параметр `fParam` объемлющей функции `f`:

```
> // Вложенные функции
let moduleValue = 1

let f fParam =

    let g gParam = fParam + gParam + moduleValue

        let a = g 1
        let b = g 2
        a + b;;

val moduleValue : int = 1
val f : int -> int
```

Может показаться, что определение функции внутри другой функции только приводит к путанице, однако в действительности возможность ограничивать область видимости функций является очень удобной. Она помогает предотвратить засорение пространства имен родительского модуля, позволяя создавать маленькие специфические функции, доступные ровно в той области видимости, где они используются. Это преимущество станет более очевидным, как только мы начнем изучать функциональный стиль программирования в главе 3.

При использовании вложенных функций может оказаться достаточно сложным уследить за всеми значениями в области видимости. Что если вам потребуется объявить значение с именем `x`, но это значение уже используется в родительской области видимости? В языке F# наличие двух значений с одним и тем же именем не приводит к ошибке компиляции – просто новое объявление *маскирует* (*shadowing*) прежнее. В таких ситуациях оба значения одновременно присутствуют в памяти, но нет никакого способа получить доступ к значению, объявленному ранее, – «побеждает» последнее объявление.

В следующем фрагменте объявляется функция, которая принимает параметр `x`, а затем объявляет несколько новых значений с именем `x`:

```
> // Преобразование количества байтов в гигабайты
let bytesToGB x =
    let x = x / 1024I // Переводим из байтов в килобайты
    let x = x / 1024I // из килобайтов в мегабайты
    let x = x / 1024I // из мегабайтов в гигабайты
    x;;

val bytesToGB : System.Numerics.BigInteger -> System.Numerics.BigInteger

> let hardDriveSize = bytesToGB 268435456000I;;
val hardDriveSize : System.Numerics.BigInteger = 250
```

В предыдущем примере после каждого оператора связывания `let` значение с именем `x` маскируется и замещается новым значением. На первый взгляд может показаться, что значение `x` просто изменяется, но в действительности каждый раз создается новое значение с тем же самым именем `x`. Следующий фрагмент демонстрирует, во что компилируется этот код:

```
let bytesToGB x =
    let x_2 = x / 1024I   // Переводим из байтов в килобайты
    let x_3 = x_2 / 1024I // из килобайтов в мегабайты
    let x_4 = x_3 / 1024I // из мегабайтов в гигабайты
    x_4
```

Этот прием преднамеренного маскирования значений удобно использовать для создания иллюзии обновления значений без фактического их изменения. Если вам действительно потребуется обновить значение `x`, вам следует объявить его с использованием ключевого слова `mutable`, о чем рассказывается в главе 4.

Управление потоком выполнения

Внутри функций можно изменять поток выполнения с помощью ключевого слова `if`. Условное выражение должно иметь тип `bool`, и если результатом его вычисления является значение `true`, выполняется вложенный блок кода, что в следующем примере выводит сообщение на консоль:

```
> // Условные инструкции
let printGreeting shouldGreet greeting =
    if shouldGreet then
        printfn "%s" greeting;;

val printGreeting : bool -> string -> unit

> printGreeting true "Hello!";;
Hello!
val it : unit = ()
> printGreeting false "Hello again!";;
val it : unit = ()
```

Более сложные ветвления могут быть реализованы с помощью *условных выражений (if-expression)*.

Условные выражения действуют именно так, как от них и ожидается: если условное выражение возвращает `true`, то выполняется первый блок кода, в противном случае выполняется второй блок. Однако есть одно обстоятельство, которое существенно отличает F# от других языков программирования: условные выражения в языке F# возвращают значение.

В следующем примере результат, возвращаемый условным выражением, связывается с именем `result`. То есть если условие `x % 2 = 0` выполняется, то значением `result` будет `"Yes it is"`, в противном случае `"No it is not"`:

```
> // Условные выражения
let isEven x =
    let result =
        if x % 2 = 0 then
            "Yes it is"
        else
            "No it is not"
    result;;

val isEven : int -> string

> isEven 5;;
val it : string = "No it is not"
```

Для организации более сложных ветвлений допускается использовать условные выражения, вложенные друг в друга, но сопровождать такой программный код становится намного сложнее:

```
let isWeekend day =
    if day = "Sunday" then
        true
    else
        if day = "Saturday" then
            true
        else
            false
```

В языке F# имеется синтаксический помощник, помогающий избегать использования глубоко вложенных условных выражений, – ключевое слово `elif`. С его помощью можно связывать цепочки условных выражений, избегая вложенности:

```
let isWeekday day =
    if day = "Monday"      then true
    elif day = "Tuesday"   then true
```

```
    elif day = "Wednesday" then true
    elif day = "Thursday"  then true
    elif day = "Friday"    then true
    else false
```

Поскольку результатом условного выражения является значение, все операторы (clause) в таком выражении должны возвращать значения одного и того же типа:

```
> // ОШИБКА: Операторы в условном выражении возвращают значения различных типов
let x =
    if 1 > 2 then
        42
    else
        "a string";;

      else          "a string";;
  ------------------^^^^^^^^^^^

stdin(118,19): error FS0001: This expression was expected to have type
        int
but here has type
        string.
stopped due to error
```

(В данном выражении требовалось наличие типа int, но получен тип string, остановлено из-за ошибки.)

Но что если у вас есть только ветка `if` без соответствующей ветки `else`? В этом случае операторы должны возвращать тип `unit`. Тип `unit` — это специальный тип в языке F#, который по сути означает «нет значения». (Разработчики на C# могут рассматривать тип `unit` как аналог типа `void`.) Подробнее этот тип мы будем рассматривать чуть ниже.

Условные выражения в следующем примере не вызывают сообщения об ошибках только благодаря тому, что функция `printfn` возвращает тип `unit`:

```
> // Тело инструкции if возвращает unit
let describeNumber x =
    if x % 2 = 0 then
        printfn "x is a multiple of 2"
    if x % 3 = 0 then
        printfn "x is a multiple of 3"
    if x % 5 = 0 then
        printfn "x is a multiple of 5"
     ();;

val describeNumber : int -> unit

> describeNumber 18;;
x is a multiple of 2
```

```
x is a multiple of 3
val it : unit = ()
```

Основные типы

Выше мы познакомились с элементарными типами данных, доступными на платформе .NET, но одних этих типов недостаточно, чтобы писать более или менее разумные программы. Библиотека F# включает несколько ключевых типов, позволяющих организовывать, обрабатывать и управлять данными. В табл. 2.9 перечислены фундаментальные типы, которые используются практически во всех приложениях на языке F#.

Таблица 2.9. Основные типы данных в языке F#

Сигнатура	Название	Описание	Пример
`unit`	Пустой тип	Пустое значение	`()`
`int, float`	Определенный тип	Определенный тип	`42, 3.14`
`'a, 'b`	Обобщенный тип	Обобщенный (свободный) тип	
`'a -> 'b`	Функция	Функция, возвращающая значение	`fun x -> x + 1`
`'a * 'b`	Кортеж	Упорядоченная коллекция значений	`(1, 2), ("eggs", "ham")`
`'a list`	Список	Список значений	`[ 1; 2; 3], [1 .. 3]`
`'a option`	Необязательный тип	Необязательное значение	`Some(3), None`

Тип unit

Тип `unit` — это пустое значение. Тип `unit` можно рассматривать как аналог типа `void`, а значения этого типа представляются в коде как `()`:

```
> let x = ();;

val x : unit

> ();;
val it : unit = ()
```

Условные выражения без ветки `else` должны возвращать значение типа `unit`, потому что в противном случае непонятно, что должно быть в результате, если это условие не будет выполнено. Кроме того, в языке F# все функции должны возвращать некоторое значение, поэтому если функция в принципе ничего не возвращает, как, например, функция `printf`, она должна возвращать значение типа `unit`.

Если вы хотите вернуть значение unit, можно воспользоваться функцией ignore, которая затеняет значение, возвращаемое этой функцией. Обычно она используется в случаях, когда функция вызывается ради побочного эффекта и возвращаемое значение не представляет интереса:

```
> let square x = x * x;;

val square : int -> int

> ignore (square 4);;
val it : unit = ()
```

Кортежи

Кортеж (tuple) – это упорядоченная коллекция данных и наиболее простой способ группировки различных элементов в одну структуру. Например, кортежи могут использоваться для отслеживания промежуточных результатов вычислений.

В языке F# кортежи основаны на типе System.Tuple<_>, однако на практике вам никогда не придется использовать класс Tuple<_> напрямую.

Чтобы создать экземпляр кортежа, достаточно указать список значений, разделенных запятыми и (необязательно) заключить его в круглые скобки. Тип кортежа обозначается с помощью списка типов его элементов, разделенных символами *. В следующем примере dinner – это экземпляр кортежа, а string * string – это тип данного кортежа:

```
> let dinner = ("green eggs", "ham");;

val dinner : string * string = ("green eggs", "ham")
```

Кортежи могут содержать любое количество элементов любых типов. Фактически можно даже создать кортеж, который будет содержать другие кортежи!

Следующий фрагмент определяет два кортежа. Первый, с именем zeros, представляет собой кортеж, содержащий различные представления нулевых значений. Второй, nested, содержит вложенные кортежи. Этот кортеж содержит три элемента, из которых второй и третий сами являются кортежами:

```
> let zeros = (0, 0L, 0I, 0.0);;

val zeros : int * int64 * bigint * float = (0, 0L, 0I, 0.0)

> let nested = (1, (2.0, 3M), (4L, "5", '6'));;

val nested : int * (float * decimal) * (int64 * string * char) = ...
```

Для извлечения значений из кортежа, содержащего два элемента, можно использовать функции fst и snd. Функция fst возвращает первый элемент кортежа, а функция snd – второй:

```
> let nameTuple = ("John", "Smith");;

val nameTuple : string * string = ("John", "Smith")

> fst nameTuple;;
val it : string = "John"
> snd nameTuple;;
val it : string = "Smith"
```

Альтернативный способ извлечения элементов кортежей заключается в использовании оператора связывания let. Если сразу за оператором let указать несколько идентификаторов, разделенных запятыми, с этими именами будут связаны значения элементов кортежа.

В следующем примере создается кортеж с именем snacks, а затем из него извлекаются значения элементов и связываются с тремя новыми идентификаторами x, y и z:

```
> let snacks = ("Soda", "Cookies", "Candy");;

val snacks : string * string * string = ("Soda", "Cookies", "Candy")

> let x, y, z = snacks;;

val z : string = "Soda"
val y : string = "Cookies"
val x : string = "Candy"

> y, z;;
val it : string * string = ("Cookies", "Candy")
```

Вы получите ошибку компиляции, если попытаетесь извлечь из кортежа слишком много или слишком мало значений:

```
> let x, y = snacks;;

  let x, y = snacks;;
  -----------^^^^^^

stdin(8,12): error FS0001: Type mismatch. Expecting a
        string * string
but given a
        string * string * string.
The tuples have differing lengths of 2 and 3.
```

*(Несоответствие типов. Требуется string * string, но получен string * string * string. Кортежи имеют разную длину – 2 и 3.)*

Кортежи можно передавать функциям в виде параметров, как и любые другие значения. Функции, принимающие единственный кортеж, име-

ют совершенно иное значение, когда речь заходит о функциональном стиле программирования; см. раздел «Частичное применение функции» в главе 3.

В следующем примере функция `tupledAdd` принимает два параметра, `x` и `y`, в виде кортежа. Обратите внимание на различия в сигнатурах функций `add` и `tupledAdd`:

```
> let add x y = x + y;;

val add : int -> int -> int

> let tupledAdd(x, y) = x + y;;

val tupledAdd : int * int -> int

> add 3 7;;
val it : int = 10

> tupledAdd(3, 7);;
val it : int = 10
```

Списки

Кортежи группируют значения в единую сущность, а списки объединяют данные в виде цепочки. Такой подход позволяет обрабатывать сразу все элементы списка с помощью агрегатных операторов, которые обсуждаются чуть ниже.

Самый простой способ объявить список состоит в том, чтобы указать список значений, разделенных точками с запятой, заключенный в квадратные скобки. Позднее вы узнаете, как объявлять списки с использованием более мощного синтаксиса генераторов списков (list comprehension syntax). Пустой список, не имеющий элементов, объявляется с помощью пустых квадратных скобок `[]`:

```
> // Объявление списков
let vowels = ['a'; 'e'; 'i'; 'o'; 'u']
let emptyList = [];;

val vowels : char list = ['a'; 'e'; 'i'; 'o'; 'u']
val emptyList : 'a list = []
```

В этом примере пустой список имеет тип `'a list`, потому что он может иметь любой тип и механизм вывода типов не способен определить конкретный тип.

В отличие от списков в других языках программирования, списки в языке F# имеют весьма ограниченные возможности доступа к элементам и управления ими. Фактически списки поддерживают всего две операции. (Чтобы увидеть, как это ограничение можно превратить в преимущество, обращайтесь к главе 7.)

Первая элементарная операция над списками – операция добавления, которая выполняется с помощью оператора ::. Этот оператор добавляет элемент в начало списка. В следующем примере значение 'y' добавляется в начало списка vowels:

```
> // Использование оператора добавления в начало
let sometimes = 'y' :: vowels;;

val sometimes : char list = ['y'; 'a'; 'e'; 'i'; 'o'; 'u']
```

Вторая элементарная операция над списками – операция объединения, которая выполняется с помощью оператора @. Этот оператор объединяет два списка. В следующем примере выполняется объединение двух списков, odds и evens, в результате чего создается новый список:

```
> // Использование оператора объединения
let odds = [1; 3; 5; 7; 9]
let evens = [2; 4; 6; 8; 10]

val odds : int list = [1; 3; 5; 7; 9]
val evens : int list = [2; 4; 6; 8; 10]

> odds @ evens;;
val it : int list = [1; 3; 5; 7; 9; 2; 4; 6; 8; 10]
```

Диапазоны списков

Объявление элементов списков в виде перечней значений, разделенных точками с запятой, быстро превращается в утомительное занятие, особенно при работе с длинными списками. Объявить список упорядоченных числовых значений можно с помощью синтаксиса диапазонов. Первое выражение в диапазоне определяет минимальное значение, а второе – максимальное. В результате получается список последовательных значений от минимального до максимального с шагом, равным 1:

```
> let x = [1 .. 10];;

val x : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Если указано необязательное значение шага, то в результате будет создан список значений в диапазоне между двумя числами с заданным шагом. Обратите внимание, что шаг может иметь отрицательное значение:

```
> // Диапазоны списков
let tens = [0 .. 10 .. 50]
let countDown = [5L .. -1L .. 0L];;

val tens : int list = [0; 10; 20; 30; 40; 50]
val countDown : int list = [5L; 4L; 3L; 2L; 1L; 0L]
```

Генераторы списков

Наиболее выразительный способ создания списков заключается в использовании *генераторов списков (list comprehensions)*, которые позволяют создавать списки непосредственно в коде F#. В самом простом случае генератор списков – это некоторый программный код, заключенный в квадратные скобки [ ]. Тело генератора списков будет выполняться, пока не завершится, а сам список будет создан из элементов, возвращаемых с помощью ключевого слова `yield`. (Обратите внимание, что список создается в памяти целиком – элементы списков не вычисляются отложенно (lazily), как в случае использования типа `seq<_>`, который обсуждается в следующей главе.)

```
> // Простой генератор списков
let numbersNear x =
    [
        yield x - 1
        yield x
        yield x + 1
    ];;

val numbersNear : int -> int list

> numbersNear 3;;
val it : int list = [2; 3; 4]
```

Внутри генераторов списков допускается использовать практически любые возможности языка F#, включая объявления функций и циклы `for`. В следующем фрагменте демонстрируется генератор списков, который объявляет функцию `negate` и возвращает числа от 1 до 10, инвертируя знак четных значений:

```
> // Более сложный генератор списков
let x =
    [ let negate x = -x
      for i in 1 .. 10 do
          if i % 2 = 0 then
              yield negate i
          else
            yield i ];;

val x : int list = [1; -2; 3; -4; 5; -6; 7; -8; 9; -10]
```

При использовании циклов `for` внутри генераторов списков программный код можно упростить, используя `->` вместо `do yield`. Ниже приводятся два идентичных фрагмента кода:

```
// Генерирует первые десять чисел, кратных заданному числу
let multiplesOf x = [ for i in 1 .. 10 do yield x * i ]
```

```
// Упрощенный генератор списков
let multiplesOf2 x = [ for i in 1 .. 10 -> x * i ]
```

Синтаксис генераторов списков позволяет быстро и лаконично создавать списки данных, которые затем могут обрабатываться в вашем коде. В примере 2.3 показано, как можно использовать генератор списков для создания списка простых чисел, меньших заданного значения.

В этом примере выполняется обход всех чисел от 1 до указанного максимального значения. Затем с помощью генератора списков создаются все множители каждого числа. Если полученный список множителей содержит всего два элемента, то это число возвращается инструкцией `yield`, так как оно является простым. Безусловно, существуют более эффективные способы нахождения простых чисел, но этот пример показывает, насколько выразительными могут быть генераторы списков.

Пример 2.3. Использование генератора списков для нахождения простых чисел

```
> // Использование генератора списков для нахождения простых чисел
let primesUnder max =
    [
        for n in 1 .. max do
            let factorsOfN =
                [
                    for i in 1 .. n do
                        if n % i = 0 then
                            yield i
                ]

            // n - простое число, если для него имеется
            // всего два делителя, 1 и n
            if List.length factorsOfN = 2 then
                yield n
    ];;

val primesUnder : int -> int list

> primesUnder 50;;
val it : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47]
```

Функции модуля List

Модуль `List` из стандартной библиотеки F# содержит множество методов, упрощающих обработку списков. Эти встроенные методы, перечисленные в табл. 2.10, станут для вас основными инструментами обработки списков.

Таблица 2.10. Функции в модуле List

Функция и тип	Описание
`List.length` `'a list -> int`	Возвращает длину списка.
`List.head` `'a list -> 'a`	Возвращает первый элемент списка.
`List.tail` `'a list -> 'a list`	Возвращает часть списка без первого элемента.
`List.exists` `('a -> bool) -> 'a list -> bool`	Возвращает признак того, удовлетворяет ли какой-либо элемент списка функции поиска или нет.
`List.rev` `'a list -> 'a list`	Изменяет порядок следования элементов в списке на противоположный.
`List.tryfind` `('a -> bool) -> 'a list -> 'a option`	Возвращает `Some(x)`, где `x` – первый элемент списка, для которого указанная функция вернула значение `true`. В противном случае возвращает `None`. (Значения `Some` и `None` мы рассмотрим чуть ниже.)
`List.zip` `'a list -> 'b list -> ('a * 'b) list`	Принимает два списка одинаковой длины. Возвращает объединенный список кортежей.
`List.filter` `('a -> bool) -> 'a list -> 'a list`	Возвращает список, содержащий только те элементы, для которых указанная функция вернула значение `true`.

`List.partition`
`('a -> bool) -> 'a list ->`
`('a list * 'a list)`
	Принимает функцию-предикат и список, возвращает два новых списка. Первый список содержит элементы, для которых указанная функция вернула значение `true`, а второй – элементы, для которых эта функция вернула значение `false`.

Вначале может быть не совсем понятно, как правильно использовать некоторые функции модуля `List`, но очень скоро вы сможете определять, что делает функция, по ее сигнатуре.

В следующем примере демонстрируется использование функции `List.partition`, которая разбивает список чисел от 1 до 15 на два новых списка: один из них содержит числа, кратные пяти, а второй – все остальные. Обратите внимание, что функция `List.partition` возвращает кортеж, а значения `multOf5` и `nonMultOf5` связываются с элементами этого кортежа одновременно:

```
> // Использование функции List.partition
let isMultipleOf5 x = (x % 5 = 0)
```

```
let multOf5, nonMultOf5 =
    List.partition isMultipleOf5 [1 .. 15];;

val isMultipleOf5 : int -> bool
val nonMultOf5 : int list = [1; 2; 3; 4; 6; 7; 8; 9; 11; 12; 13; 14]
val multOf5 : int list = [5; 10; 15]
```

Что представляет собой модуль `List`? Все эти функции определены в модуле `List`, в пространстве имен `Microsoft.FSharp.Collections`. Модуль `Microsoft.FSharp.Collections` импортируется автоматически, поэтому чтобы вызвать любой из этих методов, достаточно квалифицировать имя требуемой функции с помощью имени `List`.

Агрегатные операторы

Хотя списки предоставляют способ объединения данных, тем не менее в этом нет ничего особенного. Истинная мощь списков заключается в *агрегатных операторах (aggregate operators)*, которые представляют собой набор мощных функций, позволяющих манипулировать любыми коллекциями. Вы еще раз встретитесь с этим набором функций, когда мы будем обсуждать последовательности (глава 3) и массивы (глава 4).

Функция List.map

Функция `List.map` – это операция проекции, которая создает новый список на основе заданной функции. Каждый элемент нового списка является результатом выполнения функции. Функция `List.map` имеет тип:

```
('a -> 'b) -> 'a list -> 'b list
```

Вы можете наглядно представить себе функцию отображения `f` на списке `[x; y; z]`, как показано на рис. 2.1.

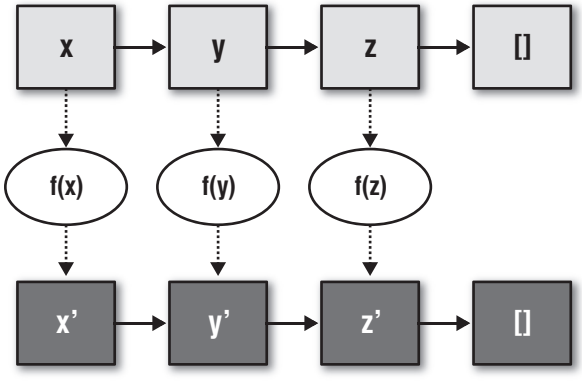


Рис. 2.1. *Визуальное представление работы функции List.map*

В примере 2.4 демонстрируется результат отображения функции вычисления квадрата числа на списке целых чисел.

Пример 2.4. Использование функции List.map для создания списка квадратов чисел

```
> let squares x = x * x;;

val squares : int -> int

> List.map squares [1 .. 10];;
val it : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

Возможно, сейчас в это трудно поверить, тем не менее функция `List.map` является одной из самых полезных функций в языке F#. Она обеспечивает элегантный способ преобразования данных и при многократном использовании способна упростить структуру кода.

Функция List.fold

Свертки (folds) представляют наиболее мощный тип агрегатных операторов, и неудивительно, что они же являются самыми сложными. Когда имеется некоторый список значений и его необходимо свернуть до единственного элемента данных, используется операция свертки.

Существуют два основных типа операций свертки, которые можно применять к спискам. Начнем с функции `List.reduce`, которая имеет следующий тип:

```
('a -> 'a -> 'a) -> 'a list -> 'a
```

Функция `List.reduce` проходит по всем элементам списка и накапливает *значение аккумулятора*, представляющее собой результат обработки списка к текущему моменту. После обработки всех элементов списка функция возвращает окончательное значение аккумулятора. В качестве начального значения аккумулятора функция `List.reduce` принимает первый элемент списка.

В примере 2.5 демонстрируется использование функции `List.reduce` со списком строк, разделенных точками с запятой. Функция `insertCommas` принимает значение аккумулятора и элемент списка и просто возвращает новую строку, которая включает аккумулятор и элемент списка, отделенные запятой. В качестве начального значения аккумулятора функция `List.reduce` использует первый элемент списка, поэтому в результате обработки получается строка, содержащая все элементы списка, разделенные запятыми.

Пример 2.5. Строка из элементов списка, разделенных запятыми, полученная с помощью List.reduce

```
> let insertCommas (acc : string) item = acc + ", " + item;;
```

```
val insertCommas : string -> string -> string

> List.reduce insertCommas ["Jack"; "Jill"; "Jim"; "Joe"; "Jane"];;
val it : string = "Jack, Jill, Jim, Joe, Jane"
```

В следующей таблице приводится состояние аккумулятора после обработки каждого элемента списка:

Аккумулятор	Элемент списка
"Jack" (первый элемент списка)	"Jill" (второй элемент списка)
"Jack, Jill"	"Jim"
"Jack, Jill, Jim"	"Joe"
"Jack, Jill, Jim, Joe"	"Jane"

Несмотря на всю полезность свертки `reduce`, она возвращает аккумулятор того же типа, что и элементы списка. Но как быть в случаях, когда требуется реализовать что-то иное? Например, получить общую стоимость товаров в покупательской корзине.

В случаях, когда требуется использовать произвольный тип аккумулятора, можно воспользоваться функцией `List.fold`. Функция `fold` принимает три параметра. Первый – функция, которая получает аккумулятор и элемент списка и возвращает новое значение аккумулятора. Второй – начальное значение аккумулятора. И последний параметр – список, для которого требуется получить свертку. Возвращаемым значением функции `fold` является окончательное значение аккумулятора. Формально она имеет тип:

```
('acc -> 'b -> 'acc) -> 'acc -> 'b list -> 'acc
```

В качестве простого примера рассмотрим свертку списка целых чисел в их сумму:

```
> let addAccToListItem acc i = acc + i;;

val addAccToListItem : int -> int -> int

> List.fold addAccToListItem 0 [1; 2; 3];;
val it : int = 6
```

Напомню еще раз, что функция `fold` не требует, чтобы аккумулятор имел тот же тип, что и элементы списка. В примере 2.6 демонстрируется свертка строки символов в кортеж, содержащий количество появлений каждой гласной буквы. (Количество символов a, e, i и так далее.)

Функция свертки применяется к каждому символу строки. Если символ представляет собой гласную букву, она возвращает измененное значение аккумулятора, в противном случае возвращается текущее значение аккумулятора.

Пример 2.6. Подсчет количества гласных букв с помощью List.fold

```
> // Подсчет количества гласных букв в строке
let countVowels (str : string) =
    let charList = List.ofSeq str

    let accFunc (As, Es, Is, Os, Us) letter =
        if letter = 'a' then (As + 1, Es, Is, Os, Us)
        elif letter = 'e' then (As, Es + 1, Is, Os, Us)
        elif letter = 'i' then (As, Es, Is + 1, Os, Us)
        elif letter = 'o' then (As, Es, Is, Os + 1, Us)
        elif letter = 'u' then (As, Es, Is, Os, Us + 1)
        else (As, Es, Is, Os, Us)

    List.fold accFunc (0, 0, 0, 0, 0) charList;;

val countVowels : string -> int * int * int * int * int

> countVowels "The quick brown fox jumps over the lazy dog";;
val it : int * int * int * int * int = (1, 3, 1, 4, 2)
```

Свертка справа налево

Функции List.reduce и List.fold обрабатывают списки в направлении слева направо. При этом существуют альтернативные им функции List.reduceBack и List.foldBack, обрабатывающие списки справа налево. В некоторых ситуациях обработка списков в обратном порядке может оказывать существенное влияние на производительность. (Более подробно проблема производительности при обработке списков рассматривается в главе 7.)

List.iter

Последний агрегатный оператор, List.iter, проходит по всем элементам списка и вызывает функцию, переданную вами в виде параметра. Она имеет тип:

```
('a -> unit) -> 'a list -> unit
```

Функция List.iter возвращает значение unit, поэтому она прежде всего используется для получения побочного эффекта вызова указанной функции. Под термином «побочный эффект» подразумевается, что вызов функции приводит к некоторому побочному эффекту помимо возвращаемого значения. Например, побочным эффектом вызова функции printfn является вывод текста на консоль помимо возвращаемого значение типа unit.

В примере 2.7 функция List.iter используется для обхода всех чисел в списке и вывода их на консоль.

Пример 2.7. Использование функции List.iter для вывода чисел из списка

```
> // Использование функции List.iter
let printNumber x = printfn "Printing %d" x
List.iter printNumber [1 .. 5];;

Printing 1
Printing 2
Printing 3
Printing 4
Printing 5

val printNumber : int -> unit
```

Тип option

Если вам нужно представить значение, которое может существовать, а может и нет, лучше всего использовать тип `option`. Тип `option` имеет всего два возможных значения: `Some('a)` и `None`.

Рассмотрим в качестве примера преобразование значения `string` в `int`. Если строка содержит допустимое представление числа, эта функция должна вернуть целочисленное значение, но как быть, если строка содержит недопустимый формат? Это как раз тот случай, когда пригодится тип `option`.

В примере 2.8 определяется функция `isInteger`, которая пытается преобразовать строку в целое число с помощью функции `Int32.TryParse`. В случае успешного преобразования функция возвращает `Some(result)`, в противном случае — `None`. Это позволит вызывающему коду определить, когда входная строка не может быть преобразована в число.

Пример 2.8. Использование типа option при преобразовании строки в целое число

```
> // Использование типа option для возвращаемого значения
open System

let isInteger str =
    let successful, result = Int32.TryParse(str)
    if successful
    then Some(result)
    else None;;

val isInteger : string -> int option

> isInteger "This is not an int";;
val it : int option = None
> isInteger "400";;
val it : int option = Some 400
```

В языке C# для обозначения отсутствующего значения обычно используется значение `null`. Однако `null` точно так же используется для обозначения неинициализированного значения. Такая двойственность может приводить к путанице и ошибкам. При использовании типа `option` никаких вопросов относительно того, что представляет собой это значение, не возникает.

Тип `option` можно рассматривать как аналог типа `System.Nullable`.

Чтобы получить значение объекта типа `option`, можно воспользоваться функцией `Option.get`. (Если функции `Option.get` передать значение `None`, то она сгенерирует исключение.) В следующем фрагменте определяется функция `containsNegativeNumbers`, которая возвращает `Some(_)` со списком всех отрицательных чисел, имеющихся в исходном списке. Затем в этом примере мы извлекаем отрицательные числа с помощью `Option.get`:

```
> // Использование функции Option.get
let isLessThanZero x = (x < 0)

let containsNegativeNumbers intList =
    let filteredList = List.filter isLessThanZero intList
    if List.length filteredList > 0
    then Some(filteredList)
    else None;;

val containsNegativeNumbers : int list -> int list option

> let negativeNumbers = containsNegativeNumbers [6; 20; -8; 45; -5];;

val negativeNumbers : int list option = Some [-8; -5]

> Option.get negativeNumbers;;
val it : int list = [-8; -5]
```

Модуль `Option` содержит ряд других полезных функций, которые перечислены в табл. 2.11.

Таблица 2.11. Методы модуля Option

Функция и ее тип	Описание
`Option.isSome` `'a option -> bool`	Для значения `Some` возвращает `true`, в противном случае возвращает `false`.
`Option.isNone` `'a option -> bool`	Для значения `Some` возвращает `false`, в противном случае возвращает `true`.

Функция printfn

Вывод данных в консоль – самая простая разновидность операций ввода-вывода, и они выполняются с помощью семейства функций `printf`.

Семейство функций printf содержит три основных версии: printf, printfn и sprintf.

Функция printf принимает входные данные и выводит их на экран. Функция printfn также выводит данные на экран, но она выполняет еще и переход на другую строку:

```
> // printf и printfn
printf "Hello, "
printfn "World";;

Hello, World
```

Для вывода текста на экран можно использовать класс System.Console, имеющийся в платформе .NET, но для функционального стиля программирования лучше подходит функция printf, потому что ее аргументы строго типизированы и способствуют механизму вывода типов. При этом для ввода данных по-прежнему следует использовать класс System.Console.

Вывод текста в консоль не является чем-то захватывающим, однако функция printf обладает дополнительными возможностями, такими как форматирование и встроенные проверки типов. Благодаря *спецификаторам формата (format specifier)*, перечисленным в табл. 2.12, можно передавать функции printf дополнительные данные. Это существенно упрощает вывод информации:

```
> // Спецификаторы формата
let mountain = "K2"
let height = 8611
let units = 'm';;

val mountain : string = "K2"
val height : int = 8611
val units : char = 'm'

> printfn "%s is %d%c high" mountain height units;;
K2 is 28251m high
val it : unit = ()
```

Что самое интересное, благодаря механизму вывода типов компилятор F# сообщит об ошибке, если данные не соответствуют указанному спецификатору формата:

```
> printfn "An integer = %d" 1.23;;

  printfn "An integer = %d" 1.23;;
  -------------------------^^^^^

stdin(2,27): error FS0001: The type 'float' is not compatible with any of the
types byte,int16,int32,int64,sbyte,uint16,uint32,uint64,nativeint,unativeint,
arising from the use of a printf-style format string.
stopped due to error
```

(Тип 'float' несовместим с любыми типами byte, int16, int32, int64, sbyte, uint16, uint32, uint64, nativeint, unativeint, возникающими при использовании строки формата printf-style. Остановлено из-за ошибки)

Таблица 2.12. Спецификаторы формата функции printf

Спецификатор	Описание	Пример	Результат
%d, %i	Выводит целое значение	printf "%d" 5	5
%x, %o	Выводит любое целое значение в шестнадцатеричном и восьмеричном форматах соответственно	printfn "%x" 255	ff
%s	Выводит любую строку	printf "%s" "ABC"	ABC
%f	Выводит любое вещественное число	printf "%f" 1.1M	1.100000
%c	Выводит любой символ	printf "%c" '\097'	a
%b	Выводит любое булево значение	printf "%b" false	false
%0	Выводит любой объект	printfn "%0" (1,2)	(1,2)
%A	Выводит значение любого типа	printf "%A" (1, [])	(1, [])

Кроме того, благодаря тому что компилятор F# способен определять ожидаемые типы исходя из списка спецификаторов формата, механизм вывода типов может использовать эту информацию для определения типов значений. Например, в следующем фрагменте типы параметров функции выводятся на основании их использования:

```
> // Выведение типов на основе вызова функции printf
let inferParams x y z =
    printfn "x = %f, y = %s, z = %b" x y z;;

val inferParams : float -> string -> bool -> unit
```

Спецификатор формата %0 приводит к упаковке объекта с последующим вызовом виртуального метода Object.ToString. Спецификатор формата %A действует похожим образом, за исключением того, что он проверяет наличие дополнительных данных в атрибуте [<StructuredFormatDisplay>], прежде чем вызвать метод Object.ToString.

Функция sprintf используется в случаях, когда необходимо сохранить строковое представление объекта в виде строки:

```
> let location = "World";;
```

```
val location : string = "World"

> sprintf "Hello, %s" location;;
val it : string = "Hello, World"
```

Строение программы на языке F#

К настоящему моменту у вас вполне могло возникнуть желание узнать, как весь этот код F#, который мы вводили в окне FSI, превратить в настоящую программу. В действительности каждый фрагмент кода, который вы видели выше, уже является полноценной программой!

Во многих других языках программирования, таких как C#, требуется явно определить точку входа в программу, которая часто называется *главным* (*main*) методом. В наших программах F# до сих пор отсутствовал какой-либо специальный признак, который указывал бы, откуда должно начинаться выполнение программы. В языке F# для приложений, состоящих из одного файла, код выполняется в направлении от начала и до конца (при этом не требуется объявлять специальный главный метод).

Однако для проектов, состоящих из нескольких файлов, программный код должен делиться на организационные единицы, называемые *модулями* и *пространствами имен*.

Модули

Весь код, который мы писали до сих пор, находился в одном модуле. По умолчанию компилятор F# помещает весь программный код в *анонимный модуль* с тем же именем, что и имя файла, с заглавным первым символом имени. То есть если в коде имеется значение с именем `value1`, а файл называется *file1.fs*, вы можете обратиться к нему через полностью квалифицированное имя: `File1.value1`.

Создание модулей

Вы можете явно указать имя модуля с помощью ключевого слова `module` в самом начале файла. После этого объявления все значения, функции или определения типов будут принадлежать этому модулю:

```
module Alpha

// Чтобы использовать следующее значение вне модуля,
// следует использовать: Alpha.x
let x = 1
```

Вложенные модули

Файлы могут также содержать вложенные модули. Чтобы объявить вложенный модуль, нужно с помощью ключевого слова `module` объявить

имя вложенного модуля и добавить знак = после имени. Вложенные модули должны оформляться дополнительными отступами, чтобы их можно было отличить от модуля «верхнего уровня»:

```
module Utilities

module ConversionUtils =

    // Utilities.ConversionUtils.intToString
    let intToString (x : int) = x.ToString()

    module ConvertBase =
        // Utilities.ConversionUtils.ConvertBase.convertToHex
        let convertToHex x = sprintf "%x" x

        // Utilities.ConversionUtils.ConvertBase.convertToOct
        let convertToOct x = sprintf "%o" x

module DataTypes =
    // Utilities.DataTypes.Point
    type Point = Point of float * float * float
```

Пространства имен

Еще одной организационной единицей, альтернативной модулям, являются пространства имен. Пространства имен во многом аналогичны модулям, с той лишь разницей, что они не могут содержать значений – только объявления типов. Кроме того, пространства имен не могут вкладываться друг в друга таким же образом, как модули. Вместо этого вы можете просто добавить несколько пространств имен в один файл.

В примере 2.9 объявляется несколько типов в двух пространствах имен.

Пример 2.9. Пространства имен

```
namespace PlayingCards

// PlayingCards.Suit
type Suit =
  | Spade
  | Club
  | Diamond
  | Heart

// PlayingCards.PlayingCard
type PlayingCard =
  | Ace       of Suit
  | King      of Suit
  | Queen     of Suit
  | Jack      of Suit
  | ValueCard of int * Suit
```

```
namespace PlayingCards.Poker

// PlayingCards.Poker.PokerPlayer
type PokerPlayer = { Name : string; Money : int; Position : int }
```

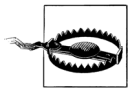
Кому-то может показаться непонятным, что в языке F# предусмотрены сразу две организационные единицы – пространства имен и модули. Модули прекрасно подходят для быстрого создания прототипов и быстрого поиска решений, как вы уже могли убедиться ранее. Пространства имен, в свою очередь, предназначены для использования в крупных проектах на основе объектно-ориентированных решений.

Запуск программы

В языке F# выполнение программы начинается с начала последнего файла, который обязательно должен быть модулем. Рассмотрим следующий простой пример программы на языке F#, состоящей из одного файла:

```
// Program.fs
let numbers = [1 .. 10]
let square x = x * x

let squaredNumbers = List.map square numbers

printfn "SquaredNumbers = %A" squaredNumbers

open System

printfn "(press any key to continue)"
Console.ReadKey(true)
```

Теперь откройте этот проект в Visual Studio и добавьте новый пустой файл F#. Если после этого вы нажмете клавишу F5, чтобы запустить программу, ничего не произойдет. Это обусловлено тем, что новый файл, добавленный в проект «последним», не содержит программного кода, который можно было бы выполнить.



Исполнение программного кода вроде бы в случайном порядке должно, вероятно, вызывать у вас тревогу. Модули не рекомендуется использовать в крупных проектах, однако мы рассмотрим использование модулей в реальных приложениях в главе 7.

Для формализации запуска приложения вы можете явно задать главный метод с помощью атрибута [<EntryPoint>]. Для этого главный метод должен:

- Быть последней функцией в последнем компилируемом файле проекта. Это гарантирует отсутствие неоднозначности в определении того, откуда должна запускаться программа.

- Принимать единственный параметр типа `string array`, содержащий аргументы командной строки. (Массивы рассматриваются в главе 4.)
- Возвращать целое число – код завершения программы.

Чтобы явно указать главный метод, предыдущий пример приложения можно переписать следующим образом:

```
// Program.fs
open System

[<EntryPoint>]
let main (args : string[]) =
    let numbers = [1 .. 10]
    let square x = x * x

    let squaredNumbers = List.map square numbers

    printfn "SquaredNumbers = %A" squaredNumbers

    printfn "(press any key to continue)")
    Console.ReadKey(True) |> ignore

    // Вернуть 0
    0
```

Теперь у вас есть все инструменты, необходимые для создания простых программ на языке F#. В следующей главе вы научитесь программировать в функциональном стиле, что позволит писать более мощные приложения на языке F# и приблизит вас к девятому дану владения языком F#.

3

Функциональное программирование

Познакомившись с основами языка F# в целом, мы можем приступить к его исследованию с точки зрения конкретного стиля программирования. Эта глава посвящена основной парадигме F#: функциональному программированию. Если говорить в двух словах, то функциональное программирование приводит к более декларативному коду. При использовании императивного стиля программировании (рассматривается в главе 4) большую часть своего времени программист тратит на описание конкретных шагов решения задачи. При использовании функционального стиля программирования программист описывает, *что* требуется сделать, а не *как это сделать*. И хотя функциональное программирование не является «серебряной пулей», при его использовании программы получаются более понятными, а некоторые проблемы, такие как многопоточность, решаются намного проще.

Функциональное программирование не должно рассматриваться как замена императивному или объектно-ориентированному – оно лишь предоставляет иной подход к разработке, который в некоторых ситуациях может быть значительно более эффективным.

Язык, претендующий на звание «функционального», обычно должен обладать следующими ключевыми особенностями:

- Неизменяемость данных
- Возможность композиции функций
- Функции могут рассматриваться как данные
- Отложенные (lazy) вычисления
- Сопоставления с образцом (pattern matching)

В этой главе мы рассмотрим каждую из этих концепций функционального программирования и что они могут нам предложить. К концу этой

главы вы сможете писать чистый функциональный код и использовать простоту и элегантность декларативного программирования. Более глубокое изучение концепций функционального программирования, таких как хвостовая рекурсия (tail recursion) и замыкание (closure), будет продолжено в главе 7.

Программирование с помощью функций

В основе функционального программирования лежит представление о коде в терминах математических функций. Рассмотрим две функции, `f` и `g`:

```
f(x) = x^2 + x
g(x) = x + 1
```

Из этого следует, что:

```
f(2) = (2)^2 + (2)
g(2) = (2) + 1
```

И если теперь объединить эти две функции, получим следующее:

```
f g (2) = f(g(2))
        = (g(2))^2 + (g(2))
        = (2+1)^2 + (2+1)
        = 12
```

Вам не нужно быть математиком, чтобы программировать на F#, но некоторые идеи функционального программирования пришли именно из этой науки. Например, в предыдущих фрагментах кода нигде явно не указывается тип возвращаемого значения. Как понять, аргумент какого типа принимает функция `f(x)` – целое число или вещественное? Такая математическая форма записи не имеет отношения к типам параметров или возвращаемых значений. Эквивалентный ей код на F# имеет следующий вид:

```
let f x = x ** 2.0 + x
let g x = x + 1.0
```

То, что этот код напоминает математическую нотацию, не случайное совпадение. По сути функциональное программирование как раз и заключается в представлении вычислений подобным абстрактным способом. Напомню еще раз, что в функциональном программировании программист описывает, *что* требуется сделать, а не *как* это сделать.

Вы можете даже представить себе всю программу целиком, как единую функцию, которая на входе принимает сигналы от мыши и клавиатуры, а на выходе возвращает код завершения приложения. Когда программирование начинает рассматриваться с этой точки зрения, уходят многие сложности, присущие обычным моделям программирования.

Во-первых, если представить себе программу как последовательность функций, то не придется тратить все свое время, чтобы во всех подробностях описать каждый шаг, необходимый для решения задачи. Функции просто принимают входные данные и возвращают некоторый результат. Во-вторых, алгоритмы обычно выражаются в терминах функций, а не классов или объектов, поэтому их проще выражать в функциональном стиле.

В этой главе вы наглядно увидите, как функциональный стиль программирования может упростить сложный код, но сначала вам необходимо начать думать в терминах функций. Для этого вам следует забыть опыт, накопленный в результате использования императивных языков программирования. В следующих разделах я представлю вам такие понятия, как неизменяемость, функции как значения и композиция функций, чтобы продемонстрировать, как использовать функциональные программирование.

Неизменяемость

Вероятно, вы заметили, что ранее я нигде не использовал термин *переменная*, и вместо него я использовал термин *значение*. Причина заключается в том, что в функциональном программировании имена, которые объявляются в программе, по умолчанию являются неизменяемыми, в том смысле, что их нельзя изменить.

Если функция каким-то образом изменяет состояние программы, например записывает данные в файл или изменяет значение глобальной переменной в памяти, это называют *побочным эффектом*. Например, функция `printfn` возвращает результат типа `unit`, но она содержит побочный эффект в виде вывода текста на экран. Аналогично если функция изменяет некоторое значение в памяти, это также является побочным эффектом – тем, что делает функция помимо вычисления возвращаемого значения.

Побочные эффекты – это не всегда плохо, но непредусмотренные побочные эффекты являются источником многих ошибок. Даже с самыми благими намерениями можно допустить ошибку, если не задумываться о побочных эффектах функции. Неизменяемые значения помогают писать надежный код, потому что нельзя испортить то, что невозможно изменить.

Если у вас есть опыт использования императивных языков программирования, отсутствие переменных может показаться вам весьма неудобным. Тем не менее неизменяемость имеет свои преимущества. Рассмотрим пример 3.1, в котором обе функции просто суммируют квадраты списка чисел, но одна функция написана в императивном стиле и изменяет данные, а другая написана в функциональном стиле. Императивная функция использует *изменяемую* переменную, то есть значение `total` изменяется в процессе выполнения функции `imperativeSum`.

Пример 3.1. Вычисление суммы квадратов списка чисел в императивном и функциональном стилях

```
let square x = x * x

let imperativeSum numbers =
    let mutable total = 0
    for i in numbers do
        let x = square i
        total <- total + x
    total

let functionalSum numbers =
    numbers
    |> Seq.map square
    |> Seq.sum
```

Обе функции возвращают одно и то же значение для одних и тех же входных данных, но в чем же заключается преимущество функционального стиля? Первое, что можно заметить, – это читабельность. Чтобы понять, что происходит в коде, написанном в императивном стиле, вам придется прочитать его целиком. Функциональный код декларативен – он наглядно отражает то, что вы хотели, чтобы он делал. (В данном случае – возведение в квадрат всех элементов списка с последующим их суммированием.) Кроме того, если вам потребуется запускать императивную версию функции параллельно, вам придется переписать ее полностью. Поскольку в этом случае вы подробно описываете, как решать конкретную задачу, то если вам понадобится выполнить ее параллельно, вам придется так же подробно описать все шаги, необходимые для этого. Функциональная версия, напротив, не описывает, как выполнять конкретные действия, поэтому вы легко сможете воспользоваться параллельными реализациями `map` и `sum`. В главе 11 вы увидите, насколько легко использовать параллельное программирование на языке F#.

Функциональные языки программирования, которые не допускают возможность появления побочных эффектов, называются *чистыми*. Язык F# в этом отношении не является чистым, потому что позволяет изменять значения переменных при использовании императивного стиля программирования.

Функции как значения

В большинстве других языков программирования функции и данные рассматриваются как нечто отличное друг от друга. Однако в функциональных языках программирования функции рассматриваются как обычные элементы данных. Например, функции могут передаваться в виде параметров другим функциям. Кроме того, функции могут создавать и возвращать новые функции! Функции, которые принимают

или возвращают другие функции, называются функциями *высшего порядка* и являются ключевым понятием функционального программирования.

Данная возможность позволяет абстрагировать и повторно использовать алгоритмы в вашем коде. Вы уже видели подобные примеры в предыдущей главе, где обсуждались функции `List.iter`, `List.map` и `List.fold`.

В примере 3.2 определяется функция `negate`, которая изменяет знак одного целого числа. Если передать ее в функцию `List.map` в виде параметра, она будет применена ко всему списку и изменит знак каждого его элемента.

Пример 3.2. Пример функций высшего порядка

```
> let negate x = -x;;

val negate : int -> int

> List.map negate [1 .. 10];;
val it : int list = [-1; -2; -3; -4; -5; -6; -7; -8; -9; -10]
```

Использовать функции как значения очень удобно, но в конечном результате вам придется написать множество простых функций, которые сами по себе не имеют большой ценности. Например, наша функция `negate` из примера 3.2 едва ли будет использоваться еще для каких-либо целей, кроме как для изменения знаков элементов списка.

Вместо того чтобы давать имена всем этим мелким функциям, которые передаются в виде параметров, можно использовать *анонимные функции*, которые также известны как *лямбда-выражения*.

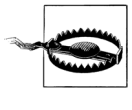
Чтобы создать лямбда-выражение, достаточно воспользоваться ключевым словом `fun`, за которым следуют параметры функции и стрелка `->`.

В следующем фрагменте создается лямбда-выражение, которому в виде параметра передается значение 5. При вызове этой функции она увеличит значение параметра `x` на 3 и вернет 8:

```
> (fun x -> x + 3) 5;;
val it : int = 8
```

Теперь мы можем переписать наш пример, изменяющий знаки чисел в списке, с использованием лямбда-выражения, которое принимает единственный параметр `i`:

```
> List.map (fun i -> -i) [1 .. 10];;
val it : int list = [-1; -2; -3; -4; -5; -6; -7; -8; -9; -10]
```



Старайтесь создавать лямбда-выражения максимально простыми. Чем длиннее лямбда-выражение вы напишете, тем сложнее его будет отлаживать. Это особенно справедливо когда вы начинаете копировать лямбда-выражения по всему своему коду.

Частичное применение функции

Еще одна характерная особенность функционального программирования – частично применимые функции, известные также как *каррированные функции (carrying)*. Начнем обсуждение этой темы с создания простой функции, которая дописывает текст в конец файла, используя существующие библиотеки .NET:

```
> // Добавить текст в конец файла
open System.IO

let appendFile (fileName : string) (text : string) =
    use file = new StreamWriter(fileName, true)
    file.WriteLine(text)
    file.Close();;

val appendFile : string -> string -> unit

> appendFile @"D:\Log.txt" "Processing Event X...";;
val it : unit = ()
```

Функция `appendFile` выглядит достаточно просто, но что если нам потребуется многократно выполнять запись в один и тот же журнальный файл? Можно было бы сохранить путь к файлу и всегда передавать его в первом параметре. Однако было бы лучше создать версию функции `appendFile`, первый параметр которой имеет фиксированное значение `@"D:\Log.txt"`.

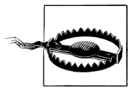
Создание частично применимой функции означает возможность задать значения некоторых параметров функции и создать новую функцию с фиксированными значениями параметров. Мы можем «каррировать» (зафиксировать) первый параметр функции `appendFile` и создать новую функцию, принимающую единственный параметр – текст сообщения для вывода в журнальный файл:

```
> // Каррировать appendFile, зафиксировав значение первого параметра
let curriedAppendFile = appendFile @"D:\Log.txt";;

val curriedAppendFile : (string -> unit)

> // Добавить текст в файл 'D:\Log.txt'
curriedAppendFile "Processing Event Y...";;
val it : unit = ()
```

Карринг объясняет наличие в типе функции стрелки между ее аргументами. Функция `appendFile` имела тип:

```
string -> string -> unit
```

так что после передачи первого параметра типа `string` получается функция, которая принимает параметр типа `string` и возвращает значение типа `unit`, то есть `string -> unit`.



Каррированные функции могут упростить код, но они также могут усложнить его отладку. Старайтесь не злоупотреблять каррингом, чтобы не сделать программу более сложной, чем это необходимо.

Все, что необходимо для каррирования функции, – это указать подмножество параметров; в результате вы получите функцию, которой требуется передать только параметры, которые не были указаны до этого. По этой причине каррировать параметры можно только слева направо.

Карринг и частично применимые функции не выглядят особенно мощным инструментом, но они могут существенно повысить элегантность вашего кода. Представьте вызов функции `printf`, которой передается строка формата и набор данных, соответствующих этой строке. Если создать каррированную версию, первый параметр которой имеет фиксированное значение `"%d"`, в результате мы получим частично применимую функцию, которая принимает целое число и выводит его на экран.

В следующем примере демонстрируется, как можно передать частично применимую версию функции `printf`, чтобы избежать необходимости создавать лямбда-выражение:

```
> // Некаррированная версия
List.iter (fun i -> printfn "%d" i) [1 .. 3];;
1
2
3
val it : unit = ()
> // Использование каррированной версии printfn
List.iter (printfn "%d") [1 .. 3];;
1
2
3
val it : unit = ()
```

Как полностью использовать все преимущества частично применимых функций, вы увидите, когда мы будем обсуждать композицию функций и конвейерный оператор (pipe-forward operator).

Функции, возвращающие функции

Благодаря тому что в функциональном программировании функции интерпретируются как данные, функции могут возвращать другие функции как обычные значения. Это может приводить к весьма интересным результатам, если рассматривать эти значения с их областями видимости.

В примере 3.3 определяется функция `generatePowerOfFunc`, возвращающая функцию, которая в свою очередь вычисляет степень заданного числа. Далее определяются две функции, `powerOfTwo` и `powerOfThree`, которые возводят двойку и тройку в заданную степень.

Пример 3.3. Функции, возвращающие функции

```
> // Функции, возвращающие функции
let generatePowerOfFunc baseValue = (fun exponent -> baseValue ** exponent);;

val generatePowerOfFunc : float -> float -> float

> let powerOfTwo = generatePowerOfFunc 2.0;;

val powerOfTwo : (float -> float)

> powerOfTwo 8.0;;
val it : float = 256.0
> let powerOfThree = generatePowerOfFunc 3.0;;

val powerOfThree : (float -> float)

> powerOfThree 2.0;;
val it : float = 9.0
```

Если внимательно рассмотреть функцию `generatePowerOfFunc`, можно заметить, что ее параметр `baseValue` используется в двух возвращаемых лямбда-выражениях. На первый взгляд может показаться непонятным, откуда берется значение `baseValue` позднее, когда вызываются функции `powerOfTwo` и `powerOfThree`, потому что оно отсутствует в списках параметров этих функций. И не совсем понятно, где сохраняется число 2.0, когда функция `generatePowerOfFunc` вызывается первый раз, со значением 2.0 в виде параметра?

Здесь мы имеем дело с толикой магии под названием *замыкание (closure)*. Пока не нужно стараться понять, что это такое и как это работает. Сейчас вам достаточно знать, что если некоторое значение находится в области видимости функции, оно может использоваться и при необходимости возвращаться этой функцией. В главе 7 мы подробно обсудим замыкания и проверим на прочность возможности этой магии, предоставляемой компилятором F#.

Рекурсивные функции

Функция, которая вызывает саму себя, называется *рекурсивной*. Рекурсивные функции могут быть чрезвычайно полезны при использовании функционального стиля программирования, в чем вы вскоре убедитесь.

Чтобы определить рекурсивную функцию, достаточно просто добавить ключевое слово `rec`. В следующем фрагменте определяется функция, вычисляющая факториал. (То есть произведение всех положительных целых чисел до заданного числа включительно. Например, факториал числа 4 равен $4 \times 3 \times 2 \times 1$.)

```
> // Определение рекурсивной функции
let rec factorial x =
```

```
    if x <= 1 then
        1
    else
        x * factorial (x - 1);;

val factorial : int -> int

> factorial 5;;
val it : int = 120
```

Ключевое слово `rec` может показаться ненужным излишеством, так как другие языки программирования не требуют явного обозначения рекурсивных функций. Настоящая цель введения ключевого слова `rec` состоит в том, чтобы сообщить механизму выведения типов, что функция может использоваться в процессе определения типов. Ключевое слово `rec` позволяет вызывать функцию еще до того, как механизм выведения типов определит тип функции.

Используя рекурсию в комбинации с функциями высшего порядка, вы легко сможете смоделировать конструкции циклов, которые можно найти в императивных языках программирования, не используя изменяемые значения. В следующем примере демонстрируются функциональные версии циклов `for` и `while`. Обратите внимание, что в примере с циклом `for` измененное значение счетчика просто передается рекурсивному вызову в виде параметра:

```
> // Функциональная версия цикла for
let rec forLoop body times =
    if times <= 0 then
        ()
    else
        body()
        forLoop body (times - 1)

// Функциональная версия цикла while
let rec whileLoop predicate body =
    if predicate() then
        body()
        whileLoop predicate body
    else
        ();;

val forLoop : (unit -> unit) -> int -> unit
val whileLoop : (unit -> bool) -> (unit -> unit) -> unit

> forLoop (fun () -> printfn "Looping...") 3;;
Looping...
Looping...
Looping...
val it : unit = ()
```

```
> // Типичная рабочая неделя...
open System

whileLoop
    (fun () -> DateTime.Now.DayOfWeek <> DayOfWeek.Saturday)
    (fun () -> printfn "I wish it were the weekend...");;
I wish it were the weekend...
I wish it were the weekend...
I wish it were the weekend...
    * * * Так будет продолжаться несколько дней * * *
val it : unit = ()
```

Взаимная рекурсия

Две функции, вызывающие друг друга, называются *взаимно рекурсивными*. Взаимно рекурсивные функции представляют определенную сложность для механизма вывода типов в языке F#. Чтобы определить тип первой функции, необходимо знать тип второй функции, и наоборот.

В примере 3.4 происходит ошибка компиляции взаимно рекурсивных функций, потому что к моменту компиляции функции `isOdd` функция `isEven` еще не определена.

Пример 3.4. Взаимно рекурсивные функции

```
> // Ошибка: Невозможно определить isOdd без определения isEven, и наоборот
let isOdd  n = if n = 0 then false elif n = 1 then true else (isEven (n - 1))
let isEven n = if n = 0 then true elif n = 1 then false else (isOdd (n - 1));;

 let isOdd  n = if n = 0 then false elif n = 1 then true else (isEven (n - 1))
 ------------------------------------------------------------------^^^^^^^

stdin(4,44): error FS0039: The value or constructor 'isEven' is not defined.
stopped due to error
```

(Значение или конструктор 'isEven' не определено. Остановлено из-за ошибки.)

Чтобы определить взаимно рекурсивные функции, необходимо объединить их с помощью ключевого слова `and`, которое говорит компилятору F# выводить типы обеих функций одновременно:

```
> // Определение взаимно рекурсивных функций
let rec isOdd n = if n = 0 then false elif n = 1 then true else isEven (n - 1)
and   isEven n = if n = 0 then true elif n = 1 then false else isOdd (n - 1);;

val isOdd : int -> bool
val isEven : int -> bool

> isOdd 13;;
val it : bool = true
```

Символьные операторы

Представьте, насколько сложно было бы программировать, если бы всякий раз вместо выражения `1 + 2` приходилось бы писать `add 1 2`. К счастью, F# не только поддерживает встроенные символьные операторы для выполнения таких операций, как сложение и вычитание, но и дает нам возможность определять собственные операторы, позволяя писать более понятный и выразительный код.

Символьные функции не следует рассматривать как разновидность перегрузки операторов, скорее это функции, имена которых составлены из специальных символов.

Символьный оператор может представлять собой любую комбинацию из следующих символов: `!%&*+-./<=>?@^|~` (включая `:`, при условии, что он не будет первым в комбинации). Следующий фрагмент определяет новую функцию `!`, которая вычисляет факториал числа:

```
> // Факториал
let rec (!) x =
    if x <= 1 then 1
    else x * !(x - 1);;

val ( ! ) : int -> int

> !5;;
val it : int = 120
```

По умолчанию при использовании символьных функций, принимающих более одного параметра, применяется *инфиксная нотация*. То есть первый параметр предшествует символу, что является привычной формой использования символьных функций. В следующем примере определяется функция `===`, которая сравнивает строку на основе регулярного выражения:

```
> // Определение (===) для сравнения строк на основе регулярного выражения
open System.Text.RegularExpressions;;

let (===) str (regex : string) =
    Regex.Match(str, regex).Success;;

val ( === ) : string -> string -> bool

> "The quick brown fox" === "The (.*) fox";;
val it : bool = true
```

Чтобы получить символьный оператор, который записывается перед своими параметрами, что известно как *префиксная нотация*, имя функции должно начинаться с тильды `~`, восклицательного знака `!` или

знака вопроса ?. В следующем фрагменте имя функции ~+++ начинается с тильды, поэтому для его вызова необходимо писать ~+++ 1 2 3, а не 1 ~+++ 2 3. Это позволяет выбирать для символьных операторов форму записи, наиболее естественно соответствующую определяемой функции:

```
> let (~+++) x y z = x + y + z;;
val ( ~+++ ) : int -> int -> int -> int
> ~+++ 1 2 3;;
val it : int = 6
```

Помимо возможности создавать имена функций, которые точно соответствуют математической нотации, вы можете передавать символьные операторы функциям высшего порядка – для этого достаточно заключить оператор в круглые скобки. Например, если вам потребуется найти сумму или произведение элементов списка, вы можете просто записать:

```
> // Вычисление суммы элементов списка
// с использованием символьной функции (+)
List.fold (+) 0 [1 .. 10];;
val it : int = 55

> // Вычисление произведения элементов списка
// с использованием символьной функции (*)
List.fold (*) 1 [1 .. 10];;
val it : int = 3628800

> let minus = (-);;

val minus : (int -> int -> int)

> List.fold minus 10 [3; 3; 3];;
val it : int = 1
```

Композиция функций

Как только вы освоите работу с функциями, вы можете начинать объединять их в более крупные и мощные функции. Этот прием известен как *композиция функций* и является еще одним основополагающим принципом функционального программирования.

Но прежде чем перейти к объединению функций, давайте посмотрим, какую проблему это решает. Ниже приводится пример, как не надо делать: вся реализация помещена внутрь одной огромной функции. Этот код определяет размер указанной папки на диске (я добавил аннотации типов, чтобы помочь определить тип возвращаемого значения):

```
open System
open System.IO

let sizeOfFolder folder =
```

```
// Получить список всех файлов в папке
let filesInFolder : string [] =
    Directory.GetFiles(
        folder, "*.*",
        SearchOption.AllDirectories)

// Отобразить имена файлов на соответствующие им объекты FileInfo
let fileInfos : FileInfo [] =
    Array.map
        (fun file -> new FileInfo(file))
        filesInFolder

// Отобразить объекты fileInfo на значения размеров файлов
let fileSizes : int64 [] =
    Array.map
        (fun (info : FileInfo) -> info.Length)
        fileInfos

// Суммарный размер файлов
let totalSize = Array.sum fileSizes

// Вернуть суммарный размер файлов
totalSize
```

У этого кода есть три основные проблемы:

- Механизм вывода типов не способен правильно определить типы, поэтому пришлось добавить аннотации типов параметров в каждое лямбда-выражение. Это обусловлено тем, что механизм вывода типов просматривает код слева направо, сверху вниз и встречает лямбда-выражение, передаваемое функции `Array.map`, до того, как он сможет определить тип элементов массива. (То есть тип параметров лямбда-выражения неизвестен.)
- Результат каждого этапа вычислений передается в виде параметра следующему этапу, поэтому инструкции `let` в функции выглядят ненужным излишеством.
- Код попросту ужасен. Чтобы понять, что он делает, придется потратить времени больше, чем следует.

Композиция функций позволяет разбить подобный код на несколько маленьких функций, а затем объединить их в окончательную реализацию.

В предыдущем примере результаты одного вычисления передаются следующему. На языке математики передача результата функции `f(x)` в функцию `g(x)` записывается так: `g(f(x))`. Мы могли бы избежать всех этих операторов связывания `let`, используя вложенное вычисление промежуточных результатов, но такой код крайне сложно читать, как можно убедиться по следующему фрагменту:

```
let uglySizeOfFolder folder =
    Array.sum
        (Array.map
            (fun (info : FileInfo) -> info.Length)
            (Array.map
                (fun file -> new FileInfo(file))
                (Directory.GetFiles(
                    folder, "*.*",
                    SearchOption.AllDirectories))))
```

Прямой конвейерный оператор

К счастью, язык F# предоставляет лаконичное решение проблемы передачи промежуточных результатов от функции к функции с помощью прямого конвейерного оператора (pipe-forward opeator) `|>`. Он определяется следующим образом:

```
let (|>) x f = f x
```

Прямой конвейерный оператор позволяет переупорядочить параметры функции так, что при вызове функции последний ее параметр указывается первым. Если учесть, что последним параметром функции `List.iter` передается список, по которому выполняются итерация, то, используя прямой конвейерный оператор, можно отправить этот список функции `List.iter` «по конвейеру», поместив его первым:

```
> [1 .. 3] |> List.iter (printfn "%d");;
1
2
3
val it : unit = ()
```

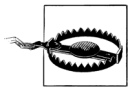
С технической точки зрения прямой конвейерный оператор и его брат, обратный конвейерный оператор, в действительности не являются механизмом композиции функций. Скорее они связаны с применением функций.

Преимущество прямого конвейерного оператора состоит в том, что с его помощью можно составлять целые цепочки вызовов функций. То есть передавать результат предыдущей функции на вход следующей функции. Теперь мы можем переписать функцию `sizeOfFolder`, как показано ниже. Обратите внимание, что функция `Directory.GetFiles` принимает свои параметры в виде кортежа, поэтому она не может быть каррирована:

```
let sizeOfFolderPiped folder =

    let getFiles folder =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)

    let totalSize =
        folder
```

```
    |> getFiles
    |> Array.map (fun file -> new FileInfo(file))
    |> Array.map (fun info -> info.Length)
    |> Array.sum

totalSize
```

Простота, достигнутая с помощью прямого конвейерного оператора, является наглядным примером того, насколько полезным может оказаться каррирование функций. Прямой конвейерный оператор принимает значение и функцию с единственным параметром. Однако использованная нами функция `Array.map` принимает два параметра (функцию, выполняющую отображение, и сам массив). Причина, по которой наш программный код оказался работоспособным, заключается в том, что мы каррировали один аргумент, получили в результате функцию с единственным входным параметром и благодаря этому смогли использовать ее с прямым конвейерным оператором.



Прямой конвейерный оператор способен существенно упростить код за счет устранения ненужных объявлений промежуточных результатов, однако отладка последовательностей функций, объединенных конвейером, становится намного сложнее, потому что вы лишаетесь возможности проверить значения промежуточных результатов.

Дополнительным преимуществом прямого конвейерного оператора заключается в том, что он оказывает помощь механизму выведения типов. Вы не сможете получить доступ ни к одному свойству или методу значения, если компилятор не может определить его тип. Поэтому для явного указания типов в подобных ситуациях приходится использовать аннотации:

```
> // ОШИБКА: Компилятору неизвестно, что значение s имеет свойство Length
List.iter
    (fun s -> printfn "s has length %d" s.Length)
    ["Pipe"; "Forward"];;

      (fun s -> printfn "s has length %d" s.Length)
  ----------------------------------------^^^^^^^^^

stdin(89,41): error FS0072: Lookup on object of indeterminate type based on
information prior to this program point. A type annotation may be needed prior
to this program point to constrain the type of the object. This may allow the
lookup to be resolved
stopped due to error
```

(Поиск объекта неопределенного типа, основанного на информации до данной точки программы. Возможно, перед данной точкой программы потребуется аннотация типа с целью ограничения типа объекта. Возможно, это позволит разрешить поиск. Остановлено из-за ошибки.)

Прямой конвейерный оператор помогает компилятору «увидеть» последний параметр функции ранее, благодаря чему механизм выведения типов может определить правильные типы функций без дополнительных аннотаций.

В следующем примере благодаря использованию прямого конвейерного оператора параметр лямбда-выражения, передаваемого функции `List.iter`, определяется как значение типа `string`, вследствие чего отпадает необходимость в аннотации типа:

```
> ["Pipe"; "Forward"] |> List.iter (fun s -> printfn "s has length %d"
s.Length);;
s has length 4
s has length 7
val it : unit = ()
```

Прямой оператор композиции

Прямой оператор композиции (forward composition operator) >> объединяет две функции, при этом функция слева вызывается первой:

```
let (>>) f g x = g(f x)
```

При использовании прямого конвейерного оператора необходимо указать переменную, чтобы «запустить» конвейерную обработку. В нашем последнем примере функция принимала параметр `folder`, который передавался первой функции в конвейере:

```
let sizeOfFolderPiped2 folder =

    let getFiles folder =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)

    folder
    |> getFiles
    |> Array.map (fun file -> new FileInfo(file))
    |> Array.map (fun info -> info.Length)
    |> Array.sum
```

Однако при использовании оператора композиции функций никакие параметры не требуется. Мы можем просто объединить все эти функции воедино и получить в результате новую функцию, принимающую параметр и возвращающую результат:

```
> // Композиция функций

open System.IO

let sizeOfFolderComposed (*Нет параметров!*) =

    let getFiles folder =
        Directory.GetFiles(folder, "*.*", SearchOption.AllDirectories)
```

```
    // Результатом этого выражения является функция, принимающая
    // один параметр, который будет передан функции getFiles и далее
    // по конвейеру последующим функциям.
    getFiles
    >> Array.map (fun file -> new FileInfo(file))
    >> Array.map (fun info -> info.Length)
    >> Array.sum;;

val sizeOfFolderComposed : (string -> int64)

> sizeOfFolderComposed
    (Environment.GetFolderPath(Environment.SpecialFolder.MyPictures));;
val it : int64 = 904680821L
```

Ниже приводится более простой пример композиции простых функций:

```
> // Композиция простых функций
let square x = x * x
let toString (x : int) = x.ToString()
let strLen (x : string) = x.Length
let lenOfSquare = square >> toString >> strLen;;

val square : int -> int
val toString : int -> string
val strLen : string -> int
val lenOfSquare : (int -> int)

> square 128;;
val it : int = 16384
> lenOfSquare 128;;
val it : int = 5
```

Обратный конвейерный оператор

Обратный конвейерный оператор (pipe-backward operator) <|, как можно было бы догадаться, принимает функцию слева и передает ей значение направо. На первый взгляд в этом нет никакой необходимости, потому что этот оператор всего лишь отделяет функцию от ее последнего параметра, что можно сделать и так, без дополнительного оператора:

```
let (<|) f x = f x
```

Ниже приводится пример использования обратного конвейерного оператора:

```
> List.iter (printfn "%d") [1 .. 3];;
1
2
3

val it : unit = ()
```

```
> List.iter (printfn "%d") <| [1 .. 3];;
1
2
3
val it : unit = ()
```

Возможно, вы будете удивлены, узнав, что в действительности обратный конвейерный оператор играет важную роль: он позволяет изменить порядок вычислений. До сих пор я ничего не говорил о порядке выполнения операторов; это порядок, в котором вызываются функции. Аргументы функций вычисляются в направлении слева направо, то есть если вы хотите вызвать функцию и передать ей результат другой функции, вы можете заключить выражение в круглые скобки или воспользоваться обратным конвейерным оператором:

```
> printfn "The result of sprintf is %s" (sprintf "(%d, %d)" 1 2);;
The result of sprintf is (1, 2)
val it : unit = ()

> printfn "The result of sprintf is %s" <| sprintf "(%d, %d)" 1 2;;
The result of sprintf is (1, 2)
val it : unit = ()
```

Обратный конвейерный оператор используется значительно реже, чем прямой конвейерный оператор или оператор композиции функций, но он также играет важную роль в упрощении кода F#.

Обратный оператор композиции

Подобно паре конвейерных операторов, точно так же существует обратный оператор композиции (backward composition operator), <<. Обратный оператор композиции принимает две функции и сначала вызывает функцию справа, а затем слева. Его удобно использовать, когда вы хотите выразить идею в обратном порядке. Он имеет следующее определение:

```
let (<<) f g x = f(g x)
```

Следующий фрагмент демонстрирует, как получить квадрат отрицательного числа. Использование обратного оператора композиции позволяет читать текст программы именно так, как он будет выполняться, то есть записывать программный код, который будет читаться в точном соответствии с тем, как он работает:

```
> // Обратная композиция
let square x = x * x
let negate x = -x;;

val square : int -> int
val negate : int -> int
```

```
> // Использование (>>)изменяет знак квадрата числа
(square >> negate) 10;;
val it : int = -100
> (* Но мы хотим получить квадрат отрицательного числа, поэтому должны
использовать оператор (<<) *)
(square << negate) 10;;
val it : int = 100
```

Еще один пример использования обратного оператора композиции – удаление пустых списков в списке списков. Здесь обратный оператор композиции также используется для изменения порядка чтения кода:

```
> // Фильтрация списков
[ [1]; []; [4;5;6]; [3;4]; []; []; []; [9] ]
|> List.filter(not << List.isEmpty);;

val it : int list list = [[1]; [4; 5; 6]; [3; 4]; [9]]
```

Сопоставление с образцом

В любых программах возникает необходимость фильтровать и сортировать данные. Для этой цели в функциональном программировании используется сопоставление с образцом (pattern matching). Сопоставление с образцом напоминает инструкцию `switch` в языках C# и C++, но обладает более широкими возможностями. По сути это набор правил, которые выполняются при совпадении входного значения с образцом. Затем выражение сопоставления с образцом возвращает результат правила, для которого было найдено соответствие. Вследствие этого все правила должны возвращать значения одного и того же типа.

Для использования сопоставления с образцом применяются ключевые слова `match` и `with` с набором правил, за каждым из которых следует стрелка `->`. Следующий фрагмент демонстрирует использование сопоставления с результатом выражения `isOdd x` для имитации поведения условного выражения. Первое правило соответствует значению `true`, и, если будет найдено совпадение с этим правилом, в консоль будет выведено сообщение "`x is odd`" (`x` – нечетное число):

```
> // Простое сопоставление с образцом
let isOdd x = (x % 2 = 1)

let describeNumber x =
    match isOdd x with
    | true  -> printfn "x is odd"
    | false -> printfn "x is even";;

val isOdd : int -> bool
val describeNumber : int -> unit
```

```
> describeNumber 4;;
x is even
val it : unit = ()
```

Сопоставление с константами – это простейшая форма сопоставления с образцом. Пример 3.5 создает таблицу истинности для булевой функции `And`, сопоставляя оба значения кортежа одновременно.

Пример 3.5. Создание таблицы истинности с помощью сопоставления с образцом

```
> // Таблица истинности для функции AND с помощью сопоставления с образцом
let testAnd x y =
    match x, y with
    | true,  true  -> true
    | true,  false -> false
    | false, true  -> false
    | false, false -> false;;

val testAnd : bool -> bool -> bool

> testAnd true true;;
val it : bool = true
```

Обратите внимание, что механизм вывода типов работает с выражениями сопоставления с образцом. В примере 3.5 первое правило сопоставляет `x` и `y` с булевыми величинами, механизм выведения типов определяет, что `x` и `y` являются булевыми величинами.

Символ подчеркивания `_` в выражениях сопоставления играет роль группового символа (wildcard) и совпадает с любыми значениями. Благодаря этому мы можем упростить предыдущий пример, добавив правило с групповым символом, которое будет выполняться для любых комбинаций входных значений кроме `true true`:

```
let testAnd x y =
    match x, y with
    | true, true -> true
    | _, _       -> false
```

Правила сопоставления с образцом проверяются в порядке их объявления. Поэтому если вставить правило с групповыми символами первым, все последующие правила никогда не будут проверяться.

Отсутствие совпадений

Возможно, вас уже заинтересовал вопрос: что произойдет, если убрать одно из правил таблицы истинности в исходной версии функции `testAnd`? Например:

```
let testAnd x y =
    match x, y with
```

```
    | true, true -> true
    | true, false -> false
    // | false, true -> false - Упс! Случай false, true отсутствует!
    | false, false -> false
```

Если в процессе сопоставления с образцом не будет найдено совпадение, будет сгенерировано исключение `Microsoft.FSharp.Core.MatchFailureException`. Вы можете избежать этого, определив правила для всех возможных значений. К счастью, компилятор F# выводит предупреждение, когда оказывается в состоянии обнаружить, что набор правил неполон:

```
> // Неполный набор правил.
// ВНИМАНИЕ! Нет совпадений для каждого символа.
let letterIndex l =
    match l with
    | 'a' -> 1
    | 'b' -> 2;;

      match l with
  ----------^
stdin(48,11): warning FS0025: Incomplete pattern matches on this expression.
For example, the value '' '' may indicate a case not covered by the pattern(s).
```

(Незавершенный шаблон соответствует данному выражению. К примеру, значение "' '" может указывать на случай, не покрытый шаблоном(ами).)

```
val letterIndex : char -> int
> letterIndex 'k';;
Microsoft.FSharp.Core.MatchFailureException: The match cases were incomplete
    at FSI_0040.letterIndex(Char l)
    at <StartupCode$FSI_0041>.$FSI_0041.main@()
stopped due to error
```

*(Варианты сопоставления не были завершены в...
Остановлено из-за ошибки)*

Именованные образцы

До сих пор мы выполняли сопоставление с константами, но точно так же можно использовать именованные образцы при извлечении данных и привязке их к новым значениям. Взгляните на следующий пример. В нем выполняется сопоставление с определенными строками, а в случае несоответствия используется значение, связанное с именем `x`:

```
> // Именованные образцы
let greet name =
    match name with
    | "Robert"  -> printfn "Hello, Bob"
    | "William" -> printfn "Hello, Bill"
    | x         -> printfn "Hello, %s" x;;
```

```
val greet : string -> unit

> greet "Earl";;
Hello, Earl
val it : unit = ()
```

Последнее правило не выполняет сопоставление с константой. Вместо этого оно связывает сопоставляемое значение с именем `x`, которое затем может использоваться в теле правила. Захват значений, подобно групповому символу, соответствуют любым значениям, поэтому такие правила всегда должны быть последними.

Сопоставление с литералами

Возможность использования именованных образцов хороша сама по себе, но она не дает возможности использовать существующие значения в качестве образцов. Использование жестко заданных имен в выражении сопоставления, как это сделано в предыдущем примере, может существенно осложнить отладку кода и привести к ошибкам.

Если вы хотите использовать сопоставление с часто используемым значением, но не хотите копировать эти значения во все правила, с именованными образцами вы можете допустить ошибку. В этом случае вы не получите ожидаемый результат, так как такое правило просто будет дополнительно вводить такое же имя во вложенную область видимости.

В следующем примере первое правило не будет сравнивать значение `name` со значением `bill`, вместо этого объявляется новое значение с именем `bill`. Именно по этой причине второе правило вызовет предупреждение, так как предыдущее правило уже соответствует любым возможным входным значениям:

```
> // ОШИБКА: Непреднамеренный захват значения
let bill = "Bill Gates"
let greet name =
    match name with
    | bill -> "Hello Bill!"
    | x    -> sprintf "Hello, %s" x;;

      | x -> sprintf "Hello, %s" x;;
  ------^^

stdin(56,7): warning FS0026: This rule will never be matched.
```

(Данное правило никогда не будет сопоставлено.)

```
val bill : string = "Bill Gates"
val greet : string -> string
```

Для сопоставления с имеющимся значением необходимо добавить атрибут `[<Literal>]`, что позволит использовать любое литеральное значение

(константу) внутри выражения сопоставления. Обратите внимание, что имена значений, помеченных атрибутом [<Literal>], должны начинаться с заглавной буквы.

Атрибуты – это способ аннотирования программного кода в .NET. Дополнительная информация об атрибутах и метаданных .NET приводится в главе 12.

```
> // Определение литерального значения
[<Literal>]
let Bill = "Bill Gates";;

val Bill : string = "Bill Gates"
> // Сопоставление с литеральными значениями

let greet name =
    match name with
    | Bill -> "Hello Bill!"
    | x    -> sprintf "Hello, %s" x;;

val greet : string -> string
> greet "Bill G.";;
val it : string = "Hello, Bill G."
> greet "Bill Gates";;
val it : string = "Hello Bill!"
```

Атрибутом [<Literal>] могут быть помечены только целые числа, символы, булевы значения, строки и вещественные числа. Если вам потребуется использовать сопоставление со значениями более сложных типов, такими как словари или отображения, придется использовать ограничение when.

Ограничение when

Сопоставление с образцом – достаточно мощная концепция, но иногда бывает необходимо добавить дополнительную логику, чтобы определить, соответствует значение данному правилу или нет. Именно для этого предназначено ограничение when. Если сопоставление с образцом найдено, вычисляется необязательное ограничение when, и правило выполняется тогда и только тогда, когда выражение when возвращает true.

В следующем примере реализована простая игра, в которой вам нужно угадать случайное число. Ограничение when используется, чтобы определить, является ли предложенное вами число больше, меньше или равно секретному числу:

```
> // Игра больше-меньше
open System
```

```
let highLowGame () =

    let rng = new Random()
    let secretNumber = rng.Next() % 100

    let rec highLowGameStep () =

        printfn "Guess the secret number:"
        let guessStr = Console.ReadLine()
        let guess = Int32.Parse(guessStr)
        match guess with
        | _ when guess > secretNumber
            -> printfn "The secret number is lower."
               highLowGameStep()

        | _ when guess = secretNumber
            -> printfn "You've guessed correctly!"
               ()

        | _ when guess < secretNumber
            -> printfn "The secret number is higher."
               highLowGameStep()

    // Начало игры
    highLowGameStep();;

val highLowGame : unit -> unit

> highLowGame();;
Guess the secret number: 50
The secret number is lower.
Guess the secret number: 25
The secret number is higher.
Guess the secret number: 37
You've guessed correctly!
val it : unit = ()
```

Группировка образцов

По мере того как в ваших выражениях сопоставления с образцом будут содержать все больше и больше правил, у вас может появиться желание объединить некоторые из образцов. Существует два способа объединения образцов. Первый – объединение по «ИЛИ» с помощью символа вертикальной черты |. В таком случае объединения правило будет запускаться, если значение совпадет с любым образцом в группе. Второй – объединение по «И» с помощью символа амперсанда &. В этом случае правило будет выполнено, только если значение соответствует всем образцам в группе:

```
let vowelTest c =
    match c with
```

```
    | 'a' | 'e' | 'i' | 'o' | 'u'
        -> true
    | _ -> false

let describeNumbers x y =
    match x, y with
    | 1, _
    | _, 1
        -> "One of the numbers is one."
    | (2, _) & (_, 2)
        -> "Both of the numbers are two"
    | _ -> "Other."
```

Объединение по «И» обычно используется редко, однако эта возможность неоценима при использовании активных образцов (глава 7).

Сопоставление структур данных

Сопоставление с образцом может также выполняться со структурами данных.

Кортежи

Вы уже видели, как можно использовать сопоставление с кортежами. Если в сопоставлении элементы кортежа отделены друг от друга запятыми, то все элементы кортежа будут сопоставляться по отдельности. Однако если с именованным образцом используется кортеж, связываемое значение должно иметь тип кортежа.

В следующем примере первое правило связывает значение с именем `tuple` с кортежем и захватывает оба значения `x` и `y`. Другие правила выполняют сопоставление элементов кортежа по отдельности:

```
let testXor x y =
    match x, y with
    | tuple when fst tuple <> snd tuple
        -> true
    | true, true -> false
    | false, false -> false
```

Списки

В примере 3.6 демонстрируется использование сопоставления с образцом совместно со списками. Функция `listLength` сначала находит сопоставление со списками фиксированной длины, и если совпадение не найдено, она рекурсивно вызывает саму себя со списком без первого элемента.

Пример 3.6. Определение длины списка

```
let rec listLength l =
   match l with
   | []         -> 0
   | [_]        -> 1
```

```
| [_; _]     -> 2
| [_; _; _]  -> 3
| hd :: tail -> 1 + listLength tail
```

Первые четыре правила находят сопоставления со списками определенной длины, используя групповые символы, чтобы показать, что значения элементов списка не важны. Однако в последней строке сопоставления с образцом используется оператор добавления элемента `::`, чтобы найти соответствие первого элемента списка, `hd`, оставшейся его части, `tail`. `tail` может быть любым списком: как пустым `[]`, так и содержащим миллион элементов. (Впрочем, учитывая предыдущие правила в функции, можно смело заявить, что список `tail` будет содержать не менее трех элементов.)

Необязательные значения

Сопоставление с образцом предоставляет более функциональный подход к использованию типов `option`:

```
let describeOption o =
    match o with
    | Some(42) -> "The answer was 42, but what was the question?"[1]
    | Some(x)  -> sprintf "The answer was %d" x
    | None     -> "No answer found."
```

За пределами выражений сопоставления

Сопоставление с образцом – достаточно мощная концепция, но самое интересное, что оно используется не только внутри выражений `'match with'`. Сопоставление с образцом в языке F# происходит повсеместно.

Операторы связывания let

Операторы связывания `let` в действительности являются правилами сопоставления с образцом. Так, конструкцию:

```
let x = f()
...
```

можно рассматривать так:

```
match f() with
| x -> ...
```

Следующий код поясняет, как извлекаются значения из кортежей:

```
// Следующая инструкция...
```

[1] Здесь автор шутит по поводу знаменитого вопроса из книги Дугласа Адамса «Путеводитель для путешествующих автостопом по галактике» – «Ответ на главный вопрос жизни, вселенной и всего такого». На нахождение ответа потребовалось семь с половиной миллионов лет вычислений, в результате чего был дан краткий ответ: «42». – *Прим. науч. ред.*

```
let x, y = (100, 200)

// ... аналогична такой конструкции...
match (100, 200) with
| x, y -> ...
```

Анонимные функции

Параметры анонимных функций тоже являются замаскированными операциями сопоставления!

```
// Принимает кортеж необязательных значений и возвращает их сумму
let addOptionValues = fun (Some(x), Some(y)) -> x + y
```

Сопоставления с групповыми символами

Представьте, что вам требуется написать функцию, один из параметров которой вам не требуется, или вам не нужны некоторые значения в кортеже. В этом случае можно воспользоваться групповым символом:

```
> List.iter (fun _ -> printfn "Step...") [1 .. 3];;
Step...
Step...
Step...
val it : unit = ()

> let _, second, _ = (1, 2, 3);;

val second : int = 2
```

Альтернативный синтаксис лямбда-выражений

Наконец, сопоставления с образцом можно использовать как упрощенный синтаксис лямбда-выражений. В процессе разработки программ на языке F# обычной практикой является передача параметра непосредственно в выражение сопоставления с образцом. Например:

```
let rec listLength theList =
    match theList with
    | [] -> 0
    | [_] -> 1
    | [_; _] -> 2
    | [_; _; _] -> 3
    | hd :: tail -> 1 + listLength tail
```

Эту конструкцию можно переписать немного проще с помощью ключевого слова `function`. Оно действует практически так же, как ключевое слово `fun`, с помощью которого создаются лямбда-выражения, за исключением того, что лямбда-выражения, созданные с помощью `function`, принимают единственный параметр, который должен передаваться выражению сопоставления с образцом. В следующем примере представлена переделанная версия функции `listLength`, в которой используется ключевое слово `function`:

```
> // Ключевое слово 'function'
let rec funListLength =
    function
    | [] -> 0
    | [_] -> 1
    | [_; _] -> 2
    | [_; _; _] -> 3
    | hd :: tail -> 1 + funListLength tail;;

val funListLength : 'a list -> int

> funListLength [1 .. 5];;
val it : int = 5
```

Размеченные объединения

Одним из основополагающих типов в функциональном программировании являются *размеченные объединения (discriminated unions)*. Это тип данных, который может принимать только одно из множества возможных значений. Каждое возможное значение размеченного объединения называется *вариантом объединения (union case)*. Благодаря тому что размеченные объединения могут иметь только одно из множества допустимых значений, компилятор может выполнять дополнительные проверки корректности кода, в частности проверить полноту набора правил сопоставления с образцом, когда оно используется с размеченными объединениями.

Размеченные объединения объявляются с помощью ключевого слова `type`, за которым указывается имя типа и варианты объединения, разделенные символом вертикальной черты `|`. Так, масть карты из стандартной колоды может быть представлена следующим размеченным объединением:

```
> // Размеченное объединение, представляющее масть карты
type Suit =
    | Heart
    | Diamond
    | Spade
    | Club;;

type Suit =
    | Heart
    | Diamond
    | Spade
    | Club

> let suits = [ Heart; Diamond; Spade; Club ];;

val suits : Suit list = [Heart; Diamond; Spade; Club]
```

При необходимости с каждым вариантом объединения может быть связано некоторое значение. Если продолжить пример с игральными картами, каждая карта с картинкой может быть ассоциирована с определенной мастью, а все остальные карты – с целым значением и мастью, в виде пары целое число/масть. (Конструкция `int * Suit` может показаться вам знакомой – это сигнатура кортежа.)

```
// Размеченное объединение для представления игральных карт
type PlayingCard =
    | Ace   of Suit
    | King  of Suit
    | Queen of Suit
    | Jack  of Suit
    | ValueCard of int * Suit

// Использовать генератор списка для создания колоды карт
let deckOfCards =
    [
        for suit in [ Spade; Club; Heart; Diamond ] do
            yield Ace(suit)
            yield King(suit)
            yield Queen(suit)
            yield Jack(suit)
            for value in 2 .. 10 do
                yield ValueCard(value, suit)
    ]
```

Кроме того, имеется возможность объявлять размеченные объединения в одной строке. В этом случае первый символ вертикальной черты | становится необязательным.

```
type Number = Odd | Even
```

Размеченные объединения могут образовывать рекурсивные структуры. Если вам потребуется объявить множество взаимно рекурсивных размеченных объединений, то, как и в случае с функциями, вам необходимо будет связать их с помощью ключевого слова `and`.

Ниже приводится пример простого формата для описания языка программирования:

```
// Инструкции
type Statement =
    | Print    of string
    | Sequence of Statement * Statement
    | IfStmt   of Expression * Statement * Statement

// Выражения
and Expression =
    | Integer     of int
    | LessThan    of Expression * Expression
    | GreaterThan of Expression * Expression
```

```
(*
    if (3 > 1)
        print “3 is greater than 1”
    else
        print “3 is not”
        print “greater than 1”
*)
let program =
    IfStmt(
        GreaterThan(
            Integer(3),
            Integer(1)),
        Print(“3 is greater than 1”),
        Sequence(
            Print(“3 is not”),
            Print(“greater than 1”)
        )
    )
```

Использование размеченных объединений для создания древовидных структур

Размеченные объединения идеально подходят для представления древовидных структур. В примере 3.7 определяется двоичное дерево и функция для его обхода, причем все это занимает всего 11 строк кода. Такая выразительность размеченных объединений делает их идеальными для использования в ситуациях, когда необходимо обрабатывать данные с древовидной структурой.

Пример 3.7. Реализация двоичного дерева на основе размеченного объединения

```
type BinaryTree =
    | Node of int * BinaryTree * BinaryTree
    | Empty

let rec printInOrder tree =
    match tree with
    | Node (data, left, right)
        -> printInOrder left
           printfn “Node %d” data
           printInOrder right
    | Empty
        -> ()

(*
        2
       / \
      1   4
         / \
        3   5
```

```
*)
let binTree =
    Node(2,
        Node(1, Empty, Empty),
        Node(4,
            Node(3, Empty, Empty),
            Node(5, Empty, Empty)
        )
    )
```

Если выполнить этот пример в окне FSI, то получим следующее:

```
> printInOrder binTree;;
Node 1
Node 2
Node 3
Node 4
Node 5
val it : unit = ()
```

Сопоставление с образцом

Вы можете использовать сопоставления с образцом совместно с размеченными объединениями путем использования вариантов объединения в качестве образцов. Если вариант объединения ассоциирован с какими-либо данными, можно использовать сопоставление с константой, с групповым символом или захватывать значения, как при обыкновенном сопоставлении с образцом.

Следующий пример демонстрирует мощь сопоставления с образцом и размеченных объединений для описания пар карт, имеющих особое значение при игре в покер:

```
// Описывает пары карт, имеющих особое значение при игре в покер
let describeHoleCards cards =
    match cards with
    | []
    | [_]
        -> failwith "Too few cards."
    | cards when List.length cards > 2
        -> failwith "Too many cards."

    | [ Ace(_); Ace(_) ] -> "Pocket Rockets"
    | [ King(_); King(_) ] -> "Cowboys"

    | [ ValueCard(2, _); ValueCard(2, _)]
        -> "Ducks"

    | [ Queen(_); Queen(_) ]
    | [ Jack(_); Jack(_) ]
        -> "Pair of face cards"
```

```
    | [ ValueCard(x, _); ValueCard(y, _) ] when x = y
        -> "A Pair"

    | [ first; second ]
        -> sprintf "Two cards: %A and %A" first second
```

В сопоставлении с образцом допускается также использовать рекурсивные размеченные объединения. Обратите внимание, что в следующем примере третье правило использует вложенный образец, который соответствует только тем значениям `Manager` (руководитель), в подчинении у которых имеется точно два `Worker` (подчиненных):

```
type Employee =
    | Manager of string * Employee list
    | Worker of string

let rec printOrganization worker =
    match worker with
    | Worker(name) -> printfn "Employee %s" name

    // Руководитель, в списке подчиненных у которого всего один элемент
    | Manager(managerName, [ Worker(employeeName) ])
        -> printfn "Manager %s with Worker %s" managerName employeeName

    // Руководитель, в списке подчиненных у которого два элемента
    | Manager(managerName, [ Worker(employee1); Worker(employee2) ])
        -> printfn
                "Manager %s with two workers %s and %s"
                managerName employee1 employee2

    // Руководитель со списком подчиненных
    | Manager(managerName, workers)
        -> printfn "Manager %s with workers..." managerName
           workers |> List.iter printOrganization
```

Если предыдущий пример запустить в окне FSI, он выведет следующее:

```
> let company = Manager("Tom", [ Worker("Pam"); Worker("Stuart") ] );;

val company : Employee = Manager ("Tom",[Worker "Pam"; Worker "Stuart"])

> printOrganization company;;
Manager Tom with two workers Pam and Stuart
val it : unit = ()
```

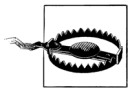
Благодаря тому что компилятору известны все возможные варианты размеченного объединения, любой неполный набор образцов будет вызывать предупреждение. Например, если вариант `Ace` исключить из выражения сопоставления, компилятор F# выведет следующее:

```
> // ВНИМАНИЕ! Забыли об Ace...
let getCardValue card =
    match card with
```

```
    | King(_) | Queen(_) | Jack(_) -> 10
    | ValueCard(x, _) -> x;;

      match card with
  ----------^^^^
stdin(53,11): warning FS0025: Incomplete pattern matches on this expression.
For example, the value 'Ace (_)' may indicate a case not covered by the
pattern(s).
```

(Незавершенный шаблон соответствует данному выражению. К примеру, значение "Ace (_)" может указывать на случай, не покрытый шаблоном(ами).)

```
val getCardValue : PlayingCard -> int
```



Будьте внимательны при использовании группового символа для сопоставления с размеченными объединениями. Если позднее объединение будет дополнено новыми вариантами, компилятор будет выводить предупреждения для каждого сопоставления с неполным набором вариантов. Однако если в выражении используется групповой символ, никаких предупреждений выводиться не будет, потому что этот вариант будет соответствовать всем новым случаям. С получением предупреждений о коде, который может приводить к ошибкам времени выполнения при сопоставлении, начинается долгий путь по предотвращению ошибок.

Методы и свойства

Вы можете расширять возможности размеченных объединений за счет добавления свойств и методов, которые являются функциями, вызываемыми у экземпляров размеченного объединения. (Дополнительная информация о добавлении свойств и методов приводится в главе 5.)

```
type PlayingCard =
    | Ace of Suit
    | King of Suit
    | Queen of Suit
    | Jack of Suit
    | ValueCard of int * Suit

    member this.Value =
        match this with
        | Ace(_) -> 11
        | King(_) | Queen (_)| Jack(_) -> 10
        | ValueCard(x, _) when x <= 10 && x >= 2
                        -> x
        | ValueCard(_) -> failwith "Card has an invalid value!"

let highCard = Ace(Spade)
let highCardValue = highCard.Value
```

Записи

Размеченные объединения с успехом могут использоваться для определения иерархических данных, но когда вы пытаетесь получить данные из размеченного объединения, возникает та же проблема, что и с кортежами, а именно: значения не имеют какого-то особого смысла – скорее, это просто данные, сваленные в общую кучу в некотором фиксированном порядке. Например, взгляните на следующее размеченное объединение, предназначенное для описания человека. Первая строковое поле в нем описывает имя, а второе – фамилию, или наоборот? Такая неопределенность может приводить к путанице и появлению ошибок.

```
type Person =
    | Person of string * string * int

let steve = Person("Steve", "Holt", 17)
let gob = Person("Bluth", "George Oscar", 36)
```

Если вам необходимо сгруппировать данные в структуру с помощью достаточно простого синтаксиса, вы можете использовать тип *запись (record)*. Записи позволяют организовать значения в типы, а также присвоить имена отдельным *полям*.

Для определения записи нужно задать последовательность пар имя/тип и заключить их в фигурные скобки. Чтобы создать экземпляр записи, достаточно определить значения каждого поля записи, а все остальное сделает механизм вывода типов, как показано в примере 3.8.

Пример 3.8. Конструирование и использование записей

```
> // Определение типа записи
type PersonRec = { First : string; Last : string; Age : int};;

type PersonRec =
  {First: string;
   Last: string;
   Age: int;}

> // Конструирование экземпляра записи
let steve = { First = "Steve"; Last = "Holt"; Age = 17 };;

val steve : PersonRec = {First = "Steve";
                         Last = "Holt";
                         Age = 17;}

> // Использование '.field'
// для доступа к полям записи
printfn "%s is %d years old" steve.First steve.Age;;
Steve is 17 years old
val it : unit = ()
```

Опытному разработчику приложений на платформе .NET записи могут показаться упрощенной версией стандартных классов .NET. В языке F# легко можно добавлять свойства и методы, тогда зачем вообще могут понадобиться записи? Записи предлагают ряд явных преимуществ перед традиционными объектно-ориентированными структурами данных:

- Механизм вывода типов способен автоматически определять типы записей, без дополнительных аннотаций типов.
- Поля записи по умолчанию являются неизменяемыми, тогда как классы не обладают такой особенностью.
- Записи не могут наследоваться, гарантируя свою неизменность на будущее.
- Записи могут использоваться в выражениях сопоставления с образцом, тогда как классы – нет, без применения активных шаблонов или ограничения when.
- Записи (и размеченные объединения) автоматически могут использоваться в операциях сравнения. Особенности операций сравнения в .NET рассматриваются в главе 6.

Клонирование записей

Записи легко могут клонироваться с помощью ключевого слова `with`:

```
type Car =
    {
        Make : string
        Model : string
        Year : int
    }

let thisYear's = { Make = "FSharp"; Model = "Luxury Sedan"; Year = 2010 }
let nextYear's = { thisYear's with Year = 2011 }
```

Что эквивалентно следующему:

```
let nextYear's =
    {
        Make  = thisYear's.Make
        Model = thisYear's.Model
        Year  = 2011
    }
```

Сопоставление с образцом

Записи могут участвовать в сопоставлении с образцом с возможностями захвата значений и сопоставления с литералами. Обратите внимание, что не все поля записи должны участвовать в сопоставлении с образцом.

В следующем примере выполняется фильтрация исходного списка записей типа `Car`, чтобы в новом списке остались только записи, в которых поле `Model` имеет значение `"Coup"`:

```
let allCoups =
    allNewCars
    |> List.filter
       (function
         | { Model = "Coup" } -> true
         | _                  -> false)
```

Вывод типов

Важным преимуществом записей является способ их взаимодействия с механизмом вывода типов в языке F#. Для использования классов .NET необходимо указывать аннотации типов, тогда как тип записи может быть выведен по именам используемых полей.

В примере 3.9 отсутствуют аннотации типов для значений `pt1` и `pt2`. Благодаря тому что в этом фрагменте выполняются обращения к полям `X` и `Y`, компилятор знает о существовании типа записи с полями `X` и `Y`, поэтому он определяет, что типом значений `pt1` и `pt2` является `Point`.

Пример 3.9. Определение типов записей

```
> type Point = { X : float; Y : float };;

type Point =
  {X: float;
   Y: float;}

> // Расстояние между двумя точками
let distance pt1 pt2 =
    let square x = x * x
    sqrt <| square (pt1.X - pt2.X) + square (pt1.Y - pt2.Y);;

val distance : Point -> Point -> float

> distance {X = 0.0; Y = 0.0} {X = 10.0; Y = 10.0};;
val it : float = 14.14213562
```

Если в программе используются два типа записей, имеющие поля с одинаковыми именами, механизм вывода типов либо выдаст сообщение об ошибке, либо определит типы значений, которые могут отличаться от предполагаемых. Чтобы предотвратить подобные проблемы, необходимо добавить аннотации типов или использовать полные квалифицированные имена полей:

```
type Point = { X: float; Y: float;}
type Vector3 = { X : float; Y : float; Z : float }
```

```
// Аннотации типов не позволят компилятору определить
// тип pt1 и pt2 как Vector3 (поскольку тип Vector3 определен последним
// и содержит поля X и Y)
let distance (pt1 : Point) (pt2 : Point) =
    let square x = x * x
    sqrt <| square (pt1.X - pt2.X) + square (pt1.Y - pt2.Y)

// Устранить неоднозначность можно также за счет использования
// полностью квалифицированных имен полей.
let origin = { Point.X = 0.0; Point.Y = 0.0 }
```

Тем самым вы предоставите механизму вывода типов всю необходимую информацию, которая позволит ему понять, что же вы имели в виду.

Методы и свойства

Как и для размеченных объединений, вы можете добавлять к записям свойства и методы:

```
> // Добавление свойства к записи типа Vector =
type Vector =
    { X : float; Y : float; Z : float }
    member this.Length =
        sqrt <| this.X ** 2.0 + this.Y ** 2.0 + this.Z ** 2.0;;

type Vector =
  {X: float;
   Y: float;
   Z: float;}
  with
    member Length : float
  end

> let v = { X = 10.0; Y = 20.0; Z = 30.0 };;

val v : Vector = {X = 10.0;
                  Y = 20.0;
                  Z = 30.0;}

> v.Length;;
val it : float = 37.41657387
```

Отложенные вычисления

Программный код, который мы писали до сих пор, выполнялся *немедленно (eagerly)*, то есть он выполнялся, как только мы заканчивали его ввод в окне FSI. То же самое произойдет, если скомпилировать и запустить программу на языке F#. Однако иногда возникают ситуации, когда необходимо, чтобы код выполнялся по требованию, например,

когда вы хотите, чтобы дорогостоящие вычисления выполнялись только в случае необходимости. Этот прием известен как *отложенные вычисления (lazy evaluation)*.

Использование отложенных вычислений может привести к существенному снижению расхода памяти, поскольку в этом случае значения создаются в памяти только при необходимости.

Язык F# поддерживает две разновидности отложенных вычислений: с помощью типа `Lazy` и с помощью последовательностей (тип `seq<'a>`).

Типы Lazy

Тип `Lazy` является заглушкой, или заменителем, для некоторых вычислений. После создания вы можете передавать экземпляр типа `Lazy`, как если бы его значение было вычислено. Но в действительности значение будет вычислено всего один раз и только при необходимости.

В примере 3.10 создаются два значения, `x` и `y`, вычисление каждого из которых содержит побочный эффект – вывод сообщения в консоль. Поскольку они инициализируются отложенно, они не вычисляются в момент объявления. Вычисление `y` происходит, только когда это значение оказывается необходимым, при этом происходит принудительное вычисление значения `x`.

Для создания отложенного значения вы можете использовать ключевое слово `lazy` или `Lazy<_>.Create`.

Пример 3.10. Использование отложенных вычислений

```
> // Определяются два отложенных значения
let x = Lazy<int>.Create(fun () -> printfn "Evaluating x..."; 10)
let y = lazy (printfn "Evaluating y..."; x.Value + x.Value);;

val x : Lazy<int> = <unevaluated>
val y : Lazy<int> = <unevaluated>

> // Прямое обращение к значению y приводит к его вычислению
y.Value;;
Evaluating y...
Evaluating x...
val it : int = 20

> // При повторном обращении к значению y используется сохраненное ранее
// значение (побочные эффекты отсутствуют)
y.Value;;
val it : int = 20
```

Последовательности

Наиболее часто отложенные вычисления используются при работе с последовательностями или типом `seq`, который представляет упорядочен-

ную последовательность элементов, очень похожую на тип list. В следующем фрагменте определяется последовательность из пяти элементов и выполняется итерация по ним с помощью функции Seq.iter. (Существует целый модуль, аналогичный модулю List, содержащий множество различных функций для работы с последовательностями.)

```
> let seqOfNumbers = seq { 1 .. 5 };;

val seqOfNumbers : seq<int>

> seqOfNumbers |> Seq.iter (printfn "%d");;
1
2
3
4
5
val it : unit = ()
```

Различия между типами seq и list заключается в том, что в каждый конкретный момент времени в памяти существует только один элемент последовательности. Тип seq – это всего лишь псевдоним для интерфейса System.Collections.Generic.IEnumerable <'a> в платформе .NET.

Тогда зачем нужны эти ограничения? Зачем нам два типа, когда мы просто можем использовать тип list? Так как содержимое списка целиком хранится в памяти, это означает, что вы ограничены объемом имеющихся ресурсов. Вы легко можете определить бесконечную последовательность, но бесконечный список просто не поместится в память. Кроме того, при использовании списков значение каждого его элемента должно быть известно заранее, тогда как последовательности могут развертываться динамически (это так называемая «pull»-модель).

В примере 3.11 определяется последовательность строковых представлений всех положительных 32-битовых целых чисел. Затем выполняется попытка создать эквивалентный список, но она терпит неудачу из-за нехватки памяти.

Пример 3.11. Последовательность всех целых чисел

```
> // Последовательность всех целых чисел
let allIntsSeq = seq { for i = 0 to System.Int32.MaxValue -> i };;

val allIntsSeq : seq<int>

> allIntsSeq;;
val it : seq<int> = seq [0; 1; 2; 3; ...]
> // Список всех целых чисел - ОШИБКА: не умещается в памяти!
let allIntsList = [ for i = 0 to System.Int32.MaxValue do yield i ];;
System.OutOfMemoryException: Exception of type 'System.OutOfMemoryException' was
thrown.
```

(Было возбуждено исключение типа "System.OutOfMemoryException".)

Выражения последовательности

Для определения последовательностей можно использовать синтаксис генераторов списков (такие конструкции называются *выражениями последовательности (sequence expressions)*). Определение последовательности начинается с ключевого слова `seq`, за которым следуют фигурные скобки:

```
> let alphabet = seq { for c in 'A' .. 'Z' -> c };;

val alphabet : seq<char>

> Seq.take 4 alphabet;;
val it : seq<char> = seq ['A'; 'B'; 'C'; 'D']
```

Последовательности вычисляются отложенно, поэтому каждый раз при получении очередного элемента выполняется код выражения последовательности. Попробуем выполнить предыдущий пример еще раз, но теперь добавим побочный эффект, который будет выполняться при возврате каждого элемента. Теперь выражение будет не только возвращать алфавитные символы, но и выводить их в консоль:

```
> // Последовательность с побочным эффектом
let noisyAlphabet =
    seq {
        for c in 'A' .. 'Z' do
            printfn "Yielding %c..." c
            yield c
    };;

val noisyAlphabet : seq<char>

> let fifthLetter = Seq.nth 4 noisyAlphabet;;
Yielding A...
Yielding B...
Yielding C...
Yielding D...
Yielding E...

val fifthLetter : char = 'E'
```

Интересно отметить, что выражения последовательности могут быть рекурсивными. С помощью ключевого слова `yield!` (произносится как *йилд банг*) можно вернуть подпоследовательность, которая объединяется с основной последовательностью.

Пример 3.12 демонстрирует, как использовать ключевое слово `yield!` внутри выражения последовательности. Функция `allFilesUnder` возвра-

щает значение типа seq<string> и получает имена всех файлов в указанной папке и рекурсивно во всех подпапках. Функция Directory.GetFiles возвращает массив строк, содержащий имена всех файлов в папке basePath. Поскольку тип array совместим с типом seq, инструкция yield! возвращает все эти файлы.

Пример 3.12. Последовательность с перечнем всех файлов в папке и в подпапках

```
let rec allFilesUnder basePath =
    seq {
        // Добавить все файлы из основной папки
        yield! Directory.GetFiles(basePath)
        // Добавить все файлы из подпапок
        for subdir in Directory.GetDirectories(basePath) do
            yield! allFilesUnder subdir
    }
```

Функции модуля Seq

Модуль Seq содержит множество полезных функций для работы с последовательностями.

Seq.take

Возвращает первые *n* элементов последовательности:

```
> // Последовательность случайных чисел
open System
let randomSequence =
    seq {
        let rng = new Random()
        while true do
            yield rng.Next()
    };;

val randomSequence : seq<int>

> randomSequence |> Seq.take 3;;
val it : seq<int> = seq [2101281294; 1297716638; 1114462900]
```

Seq.unfold

Генерирует последовательность, используя указанную функцию. Имеет тип ('a -> ('b * 'a) option) -> 'a -> seq<'b>.

Функция, передаваемая в виде аргумента, принимает входное значение и возвращает значение типа option с кортежем, содержащим очередное значение последовательности и входное значение для вызова функции на следующей итерации функции Seq.unfold. Для обозначения конца последовательности функция должна возвращать значение None. Пример 3.13 генерирует все числа Фибоначчи меньше 100. (Каждое последующее число Фибоначчи является суммой двух предыдущих, то есть:

1, 1, 2, 3, 5, 8 и так далее.) Кроме того, в этом примере используется функция Seq.tolist, которая преобразует последовательность в список.

Пример 3.13. Использование функции Seq.unfold

```
> // Генерирует очередной элемент последовательности чисел Фибоначчи на основе
// двух предыдущих элементов. Предназначена для использования функцией
// Seq.unfold.
let nextFibUnder100 (a, b) =
    if a + b > 100 then
        None
    else
        let nextValue = a + b
        Some(nextValue, (nextValue, a));;

val nextFibUnder100 : int * int -> (int * (int * int)) option

> let fibsUnder100 = Seq.unfold nextFibUnder100 (0, 1);;

val fibsUnder100 : seq<int>

> Seq.tolist fibsUnder100;;
val it : int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

В табл. 3.1 перечислены другие полезные функции из модуля Seq.

Таблица 3.1. Часто используемые функции из модуля Seq

Функция и ее тип	Описание
Seq.length seq<'a> -> int	Возвращает длину последовательности.
Seq.exists ('a -> bool) -> seq<'a> -> bool	Возвращает признак наличия в последовательности элемента, который удовлетворяет функции поиска.
Seq.tryFind ('a -> bool) -> seq<'a> -> 'a option	Возвращает Some(x) для первого элемента x, для которого указанная функция вернет true. В противном случае вернет None.
Seq.filter ('a -> bool) -> seq<'a> -> seq<'a>	Отфильтровывает все элементы последовательности, для которых указанная функция возвращает false.
Seq.concat (seq< #seq<'a> > -> seq<'a>	Объединяет серию последовательностей в единую последовательность seq.

Агрегатные операторы

Кроме этого, модуль Seq содержит агрегатные операторы, аналогичные операторам в модуле List.

Seq.iter

Выполняет обход всех элементов последовательности:

```
> // Выводит нечетные числа меньше 10
let oddsUnderN n = seq { for i in 1 .. 2 .. n -> i }
Seq.iter (printfn "%d") (oddsUnderN 10);;
1
3
5
7
9

val oddsUnderN : int -> seq<int>
```

Seq.map

Создает новую последовательность путем получения соответствия результатов выполнения указанной функции к элементам последовательности:

```
> // Последовательность слов (массивы совместимы с последовательностями)
let words = "The quick brown fox jumped over the lazy dog".Split( [| ' ' |]);;

val words : string [] =
  [|"The"; "quick"; "brown"; "fox"; "jumped"; "over"; "the"; "lazy"; "dog"|]

> // Получаем соответствие строк к кортежам вида: строка, длина строки
words |> Seq.map (fun word -> word, word.Length);;
val it : seq<string * int> =
  seq [("The", 3); ("quick", 5); ("brown", 5); ("fox", 3); ...]
```

Seq.fold

Свертывает последовательность к единственному значению. Поскольку обход последовательностей может выполняться только в одном направлении, в модуле `Seq` отсутствует эквивалент оператора `List.foldBack`:

```
> Seq.fold (+) 0 <| seq { 1 .. 100 };;
val it : int = 5050
```

Теперь вы знакомы со всеми основными аспектами функционального программирования. Нет ничего страшного, если преимущества таких возможностей, как композиция функций или отложенные вычисления, для вас пока неочевидны. Большинство примеров в этой книге написаны в функциональном стиле, и со временем вы сможете понять и оценить разнообразие демонстрируемых подходов к решению различных задач.

4

Императивное программирование

До сих пор большая часть написанных нами программ были *чистыми*, то есть они не изменяли свое состояние. Если функция делает еще что-то помимо возвращения некоторого значения, это называют *побочным эффектом*. Хотя чистые функции обладают рядом интересных особенностей, например возможностью композиции, тем не менее мало кому будут интересны программы, если они не выполняют что-то полезное: сохраняют данные на диск, выводят значения на экран, передают информацию по сети и так далее. Эти побочные эффекты и есть то, ради чего пишется программа.

В этой главе рассказывается об *императивном программировании*, то есть о том, как изменять состояние программы и управлять потоком выполнения. Считается, что при использовании этого стиля программирования легче допустить ошибку, чем при использовании функционального стиля, потому что в этом случае существует множество способов сделать что-либо не то. Чем более подробные инструкции приходится писать, чтобы заставить компьютер выполнить условный переход или сохранить конкретные значения в конкретной области памяти, тем вероятнее, что программист допустит ошибку. При программировании в функциональном стиле все данные являются неизменяемыми, поэтому вы не можете случайно присвоить неправильное значение. Тем не менее при благоразумном использовании императивный стиль программирования может быть очень полезен разработчику на F#.

В числе потенциальных преимуществ императивного программирования можно назвать:

- Улучшенную производительность
- Простоту сопровождения благодаря ясности кода
- Взаимодействие с существующим кодом

Императивное программирование – это такой стиль, когда программа решает свои задачи путем изменения данных в памяти. В этом случае программы обычно записываются как последовательность инструкций или команд. В примере 4.1 приводится гипотетическая программа боевого робота, используемого для покорения Земли. Функции не возвращают никаких значений, а вместо этого воздействуют на некоторые части системы, например обновляют внутренние структуры данных.

Пример 4.1. Покорение Земли с применением императивного стиля программирования

```
let robot = new GiantKillerRobot()

robot.Initialize()

robot.EyeLaserIntensity <- Intensity.Kill
robot.Target <- [| Animals; Humans; Superheroes |]

// Последовательность операций при покорении Земли
let earth = Planets.Earth
while robot.Active && earth.ContainsLife do
    if robot.CurrentTarget.IsAlive then
        robot.FireEyeLaserAt(robot.CurrentTarget)
    else
        robot.AquireNextTarget()
```

Фрагмент программы покорения Земли выглядит достаточно просто, но все сложности ее работы скрыты за кулисами. Функция `Initialize` может выполнять включение ядерного реактора, а попытка дважды вызвать функцию `Initialize` может приводить к взрыву реактора. Если бы функция `Initialize` была написана в чистом функциональном стиле, ее результат зависел бы только от значений входных параметров. Однако в данном случае действия функции `Initialize` зависят от текущего состояния памяти.

В этой главе рассказывается не о том, как программировать роботов, захватывающих планеты, а о том, как писать на языке F# программы, которые могут изменять свое окружение в процессе выполнения. Здесь вы узнаете, как объявлять *переменные*, значения которых могут изменяться в ходе работы программы. Научитесь использовать типы изменяемых коллекций, которые представляют собой простую в использовании альтернативу спискам. Наконец, вы познакомитесь с инструкциями управления потоком выполнения и с исключениями, позволяющими изменять порядок выполнения кода.

Понятие памяти в .NET

Прежде чем приступать к изменению содержимого памяти, необходимо понять, как в .NET осуществляется работа с памятью. Значения в при-

ложениях на платформе .NET сохраняются в одной из двух областей: в стеке или в «куче». (Опытные программисты наверняка знакомы с этими понятиями.) Стек – область памяти фиксированного объема, выделяемая каждому процессу, которая используется для хранения *локальных переменных*. Локальные переменные – это временные значения, такие как счетчики циклов, существующие только во время выполнения функции. Стек имеет достаточно ограниченный объем, тогда как «куча» (также известная как ОЗУ) может хранить несколько гигабайт данных. Программы на платформе .NET используют обе области памяти, стек и «кучу», размещая небольшие объемы данных в стеке, когда это возможно, и в «куче», когда требуется сохранить большие объемы данных.[1]

От того, в какой области памяти сохраняются данные, зависит, как эти данные используются.

Значимые типы и ссылочные типы

Типы данных, хранящихся в стеке, известны как *значимые типы (value types)*, а типы данных, хранящихся в «куче», известны как *ссылочные типы (reference types)*.

Значимые типы значений имеют фиксированный размер. Примерами значимых типов являются типы `int` и `float`, потому что данные этих типов имеют постоянный размер. С другой стороны, при работе со ссылочными типами в стеке хранятся лишь *указатели* на другие области памяти, где находятся фактические данные. Но несмотря на то, что сам указатель имеет фиксированный размер (обычно четыре или восемь байт), данные, на которые он указывает, могут занимать гораздо больший объем. Примерами ссылочных типов могут служить типы `list` и `string`.

Различия между значимыми и ссылочными типами наглядно показаны на рис. 4.1. Целое число 5 находится в стеке и не имеет соответствующей ему области памяти в «куче». В случае строки, напротив, в стеке находится лишь указатель на область памяти в «куче», где располагается последовательность символов.

Значения по умолчанию

До сих пор все значения, которые мы объявляли в программном коде F#, инициализировались сразу же в момент их создания, потому что

[1] На самом деле, объем занимаемой объектом памяти не имеет настолько однозначного отношения к тому, где этот объект будет располагаться. Тут правила более сложные. Наиболее точно эти правила описаны Эриком Липпертом (Eric Lippert) в статье “The Truth About Value Types” (*http://blogs.msdn.com/b/ericlippert/archive/2010/09/30/the-truth-about-value-types.aspx*). – *Прим. науч. ред.*

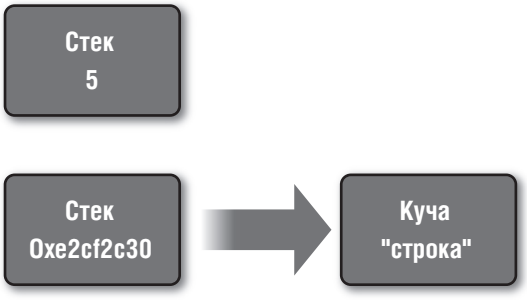


Рис. 4.1. *Значимые и ссылочные типы*

в функциональном программировании значения не могут изменяться после их объявления. Однако в императивном программировании отсутствует такая острая необходимость инициализировать значения немедленно, потому что вы всегда можете изменить их позднее. Это означает, что для значимых и ссылочных типов существует понятие *значение по умолчанию*, то есть некоторое значение, устанавливаемое до того, как переменная будет инициализирована.

Чтобы получить значение по умолчанию для любого типа, можно воспользоваться функцией типа `Unchecked.defaultof<'a>`. Она возвращает значение по умолчанию для указанного типа.

Функция типа (type function) – это особый тип функции, которая не принимает никаких аргументов, кроме параметра обобщенного типа. Существует несколько полезных функций типов, которые мы исследуем в следующих главах:

`Unchecked.defaultof<'a>`

Возвращает значение по умолчанию для типа `'a`.

`typeof<'a>`

Возвращает объект `System.Type`, описывающий тип `'a`.

`sizeof<'a>`

Возвращает размер памяти в стеке, выделяемой для объекта типа `'a`.

Для значимых типов в качестве значений по умолчанию используется нулевое битовое представление. Поскольку размер экземпляра известен в момент создания, в стеке для него выделяется необходимое количество байтов, каждый байт которого устанавливается в значение `0b00000000`.

Значения по умолчанию для ссылочных типов имеют более сложный вид. До инициализации ссылочный тип указывает на специальный адрес `null`. Он используется, чтобы отличать неинициализированные данные ссылочных типов.

Для проверки равенства ссылочного типа значению null в языке F# можно использовать ключевое слово null. В следующем примере определяется функция, которая проверяет, равен ли входной аргумент значению null или нет, а затем вызывает ее, передавая инициализированную и неинициализированную строки:

```
> let isNull = function null -> true | _ -> false;;

val isNull : obj -> bool

> isNull "a string";;
val it : bool = false
> isNull (null : string);;
val it : bool = true
```

Однако null в языке F# не является допустимым значением для ссылочных типов, то есть им нельзя присвоить значение null:

```
> type Thing = Plant | Animal | Mineral;;

type Thing =
    | Plant
    | Animal
    | Mineral

> // Ошибка: Значение Thing не может быть null
let testThing thing =
    match thing with
    | Plant -> "Plant"
    | Animal -> "Animal"
    | Mineral -> "Mineral"
    | null -> "(null)";;

    | null -> "(null)";;
------^^^^
stdin(9,7): error FS0043: The type 'Thing' does not have 'null' as a proper
value.
```

(Тип "Thing" не содержит собственное значение "null".)

Такое ограничение на первый взгляд выглядит достаточно странным, но благодаря этому отпадает необходимость в излишних проверках на равенство значению null. (Если вызвать метод на неинициализированном объекте ссылочного типа, программа сгенерирует исключение NullReferenceException. Такая защитная проверка всех параметров функций на равенство значению null является обычным делом.) Если вам потребуется представить неинициализированное состояние в программе на языке F#, то вместо ссылочного типа со значением null можно использовать тип Option, где значение None представляет неинициализированное состояние, а значение Some('a) – инициализированное.

В языке F# к типу можно добавить специальный атрибут, чтобы обеспечить возможность присваивания данным этих типов значения `null` и упростить взаимодействие с другими языками. NET. Подробнее об этом рассказывается в приложении B.

Совмещение имен ссылочных типов

Существует возможность двум ссылочным типам ссылаться на одну и ту же область памяти.[1] Этот прием известен как *совмещение имен (aliasing)*. В этом случае изменение одного значения приведет к изменению другого, потому что оба они ссылаются на одну и ту же область памяти. При неосторожном обращении такие ситуации могут приводить к ошибкам.

В примере 4.2 создается единственный экземпляр типа `array` (с которым мы скоро познакомимся поближе), но имеется два значения, указывающие на один и тот же экземпляр. При изменении значения `x` также изменяется и значение `y`, и наоборот.

Пример 4.2. Совмещение имен ссылочных типов

```
> // Значение x указывает на массив, значение y указывает
// на тот же самый массив, что и значение x
let x = [| 0 |]
let y = x;;

val x : int [] = [|0|]
val y : int [] = [|0|]

> // Если изменить значение x...
x.[0] <- 3;;
val it : unit = ()
> // ... x изменится ...
x;;
val it : int [] = [|3|]
> // ... но при этом изменится и значение y ...
y;;
val it : int [] = [|3|]
```

Изменение значений

Теперь, когда мы разобрались с основами того, как и где хранятся данные в платформе .NET, давайте рассмотрим, как эти данные можно из-

[1] Здесь автор выражается не совсем корректно. Более правильно говорить о том, что в программе может быть один объект ссылочного типа, на который ссылаются две или более ссылки. Тогда при изменении этого объекта через одну из ссылок вторая ссылка будет также ссылаться на измененный объект. – *Прим. науч. ред.*

менить. *Изменяемые значения (mutable values)* – это такие значения, которые можно изменять и которые объявляются с помощью ключевого слова `mutable`. Для изменения значения используется оператор `<-` «стрелка влево»:

```
> let mutable message = "World";;

val mutable message : string = "World"

> printfn "Hello, %s" message;;
Hello, World
val it : unit = ()

> message <- "Universe";;
val it : unit = ()
> printfn "Hello, %s" message;;
Hello, Universe
val it : unit = ()
```

Изменяемые значения имеют несколько ограничений, которые обусловлены ограничениями безопасности CLR (Common Language Runtime – общеязыковая среда исполнения). Эти ограничения препятствуют использованию изменяемых значений в некоторых ситуациях. В примере 4.3 выполняется попытка определить вложенную функцию `incrementX`, которая обращается к изменяемому значению `x` с помощью замыкания (то есть она имеет доступ к `x` несмотря на то, что это значение не передается ей в виде параметра). Это приводит к сообщению об ошибке, потому что изменяемые значения могут использоваться только внутри функций, в которых они были объявлены.

Пример 4.3. Ошибка при использовании изменяемого значения в замыкании

```
> // ОШИБКА: Изменяемые значения можно использовать только в функции,
//где они объявлены
let invalidUseOfMutable() =
    let mutable x = 0

    let incrementX() = x <- x + 1
    incrementX()

    x;;

    let incrementX() = x <- x + 1
----------------------^^^^^^^^^^^
stdin(16,24): error FS0407: The mutable variable 'x' is used in an invalid way.
Mutable variables may not be captured by closures. Consider eliminating this use
of mutation or using a heap-allocated mutable reference cell via 'ref' and '!'.
```

(Недопустимый способ использования изменяемой переменной "x". Изменяемые переменные нельзя зафиксировать с помощью замкнутых выражений. Рекомендуется исключить возможность такого изменения или использовать выделяемую в куче изменяемую ссылочную ячейку с помощью выражений "ref" и "!".)

Полный перечень ограничений, связанных с изменяемыми значениями:

- Изменяемые значения не могут возвращаться из функции (вместо этого возвращается копия).
- Изменяемые значения не могут использоваться вложенными функциями (замыканиями).

Если вы когда-либо столкнетесь с одной из этих проблем, самый простой способ решить ее состоит в том, чтобы сохранить изменяемые данные в «куче» с помощью ссылочной ячейки `ref`.

Ссылочные ячейки

Тип `ref`, который иногда называют *ссылочной ячейкой (ref cell)*, позволяет хранить изменяемые данные в «куче», что дает возможность обойти ограничения, связанные с изменяемыми значениями, хранящимися в стеке. Для получения значения типа `ref` используется символический оператор `!`, а для изменения – оператор `:=`.

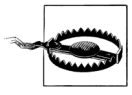
`ref` – это не только название типа, но также имя функции, которая создает значения типа `ref` и имеет сигнатуру:

```
val ref: 'a -> 'a ref
```

Функция `ref` принимает значение и возвращает его копию, заключенную в ссылочную ячейку `ref`. Пример 4.4 демонстрирует передачу списка планет функции `ref` и последующее изменение возвращенного значения.

Пример 4.4. Использование ссылочных ячеек для изменения данных

```
let planets =
    ref [
        "Mercury"; "Venus"; "Earth";
        "Mars"; "Jupiter"; "Saturn";
        "Uranus"; "Neptune"; "Pluto"
    ]

// Опа! Плутон уже не считается планетой...

// Оставить в списке планеты, кроме "Pluto"
// Получить значение planets типа ref cell с помощью оператора (!),
// затем присвоить ему новое значение с помощью оператора (:=)
planets := !planets |> List.filter (fun p -> p <> "Pluto")
```



Программисты на языке C# должны быть особенно внимательными при использовании данных типа `ref` и булевых значений. Запись `!x` означает получение значения `x`, а не логическое отрицание `x`:

```
> let x = ref true;;

val x : bool ref

> !x;;

val it : bool = true
```

В стандартной библиотеке языка F# имеется две функции, `decr` и `incr`, упрощающие увеличение и уменьшение данных типа `int ref`:

```
> let x = ref 0;;

val x : int ref = {contents = 0;}

> incr x;;
val it : unit = ()
> x;;
val it : int ref = {contents = 1;}
> decr x;;
val it : unit = ()
> x;;
val it : int ref = {contents = 0;}
```

Изменяемые записи

Изменяемыми могут быть не только простые значения – поля записей также могут быть помечены как изменяемые. Это позволяет использовать записи в императивном стиле. Чтобы сделать поле записи изменяемым, просто добавьте ключевое слово `mutable` перед его именем.

В следующем примере создается запись с изменяемым полем `Miles`, которое может модифицироваться, как обычная изменяемая переменная. Теперь у вас появилась возможность изменять поля записи без необходимости клонировать запись целиком:

```
> // Изменяемые записи
open System

type MutableCar = { Make : string; Model : string; mutable Miles : int }

let driveForASeason car =
    let rng = new Random()
    car.Miles <- car.Miles + rng.Next() % 10000;;

type MutableCar =
  {Make: string;
   Model: string;
```

```
   mutable Miles: int;}
val driveForASeason : MutableCar -> unit

> // Изменение поля записи
let kitt = { Make = "Pontiac"; Model = "Trans Am"; Miles = 0 }

driveForASeason kitt
driveForASeason kitt
driveForASeason kitt
driveForASeason kitt;;

val kitt : MutableCar = {Make = "Pontiac";
                         Model = "Trans Am";
                         Miles = 4660;}
```

Массивы

До сих пор, когда вам требовалось объединить несколько кусков данных, вы использовали списки. Списки чрезвычайно эффективны при добавлении и удалении данных в начале списка, но они не являются идеальным выбором во всех случаях. Например, случайный доступ к элементам списка выполняется очень медленно. Кроме того, чтобы изменить последний элемент списка, вам придется клонировать весь список целиком. (Подробнее о производительности списков будет рассказываться в главе 7.)

Если количество элементов коллекции известно заранее и вам необходимо иметь возможность изменять любые элементы коллекции, лучшим выбором будут массивы.

Массивы в .NET – это непрерывные блоки памяти, содержащие ноль или более элементов, каждый из которых может изменяться независимо от других (в отличие от списков, которые являются неизменяемыми).

Массивы можно создавать с помощью *генераторов массивов*, идентичных генераторам списков (которые рассматривались в главе 2), или вручную с помощью списков значений, разделенных точками с запятой и заключенных в скобки [| |]:

```
> // Использование синтаксиса генератора массивов
let perfectSquares = [| for i in 1 .. 7 -> i * i |];;

val perfectSquares : int [] = [|1; 4; 9; 16; 25; 36; 49|]

> perfectSquares;;
val it : int []= [|1; 4; 9; 16; 25; 36; 49|]

> // Объявление вручную
let perfectSquares2 = [| 1; 4; 9; 16; 25; 36; 49; 64; 81 |];;

val perfectSquares2 : int [] = [|1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

Обращение к элементам массива

Для получения элемента массива необходимо использовать оператор индексирования .[], где в квадратных скобках указывается индекс элемента (счет элементов начинается с нуля):

```
> // Обращение к элементам массива
printfn
    "The first three perfect squares are %d, %d, and %d"
    perfectSquares.[0]
    perfectSquares.[1]
    perfectSquares.[2];;
The first three perfect squares are 1, 4, and 9
val it : unit = ()
```

В примере 4.5 показан примитивный алгоритм шифрования ROT13, в котором с помощью оператора индексирования выполняется изменение отдельных элементов массива символов. Суть алгоритма ROT13 заключается в том, что он просто извлекает каждый символ и замещает его символом, отстоящим в алфавите на 13 позиций дальше. Для этого в примере каждый символ преобразуется в целое число, к этому числу добавляется 13, а затем получившееся значение преобразуется обратно в символ.

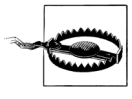
Пример 4.5. Реализация алгоритма шифрования ROT13 на языке F#

```
open System

// Шифрование символов по алгоритму ROT13
let rot13Encrypt (letter : char) =

    // Если это буква, взять символ, отстоящий от текущего на 13 позиций
    // дальше (с переходом к началу в случае необходимости).
    // Остальные символы игнорировать.
    if Char.IsLetter(letter) then
        let newLetter =
            (int letter)
            |> (fun letterIdx -> letterIdx - (int 'A'))
            |> (fun letterIdx -> (letterIdx + 13) % 26)
            |> (fun letterIdx -> letterIdx + (int 'A'))
            |> char
        newLetter
    else
        letter

// Обойти все элементы массива, шифруя каждую букву
let encryptText (text : char[]) =

    for idx = 0 to text.Length - 1 do
        let letter = text.[idx]
        text.[idx] <- rot13Encrypt letter
```

```
let text =
    Array.ofSeq "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG"

printfn "Origional = %s" <| new String(text)
encryptText(text)
printfn "Encrypted = %s" <| new String(text)

// Уникальная особенность алгоритма ROT13 состоит в том, что для
// расшифровывания достаточно повторно выполнить шифрование
encryptText(text)
printfn "Decrypted = %s" <| new String(text)
```

Результат выполнения этой простой программы следующий:

```
Origional = THE QUICK FOX JUMPED OVER THE LAZY DOG
Encrypted = GUR DHVPX SBK WHZCRQ BIRE GUR YNML QBT
Decrypted = THE QUICK FOX JUMPED OVER THE LAZY DOG
```



В отличие от языков программирования C# и VB.NET, операторы индексирования в языке F# требуют использования точечной нотации. Вы можете представлять себе индексаторы как некоторую разновидность методов или свойств:

```
// Неправильно
x[0]
// Правильно
x.[0]
```

При попытке получить доступ к элементу массива с отрицательным индексом или с индексом, большим или равным количеству элементов массива, будет сгенерировано исключение `IndexOutOfRangeException`. (Исключения рассматриваются ниже в этой главе.) К счастью, массивы имеют свойство `Length`, которое возвращает количество элементов в массиве. Так как индексирование элементов массива начинается с нуля, то для получения индекса последнего элемента значение `Length` необходимо уменьшить на 1:

```
> let alphabet = [| 'a' .. 'z' |];;

val alphabet : char [] =
  [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'; 'j'; 'k'; 'l'; 'm'; 'n'; 'o';
    'p'; 'q'; 'r'; 's'; 't'; 'u'; 'v'; 'w'; 'x'; 'y'; 'z'|]

> // Первый символ
alphabet.[0];;
val it : char = 'a'

> // Последний символ
alphabet.[alphabet.Length - 1];;
val it : char = 'z'
```

```
> // Некоторый несуществующий символ
alphabet.[10000];;
System.IndexOutOfRangeException: Index was outside the bounds of the array.
  at <StartupCode$FSI_0012>.$FSI_0012._main()
stopped due to error
```

(Индекс находился вне границ массива. Остановлено из-за ошибки.)

Срезы массивов

При анализе данных, хранящихся в массивах, иногда бывает удобнее работать только с некоторым подмножеством данных. В языке F# предусмотрен специальный синтаксис получения *срезов* (slices) массива, где от вас требуется только указать необязательные нижнюю и верхнюю границы подмножества данных. Конструкция извлечения среза имеет следующий вид:

```
массив.[нижняяГраница..верхняяГраница]
```

Если нижняя граница не указана, вместо нее используется `0`. Если не указана верхняя граница, используется длина массива `-1`. Если ни верхняя, ни нижняя границы не указаны, то можно указать `*`, чтобы создать копию всего массива.

В примере 4.6 демонстрируются различные способы получения срезов массива, но чуть позже мы рассмотрим его более подробно.

Пример 4.6. Получение среза массива

```
open System
let daysOfWeek = Enum.GetNames( typeof<DayOfWeek> )

// Стандартный способ получения среза, элементы со 2 по 4
daysOfWeek.[2..4]

// Указана только нижняя граница, элементы с 4 по последний
daysOfWeek.[4..]

// Указана только верхняя граница, элементы с 0 по 2
daysOfWeek.[..2]

// Границы не указаны, возвращаются все элементы (копия всего массива)
daysOfWeek.[*]
```

В первой операции получения среза были указаны обе границы, верхняя и нижняя. В результате были получены элементы массива внутри указанного диапазона:

```
> // Стандартный способ извлечения среза, элементы со 2 по 4
daysOfWeek.[2..4];;
val it : string [] = [|"Tuesday"; "Wednesday"; "Thursday"|]
```

Затем мы указали только нижнюю границу, а потом – только верхнюю. Эти операции вернули все элементы от нижней границы до конца массива, а затем от начала массива до указанной верхней границы:

```
> // Указана только нижняя граница, элементы с 4 по последний
daysOfWeek.[4..];;
val it : string [] = [|"Thursday"; "Friday"; "Saturday"|]

> // Указана только верхняя граница, элементы с 0 по 2
daysOfWeek.[..2];;
val it : string [] = [|"Sunday"; "Monday"; "Tuesday"|]
```

И наконец, мы скопировали массив целиком, указав `*`. Обратите внимание, что все операции получения срезов возвращают новый массив, поэтому при их использовании никогда не возникают проблемы, связанные с совмещением имен:

```
> // Границы не указаны, возвращаются все элементы (копия всего массива)
daysOfWeek.[*];;
```

Создание массивов

Генераторы массивов и явное указание всех элементов не являются единственными способами создания массивов. Вы можете также использовать функцию `Array.init`, которая принимает функцию, используемую для создания элементов массива по их индексам. Создать неинициализированный массив можно с помощью функции `Array.zeroCreate`. В этом случае элементы массива инициализируются значениями по умолчанию: ноль или `null`.

Пример 4.7 демонстрирует использование функций `Array.init` и `Array.zeroCreate` для создания массивов.

Пример 4.7. Инициализация массивов с помощью Array.init

```
> // Инициализация массива значениями синусоиды
let divisions = 4.0
let twoPi = 2.0 * Math.PI;;

val divisions : float = 4.0
val twoPi : float = 6.283185307

> Array.init (int divisions) (fun i -> float i * twoPi / divisions);;
val it : float []= [|0.0; 1.570796327; 3.141592654; 4.71238898|]
> // Конструирование пустых массивов
let emptyIntArray : int []= Array.zeroCreate 3
let emptyStringArray : string []= Array.zeroCreate 3;;

val emptyIntArray : int array = [|0; 0; 0|]
val emptyStringArray : string array = [|null; null; null|]
```

CLR ограничивает размеры массивов двумя гигабайтами даже на 64-битных платформах. Поэтому если вам потребуется создать массив для хранения огромного объема данных, используйте для этого собственные структуры данных.

Сопоставление с образцом

Сопоставление с массивами выполняется так же просто, как и сопоставление со списками. И точно так же, как и в случае со списками, при сопоставлении с массивами вы можете захватывать значения элементов или находить сопоставление с массивом целиком. Пример 4.8 демонстрирует сопоставление массива со значением `null`, а также с массивами, имеющими 0, 1 или 2 элемента.

Пример 4.8. Использование массивов с сопоставлением с образцом

```
> // Получить описание массива
let describeArray arr =
    match arr with
    | null       -> “The array is null”
    | [| |]      -> “The array is empty”
    | [| x |]    -> sprintf “The array has one element, %A” x
    | [| x; y |] -> sprintf “The array has two elements, %A and %A” x y
    | a          -> sprintf “The array had %d elements, %A” a.Length a;;

val describeArray : ‘a array -> string

> describeArray [| 1 .. 4 |];;
val it : string = “The array had 4 elements, [|1; 2; 3; 4|]”
> describeArray [| (“tuple”, 1, 2, 3) |];;
val it : string = “The array has one element, (“tuple”, 1, 2, 3)”
```

Эквивалентность массивов

В языке F# массивы сравниваются с учетом структурной эквивалентности. Два массива считаются равными, если они имеют одинаковую размерность, длину и элементы. (Понятие размерности массива мы будем рассматривать в следующем разделе.)

```
> [| 1 .. 5 |] = [| 1; 2; 3; 4; 5 |];;
val it : bool = true
> [| 1 .. 3 |] = [| |];;
val it : bool = false
```

Понятие эквивалентности массивов в языке F# отличается от аналогичного понятия в C#. В языке F# оператор = наделен специальной логикой, выполняющей сравнение массивов, поэтому для массивов не используется сравнение ссылок. Дополнительные сведения о понятии равенства объектов в .NET приводятся в главе 5.

Функции модуля Array

Подобно модулям List и Seq, существует также модуль Array, содержащий практически идентичные функции для выполнения операций над массивами, перечисленные в табл. 4.1.

Таблица 4.1. Часто используемые методы модуля Array

Функция	Описание
Array.length 'a[] -> int	Возвращает длину массива. Массивы уже имеют свойство Length, но эту функцию удобно использовать для композиции функций.
Array.init int -> (int -> 'a) -> 'a[]	Создает новый массив с указанным числом элементов – каждый элемент инициализируется значением, возвращаемым указанной функцией.
Array.zeroCreate int -> 'a[]	Создает массив указанной длины, где каждый элемент массива получает значение по умолчанию
Array.exists ('a -> bool) -> 'a[] -> bool	Проверяет наличие элемента массива, удовлетворяющего заданной функции.

Array.partition

Функция Array.partition делит указанный массив на два новых массива. Первый массив получает элементы, для которых указанная функция возвратит true, а второй массив – элементы, для которых указанная функция возвратит false:

```
> // Простая булевая функция
let isGreaterThanTen x =
    if x > 10
    then true
    else false;;

val isGreaterThanTen : int -> bool

> // Деление массива
[| 5; 5; 6; 20; 1; 3; 7; 11 |]
|> Array.partition isGreaterThanTen;;
val it : int [] * int []= ([|20; 11|], [|5; 5; 6; 1; 3; 7|])
```

Array.tryFind и Array.tryFindIndex

Функция Array.tryFind возвращает значение Some первого элемента, для которого указанная функция вернет true. В противном случае она вернет None. Функция Array.tryFindIndex действует точно так же, как Array.tryFind, только вместо самого элемента массива она возвращает его индекс в массиве:

```
> // Простая булевая функция
let rec isPowerOfTwo x =
    if x = 2 then
        true
    elif x % 2 = 1 (* is odd *) then
        false
    else isPowerOfTwo (x / 2);;

val isPowerOfTwo : int -> bool

> [| 1; 7; 13; 64; 32 |]
|> Array.tryFind isPowerOfTwo;;
val it : int option = Some 64
> [| 1; 7; 13; 64; 32 |]
|> Array.tryFindIndex isPowerOfTwo;;
val it : int option = Some 3
```

Функции `Array.tryFind` и `Array.tryFindIndex` наглядно показывают удобство использования типа `Option`. В языке C# функции, похожие на `tryFindIndex`, могли бы возвращать значение `-1` как признак отсутствия искомого элемента (в противоположность значению `None`). Однако для функции `tryFind` значение `-1` может интерпретироваться не только как признак отсутствия искомого элемента, но и как элемент со значением `-1`.

Агрегатные операторы

Модуль `Array` содержит также агрегатные операторы, аналогичные тем, что имеются в модуле `List`, а именно: `fold`, `foldBack`, `map` и `iter`. Кроме того, существуют также версии этих методов, содержащие индексы. В примере 4.9 показано использование функции `iteri`, которая ведет себя точно так же, как и функция `iter`, за исключением того, что вместе с самим элементом массива она передает его индекс.

Пример 4.9. Использование агрегатной функции Array.iteri

```
> let vowels = [| 'a'; 'e'; 'i'; 'o'; 'u' |];;

val vowels : char [] = [|'a'; 'e'; 'i'; 'o'; 'u'|]

> Array.iteri (fun idx chr -> printfn "vowel.[%d] = %c" idx chr) vowels
vowel.[0] = a
vowel.[1] = e
vowel.[2] = i
vowel.[3] = o
vowel.[4] = u
val it : unit = ()
```

Многомерные массивы

Массивы удобно использовать для хранения данных в линейном виде, но как быть, если потребуется представить данные с размерностью два, три или больше? Для этих целей можно создавать *многомерные массивы*, которые позволяют рассматривать данные как блоки, индексируемые несколькими значениями.

Многомерные массивы имеют две разновидности: прямоугольные и «невыровненные» (jagged). Различия между ними показаны на рис. 4.2. Прямоугольные массивы представляют собой непрерывные блоки данных, тогда как «невыровненные» массивы фактически являются массивами массивов.

Прямоугольный массив

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

«Невыровненный» массив

[0].[0]	[0].[1]	[0].[2]	
[1].[0]	[1].[1]		
[2].[0]	[2].[1]	[2].[2]	[2].[3]

Рис. 4.2. *Прямоугольный и «невыровненный» массивы*

Какой тип многомерного массива использовать, зависит от ситуации. «Невыровненные» массивы позволяют экономить память, если строки массива не обязательно должны иметь одну и ту же длину. Однако прямоугольные массивы обеспечивают более высокую эффективность случайного доступа к элементам, потому что в памяти элементы хранятся в виде непрерывного одномерного блока. (И поэтому получают дополнительные преимущества от использования кэша процессора.)

Прямоугольные массивы

Прямоугольные массивы – это прямоугольные блоки *n* на *m* элементов. Прямоугольные массивы обозначаются прямоугольными скобками, а каждое новое измерение отделяется запятой. Кроме того, по образу и подобию одномерных массивов существуют модули `Array2D` и `Array3D`, позволяющие создавать и инициализировать прямоугольные массивы:

```
> // Создание массива 3x3
let identityMatrix : float[,] = Array2D.zeroCreate 3 3
```

```
identityMatrix.[0,0] <- 1.0
identityMatrix.[1,1] <- 1.0
identityMatrix.[2,2] <- 1.0;;

val identityMatrix : float [,] = [[1.0; 0.0; 0.0]
                                  [0.0; 1.0; 0.0]
                                  [0.0; 0.0; 1.0]]
```

Двумерные прямоугольные массивы также позволяют получать срезы с помощью аналогичного синтаксиса путем указания диапазона для каждого измерения отдельно:

```
> // Все строки для столбцов с индексами 1 и 2
identityMatrix.[*, 1..2];;

val it : float [,] = [[0.0; 0.0]
                      [1.0; 0.0]
                      [0.0; 1.0]]
```

Невыровненные массивы

Невыровненные массивы – это просто одномерные массивы одномерных массивов. Каждая строка в основном массиве – это просто другой массив, каждый из которых должен инициализироваться отдельно:

```
> // Создание невыровненного массива
let jaggedArray : int[][] = Array.zeroCreate 3
jaggedArray.[0] <- Array.init 1 (fun x -> x)
jaggedArray.[1] <- Array.init 2 (fun x -> x)
jaggedArray.[2] <- Array.init 3 (fun x -> x);;
val jaggedArray : int [] [] = [|[|0|]; [|0; 1|]; [|0; 1; 2|]|]
```

Типы изменяемых коллекций

Часто бывает необходимо хранить данные, но заранее неизвестно необходимое количество элементов. Например, вам может потребоваться загрузить записи из базы данных. При использовании массивов вы рискуете впустую потратить память, выделенную для слишком большого количества элементов, или, что еще хуже, данных может оказаться больше, чем было выделено памяти для них.

Изменяемые коллекции позволяют динамически добавлять и удалять элементы с течением времени. Изменяемые коллекции могут вызывать проблемы при параллельном и асинхронном программировании, однако в других случаях они очень просты в использовании.

List<'T>

Тип `List` в пространстве имен `System.Collections.Generic` (не путайте с типом `list` в языке F#) – это обертка вокруг массива, которая позволяет

динамически изменять размер коллекции при добавлении и удалении элементов. Поскольку тип `List` построен на основе стандартных массивов, операции извлечения и изменения значений обычно выполняются очень быстро.[1] Но как только внутренний массив будет заполнен, необходимо создать массив большего размера и скопировать содержимое в новый массив.

Пример 4.10 демонстрирует основные приемы использования типа `List`. Здесь мы снова перестаем считать Плутон (Pluto) полноценной планетой.

Пример 4.10. Использование типа List

```
> // Создается список планет типа List<_>
open System.Collections.Generic
let planets = new List<string>();;

val planets : Collections.Generic.List<string>

> // Добавить несколько названий планет по отдельности

planets.Add("Mercury")
planets.Add("Venus")
planets.Add("Earth")
planets.Add("Mars");;
> planets.Count;;

val it : int = 4
> // Добавить сразу несколько значений
planets.AddRange( [| "Jupiter"; "Saturn"; "Uranus"; "Neptune"; "Pluto" |] );;
val it : unit = ()
> planets.Count;;
val it : int = 9
> // Прошу прощения, ребята
planets.Remove("Pluto");;
val it : bool = true
> planets.Count;;
val it : int = 8
```

В табл. 4.2 перечислены наиболее часто используемые методы типа `List`.

[1] Здесь автор не совсем прав. Извлечение элементов и запись элементов действительно выполняются быстро. Но вот добавление новых элементов (при котором количество элементов коллекции изменяется) может выполняться крайне медленно, поскольку может потребовать выделения большого куска данных и копирования существующих элементов. – *Прим. науч. ред.*

Таблица 4.2. Методы и свойства типа List<'T>

Функция и ее тип	Описание
`Add` `'a -> unit`	Добавляет элемент в конец списка.
`Clear` `unit -> unit`	Удаляет все элементы из списка.
`Contains` `'a -> bool`	Возвращает признак наличия в списке указанного элемента.
`Count` `int`	Свойство, которое возвращает количество элементов в списке.
`IndexOf` `'a -> int`	Возвращает индекс указанного элемента (отсчет индексов начинается с нуля). Если искомый элемент отсутствует, возвращает `-1`.
`Insert` `int * 'a -> unit`	Вставляет указанное значение в список, в позицию с указанным индексом.
`Remove` `'a -> bool`	Удаляет указанный элемент, если он присутствует в списке.
`RemoveAt` `int -> unit`	Удаляет элемент с указанным индексом.

Dictionary<'K,'V>

Тип `Dictionary` (словарь) находится в пространстве имен `System.Collections.Generic`, которое представляет собой структуру данных, содержащих пары ключ/значение. Словари обычно используются, когда необходимо хранить данные и требуется обеспечить дружественный способ их поиска, например найти номер телефона по имени абонента, а не по индексу.

В примере 4.11 демонстрируется использование словаря для отображения символических имен на данные типа `Atom`. Здесь используется такая особенность языка F#, как *единицы измерения (units of measure)*, для обозначения атомной массы каждого химического элемента в единицах измерения атомной массы. Подробно единицы измерения рассматриваются в главе 7.

Пример 4.11. Использование словаря

```
// Единицы измерения атомной массы
[<Measure>]
type amu

type Atom = { Name : string; Weight : float<amu> }
```

```
open System.Collections.Generic
let periodicTable = new Dictionary<string, Atom>()
periodicTable.Add( "H", { Name = "Hydrogen"; Weight = 1.0079<amu> })
periodicTable.Add("He", { Name = "Helium"; Weight = 4.0026<amu> })
periodicTable.Add("Li", { Name = "Lithium"; Weight = 6.9410<amu> })
periodicTable.Add("Be", { Name = "Beryllium"; Weight = 9.0122<amu> })
periodicTable.Add( "B", { Name = "Boron "; Weight = 10.811<amu> })
// ...

// Поиск элемента
let printElement name =

    if periodicTable.ContainsKey(name) then
        let atom = periodicTable.[name]
        printfn
            "Atom with symbol with '%s' has weight %A."
            atom.Name atom.Weight
    else
        printfn "Error. No atom with name '%s' found." name

// Альтернативный способ получения значений. Возвращает кортеж
// 'success * result'
let printElement2 name =

    let (found, atom) = periodicTable.TryGetValue(name)
    if found then
        printfn
            "Atom with symbol with '%s' has weight %A."
            atom.Name atom.Weight
    else
        printfn "Error. No atom with name '%s' found." Name
```

В табл. 4.3 перечислены наиболее часто используемые методы типа Dictionary<'k,'v>.

Таблица 4.3. Методы и свойства типа Dictionary<'k,'v>

Функция и ее тип	Описание
Add 'k * 'v -> unit	Добавляет в словарь новую пару ключ/значение.
Clear unit -> unit	Удаляет все элементы из словаря.
ContainsKey 'k -> bool	Проверяет наличие указанного ключа в словаре.
ContainsValue 'v -> bool	Проверяет наличие указанного значения в словаре.

Таблица 4.3 (продолжение)

Функция и ее тип	Описание
`Count` `int`	Возвращает количество элементов в словаре.
`Remove` `'k -> bool`	Удаляет из словаря пару ключ/значение с указанным ключом.

HashSet<'T>

Тип `HashSet`, который также определен в пространстве имен `System.Collections.Generic`, обеспечивает эффективный способ хранения неупорядоченного набора элементов. Представьте, что вы создаете приложение, которое будет просматривать содержимое веб-страниц. Вам придется каким-то образом запоминать, какие страницы уже были просмотрены, чтобы не застрять в бесконечном цикле. Однако если сохранять адреса посещенных страниц в списке типа `List`, проверка наличия очередной страницы в списке посещенных ранее будет выполняться слишком медленно. Тип `HashSet` хранит коллекцию уникальных значений на основе *хеш-кодов*, благодаря чему проверка наличия элемента в коллекции выполняется намного быстрее.

Подробнее о хеш-кодах рассказывается в следующей главе, а пока считайте их одним из способов идентификации объектов. Именно они обеспечивают высокую скорость поиска элементов в коллекциях типа `HashSet` и `Dictionary`.

В примере 4.12 показано использование коллекции `HashSet` для определения того, была ли кинокартина удостоена премии Оскар.

Пример 4.12. Использование типа HashSet

```
open System.Collections.Generic

let bestPicture = new HashSet<string>()
bestPicture.Add("No Country for Old Men")
bestPicture.Add("The Departed")
bestPicture.Add("Crash")
bestPicture.Add("Million Dollar Baby")
// ...

// Проверить, была ли удостоена премии Оскар следующая кинокартина
if bestPicture.Contains("Manos: The Hands of Fate") then
    printfn "Sweet..."
    // ...
```

В табл. 4.4 перечислены наиболее часто используемые методы типа `HashSet<'T>`.

Таблица 4.4. Методы и свойства типа HashSet<'T>

Функция и ее тип	Описание
`Add` `'a -> unit`	Добавляет новый элемент в коллекцию типа `HashSet`.
`Clear` `unit -> unit`	Удаляет все элементы из коллекции типа `HashSet`.
`Count` `int`	Возвращает количество элементов в коллекции типа `HashSet`.
`IntersectWith` `seq<'a> -> unit`	Изменяет коллекцию типа `HashSet` так, чтобы она содержала только те элементы, которые также присутствуют в указанной последовательности.
`IsSubsetOf` `seq<'a > -> bool`	Определяет, является ли коллекция типа `HashSet` подмножеством указанной последовательности. То есть последовательность содержит все элементы, имеющиеся в коллекции типа `HashSet`.
`IsSupersetOf` `seq<'a > -> bool`	Определяет, является ли коллекция типа `HashSet` надмножеством указанной последовательности. То есть коллекция типа `HashSet` содержит все элементы, имеющиеся в последовательности.
`Remove` `'a -> bool`	Удаляет указанный элемент из коллекции.
`UnionWith` `seq<'a> -> unit`	Изменяет коллекцию типа `HashSet` так, чтобы она содержала все элементы, которые имеются в данной коллекции и в указанной последовательности.

Циклы

В языке F# имеются традиционные инструкции циклов, которые можно встретить во многих императивных языках. Они позволяют вызывать функцию или выполнять фрагмент кода определенное число раз или пока не будет выполнено некоторое условие.

Циклы while

Выражения `while` продолжают выполняться до тех пор, пока булево выражение не вернет значение `false`. (По этой причине циклы `while` не могут применяться в чисто функциональном стиле, так как цикл никогда не завершится, потому что вы никогда не сможете изменить возвращаемое значение предиката.)

Обратите внимание, что условное выражение в цикле `while` должно возвращать значение типа `bool`, а тело цикла – значение `unit`. Следующий пример показывает, как можно использовать цикл `while` для перебора нескольких значений:

```
> // Цикл while
let mutable i = 0
while i < 5 do
    i <- i + 1
    printfn "i = %d" i;;
i = 1
i = 2
i = 3
i = 4
i = 5

val mutable i : int = 5
```

Условие цикла `while` проверяется еще до выполнения тела цикла, поэтому если при начальных условиях выражение вернет `false`, тело цикла не будет выполнено ни разу.

Циклы for

Когда необходимо выполнить фиксированное число итераций, можно использовать выражение `for`, которое в языке F# имеют две разновидности: простые циклы и перечисления.

Простые циклы for

Простые циклы `for` создают новое целочисленное значение и последовательно увеличивают его до определенного значения. Цикл не будет выполняться, если значение счетчика окажется больше максимального значения:

```
> // Цикл for
for i = 1 to 5 do
    printfn "%d" i;;
1
2
3
4
5
val it : unit = ()
```

Чтобы организовать счет в обратном порядке, следует использовать ключевое слово `downto`. В этом случае цикл не будет выполняться, если значение счетчика окажется меньше минимального значения:

```
> // Счет выполняется в обратном порядке
for i = 5 downto 1 do
    printfn "%d" i;;
5
4
3
2
1
val it : unit = ()
```

Простые счетные циклы for поддерживают только целочисленные счетчики, поэтому если требуемое количество итераций больше System.Int32.MaxValue, используйте циклы-перечисления for.

Циклы-перечисления for

Более обобщенная разновидность циклов for выполняет итерации по элементам последовательности. Циклы-перечисления for могут работать с любыми типами seq, такими как array или list:

```
> // Цикл-перечисление
for i in [1 .. 5] do
    printfn "%d" i;;
1
2
3
4
5
val it : unit = ()
```

Циклы-перечисления for обладают более широкими возможностями, чем циклы foreach в языке C#, благодаря тому, что они могут использовать сопоставление с образцом. Элемент, который следует за ключевым словом for, в действительности является правилом сопоставления с образцом, поэтому если это новый идентификатор, такой как i, он просто захватывает значение сопоставления. Однако имеется возможность использовать более сложные правила.

В примере 4.13 показано использование объединения, имеющего всего два варианта, Cat и Dog. Цикл for выполняет обход списка, но тело цикла выполняется, только когда элемент списка является Dog.

Пример 4.13. Циклы for с сопоставлением с образцом

```
> // Тип животного
type Pet =
    | Cat of string * int // Кличка, Возраст
    | Dog of string       // Кличка
;;

type Pet =
    | Cat of string * int
    | Dog of string

> let famousPets = [ Dog("Lassie"); Cat("Felix", 9); Dog("Rin Tin Tin") ];;

val famousPets : Pet list = [Dog "Lassie"; Cat ("Felix",9); Dog "Rin Tin Tin"]

> // Вывести клички собак
for Dog(name) in famousPets do
    printfn "%s was a famous dog." name;;
Lassie was a famous dog.
```

```
Rin Tin Tin was a famous dog.
val it : unit = ()
```

В языке F# нет ключевых слов `break` и `continue`. Если вам потребуется преждевременный выход из цикла, создайте изменяемое значение и используйте цикл `while` вместо `for`.

Исключения

Каждый программист знает, что не всегда все идет так, как планировалось. До сих пор я проявлял особую осторожность в примерах, старясь не показывать вам код, который терпит неудачу. Однако готовность к неожиданным событиям является важным аспектом любой программы. К счастью, платформа .NET поддерживает мощный механизм для обработки неожиданностей.

Исключение – это сбой в программе на платформе .NET, который приводит к отклонению программы от обычного потока исполнения. При возникновении исключения происходит немедленный выход из текущей функции и из вызвавшей ее функции и так далее, пока исключение не будет *перехвачено обработчиком исключений*. Если исключение не будет перехвачено программой, она завершит свою работу.

Самый простой способ сообщить об ошибке в программе на языке F# заключается в использовании функции `failwith`. Эта функция принимает строку в виде параметра и генерирует исключение `System.Exception`. Альтернативная версия, функция `failwithf`, принимает строку формата, подобно функциям `printf` и `sprintf`:

```
> // Использование функции failwithf
let divide x y =
    if y = 0 then failwithf "Cannot divide %d by zero!" x
    x / y;;

val divide : int -> int -> int

> divide 10 0;;
System.Exception: Cannot divide 10 by zero!
  at FSI_0003.divide(Int32 x, Int32 y)
  at <StartupCode$FSI_0004>.$FSI_0004._main()
stopped due to error
```

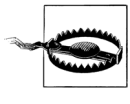
В предыдущем примере окно FSI информирует о возникшем исключении и выводит значения двух наиболее важных свойств исключения: текст сообщения и трассировку стека. Каждое исключение обладает свойством `Message`, которое содержит описание возникшей проблемы. Свойство `Stacktrace` – это строка со списком всех функций, ожидавших завершения текущей функции перед тем, как возникло исключение; она содержит информацию, которая может пригодиться при поиске

причин, вызвавших исключение. Поскольку *раскручивание* стека вызовов начинается сразу же после возникновения исключения, это исключение может быть перехвачено далеко от места, где оно возникло.

Хотя описание может помочь программисту в отладке, тем не менее более оптимальным в .NET считается использование исключений определенного типа. Чтобы генерировать конкретный тип исключения, необходимо воспользоваться функцией `raise`. Она принимает тип исключения (любой тип, наследующий `System.Exception`) и генерирует исключения этого типа, как это делает функция `failwith`:

```
> // Генерация исключения DivideByZeroException
let divide2 x y =
    if y = 0 then raise <| new System.DivideByZeroException()
    x / y;;

val divide2 : int -> int -> int

> divide2 10 0;;

System.DivideByZeroException: Attempted to divide by zero.
  at FSI_0005.divide2(Int32 x, Int32 y)
  at <StartupCode$FSI_0007>.$FSI_0007._main()
stopped due to error
```

(Попытка деления на нуль. Остановлено из-за ошибки)



Весьма заманчивой выглядит возможность генерировать исключения всякий раз, когда программа переходит в непредусмотренное состояние. Однако генерация исключений может существенно снизить производительность программы. Любые ситуации, которые могут привести к генерации исключения, должны устраняться по мере возможности.

Обработка исключений

Чтобы обработать исключение, его следует перехватить с помощью выражения `try-catch`. Любые исключения, которые могут быть сгенерированы в пределах выражения `try-catch`, будут обработаны блоком `with`, который фактически является выражением сопоставления с образцом по типу исключений.

Выбор обработчика исключений производится путем сопоставления с образцом, поэтому вы можете объединять обработчики по «ИЛИ» или использовать групповой символ для перехвата исключений любых типов. Если исключение будет возбуждено внутри выражения `try-catch`, но соответствующий ему обработчик не будет найден, это исключение продолжит подниматься вверх по стеку вызовов, пока не будет перехвачено или не приведет к завершению программы.

Выражения try-catch точно так же возвращают значение, как выражения сопоставления с образцом или условное выражение. Поэтому последнее выражение в блоке try должно иметь тот же тип, что и все правила в выражении with.

В примере 4.14 приводится код, который движется по минному полю потенциальных проблем. Для каждого возможного исключения предусмотрен соответствующий обработчик. В примере используется оператор динамической проверки типа :?, с помощью которого выполняется сопоставление с типом исключения; подробнее этот оператор будет рассматриваться в следующей главе.

Пример 4.14. Выражения try-catch

```
open System.IO

[<EntryPoint>]
let main (args : string[]) =

    let exitCode =
        try
            let filePath = args.[0]

            printfn "Trying to gather information about file:"
            printfn "%s" filePath

            // Существует ли устройство?
            let matchingDrive =
                Directory.GetLogicalDrives()
                |> Array.tryFind (fun drivePath -> drivePath.[0] = filePath.[0])

            if matchingDrive = None then
                raise <| new DriveNotFoundException(filePath)

            // Указанная папка существует?
            let directory = Path.GetPathRoot(filePath)
            if not <| Directory.Exists(directory) then
                raise <| new DirectoryNotFoundException(filePath)

            // Файл существует?
            if not <| File.Exists(filePath) then
                raise <| new FileNotFoundException(filePath)

            let fileInfo = new FileInfo(filePath)
            printfn "Created = %s" <| fileInfo.CreationTime.ToString()
            printfn "Access  = %s" <| fileInfo.LastAccessTime.ToString()
            printfn "Size    = %d" fileInfo.Length

            0
```

```
        with
        // Объединение образцов по "ИЛИ"
        | :? DriveNotFoundException
        | :? DirectoryNotFoundException
            -> printfn "Unhandled Drive or Directory not found exception"
               1
        | :? FileNotFoundException as ex
            -> printfn "Unhandled FileNotFoundException: %s" ex.Message
               3
        | :? IOException as ex
            -> printfn "Unhandled IOException: %s" ex.Message
               4
        // Групповой символ (результат будет иметь тип System.Exception)
        | _ as ex
            -> printfn "Unhandled Exception: %s" ex.Message
               5

    // Вернуть код завершения
    printfn "Exiting with code %d" exitCode
    exitCode
```

Поскольку неперехваченные исключения могут оставлять неосвобожденными неуправляемые ресурсы, такие как дескрипторы файлов или буферы ввода/вывода, существует еще один способ обработки исключений: выражения `try-finally`. Блок `finally` в выражении `try-finally` выполняется независимо от того, было ли сгенерировано исключение или нет, давая возможность выполнить необходимые операции по освобождению ресурсов.

В примере 4.15 демонстрируется использование выражения `try-finally`.

Пример 4.15. Выражения try-finally

```
> // Выражения try-finally
let tryFinallyTest() =
    try
        printfn "Before exception..."
        failwith "ERROR!"
        printfn "After exception raised..."
    finally
        printfn "Finally block executing..."

let test() =
    try
        tryFinallyTest()
    with
    | ex -> printfn "Exception caught with message: %s" ex.Message;;

val tryFinallyTest : unit -> unit
val test : unit -> unit
```

```
> test();;
Before exception...
Finally block executing...
Exception caught with message: ERROR!
val it : unit = ()
```

В отличие от языка C#, в F# нет выражения `try-catch-finally`. Если вам нужно освободить ресурсы и обработать исключения, необходимо сделать это в каждом блоке обработчика или сразу после блока `try-catch`.

Повторная генерация исключений

Иногда, несмотря на все усилия по исправлению ошибки, бывает невозможно устранить проблему. В таких ситуациях вы можете повторно сгенерировать исключение, что позволит ему продолжить подъем вверх по стеку вызовов.

Пример 4.16 показывает возможность повторной генерации исключения с помощью функции `reraise`.

Пример 4.16. Повторная генерация исключений

```
open Checked

let reraiseExceptionTest() =
    try
        let x = 0x0fffffff
        let y = 0x0fffffff

        x * y
    with
    | :? System.OverflowException as ex
        -> printfn "An overflow exception occured..."
           reraise()
```

Определение новых типов исключений

Генерация специализированных исключений позволяет пользователям вашего кода перехватывать только те исключения, которые они знают, как обработать. Остальные исключения будут продолжать подниматься до соответствующего обработчика исключений.

Вы можете определять свои собственные исключения, создавая типы, наследующие `System.Exception`, что будет рассмотрено подробнее в главе 5. Однако в языке F# существует более простой способ определения собственных исключений с использованием упрощенного синтаксиса. В этом случае исключения объявляются точно так же, как размеченные объединения.

В примере 4.17 демонстрируется создание нескольких новых типов исключений, причем некоторые из них ассоциированы с дополнительны-

ми данными. Преимущество такого упрощенного способа определения исключений состоит в том, что при обработке таких исключений из них проще извлечь соответствующие данные, потому что для этого вы можете использовать знакомый синтаксис выражений сопоставления с размеченными объединениями. Кроме того, отпадает необходимость в использовании оператора динамического определения типа :?, который мы видели в предыдущих примерах.

Пример 4.17. Упрощенный синтаксис определения исключений в языке F#

```
open System
open System.Collections.Generic

exception NoMagicWand
exception NoFullMoon of int * int
exception BadMojo of string

let castHex (ingredients : HashSet<string>) =
    try

        let currentWand = Environment.MagicWand

        if currentWand = null then
            raise NoMagicWand

        if not <| ingredients.Contains("Toad Wart") then
            raise <| BadMojo("Need Toad Wart to cast the hex!")

        if not <| isFullMoon(DateTime.Today) then
            raise <| NoFullMoon(DateTime.Today.Month, DateTime.Today.Day)

        // Начинаем колдовство...
        let mana =
            ingredients
            |> Seq.map (fun i -> i.GetHashCode())
            |> Seq.fold (+) 0

        sprintf "%x" mana

    with
    | NoMagicWand
        -> "Error: A magic wand is required to hex!"
    | NoFullMoon(month, day)
        -> "Error: Hexes can only be cast during a full moon."
    | BadMojo(msg)
        -> sprintf "Error: Hex failed due to bad mojo [%s]" msg
```

В главе 3 мы познакомились с функциональным стилем программирования, который позволяет использовать интересные способы записи кода, но не является самодостаточным. Функциональный стиль в чи-

стом виде плохо взаимодействует с существующими библиотеками классов .NET и иногда требует сложного решения для простых задач.

В этой главе вы узнали, как изменять значения, что открывает вам широкие горизонты. Теперь вы можете использовать весьма эффективные коллекции для сохранения результатов, выполнять циклы и генерировать исключения при возникновении каких-либо проблем.

Кроме того, вы можете выбирать подход к решению задачи, и понятие мультипарадигмального программирования начинает обретать смысл. Некоторые проблемы могут быть решены просто за счет создания изменяемой структуры данных, а другие – за счет объединения простых функций, выполняющих трансформацию неизменяемых данных. Язык F# дает вам возможность выбора.

В следующей главе мы познакомимся с объектно-ориентированным программированием. Эта третья парадигма языка F# не всегда упрощает вычисления, но она дает программистам возможность организовать и абстрагировать код. Композиция объектов очень похожа на композицию обычных функций.

5

Объектно-ориентированное программирование

В этой главе мы рассмотрим парадигму программирования, которая на сегодняшний день получила наиболее широкое распространение, – объектно-ориентированное программирование. Владение навыками объектно-ориентированного программирования (ООП) совершенно необходимо для использования существующих фреймворков и библиотек .NET, а также для создания кода на языке F#, который легко интегрируется с этими библиотеками.

Программирование с применением объектов

Программные системы – это одни из самых сложных творений, когда-либо создававшихся человеком. Давайте посмотрим на типичную программу на платформе .NET: тысячи, если не миллионы, строк кода, преобразованные компиляторами в инструкции некоторого промежуточного языка, которые затем компилируются JIT-компилятором в машинные инструкции, которые затем выполняются процессором. Изучение подробностей работы каждого этапа – слишком тяжелый труд.

Вместо того чтобы корпеть над всеми деталями, объектно-ориентированное программирование позволяет разбить проблему на ряд концептуальных объектов и манипулировать ими. Моделирование реального мира в терминах абстрактных объектов позволяет уменьшить сложность кода.

Например, представьте, что у вас имеется некоторый сетевой лог-файл. Вы можете написать несколько отдельных функций, разбирающих и анализирующих данные в этом файле. Но с концептуальной точки зрения это обычный лог-файл, который имеет такие свойства, как размер файла и количество записей, и предусматривает выполнение таких

операций, как удаление, копирование, поиск, резервное копирование и так далее. Добавив еще один уровень абстракции, вы можете забыть о сложностях, лежащих уровнем ниже.

Преимущества ООП

Объектно-ориентированное программирование обладает несколькими преимуществами:

Упрощение повторного использования кода

Инкапсулируя код в объекты, вы получаете возможность его повторного использования, что в конечном итоге экономит ваше время и повышает производительность труда.

Уменьшение сложности

Вместо того чтобы работать одновременно с множеством отдельных функций и глобальных переменных, ООП позволяет вам работать с одним элементом за раз. Таким образом, любые изменения состояния ограничиваются областью объекта.

Специализация через наследование

Благодаря полиморфизму вы можете писать код, который взаимодействует с объектами базового типа, но при этом будет в состоянии работать с любыми специализированными экземплярами данного типа. Это еще одна разновидность приема повторного использования кода; она будет обсуждаться ниже в этой главе.

Недостатки ООП

Объектно-ориентированное программирование прекрасно подходит для моделирования определенных концепций, но не следует считать его панацеей. Как заметил Бернард Барух (Bernard Baruch):

> Когда в руках молоток, все предметы вокруг кажутся гвоздями.

Алгоритмы и абстрактные концепции в ООП реализуются сложнее. Для представления этих концепций появились *шаблоны проектирования (design patterns)*, облегчающие описание абстракций в терминах объектов и взаимоотношений между ними.

Шаблоны проектирования, конечно, полезны, но это всего лишь компенсация невозможности ООП достаточно просто выразить определенные концепции. Кроме того, они ложатся тяжким бременем на плечи программиста, вынуждая его писать однотипный код для определения контракта и взаимоотношений между объектами.

Язык F# предлагает не только объектно-ориентированные конструкции, но и новый набор функциональных примитивов, таких как размеченные объединения и функции-значения. Эти особенности дополняют существующие объектно-ориентированные механизмы и в значительной степени уменьшают необходимость в однотипном коде, который появляется при реализации шаблонов проектирования. Фактически при

использовании функционального языка программирования некоторые шаблоны проектирования вообще становятся ненужными. Например, шаблон «Команда» (Command Pattern) (описание его можно найти в книге «Design Patterns»[1] Эриха Гаммы (Erich Gamma) и др., Addison-Wesley) описывает объект, который может передаваться между программными компонентами и выполняться. Тот же результат можно получить с помощью функций высших порядков, которые поддерживаются языком F#.

System.Object

Платформа .NET предлагает богатую систему типов, которая способна проверять идентичность объектов и гарантирует безопасность типов во время выполнения. Благодаря наличию информации о типах, среда времени выполнения платформы .NET не позволит заткнуть круглое отверстие квадратной пробкой.

Идентичность объектов обеспечивается тем, что все компоненты, начиная от целых чисел и заканчивая строками и размеченными объединениями, являются экземплярами типа `System.Object`, который в языке F# обозначается как `obj`. Экземпляры `System.Object` не имеют самостоятельной ценности, потому что они не обладают никакими дополнительными особенностями. Тем не менее важно знать, какие методы есть у типа `System.Object`, потому что они доступны в любых объектах, с которыми вы можете столкнуться в .NET.

Общие методы

Каждый экземпляр типа `System.Object` обладает несколькими методами, которые могут быть переопределены в наследниках. Например, для ряда типов языка F# (размеченные объединения и записи) компилятор языка F# переопределяет некоторые из этих методов.

ToString

Метод `ToString` возвращает удобочитаемое строковое представление объекта. В языке F# то же значение выводится в консоль функцией `printf` со спецификатором формата `%A` или `%O`. Формально никак не оговаривается, что должен возвращать этот метод; единственное требование – метод `ToString` должен возвращать результат, который позволит отличать объекты друг от друга и пригодится при отладке. По умолчанию возвращается полное имя класса.

В следующем примере приводится определение типа `PunctionMark`, который предоставляет собственную реализацию метода `ToString`. Если бы

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб.: Питер, 2007.

этот метод не был переопределен, сообщение по умолчанию выглядело бы примерно так: FSI_0002+PunctuationMark:

```
> // Переопределение метода ToString
type PunctuationMark =
    | Period
    | Comma
    | QuestionMark
    | ExclamationPoint
    override this.ToString() =
        match this with
        | Period -> "Period (.)"
        | Comma -> "Comma (,)"
        | QuestionMark -> "QuestionMark (?)"
        | ExclamationPoint -> "ExclamationPoint (!)";;

type PunctuationMark =
    | Period
    | Comma
    | QuestionMark
    | ExclamationPoint
    with
        override ToString : unit -> string
    end

> let x = Comma;;

val x : PunctuationMark = Comma

> x.ToString();;
val it : string = "Comma (,)"
```

GetHashCode

Метод GetHashCode возвращает хеш-значение объекта. Переопределение этого метода имеет большое значение для типов, которые будут использоваться в таких коллекциях, как Dictionary, HashSet и Set. Хеш-значение – это псевдоуникальный идентификатор, описывающий определенный экземпляр типа. Это делает операцию определения идентичности двух объектов намного более эффективной.

```
> "alpha".GetHashCode();;
val it : int = -1898387216
> "bravo".GetHashCode();;
val it : int = 1946920786
> "bravo".GetHashCode();;
val it : int = 1946920786
```

Реализация по умолчанию вполне пригодна для большинства приложений, но если вы переопределяете метод Equals, вы должны также переопределить метод GetHashCode.

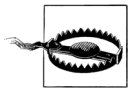
Переопределяя метод GetHashCode, вы должны гарантировать соблюдение следующих условий, чтобы обеспечить корректное поведение экземпляров данного типа в коллекциях:

- Если два объекта равны, то для них должны возвращаться одинаковые хеш-значения. Соблюдение обратного условия не требуется – два различных объекта могут иметь одинаковые хеш-значения. Однако вы должны приложить все усилия, чтобы избежать подобных коллизий хеш-значений.
- Хеш-значение объекта должно оставаться постоянным, если этот объект никак не изменялся. Если внутреннее состояние объекта изменилось, хеш-значение также может измениться.

Equals

Метод Equals проверяет эквивалентность двух объектов. Понятие эквивалентности в .NET – достаточно сложная тема (обращайтесь к разделу «Эквивалентность объектов» ниже):

```
> "alpha".Equals(4);;
val it : bool = false
> "alpha".Equals("alpha");;
val it : bool = true
```



При переопределении метода Equals вы также должны переопределить метод GetHashCode, потому что два эквивалентных объекта должны иметь одинаковые хеш-значения.

GetType

Наконец, метод GetType возвращает экземпляр типа System.Type, который представляет тип фактического объекта. Этот метод имеет большое значение для определения типов во время выполнения, и мы будем говорить об этом гораздо подробнее, когда будем рассматривать механизм рефлексии в главе 12.

```
> let stringType = "A String".GetType();;

val stringType : System.Type

> stringType.AssemblyQualifiedName;;
val it : string
= "System.String, mscorlib, Version=4.0.0.0, Culture=neutral, Public
KeyToken=b77a5c561934e089"
```

Эквивалентность объектов

Эквивалентность типов в .NET – это достаточно сложная тема, обращающаяся вокруг понятий типа и идентификации объектов. Как упоминалось в главе 4, существуют значимые и ссылочные типы, и для них

по умолчанию используются различные механизмы проверки эквивалентности.

По умолчанию для значимых типов используется побитовое сравнение, то есть для определения идентичности двух объектов в стеке выполняется побайтовое сравнение. Этого вполне достаточно для простых типов:

```
> // Эквивалентность значимых типов
let x = 42
let y = 42;;

val x : int = 42
val y : int = 42

> x = y;;
val it : bool = true
```

С другой стороны, семантика сравнения объектов ссылочных типов более сложная. В примере 5.1 оба значения, `x` и `y`, являются экземплярами класса `ClassType`, созданными с одними и теми же исходными данными. Однако, несмотря на одинаковые значения, они не считаются эквивалентными.

Пример 5.1. Ссылочная эквивалентность

```
> // Ссылочная эквивалентность
type ClassType(x : int) =
    member this.Value = x

let x = new ClassType(42)
let y = new ClassType(42);;

type ClassType =
  class
    new : x:int -> ClassType
    member Value : int
  end
val x : ClassType
val y : ClassType

> x = y;;
val it : bool = false
> x = x;;
val it : bool = true
```

По умолчанию эквивалентность ссылочных типов является *ссылочной эквивалентностью (referential equality)*, то есть две ссылки считаются эквивалентными, если они указывают на один и тот же объект. В предыдущем примере выражение `x = x` возвращает значение `true`, потому что значения `x` и `x` ссылаются на один и тот же адрес в памяти. Однако выражение `x = y` возвращает `false`, потому что значения `x` и `y` ссылаются на

разные адреса в памяти. Такое поведение не подходит для многих приложений.

Переопределение метода `Object.Equals` позволяет изменить семантику эквивалентности. В следующем примере определяется тип `ClassType2`, который придает более разумный смысл понятию эквивалентности:

```
> // Переопределение метода Equals
type ClassType2(x : int) =
    member this.Value = x
    override this.Equals(o : obj) =
        match o with
        | :? ClassType2 as other -> (other.Value = this.Value)
        | _ -> false
    override this.GetHashCode() = x

let x = new ClassType2(31)
let y = new ClassType2(31)
let z = new ClassType2(10000);;

type ClassType2 =
  class
    new : x:int -> ClassType2
    override Equals : o:obj -> bool
    override GetHashCode : unit -> int
    member Value : int
  end
val x : ClassType2
val y : ClassType2
val z : ClassType2

> x = y;;
val it : bool = true
> x = z;;
val it : bool = false
```

Сгенерированная эквивалентность

Кортежи, размеченные объединения и записи ведут себя именно так, как и следовало ожидать, то есть два экземпляра с одинаковыми значениями считаются эквивалентными, подобно значимым типам.

Кортежи считаются эквивалентными, если оба кортежа содержат одинаковое количество элементов и все их элементы эквивалентны:

```
> let x = (1, 'a', "str");;

val x : int * char * string = (1, 'a', "str")

> x = x;;
val it : bool = true
> x = (1, 'a', "different str");;
```

```
val it : bool = false
> // Вложенные кортежи
(x, x) = (x, (1, 'a', "str"));;
val it : bool = true
```

Записи считаются эквивалентными, если все их поля имеют одинаковые значения:

```
> // Эквивалентность записей
type RecType = { Field1 : string; Field2 : float }

let x = { Field1 = "abc"; Field2 = 3.5 }
let y = { Field1 = "abc"; Field2 = 3.5 }
let z = { Field1 = "XXX"; Field2 = 0.0 };;

type RecType =
  {Field1: string;
   Field2: float;}
val x : RecType = {Field1 = "abc";
                   Field2 = 3.5;}
val y : RecType = {Field1 = "abc";
                   Field2 = 3.5;}
val z : RecType = {Field1 = "XXX";
                   Field2 = 0.0;}

> x = y;;
val it : bool = true
> x = z;;
val it : bool = false
```

Размеченные объединения считаются эквивалентными, если оба значения относятся к одному и тому же варианту и кортежи, ассоциированные с вариантами, являются эквивалентными:

```
> // Эквивалентность размеченных объединений
type DUType =
    | A of int * char
    | B

let x = A(1, 'k')
let y = A(1, 'k')
let z = B;;

type DUType =
  | A of int * char
  | B
val x : DUType = A (1,'k')
val y : DUType = A (1,'k')
val z : DUType = B

> x = y;;
val it : bool = true
```

```
> x = z;;
val it : bool = false
```

В действительности кортежи, размеченные объединения и записи относятся к ссылочным типам. Поэтому по умолчанию они должны использовать ссылочную эквивалентность, то есть во всех предыдущих примерах операции сравнения должны возвращать `false`. Однако это не так, и причина такого их поведения заключается в том, что компилятор F# автоматически переопределяет методы `Equals` и `GetHashCode` в этих типах (реализуя так называемую семантику *структурной эквивалентности* (*structural equality*)).

В случае необходимости вы можете изменить поведение по умолчанию, «сгенерированное» компилятором, – либо переопределив методы `Equals` и `GetHashCode`, либо добавив атрибут `ReferenceEquality`.

Пример 5.2 демонстрирует создание типа размеченного объединения, для экземпляров которого используется ссылочная эквивалентность.

Пример 5.2. Переопределение сгенерированной эквивалентности

```
> // Ссылочная эквивалентность в функциональных типах
[<ReferenceEquality>]
type RefDUType =
    | A of int * char
    | B;;

type RefDUType =
  | A of int * char
  | B

> // Объявление двух, концептуально эквивалентных значений
let x = A(4, 'A')
let y = A(4, 'A');;

val x : RefDUType = A (4,'A')
val y : RefDUType = A (4,'A')

> x = y;;
val it : bool = false
> x = x;;
val it : bool = true
```

Классы

Теперь, когда вы познакомились с основными сведениями об объектах, вы можете приступать к созданию собственных типов данных. В самом простом представлении классы – это средство объединения функций и данных. Данные, ассоциированные с классом, называются *полями*. Функции, ассоциированные с классом, называются *свойствами,* или

членами. (Свойства – это обычные методы, не имеющие входных параметров.)

Постепенно я расскажу обо всех аспектах полей, методов и свойств. А пока отмечу, что от создания нового экземпляра класса будет мало пользы, если класс не предусматривает возможность инициализации экземпляров некоторыми начальными значениями. Классы имеют специальные методы, которые называются *конструкторами*, в задачу которых входит инициализация полей.

Конструктор класса вызывается в момент создания нового экземпляра с помощью ключевого слова `new`, и он не может быть вызван повторно, после того как экземпляр будет создан.

Язык F# позволяет определять в классах конструкторы двух типов: явные конструкторы и неявные конструкторы. Одни обеспечивают простой и лаконичный синтаксис, а другие позволяют более четко контролировать процесс создания экземпляров.

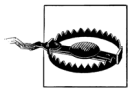
Явное создание

Использование явных конструкторов – наименее приятный способ создания экземпляров класса, но он обеспечивает наиболее полный контроль над происходящим. Когда класс определяется с явным конструктором, необходимо явно описать каждое поле класса и инициализировать его в конструкторе.

Полям класса предшествует ключевое слово `val`, и доступ к ним осуществляется с использованием *собственного идентификатора (self identifier)*; они будут рассмотрены ниже. В примере 5.3 определяется новый класс `Point`, который имеет два поля, `m_x` и `m_y`. В классе имеются также два конструктора, один из них принимает два параметра, а другой – ни одного.

Пример 5.3. Явные конструкторы класса

```
type Point =
    val m_x : float
    val m_y : float

// Конструктор 1 - принимает два параметра
new (x, y) = { m_x = x; m_y = y }

// Конструктор 2 - не имеет параметров
new () = { m_x = 0.0; m_y = 0.0 }

member this.Length =
    let sqr x = x * x
    sqrt <| sqr this.m_x + sqr this.m_y

let p1 = new Point(1.0, 1.0)
let p2 = new Point()
```



В языке C#, если вы не объявите конструктор самостоятельно, компилятор создаст конструктор по умолчанию. В языке F# дело обстоит совершенно иначе. Язык F# позволяет определять классы без конструкторов, но тогда у вас не будет возможности создавать экземпляры таких классов!

До и после инициализации полей класса может выполняться произвольный код для выполнения дополнительных операций по инициализации. Для выполнения операций до инициализации полей класса просто поместите необходимый код сразу после знака `=`. Чтобы выполнить операции после инициализации полей, используйте ключевое слово `then`, за которым должно следовать выражение, возвращающее `unit`.

В примере 5.4 в класс `Point` добавлен явный конструктор, который принимает строковый параметр с именем `text`. Этот конструктор разбирает строку и использует извлеченные из нее значения для инициализации полей `m_x` и `m_y`. После того как поля будут инициализированы, конструктор выводит сообщение в консоль.

Пример 5.4. Выполнение произвольного кода перед вызовом явного конструктора

```
open System

type Point2 =
    val m_x : float
    val m_y : float

    // Анализ строки, имеющей вид "1.0, 2.0"
    new (text : string) as this =
        // Выполнить предобработку
        if text = null then
            raise <| new ArgumentException("text")

        let parts = text.Split([| ',' |])
        let (successX, x) = Double.TryParse(parts.[0])
        let (successY, y) = Double.TryParse(parts.[1])

        if not successX || not successY then
            raise <| new ArgumentException("text")

        // Инициализация полей класса
        { m_x = x; m_y = y }
        then
            // Выполнить постобработку
            printfn
                "Initialized to [%f, %f]"
                this.m_x
                this.m_y
```

Неявное создание классов

Явное создание классов следует традиционному объектно-ориентированному стилю программирования, однако они выглядят несколько неестественно в функциональном программировании. Неявные конструкторы обеспечивают более простой способ определения классов и являются более естественными в языке F#.

При использовании неявного конструктора не требуется явно определять поля класса (с помощью ключевого слова `val`). Вместо этого внутри определения типа вы можете использовать выражения `let`. В процессе компиляции класса выражения `let` будут преобразованы в поля класса или будут просто удалены в процессе оптимизации компилятором. Несмотря на то, что неявные конструкторы не позволяют явно определять поля класса, тем не менее они помогают упростить код создания классов.

Чтобы создать класс с неявным конструктором, достаточно просто добавить круглые скобки с аргументами после имени класса. Эти аргументы будут играть роль параметров *основного конструктора (primary constructor)* класса. Любые операторы связывания `let` и `do`, присутствующие в теле класса, будут выполняться при создании его экземпляров. Любые дополнительные конструкторы должны вызывать основной конструктор, чтобы обеспечить выполнение всех операторов `let`.

В примере 5.5 демонстрируется определение класса с неявным конструктором. В этом примере точно так же реализован разбор строки с координатами точки на плоскости и добавлен еще один конструктор, который принимает два вещественных числа. Он будет играть роль основного конструктора.

Пример 5.5. Неявный конструктор класса

```
type Point3(x : float, y : float) =

    let length =
        let sqr x = x * x
        sqrt <| sqr x + sqr y
    do printfn "Initialized to [%f, %f]" x y

    member this.X = x
    member this.Y = y
    member this.Length = length

    // Определяем дополнительные конструкторы, которые должны
    // вызывать 'основной' конструктор
    new() = new Point3(0.0, 0.0)

    // Второй конструктор.
    new(text : string) =
        if text = null then
            raise <| new ArgumentException("text")
```

```
let parts = text.Split([| ',' |])
let (successX, x) = Double.TryParse(parts.[0])
let (successY, y) = Double.TryParse(parts.[1])
if not successX || not successY then
    raise <| new ArgumentException("text")
// Вызов основного конструктора класса
new Point3(x, y)
```

Параметры основного конструктора, а также значения, объявленные с помощью операторов `let`, доступны в любом месте в определении класса и не требуют использования собственного идентификатора. Значения `x` и `y` в предыдущем примере – это параметры основного конструктора, которые доступны в любом месте в определении класса, как и значение `length`, объявленное с помощью `let`.

Наличие двух видов конструкторов может показаться избыточным. Наиболее предпочтительным является использование неявных конструкторов, а явные конструкторы следует применять только в случаях, когда требуется обеспечить более тонкое управление полями класса.

Обобщенные классы

В главе 2 мы познакомились с обобщенными функциями, или с функциями, которые принимают параметры любого типа. Эту концепцию легко можно распространить и на классы, что позволит нам определять типы, которые могут быть коллекциями других типов. Например, типы `list` и `seq` в языке F# являются обобщенными, благодаря чему вы можете создавать списки и последовательности данных любых типов, такие как списки строк или последовательности вещественных чисел.

Чтобы добавить в класс параметр обобщенного типа, необходимо использовать угловые скобки после имени класса. После этого для доступа к обобщенному типу следует использовать имя параметра обобщенного типа. Например:

```
type GenericClass<'a, 'b>() = ...
```

Как и в обобщенных функциях, перед обобщенным параметром типа должен находиться символ апострофа '. В F# принято именовать обобщенные параметры типа следующим образом: 'a, 'b, 'c и так далее.

В примере 5.6 определяется обобщенный класс `Arrayify`, основной конструктор которого принимает обобщенный параметр `x` и содержит свойства, которые возвращают массивы различных размеров, заполненные значениями `x`.

Пример 5.6. Определение обобщенных классов

```
> // Определение обобщенного класса
type Arrayify<'a>(x : 'a) =
```

```
    member this.EmptyArray : 'a[] = [| |]
    member this.ArraySize1 : 'a[] = [| x |]
    member this.ArraySize2 : 'a[] = [| x; x |]
    member this.ArraySize3 : 'a[] = [| x; x; x |];;

type Arrayify<'a> =
  class
    new : x:'a -> Arrayify<'a>
    member ArraySize1 : 'a []
    member ArraySize2 : 'a []
    member ArraySize3 : 'a []
    member EmptyArray : 'a []
  end

> let arrayifyTuple = new Arrayify<int * int>( (10, 27) );;

val arrayifyTuple : Arrayify<int * int>

> arrayfiyTuple.ArraySize3;;
val it : (int * int) [] = [|(10, 27); (10, 27); (10, 27)|]
```

Важно отметить, что в вызове конструктора обобщенный параметр можно опустить, заменив его групповым символом <_>, и положиться на автоматический вывод типа параметра компилятором F#. В следующем фрагменте на основе значения параметра x, передаваемого конструктору класса Arrayify, компилятор определил, что обобщенный параметр относится к типу string:

```
> let inferred = new Arrayify<_>( "a string" );;

val inferred : Arrayify<string>
```

Записи и размеченные объединения также могут быть обобщенными. В следующем фрагменте приводится определение обобщенного размеченного объединения:

```
> // Обобщенное размеченное объединение
type GenDU<'a> =
    | Tag1 of 'a
    | Tag2 of string * 'a list;;

type GenDU<'a> =
  | Tag1 of 'a
  | Tag2 of string * 'a list

> Tag2("Primary Colors", [ 'R'; 'G'; 'B' ]);;
val it : GenDU<char> = Tag2 ("Primary Colors",['R'; 'G'; 'B'])
```

В следующем фрагменте приводится определение обобщенной записи:

```
> type GenRec<'a, 'b> = { Field1 : 'a; Field2 : 'b };;
```

```
type GenRec<'a,'b> =
  {Field1: 'a;
   Field2: 'b;}

> let x = { Field1 = "Blue"; Field2 = 'C' };;

val x : GenRec<string,char> = {Field1 = "Blue";
                               Field2 = 'C';}
```

Собственный идентификатор

В предыдущих примерах вы могли заметить слово `this` перед именем каждого метода. Это имя собственного идентификатора (self identifier), которое позволяет ссылаться на экземпляр класса внутри любого из его методов. Это не обязательно должно быть имя `this` – в действительности, в качестве собственного идентификатора можно использовать любой допустимый идентификатор.

В примере 5.7 приводится определение класса, в котором используются необычные имена собственных идентификаторов. Обратите внимание, что область действия имени, используемого в качестве собственного идентификатора, ограничивается методом или свойством.

Пример 5.7. Именование указателя this

```
open System

type Circle =
    val m_radius : float

    new(r) = { m_radius = r }
    member foo.Radius = foo.m_radius
    member bar.Area = Math.PI * bar.Radius * bar.Radius
```

Безусловно, имя собственного идентификатора не должно конфликтовать с именами членов класса.

В примерах, которые приводятся в этой книге, обычно используется имя `this` из соображений соответствия синтаксису языка C#; при этом в типичном коде на языке F# редко возникает необходимость использования собственного идентификатора, поэтому ему обычно дается более короткое имя, такое как `x` или даже `__`.

Методы и свойства

Смысл классов определяется методами и свойствами. Методы – это действия, или глаголы, описывающие, что этот класс может делать или что можно делать с ним. Свойства – это атрибуты, или прилагательные, помогающие описать класс.

Свойства

Свойства могут быть трех видов: только для чтения, только для записи и для чтения и записи. Свойство *метод чтения (getter)* определяется с помощью ключевого слова `get` и возвращает значение свойства, а *метод записи (setter)* определяется с помощью ключевого слова `set` и изменяет значение свойства.

Свойства, доступные только для чтения, определяются как простые методы без аргументов. Для явного контроля можно определить методы `with get` и `with set`.

В примере 5.8 создается простое свойство, доступное только для чтения, а также более сложное свойство с методами чтения и записи. Для вызова метода записи следует использовать оператор стрелки влево `<-`, как если бы выполнялась запись в изменяемое значение.

Пример 5.8. Определение свойств класса

```
> // Определение класса WaterBottle с двумя свойствами
[<Measure>]
type ml

type WaterBottle() =
    let mutable m_amount = 0.0<ml>

    // Свойство, доступное только для чтения
    member this.Empty = (m_amount = 0.0<ml>)

    // Свойство, доступное для чтения и записи
    member this.Amount with get ()    = m_amount
                       and set newAmt = m_amount <- newAmt;;

[<Measure>]
type ml
type WaterBottle =
  class
    new : unit -> WaterBottle
    member Amount : float<ml>
    member Empty : bool
    member Amount : float<ml> with set
  end

> let bottle = new WaterBottle();;

val bottle : WaterBottle

> bottle.Empty;;
val it : bool = true
> bottle.Amount <- 1000.0<ml>;;
val it : unit = ()
```

```
> bottle.Empty;;
val it : bool = false
```

Установка значений свойств в конструкторе

Свойства всего лишь обеспечивают простой доступ к данным, однако необходимо отметить, что вы можете устанавливать значения свойств с помощью дополнительных параметров конструктора. Это упрощает код создания объектов и позволяет инициализировать объекты так, как вы того хотите.

В примере 5.9 демонстрируются два идентичных способа создания нового объекта класса `Form` из пространства имен `System.Windows.Forms`. В первом случае создается объект и устанавливаются значения свойств по отдельности, а во втором – значения свойств определяются в вызове конструктора. Это всего лишь синтаксический сахар, позволяющий установить значения отдельных свойств после вызова конструктора.[1]

Пример 5.9. Установка значений свойств после вызова конструктора

```
open System.Windows.Forms

// Попытка первая - сложный способ
let f1 = new Form()
f1.Text    <- "Window Title"
f1.TopMost <- true
f1.Width   <- 640
f1.Height  <- 480
f1.ShowDialog()

// Попытка вторая - простой способ
let f2 = new Form(Text    = "Window Title",
                  TopMost = true,
                  Width   = 640,
                  Height  = 480)
f2.ShowDialog()
```

Методы

Для определения новых методов класса необходимо указать собственный идентификатор и вслед за ним имя функции. Как и в случае с обычными функциями, любые правила сопоставления с образцом, следующие после имени функции, являются ее параметрами. И точно так же, как в случае с обычными функциями, чтобы определить метод без параметров, достаточно просто определить для него параметр типа `unit`:

[1] Читатель, знакомый с языком программирования C#, может заметить, что данная возможность языка F# аналогична синтаксису инициализации объектов (Object Initialization Syntax) языка C#. – *Прим. науч. ред.*

```
type Television =

    val mutable m_channel : int
    val mutable m_turnedOn : bool

    new() = { m_channel = 3; m_turnedOn = true }

    member this.TurnOn () =
        printfn "Turning on..."
        this.m_turnedOn <- true

    member this.TurnOff () =
        printfn "Turning off..."
        this.m_turnedOn <- false

    member this.ChangeChannel (newChannel : int) =
        if this.m_turnedOn = false then
            failwith "Cannot change channel, the TV is not on."

        printfn "Changing channel to %d..." newChannel
        this.m_channel <- newChannel

    member this.CurrentChannel = this.m_channel
```

Методы классов могут каррироваться, подобно обычным функциям. Однако пользоваться такой возможностью не рекомендуется, потому что другие языки .NET не поддерживают каррирование функций. Рекомендуется объединять все параметры в общий кортеж. При использовании ваших классов из других языков ваши методы будут выглядеть не так, будто они принимают единственный параметр типа `Tuple`, а так, будто они принимают отдельные параметры для каждого элемента кортежа, как и следовало ожидать:

```
> // Каррируемые методы класса
type Adder() =
    // Каррируемые аргументы метода
    member this.AddTwoParams x y = x + y
    // Обычные аргументы
    member this.AddTwoTupledParams (x, y) = x + y;;

type Adder =
  class
    new : unit -> Adder
    member AddTwoParams : x:int -> y:int -> int
    member AddTwoTupledParams : x:int * y:int -> int
  end

> let adder = new Adder();;

val adder : Adder
```

```
> let add10 = adder.AddTwoParams 10;;

val add10 : (int -> int)

> adder.AddTwoTupledParams(1, 2);;
val it : int = 3
```

Наиболее важное отличие между методами классов и функциями состоит в том, что методы могут быть рекурсивными без ключевого слова `rec`.

Руководства по программированию в .NET рекомендуют называть свойства и методы в стиле Pascal Case, когда каждое слово начинается с заглавной буквы, включая и первое слово. Так, вместо идентификатора `doStuff` или `do_stuff` рекомендуется использовать `DoStuff`.

Статические методы, свойства и поля

Мы познакомились с методами и свойствами, связанными с определенным экземпляром класса. Но что если вам потребуется метод, связанный с самим классом, а не с каким-то определенным экземпляром? Например, класс `Environment` в пространстве имен `System` содержит свойства со сведениями о текущем компьютере, такие как имя компьютера. Имя компьютера концептуально никак не связано с каким-то определенным экземпляром класса `System.Environment`, поэтому было бы довольно странным требовать наличие экземпляра для получения доступа к свойству `MachineName`.

Чтобы объявить метод или свойство, доступ к которому может производиться без экземпляра класса, этот метод необходимо пометить как *статический*. Это означает, что внутри таких методов и свойств невозможно получить доступ к элементам, обращения к которым осуществляются с помощью собственного идентификатора, таким как поля класса, но они могут вызываться без экземпляра класса.

Чтобы определить статический метод, достаточно просто добавить ключевое слово `static` перед объявлением метода без определения собственного идентификатора. В примере 5.10 демонстрируется определение статического метода в классе `SomeClass`. Чтобы вызвать статический метод, необходимо использовать полное квалифицированное имя типа и имя метода, в противном случае будет ошибка компиляции.

Пример 5.10. Статические методы

```
> // Объявление статического метода
type SomeClass() =
    static member StaticMember() = 5;;

type SomeClass =
  class
    new : unit -> SomeClass
```

```
    static member StaticMember : unit -> int
  end

> SomeClass.StaticMember();;
val it : int = 5
> let x = new SomeClass();;

val x : SomeClass

> x.StaticMember();;

x.StaticMember();;
^^^^^^^^^^^^^^^^
stdin(39,1): error FS0809: StaticMember is not an instance method
```

(StaticMember не является экземплярным методом)

Статические методы обычно используются во вспомогательных классах и нетипичны в коде на языке F#. Если вы вдруг заметите, что пишете много статических методов класса, подумайте об организации их в виде модуля.

Статические поля

Поля класса также могут быть помечены ключевым словом `static`; это означает, что данные ассоциированы с самим классом и доступны всем экземплярам. (Вместо отдельных копий в каждом экземпляре класса.)

Статические поля обычно играют роль глобальных констант. Однако изменяемые статические поля также могут оказаться полезными. В примере 5.11 определяется класс `RareType` со статическим полем, значение которого уменьшается при каждом вызове основного конструктора. Как только значение статического поля `m_numLeft` достигнет нуля, генерируется исключение, препятствующее созданию новых экземпляров класса. Без значения, общего для всех экземпляров класса `RareType`, реализовать такое было бы невозможно.

Пример 5.11. Создание и использование статических полей

```
// Статические поля
type RareType() =

    // Все экземпляры класса RareType имеют одно общее значение m_numLeft
    static let mutable m_numLeft = 2

    do
        if m_numLeft <= 0 then
            failwith "No more left!"
        m_numLeft <- m_numLeft - 1
        printfn " Инициализация экземпляра RareType. Осталась возможность
                  создания только %d экземпляра(ов)" m_numLeft

    static member NumLeft = m_numLeft
```

Следующий сеанс FSI показывает действие этого счетчика:

```
> let a = new RareType();;

val a : RareType

Initialized a rare type, only 1 left!
> let b = new RareType();;

val b : RareType

Initialized a rare type, only 0 left!
> let c = new RareType();;

val c : RareType

System.Exception: No more left!
  at FSI_0012.RareType..ctor() in C:\Users\chrsmith\Desktop\Ch05.fsx:line 18
  at <StartupCode$FSI_0015>.$FSI_0015._main()
stopped due to error
> RareType.NumLeft;;
val it : int = 0
```

Перегрузка методов

Зачастую после создания некоторого очень удобного метода возникает необходимость снабдить его различными наборами параметров. Это может избавить вас от необходимости преобразовывать данные в типы, требуемые методом. Создание метода с тем же именем, но принимающего иное количество или типы аргументов, называется *перегрузкой метода (method overloading)*.

Например, существует 19 различных перегруженных версий метода `System.Console.Writeline`! На первый взгляд это кажется перебором, но если вам потребуется вывести целое число, вы сможете использовать перегруженную версию, принимающую тип `int`:

```
System.Console.WriteLine(42)
```

В противном случае вам пришлось бы сначала преобразовать параметр типа `int` в строку, что было бы крайне неудобно:

```
System.Console.WriteLine((42).ToString())
```

В примере 5.12 для метода `CountBits` создано несколько перегруженных версий, принимающих аргументы любых простых целочисленных типов.

Пример 5.12. Перегрузка методов в F#

```
type BitCounter =

    static member CountBits (x : int16) =
```

```
        let mutable x' = x
        let mutable numBits = 0
        for i = 0 to 15 do
            numBits <- numBits + int (x' &&& 1s)
            x' <- x' >>> 1
        numBits

    static member CountBits (x : int) =
        let mutable x' = x
        let mutable numBits = 0
        for i = 0 to 31 do
            numBits <- numBits + int (x' &&& 1)
            x' <- x' >>> 1
        numBits

    static member CountBits (x : int64) =
        let mutable x' = x
        let mutable numBits = 0
        for i = 0 to 63 do
            numBits <- numBits + int (x' &&& 1L)
            x' <- x' >>> 1
        numBits
```

Модификаторы доступа

Теперь вы можете создавать классы для инкапсуляции сложной внутренней логики работы ваших абстракций. Однако если пользователи ваших классов будут иметь доступ ко всем методам, свойствам или полям, они могут по неосторожности изменить внутреннее состояние и тем самым вызвать появление ошибок.

Ограничение видимости членов класса имеет большое значение. Благодаря этому вы можете ограничить доступность свойств и методов при использовании класса в других сборках.

В языке F# управление видимостью классов или методов осуществляется с помощью *модификаторов доступа (accessibility modifier)*. В примере 5.13 класс `Ruby` помечен как внутренний (`internal`), а его основной конструктор – как закрытый (`private`), тогда как все остальные конструкторы являются открытыми (public). Попытка обратиться к закрытому значению, методу, свойству или конструктору приведет к ошибке времени компиляции.

Пример 5.13. Модификаторы доступа

```
type internal Ruby private(shininess, carats) =

    let mutable m_size = carats
    let mutable m_shininess = shininess
```

```
// Полировка увеличивает отражающую способность, но уменьшает размер
member this.Polish() =
    this.Size   <- this.Size - 0.1
    m_shininess <- m_shininess + 0.1

// Открытый метод чтения и закрытый метод записи
member public this.Size with get () = m_size
member private this.Size with set newSize = m_size <- newSize

member this.Shininess = m_shininess

public new() =
    let rng = new Random()
    let s = float (rng.Next() % 100) * 0.01
    let c = float (rng.Next() % 16) + 0.1
    new Ruby(s, c)

public new(carats) =
    let rng = new Random()
    let s = float (rng.Next() % 100) * 0.01
    new Ruby(s, carats)
```

Особенности всех трех модификаторов доступа описываются в табл. 5.1. По умолчанию все типы, значения и функции в языке F# являются открытыми. Поля классов (объявленные с помощью ключевых слов `val` и `let`) по умолчанию являются закрытыми.

Таблица 5.1. Модификаторы доступа

Модификатор доступа	Видимость
`public`	Модификатор `public` означает, что метод или свойство будет доступно откуда угодно. Все классы, значения и функции в языке F# по умолчанию являются открытыми.
`private`	Модификатор `private` ограничивает область видимости значения данным классом. Закрытые значения недоступны ни внешнему коду, ни производным классам. Все поля классов по умолчанию являются закрытыми.
`internal`	Модификатор `internal` имеет то же значение, что и модификатор `public`, но он распространяется только на текущую сборку. Внутренние (`internal`) классы недоступны в других сборках, как если бы они были объявлены закрытыми (`private`).

В отличие от языка C#, в F# нет модификатора `protected`. Однако F# будет соблюдать правила видимости защищенных членов при наследовании классов, реализованных на других языках, где поддерживается этот модификатор доступа.

Модификаторы доступа значений модулей

Модификаторы могут применяться не только к классам – они могут также использоваться для значений, определенных в модулях.

В примере 5.14 определяется модуль `Logger`, в котором имеется закрытое изменяемое значение `m_filesToWriteTo`. Это значение доступно только внутри модуля, поэтому оно может использоваться методами `AddLogFile` и `LogMessage` без дополнительной проверки `m_filesToWriteTo` на равенство значению `null`. (Если бы значение `m_filesToWriteTo` было открытым, вызывающий код мог бы по ошибке изменить это значение.)

Пример 5.14. Модификаторы доступа в модулях

```
open System.IO
open System.Collections.Generic

module Logger =

    let mutable private m_filesToWriteTo = new List<string>()

    let AddLogFile(filePath) = m_filesToWriteTo.Add(filePath)

    let LogMessage(message : string) =
        for logFile in m_filesToWriteTo do
            use file = new StreamWriter(logFile, true)
            file.WriteLine(message)
            file.Close()
```

Файлы сигнатур

Модификаторы доступа являются важным средством ограничения видимости и инкапсуляции, но добавление модификаторов доступа к каждому методу класса и к каждому значению приводит к замусориванию кода. Поэтому в языке F# предусмотрена возможность управления доступностью для всего файла с помощью *файла сигнатур (signature file)*. Файл сигнатур с расширением *.fsi* позволяет определить сигнатуры для всего кода в основном файле. Все, что присутствует в файле с кодом, но отсутствует в файле сигнатур, считается закрытым. Файлы сигнатур обеспечивают простой способ аннотирования больших объемов кода.

В примере 5.15 приводится содержимое файла с кодом и содержимое соответствующего ему файла сигнатур. Файл сигнатур определяет класс с конструктором и двумя методами. В соответствующем ему файле реализации присутствуют дополнительные методы и свойства, и даже притом, что эти методы и свойства не объявлены закрытыми явно, тем не менее они являются закрытыми. Кроме того, файл сигнатур объявляет класс как внутренний (`internal`), поэтому класс будет скомпилирован именно с этой областью видимости. Если видимость члена не определена в файле *.fsi*, он считается открытым.

Если в файле с реализацией будет определена более ограниченная видимость члена, чем в файле сигнатур, компилятор выдаст сообщение об ошибке.

В этой книге все примеры были созданы без использования файлов сигнатур *.fsi*, потому что они обычно используются в крупных проектах на языке F#.

Пример 5.15. Пример файлов .fsi и .fs

```
// File.fsi
namespace MyProject.Utilities

type internal MyClass =
    new : unit -> MyClass
    member public Property1 : int
    member private Method1 : int * int -> int

// File.fs
namespace MyProject.Utilities

type internal MyClass() =
    member this.Property1 = 10
    member this.Property2 with set (x : int) = ()
    member this.Method1 (x, y) = x + y
    member this.Method2 () = true
```

Наследование

До сих пор, рассказывая о классах, я никак не касался *полиморфизма*, который обеспечивает основную мощь объектно-ориентированного программирования. Взгляните на следующие два класса, восхитительный `BLTSandwich` и обычный `TurkeyAndSwissSandwich`:

```
type BLTSandwich() =
    member this.Ingredients = ["Bacon"; "Lettuce"; "Tomato"]
    member this.Calories = 450
    override this.ToString() = "BLT"

type TurkeySwissSandwich() =
    member this.Ingredients = ["Turkey"; "Swiss"]
    member this.Calories = 330
    override this.ToString() = "Turkey and Swiss"
```

Хотя оба класса практически идентичны, они являются разными сущностями. Поэтому если вам потребуется написать функцию, принимающую экземпляры обоих классов, `BLTSandwich` и `TurkeySwissSandwich`, вам потребуется прибегнуть к перегрузке методов. Кроме того, при добавлении каждого нового вида сэндвичей нам придется добавлять новую перегруженную версию метода.

```
member this.EatLunch(sandwich : BLTSandwich) = (*...*)

member this.EatLunch(sandwich : TurkeySwissSandwich) = (*...*)

// Эту версию придется добавить позднее...
member this.EatLunch(sandwich : ClubSandwich) = (*...*)
```

Более правильным было бы представить каждую из этих вкусных закусок в виде специализации одного базового типа `Sandwich` и изменять в них общие свойства и методы класса `Sandwich`. Каждая разновидность класса `Sandwich` будет содержать свое количество калорий и список ингредиентов.

Более того, можно продолжить иерархию классов и определить еще более специфические разновидности, например создать класс `BLTWithPickelsSandwich`. Именно так действуют наследование и полиморфизм.

Наследование (inheritance) – это возможность создавать новые классы, которые наследуют методы и свойства базовых классов, а также добавляют новые методы и свойства и/или переопределяют существующие. Так, в нашем примере класс `BLTSandwich` является специализированным, или *производным классом (derived class)*, класса `Sandwich`, со своей собственной реализацией свойств `Calories` и `Ingredients`.

Полиморфизм (polymorphism) – это возможность трактовать экземпляры любых производных классов как экземпляры базового класса. (Так как известно, что производные классы содержат все методы и свойства своего базового класса.)

Для наследования в объявлении класса необходимо добавить строку `inherit` *тип*. Для классов с явными и неявными конструкторами используется немного разный синтаксис.

В классах с неявным конструктором достаточно просто добавить в основной конструктор вызов конструктора базового класса сразу после ключевого слова `inherit`. Не забудьте передать все необходимые параметры конструктору базового класса:

```
// Базовый класс
type BaseClass =
    val m_field1 : int

    new(x) = { m_field1 = x }
    member this.Field1 = this.m_field1

// Производный класс с неявным конструктором
type ImplicitDerived(field1, field2) =
    inherit BaseClass(field1)

    let m_field2 : int = field2

    member this.Field2 = m_field2
```

В классах с явными конструкторами вызов конструктора базового класса осуществляется путем указания `inherit` *тип* в разделе инициализации полей класса:

```
// Производный класс с явным конструктором
type ExplicitDerived =
    inherit BaseClass

    val m_field2 : int

    new(field1, field2) =
        {
            inherit BaseClass(field1)
            m_field2 = field2
        }

    member this.Field2 = this.m_field2
```

Переопределение методов

Ключом к специализации производных классов является прием, называемый *переопределением методов (method overriding)*, который позволяет вам изменять реализацию методов.

Чтобы сообщить компилятору, что метод или свойство может быть переопределено, в языке F# оно помечается ключевым словом `abstract`. В примере 5.16 класс `Sandwich` имеет два абстрактных свойства, `Ingredients` и `Calories`. Чтобы определить реализацию по умолчанию для этих методов, следует использовать ключевое слово `default` перед объявлением члена.[1]

Чтобы переопределить или изменить реализацию метода, унаследованного от базового класса, следует использовать ключевое слово `override` перед объявлением члена. (Точно так же, как мы выполняли переопределение метода `ToString` класса `System.Object`.) После этого при вызове этого метода будет вызываться метод производного класса.

Пример 5.16. Переопределение методов в языке F#

```
type Sandwich() =
    abstract Ingredients : string list
    default this.Ingredients = []

    abstract Calories : int
    default this.Calories = 0
```

[1] Это еще одно отличие от языка C#, в котором абстрактные члены с поведением по умолчанию называются «виртуальными» и определяются с помощью ключевого слова `virtual`, в то время как для объявления абстрактных членов применяется ключевое слово `abstract`. — *Прим. науч. ред.*

```
type BLTSandwich() =
    inherit Sandwich()

    override this.Ingredients = ["Bacon"; "Lettuce"; "Tomato"]
    override this.Calories = 330

type TurkeySwissSandwich() =
    inherit Sandwich()

    override this.Ingredients = ["Turkey"; "Swiss"]
    override this.Calories = 330
```

Для явного вызова метода базового класса необходимо использовать ключевое слово base. В примере 5.17 определяется новый класс BLTWithPickleSandwich, который обращается к базовому классу при выводе списка ингредиентов и для увеличения значения калорий.

Пример 5.17. Обращение к базовому классу

```
> // Сэндвич бекон-салат-помидор (BLT) с добавлением соленого огурца
type BLTWithPickleSandwich() =
    inherit BLTSandwich()

    override this.Ingredients = "Pickles" :: base.Ingredients
    override this.Calories = 50 + base.Calories;;

type BLTWithPickleSandwich =
  class
    inherit BLTSandwich
    new : unit -> BLTWithPickleSandwich
    override Calories : int
    override Ingredients : string list
  end

> let lunch = new BLTWithPickleSandwich();;

val lunch : BLTWithPickleSandwich

> lunch.Ingredients;;
val it : string list = ["Pickles"; "Bacon"; "Lettuce"; "Tomato"]
```

Категории классов

При наличии возможности создавать иерархии классов вам необходимо выбрать, что делать с классами, находящимися выше в дереве иерархии, и с классами, находящимися в самом низу иерархии.[1]

[1] Поскольку стандартная нотация изображения иерархии наследования подразумевает расположение базовых классов выше производных, то под понятиями «выше» и «ниже» следует понимать классы базовые или производные относительно текущего класса. – *Прим. науч. ред.*

Узлы, расположенные в дереве выше, могут быть представлены *абстрактными классами*, которые являются настолько обобщенными, что экземпляры таких классов не могут существовать в действительности. В предыдущем примере мы определили класс `Sandwich`, в котором предусмотрели реализацию по умолчанию для абстрактных членов. Однако не все абстрактные члены могут содержать осмысленную реализацию по умолчанию.

Продолжая наш пример с сэндвичами, можно предположить, что класс `Sandwich` мог бы наследовать класс `Food`, который в свою очередь наследует класс `System.Object`. Класс `Food` не может содержать разумную реализацию по умолчанию для своих методов, и поэтому они должны быть объявлены абстрактными.

Внизу дерева находятся настолько конкретные классы, что их невозможно специализировать сильнее. Хотя класс `BLTNoPicklesLightMayo` мог бы стать базовым классом для `BLTNoPicklesLightMayoOrganicLettuce`, в этом случае все равно нет никакой необходимости переопределять поведение `System.Object`.

В .NET вы можете специальным образом обозначить классы, находящиеся на самом верху и в самом низу иерархии, чтобы помочь понять, как эти классы использовать.

Абстрактные классы

Абстрактные классы (abstract classes) обычно являются корнем иерархии наследования и не являются самодостаточными. Фактически вы не можете создавать экземпляры классов, помеченных как абстрактные, так как в противном случае у вас появилась бы возможность вызвать метод, который объявлен, но не имеет реализации.

Для объявления абстрактного класса достаточно добавить атрибут `[<AbstractClass>]` к объявлению класса. В противном случае при наличии абстрактных методов, не имеющих реализации по умолчанию, компилятор выведет сообщения об ошибке.

```
> // ОШИБКА: Определение класса с членами, не имеющими реализации
type Foo() =
    member this.Alpha() = true
    abstract member Bravo : unit -> bool;;

  type Foo() =
  -----^^^

stdin(2,6): error FS0365: No implementation was given for 'abstract member
Foo.Bravo : unit -> bool'
```

(Для "abstract member Foo.Bravo : unit -> bool" не было дано реализации)

```
> // Правильно объявленный абстрактный класс
[<AbstractClass>]
type Bar() =
    member this.Alpha() = true
    abstract member Bravo : unit -> bool;;

type Bar =
  class
    new : unit -> Bar
    abstract member Bravo : unit -> bool
    member Alpha : unit -> bool
  end
```

Запечатанные классы

Если абстрактные классы – это классы, которые предназначены для того, чтобы от них наследовать, то *запечатанные классы (sealed classes)* – это классы, от которых нельзя наследоваться. Чтобы сделать класс запечатанным, достаточно добавить атрибут [<Sealed>]:

```
> // Определение запечатанного класса
[<Sealed>]
type Foo() =
    member this.Alpha() = true;;

type Foo =
  class
    new : unit -> Foo
    member Alpha : unit -> bool
  end

> // ОШИБКА: Попытка наследовать от запечатанного класс
type Bar() =
    inherit Foo()
    member this.Barvo() = false;;

      inherit Foo()
  ----^^^^^^^^^^^^^^

stdin(19,5): error FS0945: Cannot inherit a sealed type
```

(Не удается реализовать наследование от запечатанного типа)

Объявление класса запечатанным позволяет компилятору выполнить дополнительные оптимизации, потому что он знает, что виртуальные методы не могут быть переопределены. Кроме того, этот шаг ликвидирует некоторые проблемы безопасности, когда злонамеренный пользователь создает производный класс, используемый для аутентификации, и может сфальсифицировать права доступа.

Приведение типов

Важным преимуществом полиморфизма является возможность трактовать экземпляры производного класса как экземпляры базового класса. Так, в предыдущих примерах все классы в иерархии `Sandwich` имеют свойства `Calories` и `Ingredients`. Чтобы экземпляр одного типа привести к другому типу в пределах иерархии классов, необходимо использовать *оператор приведения типа (cast operator)*.

Приведение типа – это способ преобразования объектов из одного типа в другой. Существует две разновидности приведения типов – *статическое* и *динамическое*, в зависимости от того, в каком направлении (вверх или вниз) выполняется приведение в цепочке наследования.

Статическое приведение типа вверх

Под статическим приведением типа вверх (static upcasting) понимается приведение типа экземпляра производного класса к типу одного из базовых классов, то есть к типу, расположенному выше в иерархии наследования. Выполняется такое приведение с помощью оператора статического приведения типа `:>`. Результатом выполнения этого оператора является экземпляр указанного класса.

В примере 5.18 создается иерархия классов, где класс `Pomeranian` является производным от класса `Dog`, который в свою очередь является производным от класса `Animal`. Используя оператор статического приведения типа, мы можем преобразовать экземпляр класса `Pomeranian` в экземпляр класса `Dog` или `Animal`. Кроме того, экземпляр любого класса можно привести к типу `obj`.[1]

Пример 5.18. Статическое приведение типа

```
> // Определение иерархии классов
[<AbstractClass>]
type Animal() =
    abstract member Legs : int

[<AbstractClass>]
type Dog() =
    inherit Animal()
    abstract member Description : string
    override this.Legs = 4
```

[1] В отличие от большинства объектно-ориентированных языков программирования, в которых приведение от классов наследников к базовым выполняется неявно, в языке F# при использовании операторов связывания `let` необходимо явно применять оператор приведения типа (помимо тех случаев, когда используются гибкие (flexible) типы). – *Прим. науч. ред.*

```
type Pomeranian() =
    inherit Dog()
    override this.Description = "Furry";;

... вырезано ...

> let steve = new Pomeranian();;

val steve : Pomeranian

> // Приведение экземпляра steve к различным типам
let steveAsDog    = steve :> Dog
let steveAsAnimal = steve :> Animal
let steveAsObject = steve :> obj;;

val steveAsDog : Dog
val steveAsAnimal : Animal
val steveAsObject : obj
```

Динамическое приведение типа

Динамическое приведение типа (dynamic cast) выполняется, когда экземпляр базового типа приводится к производному типу, то есть к типу, расположенному ниже в иерархии наследования. (Поэтому такое приведение типа не может быть проверено компилятором статически.) Эта возможность позволяет вставить любую пробку, круглую или нет, в круглое отверстие, что может приводить к отказам во время выполнения.

Чаще всего динамическое приведение типа производится, когда имеется экземпляр класса `System.Object`, но при этом известно, что в действительности он является экземпляром некоторого другого, производного класса. Для динамического приведения типа используется оператор динамического приведения типа `:?>`. Продолжая предыдущий пример, мы можем привести значение `steveAsObj` типа `obj` к типу `Dog`, выполнив динамическое приведение типа:

```
> let steveAsObj = steve :> obj;;

val steveAsObj : obj

> let steveAsDog = steveAsObj :?> Dog;;

val steveAsDog : Dog

> steveAsDog.Description;;
val it : string = "Furry"
```

Если экземпляр преобразуется к типу, которым он в действительности не является, во время выполнения мы получим исключение `System.InvalidCastException`:

```
> let _ = steveAsObj :?> string;;
System.InvalidCastException: Unable to cast object of type 'Pomeranian' to type
'System.String'.
  at <StartupCode$FSI_0022>.$FSI_0022._main()
stopped due to error
```

(Не удалось привести тип объекта "Pomeranian" к типу "System.String". Остановлено из-за ошибки.)

Сопоставление с типами

Если динамическое приведение типа завершается неудачей, генерируется исключение, поэтому вы не можете проверить тип объекта, не выполнив дополнительную обработку исключений. К счастью, существует возможность упростить динамическую проверку типа, воспользовавшись оператором `:?` с сопоставлением с образцом.

В примере 5.19 демонстрируется функция `whatIs`, которая принимает объект и сопоставляет его с несколькими известными типами. Если тип оказывается неизвестным, она вызывает метод `GetType` класса `System.Object` и выводит имя типа на консоль.

Важно заметить, что часть `as` *идентификатор*, которая следует после динамической проверки типа в выражении сопоставления, связывает новое значение с проверяемым типом. То есть в этом примере значение `s` будет иметь тип `string`, а не `obj`.

Пример 5.19. Проверка типа с помощью выражения сопоставления с образцом

```
> // Сопоставление с типами
let whatIs (x : obj) =
    match x with
    | :? string    as s -> printfn "x is a string \"%s\"" s
    | :? int       as i -> printfn "x is an int %d" i
    | :? list<int> as l -> printfn "x is an int list '%A'" l
    | _ -> printfn "x is a '%s'" <| x.GetType().Name;;

val whatIs : obj -> unit

> whatIs [1 .. 5];;
x is an int list '[1; 2; 3; 4; 5]'
val it : unit = ()
> whatIs "Rosebud";;
x is a string "Rosebud"
val it : unit = ()
> whatIs (new System.IO.FileInfo(@"C:\config.sys"));;
x is a 'FileInfo'
val it : unit = ()
```

Хотя в ООП есть еще много чего, о чем можно было бы рассказать, и особенно об объектно-ориентированном проектировании, тем не менее теперь вы уже знаете, как создавать классы и использовать наследование для организации и повторного использования кода. Эти знания позволят вам писать код, использующий сильные стороны языка F# – анализ данных, численные алгоритмы и так далее – и готовый к интеграции с другими объектно-ориентированными языками, такими как C#.

6

Программирование на платформе .NET

Платформа .NET – это универсальная платформа программирования, которая может предоставить больше, чем только функциональное и объектно-ориентированное программирование. В этой главе рассматриваются некоторые дополнительные концепции, доступные при использовании платформы .NET и дающие дополнительные преимущества при программировании на языке F#.

В этой главе вы узнаете, как использовать интерфейсы для построения лучших абстракций, перечисления и структуры, способные повысить производительность, и как использовать объектные выражения языка F# для ускорения разработки.

Платформа .NET

Начнем наше знакомство с особенностями .NET с исследования среды времени выполнения, в которой выполняются .NET-приложения. Это поможет вам понять, как с максимальной эффективностью использовать преимущества платформы, применяя особенности языка F#.

CLI

Основой платформы .NET является спецификация общеязыковой инфраструктуры (Common Language Infrastructure, CLI). Это спецификация среды времени выполнения, которая фактически выполняет ваш код F#. Компилятор F# генерирует двоичный файл, который называется *сборкой* (*assembly*) и содержит инструкции на высокоуровневом языке ассемблера *MSIL* (Microsoft Intermediate Language – промежуточный язык компании Microsoft).

Реализация CLI компилирует инструкции MSIL в машинный код во время выполнения программы, чтобы обеспечить более высокую ско-

рость работы, чем при интерпретации. Компиляция выполняется динамически, на лету («just-in-time», или JIT).

Код, который компилируется в MSIL и затем выполняется JIT-компилятором, называется *управляемым кодом (managed code)*, в противоположность *неуправляемому коду (unmanaged code)*, который является обычным машинным кодом (как правило, написанным на языке C или C++).

Выполнение управляемого кода с помощью CLI имеет несколько преимуществ по сравнению с компиляцией кода на языке F# непосредственно в машинный код:

Взаимодействие с другими языками

Без обобщенного языка (MSIL) было бы намного сложнее обеспечить возможность совместного использования одного и того же кода в проектах на языках C#, VB.NET и F#.

Кроссплатформенность

Благодаря тому что спецификация CLI может быть реализована в любой операционной системе, появляется возможность выполнять код .NET на других платформах, таких как Microsoft Silverlight, на приставках X-Box с установленными XNA и .NET Compact Framework[1] и даже в Linux с помощью проекта Mono.

Независимость от аппаратной архитектуры

Компиляция в машинный код производится JIT-компилятором уже во время работы программы, поэтому вам не нужно заботиться о том, чтобы скомпилировать исходный код под определенную аппаратную архитектуру, такую как x86 или x64.

Кроме того, платформа .NET предлагает механизм автоматической сборки мусора. Это освобождает программиста от необходимости следить за выделением памяти и своевременным ее освобождением, благодаря чему (практически полностью) ликвидируется целый класс ошибок.

О большей части преимуществ вам не придется даже задумываться, потому что они «просто работают». Но понимание одного из аспектов программирования в .NET совершенно необходимо – вы должны знать, как действует механизм сборки мусора.

Сборка мусора

Сборка мусора – это возможность среды времени выполнения .NET автоматически освобождать память, когда она становится не нужна. Как только значение становится недоступным из кода, например после вы-

[1] XNA – это набор инструментов с управляемой средой времени выполнения (.NET), созданный Microsoft для облегчения разработки и управления компьютерными играми. – *Прим. перев.*

хода из области видимости функции, память, занимаемая объектом, помечается как доступная для освобождения и будет позднее освобождена сборщиком мусора. Например, если функция объявляет значение с именем x, то как только функция завершит работу, не останется никакой возможности обратиться к значению x, поэтому сборщик мусора может безопасно освободить эту память.

Как программисту вам, как правило, не придется беспокоиться о том, можно ли освобождать тот или иной блок памяти: сборщик мусора сам позаботится об этом.

Однако в управляемом коде все же возможны утечки памяти. Если вы по ошибке сохраните ссылку на какой-либо объект, который в действительности больше не нужен, то сборщик мусора не сможет освободить занимаемую объектом память, потому что остается возможность обратиться к этой области памяти позднее.

Сборщик мусора отлично справляется с управлением памятью, занимаемой управляемыми ресурсами, то есть ресурсами, которые создаются и используются CLI. Но проблема управления памятью осложняется, когда приходится иметь дело с неуправляемыми ресурсами, в число которых входит практически все, что существует за пределами среды CLI, например дескрипторы файлов, дескрипторы ресурсов операционной системы, разделяемая память и так далее. Управление этими ресурсами должно осуществляться явно, и освобождаться они должны самим программистом.

Если по каким-то причинам вам приходится вручную выделять и освобождать ресурсы, рекомендуется использовать интерфейс IDisposable. (Интерфейсы будут рассматриваться ниже в этой главе.) Интерфейс IDisposable содержит всего один метод, Dispose, который освобождает ресурсы. В своей реализации метода Dispose вы можете предусмотреть выполнение всех необходимых операций по освобождению ресурсов, таких как закрытие файла:

```
type IDisposable =
    interface
        abstract member Dispose : unit -> unit
    end
```

Но необходимость помнить о вызове метода Dispose представляет собой определенную проблему. Фактически этот способ точно так же может приводить к ошибкам, как ручное управление памятью. К счастью, в языке F# есть синтаксический сахар, который может придти на помощь.

При использовании ключевого слова use компилятор F# автоматически будет освобождать ресурсы, как только работа с ними будет закончена, то есть после выхода из области видимости. Синтаксически ключевое слово use эквивалентно ключевому слову let.

В следующем примере определяется новый класс с именем MultiFileLogger, который реализует интерфейс IDisposable и выводит некоторый текст в консоль при вызове метода Dispose. Поскольку значение logger создано с помощью ключевого слова use, то, как только поток управления выходит за пределы области видимости этого значения, например по завершении функции, автоматически вызывается метод Dispose:[1]

```
> // Реализация интерфейса IDisposable
open System
open System.IO
open System.Collections.Generic

type MultiFileLogger() =
    do printfn "Constructing..."
    let m_logs = new List<StreamWriter>()

    member this.AttachLogFile file =
        let newLogFile = new StreamWriter(file, true)
        m_logs.Add(newLogFile)

    member this.LogMessage (msg : string) =
        m_logs |> Seq.iter (fun writer -> writer.WriteLine(msg))

    interface IDisposable with
        member this.Dispose() =
            printfn "Cleaning up..."
            m_logs |> Seq.iter (fun writer -> writer.Close())
            m_logs.Clear();;

type MultiFileLogger =
  class
    interface IDisposable
    new : unit -> MultiFileLogger
    member AttachLogFile : file:string -> unit
    member LogMessage : msg:string -> unit
  end

> // Некоторый программный код, использующий MultiFileLogger
let task1() =
    use logger = new MultiFileLogger()
    // ...
    printfn "Exiting the function task1.."
    ();;
```

[1] Для более подробных сведений об особенностях работы сборщика мусора и управлении ресурсами см. книгу Джеффри Рихтера «CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#», Питер, Русская редакция, 2007. – *Прим. науч. ред.*

```
val task1 : unit -> unit

> task1();;
Constructing...
Exiting the function task1..
Cleaning up...
val it : unit = ()
```

Интерфейсы

Рассматривая объектно-ориентированное программирование, мы моделировали несколько различных видов взаимоотношений. Отношения вида *является (is a relationship)* моделируются с помощью наследования, а отношения вида *имеет (has a relationship)* – посредством агрегирования (полей и свойств). Например, BMW *является* автомобилем (наследует тип `Car`) и *имеет* двигатель (содержит поле с именем `m_motor` типа `Engine`).

Однако существует еще один, третий тип отношений – отношения вида *может выполнять (can do relationship)*, когда тип `X` может выполнять операции, описываемые типом `Y`. Например, люди, автомобили и вино – все они имеют возраст. Несмотря на то, что между людьми, автомобилями и вином мало общего, тем не менее все они способны стареть.

В программировании для .NET отношения вида *может выполнять* моделируются с помощью *интерфейсов*, определяющих наборы правил, которые может реализовать тот или иной тип, чтобы обеспечить выполнение определенного набора операций.

Интерфейс – это просто набор методов и свойств. Тип может объявить, что он реализует интерфейс, если содержит все методы и свойства этого интерфейса. Реализовав интерфейс, тип *сможет выполнять* все, что предусмотрено интерфейсом.

В примере 6.1 объявлено несколько типов, реализующих интерфейс `IConsumable`. Чтобы реализовать интерфейс, необходимо использовать ключевое слово `interface` с именем интерфейса и вслед за ним добавить реализацию всех методов и свойств интерфейса. Если класс реализует интерфейс `IConsumable`, это означает, что он имеет свойство `Tastiness` и метод `Eat`.

Пример 6.1. Интерфейсы в языке F#

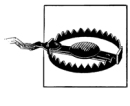
```
type Tastiness =
    | Delicious
    | SoSo
    | TrySomethingElse

type IConsumable =
    abstract Eat : unit -> unit
    abstract Tastiness : Tastiness
```

```
// Совет: ешьте по одному каждый день
type Apple() =
    interface IConsumable with
        member this.Eat() = printfn "Tastey!"
        member this.Tastiness = Delicious

// Это не так вкусно, но голод, как известно, не тетка
type CardboardBox() =
    interface IConsumable with
        member this.Eat() = printfn "Yuck!"
        member this.Tastiness = TrySomethingElse
```

В языке C# существует возможность реализовать интерфейс лишь частично и предоставить производным классам недостающую реализацию. В языке F# такая возможность отсутствует. Если тип объявляет, что он реализует интерфейс, он должен реализовать все методы и свойства.

Использование интерфейсов

В отличие от других языков .NET, в языке F# методы и свойства, реализующие интерфейсы, по умолчанию находятся вне области видимости, поэтому для вызова какого-либо метода интерфейса необходимо привести объект к типу соответствующего интерфейса. Получив экземпляр интерфейса, его можно использовать только для выполнения операций, предусмотренных интерфейсом, – ни больше, ни меньше.



Необходимость приводить типы объектов на языке F# к интерфейсам, прежде чем вызывать члены интерфейса, часто является источником непонимания при работе с существующим кодом .NET. Однако это требование делает более очевидным, какая функциональность используется – интерфейса или типа.

```
> let apple = new Apple();;

val apple : Apple

> apple.Tastiness;;

  apple.Tastiness;;
  ------^^^^^^^^^
stdin(81,7): error FS0039: The field, constructor or member 'Tastiness' is not
defined.
```

(Поле, конструктор или элемент "Tastiness" не определен.)

```
> let iconsumable = apple :> IConsumable;;

val iconsumable : IConsumable
```

```
> iconsumable.Tastiness;;
val it : Tastiness = Delicious
```

Определение интерфейсов

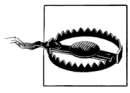
Чтобы создать интерфейс, достаточно просто определить новый класс, содержащий только абстрактные методы. Благодаря этому механизм вывода типов определит, что данный тип является интерфейсом. При желании можно использовать ключевое слово `interface`, за которым указать абстрактные методы и свойства и завершить определение ключевым словом `end`.

В коде для платформы .NET принято давать интерфейсам имена, начинающиеся с заглавной буквы `I`. Это пережиток Венгерской нотации, согласно которой имена переменных должны начинаться с префикса, обозначающего тип данных, хранящихся в переменной, или назначение этой переменной. Например, переменная, содержащая целочисленное значение с номером столбца, могла бы называться `colX` или `nX` вместо неоднозначного `x`.

Поскольку код для .NET меньше полагается на отслеживание примитивов машинного уровня и больше – на абстракции, в руководствах по программированию для платформы .NET не рекомендуется использовать подобный стиль именования. Однако интерфейсы являются исключениями из этого правила.

```
> // Объявление типа, который автоматически определяется как интерфейс
type IDoStuff =
    abstract DoThings : unit -> unit;;

// Явное определение интерфейса
type IDoStuffToo =
    interface
        abstract member DoThings : unit -> unit
    end;;

type IDoStuff =
  interface
    abstract member DoThings : unit -> unit
  end

type IDoStuffToo =
  interface
    abstract member DoThings : unit -> unit
  end
```

Интерфейсы могут наследовать друг друга, при этом производный интерфейс расширяет перечень методов и свойств базового интерфейса. Однако классы, реализующие этот интерфейс, должны реализовать всю цепочку наследования, реализуя каждый интерфейс полностью.



Это еще одно отличие интерфейсов в языке F# от интерфейсов в языке C#. Необходимость явной реализации базового интерфейса при реализации производного интерфейса может показаться странной, но благодаря этому требованию обеспечивается более высокий уровень очевидности языка F#. Чем меньше компилятор будет вынужден догадываться о чем-либо, тем меньше сюрпризов вас ожидает.

В примере 6.2 определяются два интерфейса, `IDoStuff` и `IDoMoreStuff`, причем интерфейс `IDoMoreStuff` наследует интерфейс `IDoStuff`. Если попытаться определить класс, реализующий производный интерфейс, но не имеющий реализации базового интерфейса, компилятор выдаст сообщение об ошибке.

Пример 6.2. Реализация производного интерфейса

```
> // Наследование интерфейсов
type IDoStuff =
    abstract DoStuff : unit -> unit

type IDoMoreStuff =
    inherit IDoStuff

    abstract DoMoreStuff : unit -> unit;;

... вырезано ...

> // ОШИБКА: Реализована не вся иерархия интерфейсов
type Foo() =
    interface IDoMoreStuff with
        override this.DoMoreStuff() = printfn "Stuff getting done...";;

      interface IDoMoreStuff with
  ----^
stdin(116,5): error FS0366: No implementation was given for 'abstract member
IDoStuff.DoStuff : unit -> unit'. Note that all interface members must be
implemented and listed under an appropriate 'interface' declaration,
e.g. 'interface ... with member ...'
```

(Для "abstract member IDoStuff.DoStuff : unit -> unit" не было дано реализации. Обратите внимание на то, что все элементы интерфейса должны быть реализованы и перечислены в соответствующем объявлении "interface", например, "interface ... with member ...".)

```
> // Работающий вариант
type Bar() =
    interface IDoStuff with
        override this.DoStuff() = printfn "Stuff getting done..."

    interface IDoMoreStuff with
        override this.DoMoreStuff() = printfn "More stuff getting done...";;
```

```
type Bar =
  class
    interface IDoMoreStuff
    interface IDoStuff
    new : unit -> Bar
  end
```

Объектные выражения

Интерфейсы играют важную роль в .NET, но иногда бывает необходимо просто реализовать некоторый интерфейс без определения нового класса. Например, чтобы отсортировать элементы коллекции типа `List<_>`, необходимо определить тип, реализующий интерфейс `IComparer<'a>`. Как только начнет появляться необходимость сортировать эту коллекцию типа `List<_>` различными способами, вы быстро обнаружите, что код загромождается типами, которые не делают ничего, кроме реализации единственного метода сравнения двух объектов.

В языке F# существует возможность использовать *объектные выражения (object expressions)*, которые создают анонимные классы и возвращают их экземпляры. (Термин «анонимный класс» означает лишь то, что компилятор сгенерирует класс, имя которого для вас останется неизвестным.) Это упрощает создание одноразовых типов аналогично тому, как лямбда-выражения упрощают создание одноразовых функций.

Синтаксис создания объектного выражения представляет собой пару фигурных скобок `{ }`, начинается с ключевого слова `new`, за которым следует имя интерфейса, объявляемого точно так же, как при использовании в классах. Результатом объектного выражения является экземпляр анонимного класса.

Реализация интерфейсов с помощью объектных выражений

В примере 6.3 демонстрируются использование объектных выражений и реализация интерфейса `IComparer<'a>`, которая применяется для сортировки элементов коллекции. Каждое объектное выражение определяет свой способ сортировки списка имен. Без использования объектных выражений потребовалось бы создать два отдельных типа, каждый из которых содержал бы реализацию интерфейса `IComparer<'a>`. Но благодаря использованию объектных выражений отпала необходимость в явном создании новых типов.

Пример 6.3. Сортировка списка с использованием интерфейса IComparer<'a>

```
open System.Collections.Generic

type Person =
    { First : string; Last : string }
```

```
    override this.ToString() = sprintf "%s, %s" this.Last this.First

let people =
    new List<_>(
        [|
            { First = "Jomo";  Last = "Fisher" }
            { First = "Brian"; Last = "McNamara" }
            { First = "Joe";   Last = "Pamer" }
        |] )

let printPeople () =
    Seq.iter (fun person -> printfn "\t %s" (person.ToString())) people

// Теперь отсортировать по фамилии

printfn "Initial ordering:"
printPeople()

// Сортировка по имени
people.Sort(
    {
        new IComparer<Person> with
            member this.Compare(l, r) =
                if   l.First > r.First then  1
                elif l.First = r.First then  0
                else                        -1
    } )

printfn "After sorting by first name:"
printPeople()

// Сортировать по фамилии
people.Sort(
    {
        new IComparer<Person> with
        member this.Compare(l, r) =
            if   l.Last > r.Last then  1
            elif l.Last = r.Last then  0
            else                      -1
    } )

printfn "After sorting by last name:"
printPeople()
```

Ниже приводится результат выполнения этого примера в окне FSI:

```
Initial ordering:
         Fisher, Jomo
         McNamara, Brian
         Pamer, Joe
After sorting by first name:
```

```
         McNamara, Brian
         Pamer, Joe
         Fisher, Jomo
After sorting by last name:
         Fisher, Jomo
         McNamara, Brian
         Pamer, Joe
```

Создание производных классов с помощью объектных выражений

Помимо реализации интерфейсов, объектные выражения могут использоваться для создания производных классов. Однако классы, объявляемые с помощью объектных выражений, не могут добавлять новые методы или свойства – они могут только переопределять абстрактные члены базового типа.

В примере 6.4 объектное выражение используется для создания нового экземпляра абстрактного класса `Sandwich` без явного определения нового типа.

Пример 6.4. Создание производных классов в виде объектных выражений

```
> // Абстрактный класс
[<AbstractClass>]
type Sandwich() =
    abstract Ingredients : string list
    abstract Calories : int

// Объектное выражение производного класса
let lunch =
    {
        new Sandwich() with
        member this.Ingredients = ["Peanutbutter"; "Jelly"]
        member this.Calories = 400
    };;

type Sandwich =
  class
    abstract member Calories : int
    abstract member Ingredients : string list
    new : unit -> Sandwich
  end
val lunch : Sandwich

> lunch.Ingredients;;
val it : string list = ["Peanutbutter"; "Jelly"]
```

Объектные выражения особенно удобны при создании модульных тестов. В тестах вам часто приходится создавать *мок-объекты (mock objects)*, то есть классы, имитирующие поведение, которое сложно изолировать,

например создание мок-объектов подключения к базе данных, чтобы тесты не обращались к реальной базе данных. Объектные выражения позволяют легко создавать анонимные типы, которые могут играть роль мок-объектов, без необходимости определять множество дополнительных типов.

Методы расширения

Иногда при работе с некоторым типом отсутствует возможность изменить его, либо потому что нет доступа к исходному коду, либо потому что приходится использовать старую версию. В подобных ситуациях бывает необходимо хоть как-то расширить некоторый тип.

Методы расширения (extension methods) предоставляют простой механизм, который позволяет расширять возможности типов без изменения иерархии типов или изменения существующих типов. Методы расширения – это специализированная разновидность методов, которые можно добавлять к существующим типам и использовать их, как если бы они были реальными членами этих типов. Благодаря этому отпадает необходимость пересобирать библиотеки или создавать новые производные типы ради добавления нескольких методов. После добавления методов расширения появляется возможность использовать типы со всеми дополнительными методами и свойствами, даже если они изначально отсутствовали в этих типах.

Объявление метода расширения начинается с конструкции `type` *идентификатор* `with` с последующей реализацией метода, где *идентификатор* – это полное квалифицированное имя класса.

Обратите внимание, что методы расширения в действительности не являются частью класса, поэтому они не имеют доступа к закрытым или внутренним данным. Это всего лишь методы, которые могут вызываться, как если бы они были членами класса.

В примере 6.5 демонстрируется расширение типа `System.Int32` новым методом `ToHexString`.

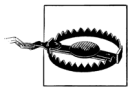
Пример 6.5. Использование метода расширения

```
> (1094).ToHexString();;

  (1094).ToHexString();;
  -------^^^^^^^^^^^^

stdin(131,8): error FS0039: The field, constructor or member 'ToHexString' is
not defined.
```

(Поле, конструктор или элемент "ToHexString" не определен)

```
> // Расширяем тип Int32 (он же тип int)
type System.Int32 with
    member this.ToHexString() = sprintf "0x%x" this;;

type Int32 with
  member ToHexString : unit -> string

> (1094).ToHexString();;
val it : string = "0x446"
```

Расширения типов могут определяться только в модулях и доступны только после того, как данный модуль будет открыт. Другими словами, те, кому необходимо воспользоваться методами расширения, должны открыть модули с соответствующими определениями.



Методы расширения, написанные на языке F#, не являются и не могут использоваться как методы расширения в языках C# и VB.NET. Методы расширения, определенные на языке F#, могут использоваться только в сборках, написанных на языке F#.

Расширение модулей

Модули в языке F# могут быть расширены за счет создания новых модулей с теми же именами. Пока пространство имен нового модуля будет открыто, программе будут доступны все новые функции, типы и значения, объявленные в модуле.

В следующем фрагменте создается новое пространство имен `FSCollectionExtensions`. Как только это пространство имен будет открыто, станут доступны два модуля с дополнительными функциями, расширяющими модули `List` и `Seq`:

```
namespace FSCollectionExtensions

open System.Collections.Generic

module List =

    /// Пропустить первые n элементов списка
    let rec skip n list =
        match n, list with
        | _, []       -> []
        | 0, list     -> list
        | n, hd :: tl -> skip (n - 1) tl

module Seq =

    /// Изменить порядок следования элементов последовательности на обратный
    let rec rev (s : seq<'a>) =
        let stack = new Stack<'a>()
```

```
s |> Seq.iter stack.Push
seq {
    while stack.Count > 0 do
        yield stack.Pop()
}
```

Перечисления

Мы уже познакомились с размеченными объединениями, которые удобно использовать для определения типов, экземпляры которых могут принимать ограниченный набор значений. Однако каждый вариант размеченного объединения – это совершенно иной класс, а кроме того, в некоторых ситуациях эта конструкция оказывается слишком тяжеловесной. Во многих случаях вполне достаточно иметь всего лишь некоторую группу взаимосвязанных констант с целочисленными значениями, и тогда можно использовать *перечисления*.

Перечисление – это простой целочисленный тип, такой как `int` или `sbyte`, содержащий группу именованных констант. Однако перечисление – это всего лишь обертка вокруг целочисленного типа, поэтому экземпляр перечисления может принимать значения, не определенные в пределах этого перечисления.

Создание перечислений

Для создания перечислений используется тот же синтаксис, что и для размеченных объединений, но при этом все варианты должны быть определены константами одного и того же типа. В примере 6.6 демонстрируется создание типа перечисления с перечнем шахматных фигур. Каждое значение в перечислении должно быть уникальным. В данном примере значения полей соответствуют ценностям шахматных фигур.

Пример 6.6. Объявление перечисления

```
type ChessPiece =
    | Empty  = 0
    | Pawn   = 1
    | Knight = 3
    | Bishop = 4
    | Rook   = 5
    | Queen  = 8
    | King   = 1000000
```

Чтобы воспользоваться значением из перечисления, достаточно просто указать полное квалифицированное имя поля. В примере 6.7 выполняется инициализация массива 8×8, представляющего размещение фигур на шахматной доске.

Наше перечисление `ChessPiece` в текущем своем виде не позволяет отличать белые и черные фигуры. Однако благодаря тому, что каждое

значение в перечислении ChessPiece является простым целым числом, мы можем схитрить и для черных фигур использовать положительные значения ChessPiece, а для белых – отрицательные. То есть значение -1 * ChessPiece.Bishop представляет фигуру белого слона. Подробнее о механизме преобразования значений перечисления будет рассказываться в следующем разделе.

Пример 6.7. Инициализация массива шахматной доски значениями перечисления

```
/// Создать 2-мерный массив со значениями перечисления ChessPiece
let createChessBoard() =
    let board = Array2D.init 8 8 (fun __ -> ChessPiece.Empty)

    // Расстановка пешек
    for i = 0 to 7 do
        board.[1,i] <- ChessPiece.Pawn
        board.[6,i] <- enum<ChessPiece> (-1 * int ChessPiece.Pawn)

    // Расстановка черных фигур
    [| ChessPiece.Rook; ChessPiece.Knight; ChessPiece.Bishop;
       ChessPiece.Queen; ChessPiece.King; ChessPiece.Bishop;
       ChessPiece.Knight; ChessPiece.Rook |]
    |> Array.iteri(fun idx piece -> board.[0,idx] <- piece)

    // Расстановка белых фигур
    [| ChessPiece.Rook; ChessPiece.Knight; ChessPiece.Bishop; ChessPiece.King;
       ChessPiece.Queen; ChessPiece.Bishop; ChessPiece.Knight; ChessPiece.Rook
    |]
    |> Array.iteri(fun idx piece ->
                       board.[7,idx] <- enum<ChessPiece> (-1 * int piece))
    // Вернуть шахматную доску с фигурами
    Board
```

Перечисления могут использоваться в выражениях сопоставления с образцом, но в отличие от размеченных объединений каждое поле перечисления должно указываться в виде полностью квалифицированного имени:

```
let isPawn piece =
    match piece with
    | ChessPiece.Pawn
        -> true
    | _ -> false
```

Преобразование

Чтобы создать значение перечисления, достаточно просто воспользоваться функцией enum<_>. Эта функция преобразует свой аргумент в перечисление, соответствующее параметру обобщенного типа.

Ниже демонстрируется преобразование значения 42 в экземпляр ChessPiece:

```
let invalidPiece = enum<ChessPiece>(42)
```

Чтобы получить значение перечисления, достаточно использовать обычную функцию преобразования типа, такую как int:

```
let materialValueOfQueen = int ChessPiece.Queen
```

Когда использовать перечисления, а когда размеченные объединения

Перечисления и размеченные объединения имеют два существенных отличия. Во-первых, перечисления не дают гарантий безопасности. Размеченные объединения, напротив, гарантируют, что экземпляры могут иметь только одно из допустимых значений. Перечисления – это всего лишь синтаксический сахар над простыми типами. Поэтому получая экземпляр перечисления, нельзя быть полностью уверенным, что он имеет значение, присутствующее в перечислении.

Например, если значение –711 преобразовать в экземпляр ChessPiece, этот экземпляр не будет иметь какой-либо смысл в терминах этого перечисления и наверняка станет источником ошибок. Когда вы получаете экземпляр перечисления из внешнего источника, обязательно проверьте его с помощью метода Enum.IsDefined:

```
> System.Enum.IsDefined(typeof<ChessPiece>, int ChessPiece.Bishop);;
val it : bool = true
> System.Enum.IsDefined(typeof<ChessPiece>, -711);;
val it : bool = false
```

Во-вторых, перечисления могут содержать только свое значение, тогда как каждый элемент данных размеченного объединения может хранить уникальный кортеж с данными.

С другой стороны, перечисления представляют общую идиому .NET и обеспечивают более высокую производительность по сравнению с размеченными объединениями. Экземпляр перечисления – это всего лишь несколько байтов в стеке, тогда как размеченное объединение – это ссылочный тип, и для размещения его экземпляров требуется выделение отдельного блока памяти. Поэтому заполнение массива большим количеством перечислений будет выполняться намного быстрее, чем заполнение того же массива размеченными объединениями.

Одной из удобных особенностей перечислений является возможность определять битовые флаги. Взгляните на пример 6.8. Определив значения перечисления в виде степеней двойки, вы можете интерпретировать перечисление как набор битовых флагов. Затем вы можете использовать оператор битового «ИЛИ», |||, для объединения значений и использовать маски для проверки установленных битов. Используя этот

прием, можно определить до 32 флагов для перечислений на основе int и до 64 флагов для перечислений на основе int64.

Пример 6.8. Объединение значений перечислений

```
open System

// Перечисление с флаговыми значениями
type FlagsEnum =
    | OptionA = 0b0001
    | OptionB = 0b0010
    | OptionC = 0b0100
    | OptionD = 0b1000

let isFlagSet (enum : FlagsEnum) (flag : FlagsEnum) =
    let flagName = Enum.GetName(typeof<FlagsEnum>, flag)
    if enum &&& flag = flag then
        printfn "Flag [%s] is set." flagName
    else
        printfn "Flag [%s] is not set." flagName
```

Ниже приводятся результаты использования определений из примера 6.8 в окне FSI:

```
> // Проверка флага
let customFlags = FlagsEnum.OptionA ||| FlagsEnum.OptionC

isFlagSet customFlags FlagsEnum.OptionA
isFlagSet customFlags FlagsEnum.OptionB
isFlagSet customFlags FlagsEnum.OptionC
isFlagSet customFlags FlagsEnum.OptionD
;;
Flag [OptionA] is set.
Flag [OptionB] is not set.
Flag [OptionC] is set.
Flag [OptionD] is not set.

val customFlags : FlagsEnum = 5
```

Структуры

Ранее мы познакомились с классами и рассмотрели сложности, связанные с переопределением метода Equals для определения семантики эквивалентности. Однако для простых типов существует более удобный способ. Почему бы вместо того чтобы пытаться имитировать семантику эквивалентности значимых типов путем переопределения метода Equals, не создать новый значимый тип? *Структуры* похожи на классы; главное их отличие состоит в том, что они являются значимыми типами, и потому экземпляры структур хранятся в стеке. Экземпляры структур занимают меньше памяти, чем экземпляры классов, и при размещении в стеке не требуют сборки мусора.

Создание структур

Чтобы создать структуру, необходимо определить тип, как это обычно делается, и добавить атрибут [<Struct>] или вставить ключевые слова struct и end до и после тела структуры:

```
[<Struct>]
type StructPoint(x : int, y : int) =
    member this.X = x
    member this.Y = y

type StructRect(top : int, bottom : int, left : int, right : int) =
    struct
        member this.Top = top
        member this.Bottom = bottom
        member this.Left = left
        member this.Right = right

        override this.ToString() =
            sprintf "[%d, %d, %d, %d]" top bottom left right
    end
```

Структуры обладают многими особенностями классов, но при этом могут храниться в стеке, а не в куче. Благодаря этому операция проверки эквивалентности (по умолчанию) выполняется в виде сравнения значений на стеке, а не в виде сравнения ссылок:

```
> // Определение двух различных экземпляров структуры
let x = new StructPoint(6, 20)
let y = new StructPoint(6, 20);;

val x : StructPoint = FSI_0011+StructPoint
val y : StructPoint = FSI_0011+StructPoint

> x = y;;
val it : bool = true
```

Одно из различий между структурами и классами заключается в способе их создания. Классы содержат только те конструкторы, которые вы определите вручную, тогда как структуры автоматически получают *конструктор по умолчанию*, который присваивает значения по умолчанию каждому полю структуры. (Если вы помните, экземпляры значимых типов по умолчанию инициализируются нулями, а экземпляры ссылочных типов – значениями null.)

```
> // Структура с описанием книги
[<Struct>]
type BookStruct(title : string, pages : int) =
    member this.Title = title
    member this.Pages = pages
```

```
    override this.ToString() =
        sprintf "Title: %A, Pages: %d" this.Title this.Pages;;

type BookStruct =
  struct
    new : title:string * pages:int -> BookStruct
    override ToString : unit -> string
    member Pages : int
    member Title : string
  end

> // Создать экземпляр структуры с помощью конструктора
let book1 = new BookStruct("Philosopher's Stone", 309);;

val book1 : BookStruct = Title: "Philosopher's Stone", Pages: 309

> // Создать экземпляр с помощью конструктора по умолчанию
let namelessBook = new BookStruct();;

val namelessBook : BookStruct = Title: <null>, Pages: 0
```

Для создания структуры с изменяемыми полями необходимо явно объявить каждое изменяемое поле с помощью ключевого слова `val`**. Кроме того, сами экземпляры структуры также должны быть объявлены изменяемыми.**

В следующем фрагменте определяется структура `MPoint` **с двумя изменяемыми полями. Затем создается изменяемый экземпляр** `pt` **этой структуры и выполняется изменение полей:**

```
> // Определение структуры с изменяемыми полями
[<Struct>]
type MPoint =
    val mutable X : int

    val mutable Y : int

    override this.ToString() =
        sprintf "{%d, %d}" this.X this.Y;;

type MPoint =
  struct
    val mutable X: int
    val mutable Y: int
    override ToString : unit -> string
  end

> let mutable pt = new MPoint();;

val mutable pt : MPoint = {0, 0}
```

```
> // Изменение полей
pt.X <- 10
pt.Y <- 7;;
> pt;;
val it : MPoint = {10, 7}
> let nonMutableStruct = new MPoint();;

val nonMutableStruct : MPoint = {0, 0}

> // ОШИБКА: Попытка изменить поле неизменяемого экземпляра структуры
nonMutableStruct.X <- 10;;

  nonMutableStruct.X <- 10;;
  ^^^^^^^^^^^^^^^^
stdin(54,1): error FS0256: A value must be mutable in order to mutate the
contents of a value type, e.g. 'let mutable x = ...'
```

(Чтобы изменить содержимое или получить адрес типа значения, значение должно быть изменяемым, например, "let mutable x = ...")

Ограничения

Структуры имеют несколько ограничений, связанных с их характеристиками:

- Структуры не могут наследоваться – они неявно помечены атрибутом [<Sealed>].
- Структуры не могут переопределять конструктор по умолчанию (потому что он существует всегда и всегда обнуляет память, занимаемую экземпляром структуры).
- Структуры не могут использовать операторы связывания let в определениях полей, как это делается в неявных конструкторах классов.

Когда использовать структуры, а когда записи

Язык F# предлагает два легковесных контейнера данных – структуры и записи, поэтому иногда могут возникать затруднения в выборе того или иного типа варианта.

Главное преимущество структур перед записями такое же, что и перед классами, а именно – иные показатели производительности. При работе с большим количеством небольших объектов выделение памяти в стеке для создания новых структур выполняется намного быстрее, чем выделение памяти в куче для того же количества объектов. Кроме того, экземпляры структур не нуждаются в сборке мусора, потому что стек очищается автоматически при завершении работы функции.

Однако нередко выигрыш в производительности при использовании структур оказывается незначительным. Фактически необдуманное использование структур может даже приводить к снижению произ-

водительности из-за необходимости копировать большие объемы при передаче их функциям в виде параметров. Кроме того, сборщик мусора и менеджер памяти в .NET оптимизированы для использования в высокопроизводительных приложениях. Если вам кажется, что у вас есть проблемы с производительностью, воспользуйтесь инструментами профилирования кода, имеющимися в составе Visual Studio, чтобы выявить узкие места, прежде чем преобразовывать все свои классы и записи в структуры.

7

Прикладное функциональное программирование

До сих пор мы в основном использовали функциональный стиль программирования. Он позволяет писать короткие и мощные программы, однако мы пока не полностью владеем всем потенциалом функционального стиля. Функциональное программирование означает значительно больше, чем просто отношение к функциям как к данным. Изучение парадигмы функционального программирования и формирование нового образа мышления позволит вам писать программы, которые сложно было бы выразить в императивном стиле.

В этой главе, опираясь на сведения о функциональном программировании, приводившиеся в главе 3, я представлю новые особенности языка, которые помогут вам повысить свою продуктивность при использовании функционального стиля. Например, использование активных шаблонов позволит вам расширить возможности выражений сопоставления с образцом и избавиться от конструкций `when`, а создание модулей, которые открываются автоматически, поможет вам расширить наиболее часто используемые модули языка F#.

Кроме того, мы рассмотрим некоторые более сложные аспекты функционального программирования. Вы узнаете, как использовать продвинутые формы рекурсии, позволяющие избежать переполнения стека, и как создавать более эффективные приложения с помощью списков. Дополнительно мы познакомимся с некоторыми основными шаблонами проектирования, применяемыми в функциональном программировании.

Наше исследование прикладного функционального программирования мы начнем с изучения того, как можно использовать мощные механизмы контроля и вывода типов в языке F# для устранения ошибок в вашем коде.

Единицы измерения

В этом мире существует несколько универсальных истин: ускорение свободного падения равно 9,8 метрам в секунду за секунду, температура кипения воды равна примерно 100 градусам Цельсия и тот факт, что любой программист, независимо от того, насколько он талантлив и внимателен, совершает ошибки, связанные с единицами измерения.

Если в будущем вам доведется писать код, работающий с реальными единицами измерения, рано или поздно вы обязательно допустите ошибку. Например, вы попытаетесь передать величину, выраженную в секундах, тогда как функция ожидает получить значение, выраженное в минутах, или спутаете ускорение со скоростью. Иногда такие ошибки могут вызывать легкое раздражение, а иногда – стоить жизни.

Проблема заключается в том, что если представить значение просто в виде вещественного числа, у вас не будет никакой дополнительной информации о том, какой смысл имеет это значение. Если я передам вам число `9.8`, вы не сможете сказать наверняка, что оно означает – мили, метры в секунду, часы или может быть даже мегабайты.

В языке F# имеется мощная возможность, позволяющая решать эти проблемы, которая называется *единицы измерения (units of measure)*. Единицы измерения позволяют передавать информацию о физическом смысле вместе с вещественными значениями – `float`, `float32`, `decimal` – или со знаковыми целыми числами и тем самым предотвратить появление целого класса ошибок. Рассмотрим пример 7.1, в котором описывается значение температуры. Обратите внимание, как определяется параметр `temp`, принимающий значения температуры только по шкале Фаренгейта (`fahrenheit`). Что означает запись `float<_>`, я опишу ниже в этом же разделе.

Пример 7.1. Преобразование значений температуры по шкале Фаренгейта в значения по шкале Цельсия

```
[<Measure>]
type fahrenheit

let printTemperature (temp : float<fahrenheit>) =

    if temp < 32.0<_> then
        printfn "Замерзаю!"
    elif temp < 65.0<_> then
        printfn "Холодновато"
    elif temp < 75.0<_> then
        printfn "В самый раз!"
    elif temp < 100.0<_> then
        printfn "Жарко!"
    else
        printfn "Адское пекло!"
```

Поскольку функция принимает только значения типа `fahrenheit`, она будет не в состоянии обрабатывать вещественные значения в других единицах измерения. Попытка передать функции значение в недопустимых единицах измерения приведет к ошибке на этапе компиляции (благодаря чему будет предотвращена возможность получения ошибочных результатов во время выполнения):

```
> let seattle = 59.0<fahrenheit>;;

val seattle : float<fahrenheit> = 59.0

> printTemperature seattle;;
Cold
val it : unit = ()
> // ОШИБКА: Другие единицы измерения
[<Measure>]
type celsius

let cambridge = 18.0<celsius>;;

[<Measure>]
type celsius
val cambridge : float<celsius> = 18.0

> printTemperature cambridge;;

  printTemperature cambridge;;
  -----------------^^^^^^^^^

stdin(18,18): error FS0001: Type mismatch. Expecting a
    float<fahrenheit>
but given a
    float<celsius>.
The unit of measure 'fahrenheit' does not match the unit of measure 'celsius'
```

(Несоответствие типов. Требуется float<fahrenheit>, но получен float<celsius>. Единица измерения "fahrenheit" не совпадает с единицей измерения "celsius".)

Единицы измерения могут также включать в себя операторы умножения или деления. Так, если разделить единицу измерения `meter` на другую единицу измерения, такую как `second`, в результате будет получена единица измерения `float<meter/second>`:

```
> // Определение единицы измерения "метры"
[<Measure>]
type m;;

[<Measure>]
type m

> // Умножение дает в результате квадратные метры
1.0<m> * 1.0<m>;;
```

```
val it : float<m ^ 2> = 1.0
> // Деление приводит к полному уничтожению единицы измерения
1.0<m> / 1.0<m>;;

val it : float = 1.0
> // Добавление еще одной операции деления даст единицу измерения 1/meters
1.0<m> / 1.0<m> / 1.0<m>;;

val it : float</m> = 1.0
```

Определение единиц измерения

Для определения единицы измерения достаточно просто добавить атрибут [<Measure>] перед объявлением типа. Типы единиц измерения могут содержать только статические методы и свойства и обычно определяются как *непрозрачные типы* (*opaque types*), то есть типы, не имеющие ни методов, ни свойств.

Иногда бывает чрезвычайно полезно добавлять функции, выполняющие преобразование из одних единиц измерения в другие. Следующий фрагмент определяет единицы измерения fahrenheit и celsius, аналогичные приведенным выше, но при этом добавляет дополнительные статические методы, выполняющие преобразования между ними. Обратите внимание на ключевое слово and, благодаря которому тип far знает о существовании типа cel, используемого в его объявлении:

```
> // Добавление методов в единицы измерения
[<Measure>]
type far =
    static member ConvertToCel(x : float<far>) =
        (5.0<cel> / 9.0<far>) * (x - 32.0<far>)

and [<Measure>] cel =
    static member ConvertToFar(x : float<cel>) =
        (9.0<far> / 5.0<cel> * x) + 32.0<far>;;

[<Measure>]
type far =
  class
    static member ConvertToCel : x:float<far> -> float<cel>
  end
[<Measure>]
and cel =
  class
    static member ConvertToFar : x:float<cel> -> float<far>
  end

> far.ConvertToCel(100.0<far>);;
val it : float<cel> = 37.77777778
```

Существует также возможность определять единицы измерения, связанные с другими единицами измерения. В примере 7.2 создается новая единица измерения `s`, секунды, и связанная с ней другая единица измерения `Hz`, герцы, которая используется для измерения количества колебаний в секунду. Поскольку одна единица измерения определяется относительно другой, два значения с одинаковой семантикой считаются эквивалентными.

Пример 7.2. Определение новых единиц измерения

```
> // Определение единиц измерения “секунды” и “герцы”
[<Measure>]
type s

[<Measure>]
type Hz = s ^ -1;;

[<Measure>]
type s
[<Measure>]
type Hz =/s

> // Если бы единица измерения Hz не была связана с s,
// следующая операция сравнения привела бы к ошибке на этапе компиляции.
3.0<s ^ -1> = 3.0<Hz>;;
val it : bool = true
```

Преобразование единиц измерения

Не все функции принимают значения с единицами измерения в виде параметров. Для преобразования значения с единицами измерения в значение базового типа достаточно передать исходное значение соответствующей функции преобразования. Пример 7.3 демонстрирует, как можно отбросить единицы измерения при вызове тригонометрических функций `sin` и `cos`, которые не принимают значения с единицами измерения.

Пример 7.3. Преобразование единиц измерения

```
> // Радианы
[<Measure>]
type rads;;

[<Measure>]
type rads

> let halfPI = System.Math.PI * 0.5<rads>;;

val halfPI : float<rads>

> // ОШИБКА: передача float<_> функции, принимающей float
sin halfPI;;
```

```
  sin halfPI;;
  ----^^^^^^^

stdin(7,5): error FS0001: The type 'float<rads>' does not match type 'float'.
```

(Тип "float<rads>" не совпадает с типом "float".)

```
> // Отбросить единицы измерения в значении halfPi
// для преобразования его из типа float<_> в тип to float
sin (float halfPI);;
val it : float = 1.0
```

Обобщенные единицы измерения

Полагаться на различные методы преобразования не всегда бывает удобно, особенно когда требуется создать обобщенную функцию. К счастью, вы можете позволить механизму вывода типов языка F# самому определять тип единицы измерения. Если отбросить тип единицы измерения и вместо него использовать конструкцию вида `float<_>`, механизм вывода типов определит, что значение типа является обобщенной единицей измерения:

```
> let squareMeter (x : float<m>) = x * x;;

val squareMeter : float<m> -> float<m ^ 2>

> let genericSquare (x : float<_>) = x * x;;

val genericSquare : float<'u> -> float<'u ^ 2>

> genericSquare 1.0<m/s>;;
val it : float<m ^ 2/s ^ 2> = 1.0
> genericSquare 9.0;;
val it : float = 81.0
```

Если вам потребуется создать тип, который будет обобщенным по отношению к единицам измерения, добавьте атрибут `[<Measure>]` перед типом обобщенного параметра. Такой обобщенный параметр позволит ссылаться на единицы измерения, но и только. Фактически скомпилированная форма типа вообще не будет содержать обобщенный параметр.

В примере 7.4 демонстрируется определение типа `Point`, который сохраняет единицы измерения.

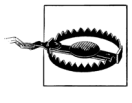
Пример 7.4. Создание типа, который является обобщенным по отношению к единицам измерения

```
// Представление точки с учетом единиц измерения ее координат
type Point< [<Measure>] 'u >(x : float<'u>, y : float<'u>) =

    member this.X = x
    member this.Y = y

    member this.UnitlessX = float x
    member this.UnitlessY = float y
```

```
    member this.Length =
        let sqr x = x * x
        sqrt <| sqr this.X + sqr this.Y

    override this.ToString() =
        sprintf
            "{%f, %f}"
            this.UnitlessX
            this.UnitlessY
```

При выполнении примера 7.4 в окне FSI результат будет следующим. Обратите внимание, что единицы измерения сохраняются в свойстве `Length`, значение которого вычисляется как корень квадратный из суммы квадратов координат:

```
> let p = new Point<m>(10.0<m>, 10.0<m>);;

val p : Point<m>

> p.Length;;
val it : float<m> = 14.14213562
```



Единицы измерения являются специфической особенностью языка F#, и они не используются в общеязыковой среде выполнения. В результате при использовании в других языках типов, созданных на языке F#, они утратят связанные с ними единицы измерения. Поэтому значения типа `float<'a>` будут доступны в языке C# как значения типа `float`.

Единицы измерения могут использоваться не только для представления значений реального мира, но и для представления более абстрактных величин, таких как количество щелчков мышью, количество пикселов, суммы в долларах и так далее.

Активные шаблоны

Сопоставление с образцом расширяет возможности программиста, предоставляя более выразительный способ организации ветвлений в коде по сравнению с условными выражениями. Они позволяют выполнять сопоставление с константами, захватывать значения и даже выполнять сопоставление со структурами данных. Однако сопоставление с образцом имеет один существенный недостаток – оно позволяет выполнять сопоставление только с литералами, такими как строки, и со значениями некоторых типов, таких как массивы и списки.

В идеале было бы здорово, если бы имелась возможность выполнять сопоставление с другими высокоуровневыми компонентами, такими как элементы последовательности `seq<_>`. Ниже приводится пример того, что можно было бы написать, но, к сожалению, этот код не компилируется:

```
// Не компилируется
let containsVowel (word : string) =
    let letters = word.Chars
    match letters with
    | ContainsAny [ 'a'; 'e'; 'i'; 'o'; 'u' ]
        -> true
    | SometimesContains [ 'y' ]
        -> true
    | _ -> false
```

Чтобы проверить наличие определенных элементов в последовательности `seq<_>`, придется использовать ограничение `when`, что в лучшем случае выглядит неэлегантно. В следующем фрагменте создается экземпляр типа `Set`, заполненный буквами, содержащимися в строке, после чего выполняется поиск определенных букв с помощью ограничения `when`:

```
let containsVowel (word : string) =
    let letters = word |> Set.ofSeq
    match letters with
    | _ when letters.Contains('a') || letters.Contains('e') ||
             letters.Contains('i') || letters.Contains('o') ||
             letters.Contains('u') || letters.Contains('y')
        -> true
    | _ -> false
```

К счастью, в языке F# есть возможность, позволяющая добиться желаемой выразительности и элегантности выражений сопоставления, под названием *активные шаблоны (active patterns)*.

Активные шаблоны – это всего лишь особые функции, которые могут использоваться внутри правил сопоставления с образцом. Применение этих функций ликвидирует необходимость использования предложений `when`, а кроме того, делает реализацию выражений сопоставления более простой и понятной, что позволяет более наглядно отразить решаемую проблему.

Существует несколько разновидностей активных шаблонов, каждая из которых принимает входные данные и преобразует их в одно или более новых значений:

- Одновариантные активные шаблоны (single-case active patterns) преобразуют входные данные в новое значение.
- Частичные активные шаблоны (partial-case active pattern) выделяют только часть пространства входных данных. Примером может служить определение строк, содержащих букву «e».
- Многовариантные активные шаблоны (multi-case active pattern) делят пространство входных данных на несколько значений. Примером может служить деление всех целых чисел на четные, нечетные и ноль.

Одновариантные активные шаблоны

Самой простой разновидностью активных шаблонов являются одновариантные активные шаблоны, которые преобразуют данные из одного типа в другой. Это позволяет использовать в выражениях сопоставления экземпляры классов и другие значения, которые иначе нельзя использовать в правилах сопоставления. Вскоре мы увидим пример использования этой возможности.

Чтобы определить активный шаблон, необходимо объявить функцию, заключенную в пары символов (| и |).

В примере 7.5 определяется активный шаблон, преобразующий имя файла в его расширение. Он позволяет выполнять сопоставление с расширениями файлов без использования предложения `when`.

Для применения активного шаблона `FileExtension` достаточно просто использовать его вместо правила сопоставления. Образец, с которым сопоставляется результат вызова активного шаблона, указывается сразу же за именем активного шаблона. Во всем остальном правило сопоставления с активным шаблоном ничем не отличается от любых других правил. Так, в примере ниже результат вызова активного шаблона сопоставляется с константой `".jpg"`. Другое правило сопоставления связывает новое значение `ext` с результатом вызова активного шаблона.

Пример 7.5. Одновариантный активный шаблон

```
open System.IO

// Преобразование имени файла в его расширение
let (|FileExtension|) filePath = Path.GetExtension(filePath)

let determineFileType (filePath : string) =
    match filePath with

    // Без применения активных шаблонов
    | filePath when Path.GetExtension(filePath) = ".txt"
        -> printfn "It is a text file."

    // Преобразование данных с помощью активного шаблона
    | FileExtension ".jpg"
    | FileExtension ".png"
    | FileExtension ".gif"
        -> printfn "It is an image file."

    // Связывает результат сопоставления с новым значением
    | FileExtension ext
        -> printfn "Unknown file extension [%s]" ext
```

Частичные активные шаблоны

Одновариантные активные шаблоны удобно использовать для повышения наглядности выражений сопоставления. Однако существует множество ситуаций, когда преобразование данных возможно не всегда. Например, что может произойти, если написать активный шаблон, который преобразует строки в целые числа?

```
> // Активный шаблон, преобразующий строки в целые числа
open System
let (|ToInt|) x = Int32.Parse(x);;

val ( |ToInt| ) : string -> int

> // Проверить, является ли входная строка изображением числа 4
let isFour str =
    match str with
    | ToInt 4 -> true
    | _ -> false;;

val isFour : string -> bool

> isFour " 4 ";;
val it : bool = true
> isFour "Not a valid Integer";;
System.FormatException: Input string was not in a correct format.
    at System.Number.StringToNumber(String str, NumberStyles options,
    NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
    at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
    at FSI_0007.IsFour(String str)
    at <StartupCode$FSI_0009>.$FSI_0009._main()
stopped due to error
```

(Входная строка имела неверный формат в ...
Остановлено из-за ошибки.)

В языке F# можно определить *частичные активные шаблоны*, которые не всегда выполняют преобразование входных данных. Для этого частичный активный шаблон должен возвращать экземпляр типа `option`. (Который, как вы помните, может иметь только два значения, `Some('a)` и `None`.) Если сопоставление завершилось успешно, возвращается значение `Some`, в противном случае – `None`. При связывании результата активного шаблона с новым значением производится обратное преобразование типа `option` в фактический тип результата.

В примере 7.6 демонстрируется, как определить частичный активный шаблон, который не генерирует исключение при получении недопусти-

мых входных данных. Чтобы определить частичный активный шаблон, достаточно заключить идентификатор *ident* активного шаблона в конструкцию (|*ident*|_|) вместо (|*ident*|). При использовании с сопоставлением с образцом совпадение будет происходить только в тех правилах, где частичный активный шаблон возвращает `Some`.

Частичные активные шаблоны можно также рассматривать как необязательные одновариантные активные шаблоны.

Пример 7.6. Частичные активные шаблоны

```
open System
let (|ToBool|_|) x =
    let success, result = Boolean.TryParse(x)
    if success then Some(result)
    else            None

let (|ToInt|_|) x =
    let success, result = Int32.TryParse(x)
    if success then Some(result)
    else            None

let (|ToFloat|_|) x =
    let success, result = Double.TryParse(x)
    if success then Some(result)
    else            None

let describeString str =
    match str with
    | ToBool  b -> printfn "%s is a bool with value %b" str b
    | ToInt   i -> printfn "%s is an integer with value %d" str i
    | ToFloat f -> printfn "%s is a float with value %f" str f
    | _         -> printfn "%s is not a bool, int, or float" str
```

Ниже приводится пример использования функции `describeString`:

```
> describeString " 3.141 ";;
3.141 is a float with value 3.141000
val it : unit = ()
> describeString "Not a valid integer";;
Not a valid integer is not a bool, int, or float
val it : unit = ()
```

Параметризованные активные шаблоны

Активные шаблоны могут также принимать дополнительные параметры. Параметры активного шаблона указываются сразу после его имени, но перед результатом.

В примере 7.7 демонстрируется создание параметризованного активного шаблона, который используется для проверки соответствия заданной строки указанному регулярному выражению. Вместо того чтобы отдельно находить соответствия с регулярным выражением, сохранять результаты и извлекать отдельные группы, вы можете использовать активные шаблоны для проверки соответствия, а также связать каждое соответствие с группой правила сопоставления с образцом.

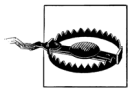
Пример 7.7. Параметризованные активные шаблоны для соответствия с регулярными выражениями

```
open System.Text.RegularExpressions

// Использование регулярного выражения
// для получения совпадений с тремя группами
let (|RegexMatch3|_|) (pattern : string) (input : string) =
    let result = Regex.Match(input, pattern)

    if result.Success then
        match (List.tail [ for g in result.Groups -> g.Value ]) with
        | fst :: snd :: trd :: []
            -> Some (fst, snd, trd)
        | [] -> failwith <| "Match succeeded, but no groups found.\n" +
                            "Use '(.*)' to capture groups"
        | _ -> failwith "Match succeeded, but did not find exactly three
groups."
    else
        None

let parseTime input =
    match input with
    // Проверить, имеет ли строка input вид "6/20/2008"
    | RegexMatch3 "(\d+)/(\d+)/(\d\d\d\d)" (month, day, year)
    // Проверить, имеет ли строка input вид "2004-12-8"
    | RegexMatch3 "(\d\d\d\d)-(\d+)-(\d+)" (year, month, day)
        -> Some( new DateTime(int year, int month, int day) )
    | _ -> None
```

Результат применения этого активного шаблона можно видеть в следующем листинге сеанса FSI:

```
> parseTime "1996-3-15";;
val it : DateTime option
= Some 3/15/1996 12:00:00 AM {Date = ...;
                              Day = 15;
                              DayOfWeek = Friday;
                              DayOfYear = 75;
                              Hour = 0;
                              Kind = Unspecified;
                              Millisecond = 0;
                              Minute = 0;
                              Month = 3;
```

```
                                    Second = 0;
                                    Ticks = 629624448000000000L;
                                    TimeOfDay = 00:00:00;
                                    Year = 1996;}
> parseTime "invalid";;
val it : DateTime option = None
```



Применение активных шаблонов способствует упрощению выражений сопоставления, но их нужно рассматривать как свойства (properties) .NET. Активные шаблоны не должны выполнять продолжительные по времени вычисления и не должны вызывать побочные эффекты. В противном случае будет очень сложно определить причину низкой производительности или заранее предсказать, где может возникнуть исключение.

Многовариантные активные шаблоны

Одновариантные активные шаблоны удобно использовать для преобразования данных, но они имеют один недостаток – одновариантные шаблоны могут преобразовать входные данные только в какой-то один новый формат. Но иногда бывает необходимо преобразовать данные сразу в несколько значений различных типов, например, когда желательно разбить пространство входных данных на несколько категорий.

С помощью *многовариантных активных шаблонов* можно разделить пространство входных данных на множество возможных значений. Чтобы определить многовариантный активный шаблон, достаточно использовать ту же конструкцию со скобками (`| |`), но при этом нужно указать несколько идентификаторов, обозначающих категории на выходе.

Например, рассмотрим реализацию сопоставления записи из таблицы `Customers`, находящейся в базе данных. Можно было бы написать несколько частичных активных шаблонов и с их помощью отнести клиента к одной из категорий: «опытный пользователь», «непользователь», «ценный клиент» и так далее. Однако вместо того чтобы создавать множество частичных активных шаблонов для определения категории входных данных, можно создать один многовариантный активный шаблон, который поделит пространство входных данных в точности на указанное количество групп возможных результатов.

Многовариантные активные шаблоны можно также представлять себе как способ преобразования входных данных к типу размеченного объединения.

В примере 7.8 демонстрируется многовариантный активный шаблон, который относит входную строку к одной из категорий: `Paragraph` (абзац), `Sentence` (предложение), `Word` (слово) и `Whitespace` (пробельная строка). Эти категории и ассоциированные с ними данные могут затем использоваться в выражениях сопоставления.

Пример 7.8. Многовариантные активные шаблоны

```
open System

// Этот активный шаблон делит все строки на четыре категории.
let (|Paragraph|Sentence|Word|WhiteSpace|) (input : string) =
    let input = input.Trim()

    if input = "" then
        WhiteSpace
    elif input.IndexOf(".") <> -1 then
        // Для категории Paragraph должен возвращаться кортеж
        // со списком предложений и его размером.
        let sentences = input.Split([|"."|], StringSplitOptions.None)
        Paragraph (sentences.Length, sentences)
    elif input.IndexOf(" ") <> -1 then
        // Для категории Sentence должен возвращаться массив слов
        Sentence (input.Split([|" "|], StringSplitOptions.None))
    else
        // Для категории Word должна возвращаться строка
        Word (input)

// Подсчет количества букв в строке
let rec countLetters str =
    match str with
    | WhiteSpace -> 0
    | Word x -> x.Length
    | Sentence words
        -> words
            |> Array.map countLetters
            |> Array.sum
    | Paragraph (_, sentences)
        -> sentences
            |> Array.map countLetters
            |> Array.sum
```

Использование активных шаблонов

Активные шаблоны могут использоваться не только в выражениях сопоставления, как мы видели это до сих пор. Активные шаблоны могут объединяться, вкладываться друг в друга и использоваться за пределами выражений сопоставления, как любые другие правила сопоставления с образцом.

Активные шаблоны могут использоваться вместо любого элемента, выполняющего сопоставление, которые, как вы наверняка помните, могут использоваться в операторах связывания `let` или в качестве параметров лямбда-выражений.

В следующем фрагменте демонстрируется применение активного шаблона `ToUpper` к первому параметру функции `f`:

```
> let (|ToUpper|) (input : string) = input.ToUpper();;

val ( |ToUpper| ) : string -> string

> let f ( ToUpper x ) = printfn "x = %s" x;;

val f : string -> unit

> f "this is lower case";;
x = THIS IS LOWER CASE
val it : unit = ()
```

Комбинирование активных шаблонов

Активные шаблоны могут объединяться подобно обычным правилам сопоставления с использованием объединения по «ИЛИ» или по «И». В примере 7.9 демонстрируется объединение нескольких активных шаблонов в одно правило сопоставления с использованием объединения по «ИЛИ», |.

Пример 7.9. Объединение активных шаблонов по "ИЛИ"

```
// Классификация фильмов
let (|Action|Drama|Comedy|Documentary|Horror|Romance|) movie =
    // ...

// Название награды, которую получил фильм
let (|WonAward|_|) awardTitle movie =
    // ...

// Оценка фильма
let goodDateMovie movie =
    match movie with
    // Варианты сопоставления многовариантного активного шаблона
    | Romance
    | Comedy

    // Параметризованный активный шаблон
    | WonAward "Best Picture"
    | WonAward "Best Adaptation"
        -> true

    | _ -> false
```

Оператор & позволяет проверять на соответствие сразу нескольким активным шаблонам. В примере 7.10 активные шаблоны объединяются для определения, не является ли файл изображения слишком большим для передачи по электронной почте. В этом примере объединяются многовариантный активный шаблон, преобразующий имя файла в один из нескольких типоразмеров, и частичный активный шаблон, определяющий, является ли указанный файл файлом изображения.

Пример 7.10. Объединение активных шаблонов по «И»

```
open System.IO

let (|KBInSize|MBInSize|GBInSize|) filePath =
    let file = File.Open(filePath, FileMode.Open)
    if file.Length < 1024L * 1024L then
        KBInSize
    elif file.Length < 1024L * 1024L * 1024L then
        MBInSize
    else
        GBInSize

let (|IsImageFile|_|) filePath =
    match filePath with
    | EndsWithExtension ".jpg"
    | EndsWithExtension ".bmp"
    | EndsWithExtension ".gif"
        -> Some()
    | _ -> None

let ImageTooBigForEmail filePath =
    match filePath with
    | IsImageFile & (MBInSize | GBInSize)
        -> true
    | _ -> false
```

Вложенные активные шаблоны

Несмотря на широкие возможности активных шаблонов по преобразованию данных в выражениях сопоставления, вы очень скоро столкнетесь с проблемой многократного преобразования данных. Однако получение результата активного шаблона и немедленное его сопоставление с другим образцом – достаточно трудоемкое занятие, которое может приводить к появлению ошибок.

К счастью, как и при сопоставлении с образцом, имеется возможность вкладывать активные шаблоны друг в друга. Кроме того, имеется возможность использовать активные шаблоны вместо образцов, с которыми выполняется сопоставление результатов активных шаблонов, что позволяет применять результат первого активного шаблона ко второму.

В примере 7.11 определяется несколько простых активных шаблонов, предназначенных для анализа элементов XML. Один выполняет преобразование узла XML в его дочерние элементы, другой выполняет преобразование значений атрибутов элемента XML и так далее. Создав вложенную структуру из активных шаблонов в пределах единственной инструкции сопоставления, можно реализовать сопоставление с полной структурой документа XML в виде единственного правила. Этот пример в полной мере демонстрирует широту возможностей использования активных шаблонов для сопоставления с образцами.

Пример 7.11. Вложенные активные шаблоны в выражении сопоставления

```
// Для этого примера необходима ссылка на System.Xml.dll
#r "System.Xml.dll"
open System.Xml

// Сопоставление с XML элементом
let (|Elem|_|) name (inp : XmlNode) =
    if inp.Name = name then Some(inp)
    else                    None

// Извлечение атрибутов элемента
let (|Attributes|) (inp : XmlNode) = inp.Attributes

// Сопоставление с именем атрибута
let (|Attr|) attrName (inp : XmlAttributeCollection) =
    match inp.GetNamedItem(attrName) with
    | null -> failwithf "Attribute %s not found" attrName
    | attr -> attr.Value

// Анализируемые элементы
type Part =
    | Widget   of float
    | Sprocket of string * int

let ParseXmlNode element =
    match element with
    // Анализ элемента Widget без использования вложенных активных шаблонов
    | Elem "Widget" xmlElement
        -> match xmlElement with
           | Attributes xmlElementsAttributes
               -> match xmlElementsAttributes with
                  | Attr "Diameter" diameter
                      -> Widget(float diameter)

// Анализ элемента Sprocket с использованием вложенных активных шаблонов
| Elem "Sprocket" (Attributes (Attr "Model" model & Attr "SerialNumber" sn))
    -> Sprocket(model, int sn)

|_ -> failwith "Unknown element"
```

Результат выполнения примера 7.11 в окне FSI следующий:

```
> // Исходный документ XML
let xmlDoc =
    let doc = new System.Xml.XmlDocument()
    let xmlText =
        "<?xml version=\"1.0\" encoding=\"utf-8\"?>
        <Parts>
            <Widget Diameter='5.0' />
            <Sprocket Model='A' SerialNumber='147' />
            <Sprocket Model='B' SerialNumber='302' />
```

```
        </Parts>
        “
    doc.LoadXml(xmlText)
    doc;;

val xmlDoc : XmlDocument

> // Анализ узлов документа
xmlDoc.DocumentElement.ChildNodes
|> Seq.cast<XmlElement>
|> Seq.map ParseXmlNode;;
val it : seq<Part> = seq [Widget 5.0; Sprocket (“A”,1024); Sprocket (“B”,306)]
```

Используя активные шаблоны, можно легко обеспечить высочайшую гибкость операций сопоставления. Активные шаблоны лучше всего подходят для ситуаций, когда возникает потребность написать множество похожих предложений `when` или когда появляется желание сделать выражение сопоставления более выразительным. Например, документы XML сами по себе мало пригодны для сопоставления с образцом, но, написав несколько простых активных шаблонов, их можно будет использовать как часть обыкновенного выражения сопоставления.

Использование модулей

Как вы наверняка помните, в главе 2 говорилось, что пространства имен являются способом организации типов. Модули ведут себя почти так же, как пространства имен, за исключением того, что модули могут содержать не только типы, но и значения.

Несмотря на то, что способность модулей содержать значения упрощает создание простых приложений, тем не менее они малопригодны для совместного использования с другими программистами и в других языках .NET. Хорошо спроектированный класс прост в использовании потому, что все его внутренние механизмы скрыты от глаз. В случае с модулем вы получаете разрозненную коллекцию значений, функций и типов, взаимоотношения между которыми не всегда очевидны.

Понимание, когда и как следует преобразовать модуль в пространство имен и в класс, важно для освоения функционального программирования на языке F#.

Преобразование модулей в классы

Говоря простым языком, код в модулях не может масштабироваться с той же легкостью, как объектно-ориентированные иерархии. Десяток модулей с десятками значений и типов в них еще управляемы, но сотни модулей с тысячами значений и типов – нет. Устранение независимых значений и переход к классам – это один из способов борьбы со сложностью.

В примере 7.12 приводится реализация простой программы, которая загружает изображения по указанному URL-адресу. Код, подобный этому, может вырасти из длительного сеанса FSI: вводим некоторый код в окне FSI, доводим его до состояния работоспособности, а затем копируем его в окно редактора. Однако несмотря на то, что подобный код отлично работает в контексте одного файла или интерактивного сеанса, его совсем непросто будет интегрировать в существующий проект.

Пример 7.12. Загрузчик изображений в виде модуля

```
open System.IO
open System.Net
open System.Text.RegularExpressions

let url = @"http://oreilly.com/"

// Загрузка веб-страницы
let req = WebRequest.Create(url)
let resp = req.GetResponse()
let stream = resp.GetResponseStream()
let reader = new StreamReader(stream)
let html = reader.ReadToEnd()

// извлечение информации об изображениях
let results = Regex.Matches(html, "<img src=\"([^\"]*)\"")
let allMatches =
    [
        for r in results do
            for grpIdx = 1 to r.Groups.Count - 1 do
                yield r.Groups.[grpIdx].Value
    ]

let fullyQualified =
    allMatches
    |> List.filter (fun url -> url.StartsWith("http://"))

// Загрузка изображений
let downloadToDisk (url : string) (filePath : string) =
    use client = new System.Net.WebClient()
    client.DownloadFile (url, filePath)

fullyQualified
|> List.map(fun url -> let parts = url.Split( [| '/' |] )
                       url, parts.[parts.Length - 1])
|> List.iter(fun (url, filename) -> downloadToDisk url (@"D:\Images\" + filename))
```

Недостаток примера 7.12 заключается в том, что он представляет собой неорганизованную последовательность значений. Хотя этот код имеет определенный смысл, если рассматривать его как выполнение последовательности операций сверху вниз, тем не менее другой программист

увидит лишь свойства модуля, такие как `results` и `allMatches`, бессмысленные вне контекста.

На первом этапе преобразования этого кода необходимо превратить его в последовательность функций, устранив все локальные значения. Если добавить объявления функций и дополнительные отступы во все строки, все содержимое модуля можно разместить в функциях. Это позволит нам разбить код на несколько частей:

```
let downloadWebpage (url : string) =
    let req = WebRequest.Create(url)
    let resp = req.GetResponse()
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    reader.ReadToEnd()

let extractImageLinks html =
    let results = Regex.Matches(html, "<img src=\"([^\"]*)\"")
    [
        for r in results do
            for grpIdx = 1 to r.Groups.Count - 1 do
                yield r.Groups.[grpIdx].Value
    ] |> List.filter (fun url -> url.StartsWith("http://"))

let downloadToDisk (url : string) (filePath : string) =
    use client = new System.Net.WebClient()
    client.DownloadFile (url, filePath)

let scrapeWebsite destPath (imageUrls : string list) =
    imageUrls
    |> List.map(fun url ->
            let parts = url.Split( [| '/' |] )
            url, parts.[parts.Length - 1])
    |> List.iter(fun (url, filename) ->
            downloadToDisk url (Path.Combine(destPath, filename)))
```

Благодаря этим изменениям мы сможем загружать изображения следующим образом:

```
downloadWebpage "http://oreilly.com/"
|> extractImageLinks
|> scrapeWebsite @"C:\Images\"
```

Но независимые друг от друга функции не очень удобны в использовании по тем же причинам, что и независимые друг от друга значения. Они ничего не упрощают – вам все равно потребуется выяснять, как объединить эти функции для получения конечного результата. Поэтому следующим шагом должно быть создание классов, которые затем могли бы использоваться другими программистами. Все, что для этого необходимо сделать, – заменить функцию типом, а ее параметры просто превратить в параметры конструктора.

В примере 7.13 все независимые функции спрятаны внутри класса. В результате доступным для нас остался только класс с конструктором и членом SaveImagesToDisk.

Пример 7.13. Загрузчик изображений, присутствующих на веб-странице, в виде класса

```
type WebScraper(url) =

    let downloadWebpage (url : string) =
        let req = WebRequest.Create(url)
        let resp = req.GetResponse()
        let stream = resp.GetResponseStream()
        let reader = new StreamReader(stream)
        reader.ReadToEnd()

    let extractImageLinks html =
        let results = Regex.Matches(html, "<img src=\"([^\"]*)\"")
        [
            for r in results do
                for grpIdx = 1 to r.Groups.Count - 1 do
                    yield r.Groups.[grpIdx].Value
        ] |> List.filter (fun url -> url.StartsWith("http://"))

    let downloadToDisk (url : string) (filePath : string) =
        use client = new System.Net.WebClient()
        client.DownloadFile (url, filePath)

    let scrapeWebsite destPath (imageUrls : string list) =
        imageUrls
        |> List.map(fun url ->
                let parts = url.Split( [| '/' |] )
                url, parts.[parts.Length - 1])
        |> List.iter(fun (url, filename) ->
                downloadToDisk url (Path.Combine(destPath, filename)))

    // Добавление полей класса
    let m_html = downloadWebpage url
    let m_images = extractImageLinks m_html

    // Добавление членов класса
    member this.SaveImagesToDisk(destPath) =
        scrapeWebsite destPath m_images
```

Модули играют важную роль в организации кода F#, особенно при разработке простых программ. Однако владение приемами преобразования модулей в классы совершенно необходимо, чтобы обеспечить возможность совместного использования вашего кода на языке F# в других языках .NET.

Намеренное сокрытие

В языке F# под значением понимается имя, связанное с некоторыми данными. Объявив новое значение с тем же именем, можно скрыть предыдущее значение с этим же именем:

```
let test() =
    let x = 'a'
    let x = "a string"
    let x = 3
    // Функция возвращает число 3
    x
```

Точно так же открытие модуля приводит к экспортированию множества новых пар имя/значение в текущее окружение, что может привести к сокрытию существующих идентификаторов. То есть если в текущей области видимости уже имеется значение с именем `x` и открывается модуль, который также содержит значение с именем `x`, то после открытия модуля имя `x` будет ссылаться на значение внутри этого модуля. В результате такого сокрытия мы лишимся любой возможности доступа к предыдущему значению `x`.

Несмотря на то, что такое сокрытие может приводить к трудноуловимым ошибкам, тем не менее преднамеренное сокрытие с помощью модулей может иметь свои преимущества. Именно такой способ используется в языке F# для проверки на переполнение в арифметических операциях.

Рассмотрим случай использования *арифметических операций с проверкой*, то есть таких арифметических операций, которые генерируют исключение в случае переполнения. По умолчанию арифметические операции в языке F# не генерируют исключения при переполнении, то есть когда при выполнении операции полученное значение выходит за границы целых чисел, просто выполняется циклический переход:

```
> let maxInt = System.Int32.MaxValue;;

val maxInt : int

> maxInt;;
val it : int = 2147483647
> maxInt + 1;;
val it : int = -2147483648
```

Модуль `Microsoft.FSharp.Core.Operators.Checked` определяет такие распространенные арифметические операторы, как `+`, `-` и другие, которые выполняют проверку на переполнение и генерируют исключение `OverflowException`. При открытии модуля `Checked` происходит сокрытие всех существующих арифметических функций и замена их другими функциями.

При открытии этого модуля фактически происходит переопределение всех арифметических операторов:

```
> let maxInt = System.Int32.MaxValue;;

val maxInt : int

> open Checked;;
> maxInt + 1;;
System.OverflowException: Arithmetic operation resulted in an overflow.
  at <StartupCode$FSI_0009>.$FSI_0009._main()
stopped due to error
```

(Переполнение в результате выполнения арифметической операции в <StartupCode$FSI_0009>.$FSI_0009._main(). Остановлено из-за ошибки.)

В этом примере было использовано простое имя модуля `Checked` вместо полного имени `Microsoft.FSharp.Core.Operators.Checked`, потому что компилятор F# автоматически открывает пространство имен `Microsoft.FSharp.Core.Operators`.

Такую возможность преднамеренного сокрытия значений удобно использовать для изменения поведения кода в файле, но при этом вы должны быть полностью уверены в том, что правильно понимаете семантику использования модулей, поведение которых пытаетесь изменить.

Управление порядком использования модулей

Как быть, если разработчику библиотеки требуется обеспечить возможность доступа к значениям модуля только с использованием имени модуля? Например, открытие модуля `List` привнесет в текущую область видимости множество простых функций, таких как `map` и `length`, но их легко можно спутать с другими похожими функциями, такими как `Seq.map` или `Array.length`:

```
open List
open Seq

// Непонятно, какая функция вызывается - List.length или Seq.length?
length [1 .. 10];;
```

Чтобы предотвратить возможность открытия модуля таким способом, достаточно просто добавить атрибут `[<RequireQualifiedAccess>]` при объявлении модуля, что предотвратит возможность его импортирования:

```
[<RequireQualifiedAccess>]
module Foo

let Value1 = 1

[<RequireQualifiedAccess>]
```

```
module Bar =
    let Value2 = 2
```

Если вы попытаетесь открыть модуль с атрибутом [<RequireQualifiedAccess>], то получите сообщение об ошибке. Например, модуль List из стандартной библиотеки F# требует использовать полные квалифицированные имена:

```
> open List;;

  open List;;
  ^^^^^^^^^

stdin(10,1): error FS0892: This declaration opens the module 'Microsoft.FSharp.
Collections.List',which is marked as 'RequireQualifiedAccess'.
Adjust your code to use qualified references to the elements of the module
instead, e.g., 'List.map' instead of 'map'. This will ensure that your code is
robust as new constructs are added to libraries
```

(Данное объявление открывает модуль "Microsoft.FSharp.Collections.List", помеченный как "RequireQualifiedAccess". Измените код так, чтобы вместо этого использовались квалифицированные ссылки на элементы модуля, например, 'List.map' вместо 'map'. Данное изменение обеспечит надежность кода при добавлении конструкций в библиотеки.)

Однако некоторые модули настолько полезны, что для них было бы удобнее, если бы они всегда экспортировали свои типы и значения в текущую область видимости. В главе 6 мы познакомились с методами расширения, благодаря которым имеется возможность добавлять в классы новые члены. Если вы определяете набор методов расширения, особенно полезных при использовании заданного пространства имен, вы можете пометить модуль как открываемый автоматически, что приведет к его автоматическому открытию, если родительское пространство имен было открыто с помощью атрибута [<AutoOpen>].

В примере 7.14 определяется пространство имен, содержащее автоматически открываемый модуль. После открытия родительского пространства имен с помощью open Alpha.Bravo вы автоматически получите доступ ко всем значениям модуля Charlie без необходимости использования полных квалифицированных имен при обращении к ним, так как модуль Charlie будет открыт неявно.

Пример 7.14. Атрибут AutoOpen

```
namespace Alpha.Bravo

[<AutoOpen>]
module Charlie =
    let X = 1
```

В первой половине этой главы было продемонстрировано, как можно воспользоваться преимуществами некоторых элементов и синтаксиче-

ских конструкций языка F# для написания кода в функциональном стиле. Оставшаяся часть главы будет посвящена не синтаксису, а теории. В частности, будет рассказано, как максимально полно использовать уникальные особенности функционального программирования.

Работа со списками

Массивы и тип `List<_>`[1] представляют собой основные структуры данных в императивном программировании, тогда как краеугольным камнем функционального программирования являются списки. Списки дают гарантию неизменяемости, что облегчает поддержку композиций функций и рекурсию, а также устраняет необходимость предварительно выделять память (как в случае использования массивов фиксированного размера).

Несмотря на то, что списки повсеместно используются в функциональном программировании, тем не менее я пока не рассказывал, как наиболее эффективно их применять. Списки в языке F# обладают уникальными характеристиками с точки зрения производительности, о которых вам следует знать, чтобы избежать ее неожиданного снижения.

Списки в языке F# реализованы в виде связных списков, как изображено на рис. 7.1. Каждый элемент списка содержит значение и ссылку на следующий элемент списка (за исключением, конечно, последнего элемента списка).

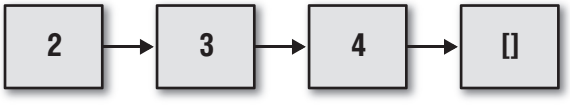


Рис. 7.1. *Структура типа FSharpList*

Операции над списками

Самым важным преимуществом типа `FSharpList` является, пожалуй, простота его использования. Модуль `List` предлагает множество различных функций для работы с существующими списками, однако когда речь заходит о динамическом конструировании списков, в вашем распоряжении имеются всего две операции: *операция добавления элемента в начало списка (cons)* и *операция объединения списков (append)*.

[1] Стандартная терминология, принятая в .Net Framework, может сбивать с толку, поскольку тип `System.Collections.Generic.List<T>` не является «списком», а реализован в виде массива с автоматически изменяемым размером и скорее относится к структуре данных типа «вектор». – *Прим. науч. ред.*

Добавление элемента в начало списка

Оператор :: добавляет новый элемент в начало существующего списка. В результате использования оператора :: получается новый список; исходный список при этом не изменяется, потому что создается новый элемент списка и его ссылке на следующий элемент присваивается начало исходного списка.

Результат работы следующего фрагмента изображен на рис. 7.2. Обратите внимание, что возвращается новый список, а старый список используется повторно:

```
> let x = [2; 3; 4];;

val x : int list = [2; 3; 4]

> 1 :: x;;
val it : int list = [1; 2; 3; 4]

> x;;
val it : int list = [2; 3; 4]
```

***Рис. 7.2.** Использование оператора ::*

Поскольку оператор :: создает единственный узел списка, он выполняется очень быстро. Напомню, что при добавлении элемента в экземпляр типа List<_>, когда внутренний массив уже заполнен, возникает необходимость в выделении памяти для нового массива, что делает операцию добавления весьма дорогостоящей. В случае списков оператор :: всегда выполняется очень быстро.

Объединение списков

Функция @ объединяет два списка, после чего последний элемент первого списка начинает указывать на начало второго списка (образуя новый, более длинный список). Однако из-за необходимости модифицировать последний элемент первого списка создается полная копия этого списка.

Результат вы можете увидеть на рис. 7.3:

```
> // Объединение списков
let x = [2; 3; 4]
let y = [5; 6; 7];;

val x : int list = [2; 3; 4]
val y : int list = [5; 6; 7]
```

```
> x @ y;;
val it : int list = [2; 3; 4; 5; 6; 7]
```

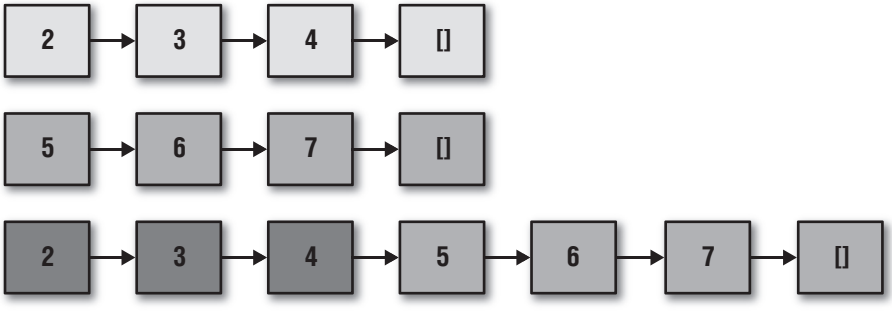


Рис. 7.3. *Оператор объединения списков*

Сравнив рис. 7.2 и 7.3, легко понять, что оператор `::` выполняется очень быстро, поскольку все, что при этом требуется сделать, – это создать один новый элемент. С другой стороны, операция объединения списков может оказаться более дорогостоящей, потому что необходимо создать копию первого списка, а это значит, что при работе с очень длинными списками операция объединения будет выполняться соответственно долго. (Необходимость копирования списка не только увеличивает время выполнения, но также приводит к увеличению расхода памяти!)

Использование списков

Познакомившись с характеристиками производительности списков, вы сможете использовать их более эффективно. За редким исключением предпочтение всегда следует отдавать оператору добавления в начало, а не объединению.

Рассмотрим пример 7.15. Обе функции удаляют одинаковые элементы списка, следующие друг за другом. Первая реализация использует функцию `fold`, выполняющую обход списка слева направо. Это делает процесс создания списка достаточно дорогостоящим, потому что при каждой итерации приходится использовать операцию объединения списков. (Для тех, кто знаком с понятием асимптотической временной оценкой сложности алгоритма, отмечу, что эта операция имеет сложность O(n^2).)

Обратите внимание, что аккумулятором для функции `fold` является кортеж, содержащий последний проверенный элемент и новый список, не содержащий повторяющихся элементов.

Вторая реализация использует функцию `foldBack`, а это означает, что она будет создавать список с использованием оператора `::`. Именно по этой причине существуют две версии функции `fold`. В данном конкретном случае использование функции `foldBack` позволяет применять опе-

ратор ::, что оказывается намного эффективнее варианта с использованием функции fold, которая требует применения оператора @.

Пример 7.15. Эффективная работа со списком

```
(*
Удаление одинаковых элементов списка, следующих друг за другом.
Например: [1; 1; 2; 2; 3; 4] => [1; 2; 3; 4]
*)

// Медленная реализация...
let removeConsecutiveDupes1 lst =

    let foldFunc acc item =
        let lastLetter, dupesRemoved = acc
        match lastLetter with
        | Some(c) when c = item
                  -> Some(c), dupesRemoved
        | Some(c) -> Some(item), dupesRemoved @ [item]
        | None    -> Some(item), [item]

    let (_, dupesRemoved) = List.fold foldFunc (None, []) lst
    dupesRemoved

// Быстрая реализация...
let removeConsecutiveDupes2 lst =
    let f item acc =
        match acc with
        | []
            -> [item]
        | hd :: tl when hd <> item
            -> item :: acc
        | _ -> acc

    List.foldBack f lst []
```

При работе со списками рассматривайте операцию объединения признаком «дурного тона» – обычно оператор @ является индикатором неоптимального решения.

Хвостовая рекурсия

Теперь вы можете использовать списки для эффективной обработки больших объемов данных, но существует еще одна проблема. Поскольку списки являются неизменяемыми, одним из основных способов их создания является рекурсия. Рассмотрим два способа создания двух списков целых чисел:

```
> // Создание списка типа List<_> 100000 целых чисел
open System.Collections.Generic
```

```
let createMutableList() =
    let l = new List<int>()
    for i = 0 to 100000 do
        l.Add(i)
    l;;

val createMutableList : unit -> List<int>

> // Создание обычного списка 100000 целых чисел
let createImmutableList() =
    let rec createList i max =
        if i = max then
             []
        else
            i :: createList (i + 1) max
    createList 0 100000;;

val createImmutableList : unit -> int list
```

Оба решения выглядят идентичными, однако функция `createImmutableList` не будет работать. (Фактически она будет «падать».) Я объясню причину «падения» функции и как это устранить при помощи хвостовой рекурсии.

Возможно, прежде вам уже приходилось слышать термин *хвостовая рекурсия (tail recursion)*. Вполне возможно, что вы уже писали функции с хвостовой рекурсией, даже не подозревая об этом. Хвостовая рекурсия – это особая форма рекурсии, когда компилятор может оптимизировать рекурсивные вызовы так, что они не будут использовать дополнительное место в стеке. Когда функция выполняет рекурсивный вызов, оставшаяся часть функции может выполнять дополнительный код, поэтому состояние функции, предшествующее вызову, сохраняется в стеке.

Однако когда функция рекурсивно вызывает себя снова и снова, может произойти переполнение стека, что приведет к появлению необрабатываемой ошибки – жуткому исключению `StackOverflowException`. Так как при хвостовой рекурсии не требуется дополнительная память в стеке, появляется возможность безопасного использования рекурсии.

Понимание хвостовой рекурсии имеет большое значение для разработки эффективного и безопасного функционального кода. Как вы видели в предыдущих разделах, для создания многих функциональных структур данных требуется рекурсия. При таком использовании рекурсии угроза переполнения стека может стать вполне реальной и ограничить объем данных, которые могут обрабатываться рекурсивными функциями. Однако этой опасности можно избежать, используя хвостовую рекурсию.

Стек

Прежде чем вы сможете написать функцию с хвостовой рекурсией, необходимо понять, как работает стек в контексте рекурсивных функций.

Всякий раз когда происходит вызов функции, в стек помещается *указатель инструкций (instruction pointer)* – адрес в программном коде, определяющий операцию, которая будет выполнена после вызова функции. Как только функция завершит свою работу, стек очищается и восстанавливается значение указателя инструкций, после чего продолжается выполнение вызывающей функции.

Взгляните на следующий простой пример рекурсивной функции, вычисляющей факториал числа:

```
let rec factorial x =
    if x <= 1
    then 1    // Базовый случай рекурсии
    else x * factorial (x - 1)
```

Если установить точку останова в строке с комментарием «Базовый случай рекурсии», вызвать функцию `factorial` со значением 10 и присоединить отладчик Visual Studio, вы увидите стек вызовов, как показано на рис. 7.4. Обратите внимание, что каждый рекурсивный вызов создает новый кадр стека. Здесь отчетливо видно, как рекурсивная функция последовательно уменьшает значение параметра `x`, пока оно не станет меньше или равно 1 и не достигнет базового случая рекурсии.

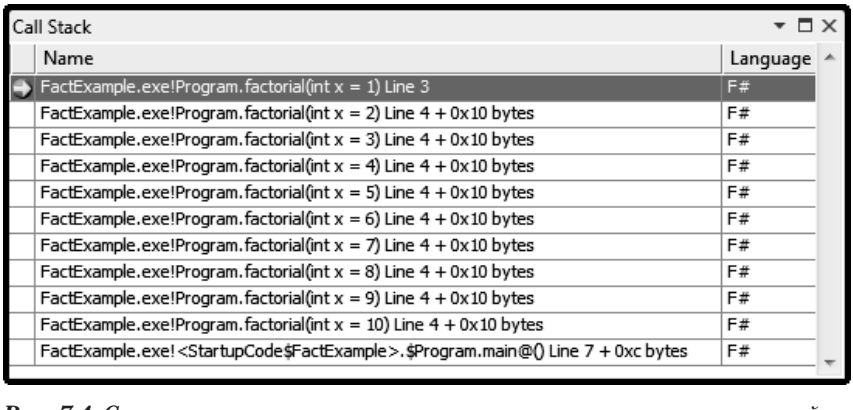


Рис. 7.4. *Состояние стека вызовов в процессе выполнения рекурсивной функции factorial*

После того как будет достигнут базовый случай рекурсии, функция вернет значение 1. Стек будет восстановлен в состояние, когда функция `factorial` была вызвана с параметром 2, и вернет значения 2×1. Затем стек будет восстановлен в состояние, когда функция `factorial` была вызвана со значением 3, и вернет значения 3×2, и так далее.

Но не забывайте, что стек имеет ограниченный размер, который обычно равен 1 Мбайт. Поэтому при попытке найти факториал очень большого числа функция `factorial` может израсходовать весь стек. Это легко можно продемонстрировать с помощью функции `infLoop` в примере 7.16. Обратите внимание, как *fsi.exe* запускается из командной строки, и при генерации исключения `StackOverflowException` процесс завершается, поэтому перехватить это исключение нельзя.

Пример 7.16. Переполнение стека с помощью рекурсии

```
C:\Program Files\Microsoft F#\v4.0>Fsi.exe

Microsoft F# Interactive, (c) Microsoft Corporation, All Rights Reserved
F# Version 1.9.8.0, compiling for .NET Framework Version v4.0.20620
Please send bug reports to fsbugs@microsoft.com
For help type #help;;

> let rec infLoop() = 1 + infLoop();;

val infLoop : unit -> int

> // Начать охоту на StackOverflowException
try
    printfn "%d" <| infLoop()
with
    | _ -> printfn "Exception caught";;

Process is terminated due to StackOverflowException.

C:\Program Files\Microsoft F#\v4.0>
```

На рис. 7.5 показаны результаты запуска той же самой функции `infLoop`, но уже под отладчиком Visual Studio.

Введение в хвостовую рекурсию

Итак, как же решить проблему переполнения стека рекурсивными вызовами? В этом нам поможет хвостовая рекурсия.

Если после рекурсивного вызова не требуется выполнять какие-либо другие инструкции, соответственно отсутствует и необходимость сохранять в стеке указатель инструкций, потому что единственное, что нужно сделать после рекурсивного вызова, – это восстановить стек предыдущей функции. То есть вместо того чтобы понапрасну изменять стек, *хвостовые вызовы (tail calls)* (или хвостовые вызовы рекурсивной функции) отбрасывают текущий кадр стека перед рекурсивным вызовом. А как только рекурсия завершится, функция вернется к исходному значению указателя инструкций.

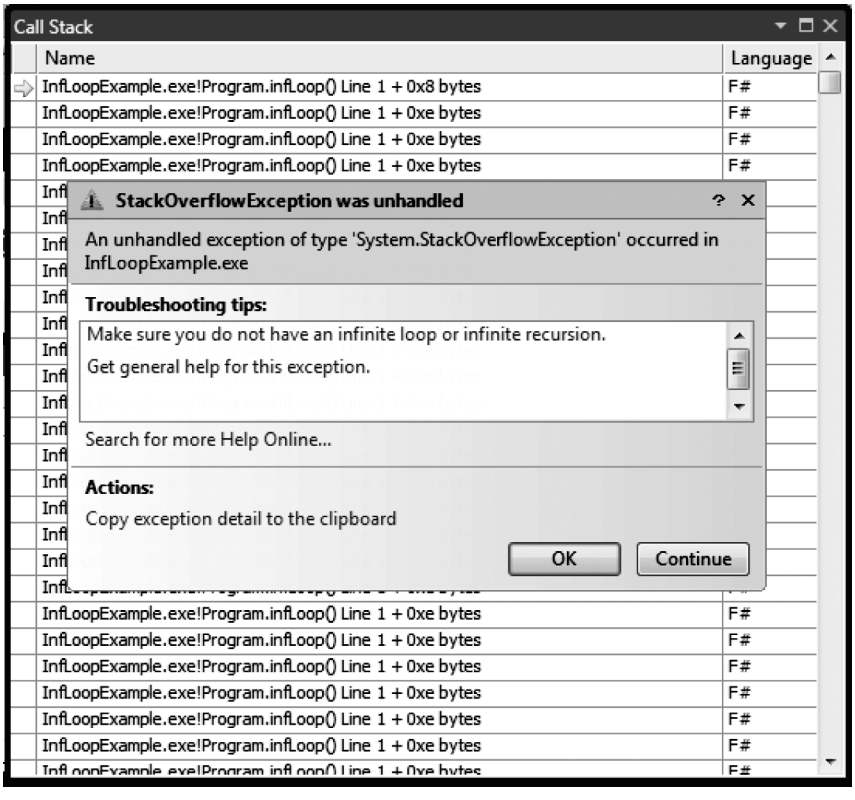


Рис. 7.5. *Жуткое исключение StackOverflowException*

Предыдущая реализация функции `factorial` не использовала хвостовую рекурсию, потому что после рекурсивного вызова `factorial (x-1)` нужно умножить результат вызова функции на значение параметра `x`. Это будет более очевидно, если переписать функцию следующим образом:

```
let rec factorial x =
    if x <= 1
    then
        1
    else
        // Рекурсивный вызов
        let resultOfRecusion = factorial (x - 1)
        // Необходимо сохранить указатель инструкций,
        // чтобы этот экземпляр функции
        // смог продолжить работу позднее.
        let result = x * resultOfRecusion
        result
```

В примере 7.17 приводится версия функции `factorial` с хвостовой рекурсией. Передача данных в виде дополнительного параметра устраняет необходимость выполнения дополнительного кода после рекурсивного вызова функции, вследствие чего отпадает потребность в дополнительном месте в стеке. Вместо этого результат конструируется и возвращается по достижении базового случая рекурсии, а не во время раскрутки рекурсивных вызовов.

Пример 7.17. Версия функции factorial с хвостовой рекурсией

```
let factorial x =
    // Сохранять значение x и аккумулятора (acc)
    let rec tailRecursiveFactorial x acc =
        if x <= 1 then
            acc
        else
            tailRecursiveFactorial (x - 1) (acc * x)
    tailRecursiveFactorial x 1
```

Когда компилятор F# определяет, что функция использует хвостовую рекурсию, он генерирует другой код. Например, для функции `tailRecursiveFactorial` может быть сгенерирован следующий код на языке C#. Обратите внимание, что рекурсия была просто заменена циклом `while`:

```
// Функция tailRecursiveFactorial, если бы она была написана
// на языке C#.
public static int tailRecursiveFactorial (int x, int acc)
{
    while (true)
    {
        if (x <= 1)
        {
            return acc;
        }
        acc *= x;
        x--;
    }
}
```

Если запустить отладчик Visual Studio и установить точку останова на базовом случае функции `tailRecursiveFactorial`, вы увидите, что используется всего один кадр стека, как показано на рис. 7.6.

Основные преимущества хвостовой рекурсии:

- Функция выполняется немного быстрее за счет уменьшения операций со стеком.
- Функция может выполнять любое количество рекурсивных вызовов.
- Не может быть получено исключение `StackOverflowException`.

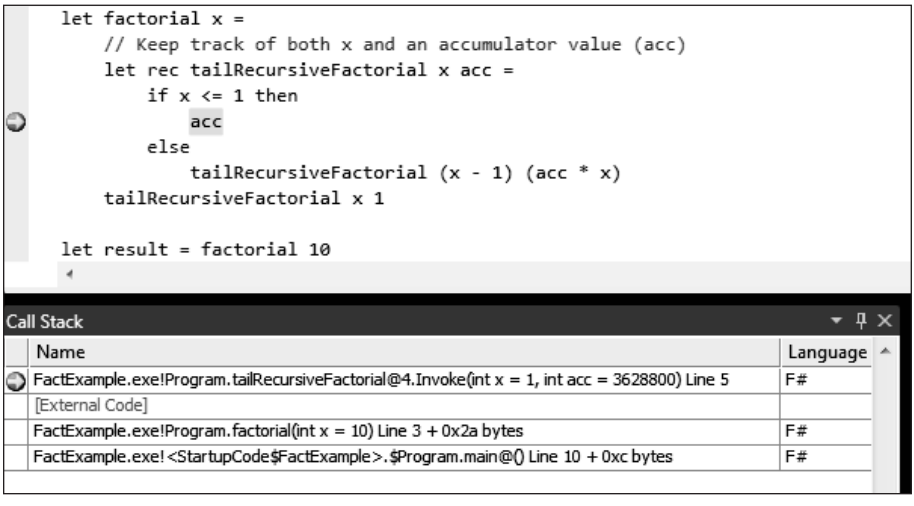


Рис. 7.6. *Стек вызовов функции с хвостовой рекурсией*

Отметим также, что рекурсия считается хвостовой, только если после хвостового вызова функция не выполняет никаких операций. В следующем разделе мы рассмотрим еще несколько примеров хвостовой рекурсии.

Среда разработки Visual Studio никак не отмечает функции, которые будут скомпилированы с хвостовой рекурсией. Когда вы пишете рекурсивные функции в функциональном стиле, старайтесь применять хвостовую рекурсию.

Шаблоны хвостовой рекурсии

Существуют два основных способа получения хвостовой рекурсии. Самый простой из них применяется, когда существует возможность передавать входные данные дальше и одновременно конструировать результат. Второй, более сложный способ, используется, когда необходимо выполнять два рекурсивных вызова с ветвлением потока выполнения.

Шаблон «Аккумулятор»

Простейший способ овладения хвостовой рекурсией заключается в следовании определенным шаблонам. Проще всего можно получить хвостовую рекурсию за счет добавления дополнительных параметров, как это было сделано в предыдущем примере функции `tailRecursiveFactorial`. Этот прием известен как *шаблон «Аккумулятор»*. Вместо того чтобы ждать, пока рекурсивный вызов вернет промежуточный результат, вы передаете рекурсивному вызову дополнительный параметр аккумулятора, чтобы по достижении базового случая рекурсии был возвращен

окончательный результат. (Указатель стека не сохраняется, потому что в функции нет другого кода, который нужно выполнить после рекурсивного вызова.)

Для иллюстрации этого шаблона напишем простую версию функции `List.map`, которая преобразует список с помощью указанной функции:

```
let rec map f list =
    match list with
    | []       -> []
    | hd :: tl -> (f hd) :: (map f tl)
```

Однако этот код не содержит хвостовую рекурсию, потому что после рекурсивного вызова функции `map` в четвертой строке будет выполнен оператор `::`. Чтобы исправить положение, можно применить шаблон «Аккумулятор» и просто сохранить промежуточный результат. В примере 7.18 определена вложенная рекурсивная функция, которая принимает результирующий список в виде параметра и возвращает его по достижении базового случая рекурсии.

Пример 7.18. Шаблон «Аккумулятор»

```
let map f list =
    let rec mapTR f list acc =
        match list with
        | []       -> acc
        | hd :: tl -> mapTR f tl (f hd :: acc)
    mapTR f (List.rev list) []
```

Для простых рекурсивных функций шаблон «Аккумулятор» – это все, что вам нужно. Но как быть, когда требуется ветвление и когда функция должна сделать два рекурсивных вызова? (Так называемая *двойная рекурсия*.) В этом случае как минимум один рекурсивный вызов никак не удастся сделать хвостовым. Рассмотрим пример 7.19 с реализацией двоичного дерева и операцией `iter`, которая применяет некоторую функцию к каждому узлу дерева.

Пример 7.19. Функция iter без хвостовой рекурсии

```
type BinTree<'a> =
    | Node of 'a * BinTree<'a> * BinTree<'a>
    | Empty

let rec iter f binTree =
    match binTree with
    | Empty -> ()
    | Node(x, l, r) ->
        f x
        iter f l // НЕ хвостовой рекурсивный вызов
        iter f r // Хвостовой рекурсивный вызов
```

Из-за двух рекурсивных вызовов мы просто не можем воспользоваться шаблоном «Аккумулятор». И что же, нет никакой надежды? Мечта о хвостовой концевой рекурсии оказалась мифом? К счастью, это не так. Но чтобы реализовать хвостовую рекурсию в подобных случаях, необходимо прибегнуть к волшебству функционального программирования.

Продолжения

Представьте, что вместо текущего состояния аккумулятора в следующий вызов функции вы передаете функцию, содержащую остаток кода, который необходимо выполнить. То есть вместо того чтобы сохранять «что осталось» в стеке, вы сохраняете это в функции. Этот прием известен как *передача продолжения (continuation passing style)*, или просто *продолжение (continuation)*.

Под продолжением понимается возвращение функции, представляющей остаток кода, который необходимо выполнить после завершения рекурсивного вызова. Использование продолжений позволяет реализовать хвостовую рекурсию, даже когда требуется выполнить множество дополнительных операций. Фактически вы обмениваете пространство стека (которое используется рекурсивными вызовами) на пространство в куче (где сохраняются значения функции).

Рассмотрим пример 7.20, который реализует вывод списка целых чисел в обратном порядке. Вывод здесь реализован как часть продолжения, которое растет с каждым рекурсивным вызовом. В базовом случае рекурсии вызывается продолжение. Функция `printListRevTR`, как и созданное ею продолжение, реализует хвостовую рекурсию, поэтому она не приводит к переполнению стека независимо от длительности обработки входных данных и результирующего продолжения.

Пример 7.20. Создание функции-значения

```
> // Вывод списка в обратном порядке с помощью продолжения
let printListRev list =
    let rec printListRevTR list cont =
        match list with
        // Для пустого списка выполнить продолжение
        | [] -> cont()
        // Для других списков добавить вывод текущего узла
        // в продолжение.
        | hd :: tl ->
            printListRevTR tl (fun () -> printf "%d " hd
                                         cont() )
    printListRevTR list (fun () -> printfn "Done!");;

val printListRev : int list -> unit
```

```
> printListRev [1 .. 10];;
10 9 8 7 6 5 4 3 2 1 Done!
val it : unit = ()
```

Конечно же, создание функции-значения – не самая простая задача. Но этот прием позволяет писать функции с хвостовой рекурсией, которую невозможно реализовать другим способом.

Теперь вернемся к операции `iter` обхода двоичного дерева. Эта операция должна выполнить два рекурсивных вызова, и поэтому применение шаблона «Аккумулятор» не поможет получить хвостовую рекурсию. Однако чтобы функция могла обрабатывать огромные деревья, нам необходимо реализовать хвостовую рекурсию.

Добиться этого можно несколькими способами с помощью продолжений, но самый простой для понимания способ состоит в использовании *собственного типа итератора (custom iterator type)*. В этом шаблоне вместо создания одного продолжения, выполняющего все необходимые вычисления для получения хвостовой рекурсии, вы *линеаризуете* необходимые операции с помощью собственного типа данных.

В примере 7.21 определяется новый тип итератора `ContinuationStep` с двумя вариантами данных, `Finished` и `Step`, при этом последний является кортежем, содержащим значение и функцию, возвращающую следующий экземпляр типа `ContinuationStep`.

Функция `linearize` использует продолжение для преобразования двоичного дерева в объект `ContinuationStep`. Каждый узел преобразуется в вариант `Step` размеченного объединения, и дополнительно производится разделение рекурсивных вызовов для обхода левого и правого поддеревьев в виде лямбда-выражений. Наконец, функция `processSteps` анализирует объект `ContinuationStep` и фактически выполняет операцию `iter` двоичного дерева.

Пример 7.21. Создание собственного итератора

```
type ContinuationStep<'a> =
    | Finished
    | Step of 'a * (unit -> ContinuationStep<'a>)

let iter f binTree =

    let rec linearize binTree cont =
        match binTree with
        | Empty -> cont()
        | Node(x, l, r) ->
            Step(x, (fun () -> linearize l (fun() -> linearize r cont)))

    let steps = linearize binTree (fun () -> Finished)

    let rec processSteps step =
        match step with
```

```
        | Finished -> ()
        | Step(x, getNext)
            -> f x
               processSteps (getNext())

    processSteps steps
```

Чтобы понять этот пример, требуется тщательно изучить его, однако это лишь один из способов, позволяющих разбить рекурсивную функцию для получения хвостовой рекурсии.

Следует признать, что такое сложное решение наверняка не является самым лучшим. Возможно, более удачным решением было бы использовать изменяемую структуру данных для хранения узлов, которые нужно обойти, в порядке их обхода. (Например, с помощью хранения узлов в изменяемом объекте типа `Stack<_>`.) Но как бы то ни было, при программировании на языке F# у вас всегда есть возможность отыскать решение, которое лучше всего соответствует вашим потребностям.

Программирование с применением функций

Хвостовая рекурсия и модули позволяют нам получить больше отдачи от использования функционального стиля программирования, однако мы еще не все знаем о применении функций как значений. Далее мы рассмотрим карринг функций и замыкания. Несмотря на то, что эти два механизма являются не самыми важными аспектами функционального программирования, тем не менее хорошее понимание их возможностей облегчит разработку кода в функциональном стиле.

Карринг

Как вы помните, карринг функции – это возможность определить подмножество параметров функции и получить новую функцию, принимающую меньшее количество параметров. На первый взгляд такая возможность кажется весьма странной. В чем выигрыш от получения функции с ограниченным подмножеством параметров? Одной из самых заметных выгод, которые несет карринг функций, является прямой конвейерный оператор, который делает обработку данных более элегантной:

```
getCustomers()
|> List.filter (fun cust -> datesEqual cust.Birthday DateTime.Today)
|> List.map (fun cust -> cust.EmailAddress)
|> List.filter Option.isSome
|> sendSpam "С Днем Рождения... Купите у нас что-нибудь!"
```

Каждая строка в этом выражении является каррированной функцией, которая получает последний параметр из предыдущей строки. Написание кода в подобном стиле позволяет записывать последовательно-

сти операций именно так, как они выстраиваются у вас в голове. Однако существует еще один способ упрощения кода с помощью карринга функций.

Передача функций в виде параметров

Карринг функций может избавить от необходимости множества лямбда-выражений. Взгляните на следующий код, который применяет функцию `f` к каждому элементу списка:

```
List.map (fun x -> f x) lst
```

Обратите внимание, что параметр `x` лямбда-выражения передается последним параметром функции `f`. Всякий раз когда вам будет встречаться подобный код, вы сможете вообще убрать лямбда-выражение, потому что функция `f` может передаваться функции `List.map` в виде параметра (тем же способом, каким передаются другие функции в функцию `List.map`):

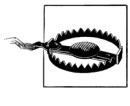
```
List.map f lst
```

Аналогичным способом можно упростить функции, даже если они принимают более одного параметра. В примере ниже функция `g` принимает четыре параметра: `a`, `b`, `c` и `x`. Однако из-за того что лямбда-выражение принимает только один параметр `x`, вы можете передать в виде параметра частично применимую функцию:

```
List.map (fun x -> g a b c x) lst
```

Этот вызов также можно упростить с помощью карринга функции:

```
List.map (g a b c) lst
```



Несмотря на то, что передача функций в виде параметров выглядит достаточно элегантно, тем не менее этот прием может усложнить чтение кода. Не забывайте, что однажды кому-то другому придется заниматься отладкой ваших программ, поэтому всегда стремитесь писать читабельный код.

Избавление от избыточного кода

Обычно основной целью рефакторинга кода является устранение его избыточности за счет повторного использования, обычно путем наследования в объектно-ориентированном коде. Однако вы можете избавиться от избыточности и в функциональном коде с помощью функций высшего порядка.

Давайте рассмотрим следующие две функции, которые помогут вам выбрать блюдо на обед. Функция `pickCheapest` возвращает самое дешевое первое блюдо из списка, а функция `pickHealthiest` возвращает самое низкокалорийное блюдо. Как вы наверняка помните, функция `reduce`

ведет себя точно так же, как и функция `fold`, за исключением того, что в качестве начального значения аккумулятора она использует первый элемент списка:

```
[<Measure>]
type usd

type Entree = { Name : string; Price : float<usd>; Calories : int }

// Выбрать самое дешевое блюдо в меню
let pickCheapest menuItems =
    List.reduce
        (fun acc item -> if item.Price < acc.Price
                         then item
                         else acc)
        menuItems

let pickHealthiest menuItems =
    List.reduce
        (fun acc item -> if item.Calories < acc.Calories
                         then item
                         else acc)
        menuItems
```

Обе эти функции практически идентичны и отличаются только способом сравнения элементов. Почему бы не вынести его за скобки? Если передать функцию сравнения в виде параметра, можно существенно упростить реализацию и сделать код более читабельным. Обратите внимание, что функции `pickCheapest2` и `pickHealthiest2` были получены за счет карринга функции `pickItem`:

```
let private pickItem cmp menuItems =
    let reduceFunc acc item =
        match cmp acc item with
        | true  -> acc
        | false -> item
    List.reduce reduceFunc menuItems

let pickCheapest2   = pickItem (fun acc item -> item.Price < acc.Price)
let pickHealthiest2 = pickItem (fun acc item -> item.Calories < acc.Calories)
```

Замыкания

Термин *замыкание (closure)* используется для обозначения внутренней, или вложенной функции, имеющей доступ к значениям, которые не передаются ей явно в виде параметров. Подобно каррингу, на первый взгляд эта концепция выглядит достаточно странной и не имеющей практического применения, но на самом деле вы использовали замыкания на протяжении всей книги, даже не подозревая об этом.

Простым примером замыкания может служить следующая функция mult, которая умножает каждый элемент списка на заданное число. Обратите внимание, что значение i – это параметр первой функции, но он не является параметром лямбда-выражения. У вас может возникнуть вопрос, как лямбда-выражение может использовать значение i, если оно не передается явно в виде параметра? В этом и заключается прелесть замыканий. Поскольку значение i находится в области видимости лямбда-выражения, оно захватывается *замыканием* этого лямбда-выражения и потому является доступным. За кулисами компилятор в действительности связывает значение i с лямбда-выражением, чтобы во время выполнения оно оставалось доступным, но все это делается незаметно для вас:

```
> let mult i lst = List.map (fun x -> x * i) lst;;

val mult : int -> int list -> int list

> mult 10 [1; 3; 5; 7];;
val it : int list = [10; 30; 50; 70]
```

По сути, с помощью замыканий функции могут захватывать любые дополнительные данные, что позволяет функциям обращаться к значениям в той области видимости, где эта функция была объявлена. Благодаря этому мы можем получать достаточно интересные результаты.

Представьте себе абстрактный тип данных, такой как класс, не имеющий явных полей, где могла бы сохраняться информация о состоянии его экземпляров. На первый взгляд это кажется невозможным – чтобы тип имел хоть какой-то смысл, он должен где-то хранить внутренние данные. Но с замыканиями такое вполне возможно.

Давайте рассмотрим пример 7.22, в котором определяется тип записи Set с двумя полями-функциями, одна из которых добавляет элементы в Set, а другая проверяет наличие элемента в нем. Для создания экземпляра типа Set достаточно обратиться к статическому свойству Empty. После получения экземпляра типа Set операции добавления и удаления элементов выполняются исключительно за счет захвата данных замыканиями. Функция makeSet принимает список, который включает элементы, содержащиеся в множестве. При добавлении в список нового элемента рекурсивный вызов makeSet возвращает управление, захватывая список элементов исходного набора, и новый элемент добавляется в замыкании функции Add.

Пример 7.22. Инкапсуляция данных с помощью замыканий

```
// Тип множества. Обратите внимание, что реализация
// хранится в полях записи...
type Set =
    {
```

```
        // Добавляет элемент в множество
        Add    : int -> Set
        // Проверяет присутствие элемента в множестве
        Exists : int -> bool
    }

    // Возвращает пустое множество
    static member Empty =
        let rec makeSet lst =
            {
                Add    = (fun item -> makeSet (item :: lst))
                Exists = (fun item -> List.exists ((=) item) lst)
            }
        makeSet []
```

Ниже приводится пример использования типа `Set` в окне FSI:

```
> // Тип Set в действии
let s = Set.Empty
let s' = s.Add(1)
let s'' = s'.Add(2)
let s''' = s''.Add(3);;

val s : Set = {Add = <fun:makeSet@15-1>;
               Exists = <fun:makeSet@16-2>;}
val s' : Set = {Add = <fun:makeSet@15-1>;
                Exists = <fun:makeSet@16-2>;}
val s'' : Set = {Add = <fun:makeSet@15-1>;
                 Exists = <fun:makeSet@16-2>;}
val s''' : Set = {Add = <fun:makeSet@15-1>;
                  Exists = <fun:makeSet@16-2>;}
> s.Exists 2;;
val it : bool = false
> s'''.Exists 2;;
val it : bool = true
```

Точно так же, как и в случае со слишком сложными продолжениями, наверняка существуют более простые способы создания типов, чем хранение внутренних данных внутри замыканий. Однако понимание того, что функции могут иметь доступ не только к своим параметрам, открывает новые возможности.

Функциональные шаблоны проектирования

Теперь, когда мы познакомились с некоторыми дополнительными концепциями функционального программирования, можно перейти к исследованию некоторых шаблонов проектирования, которые они открывают. Описание решений в терминах шаблонов проектирования упрощает объяснение и повторное использование этих решений.

Мемоизация

При использовании функционального стиля большинство функций являются «чистыми», то есть получая одни и те же входные данные, будут возвращать один и тот же результат. Однако многократный вызов одной и той же чистой функции, возвращающей одно и то же значение, может оказаться напрасной тратой ресурсов на выполнение одних и тех же операций. Было бы гораздо эффективнее сохранять вычисленные значения в экземпляре типа `Dictionary<_,_>`, отображающем входные значения на результат выполнения функции.

Процесс сохранения известных пар значений аргументы/результат называется *мемоизацией (memoization)* и может выполняться автоматически с помощью функций высшего порядка и замыканий.

В примере 7.23 определяется функция `memoize`, которая принимает функцию в виде параметра и возвращает мемоизованную версию этой функции. Мемоизованная версия хранит словарь с предыдущими результатами в замыкании, а вызовы фактической функции выполняются, только когда это действительно необходимо.

Пример 7.23. Мемоизация чистых функций

```
open System.Collections.Generic

let memoize (f : 'a -> 'b) =
    let dict = new Dictionary<'a, 'b>()

    let memoizedFunc (input : 'a) =
        match dict.TryGetValue(input) with
        | true, x -> x
        | false, _ ->
            // Вычислить результат и добавить его в словарь
            let answer = f input
            dict.Add(input, answer)
            answer

    // Вернуть мемоизованную версию функции f,
    // которая захватывает словарь dict в замыкании
    memoizedFunc
```

Следующий сеанс FSI демонстрирует использование функции `memoize`. Функция `f` в этом примере просто ожидает заданное количество секунд, поэтому два вызова этой функции с одним и тем же значением параметра приведут к двойному ожиданию. При вызове мемоизованной версии функции во второй раз просто ищется кэшированное значение, поэтому функция в действительности не вызывается:

```
> // Некоторая функция, которая долго вычисляет результат...
let f x =
    // Ждать x секунд
    System.Threading.Thread.Sleep(x * 1000)
```

```
    // Вернуть x
    x;;

val f : int -> int

> // Тестирование
#time;;

--> Таймер включен

> f 10;;
Real: 00:00:10.007, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 10
> f 10;;
Real: 00:00:10.008, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 10
> let memoizedF = memoize f;;
Real: 00:00:00.004, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0

val memoizedF : (int -> int)

> memoizedF 10;;
Real: 00:00:10.009, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 10
> memoizedF 10;;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 10
```

Мемоизация рекурсивных функций

Мемоизация рекурсивных функций может оказаться немного более сложной. В примере 7.23 функция `memoize` принимает функцию `f` и возвращает новую функцию, которая проверяет, было ли уже вычислено значение функцией `f` для заданного входного значения. Однако что произойдет, если в процессе вычисления функция `f` вызывает саму себя? К сожалению, она будет вызывать немемоизованную версию, которая не использует преимущества мемоизации. Мемоизация будет приносить свои плоды только для первого входного значения функции; в случае рекурсивных вызовов проверка кэшированных значений не выполняется.

Нам требуется, чтобы любые рекурсивные вызовы функции направлялись мемоизованной версии. Если вы задумаетесь над этим, то наверняка придете к выводу, что это невозможно, потому что мемоизованная версия должна быть вызвана еще до завершения определения мемоизированной функции! К счастью, компилятор F# способен выполнить дополнительную работу, чтобы обеспечить такую возможность, хотя и с выводом предупреждения.

В примере 7.24 приводится листинг сеанса FSI, который показывает правильный и неправильный способы мемоизации функции `fib`. Обратите внимание, как функция `fib` в определении функции `memFib2` вы-

зывает memFib2 (мемоизованную функцию) вместо рекурсивного вызова себя самой.

Пример 7.24. Мемоизация рекурсивных функций

```
> // НЕПРАВИЛЬНЫЙ способ мемоизации функции - функция fib не вызывает
// мемоизованную версию.
let wrongMemFib =
    let rec fib x =
        match x with
        | 0 | 1 -> 1
        | 2 -> 2
        | n -> fib (n - 1) + fib (n - 2)

    memoize fib;;

val wrongMemFib : (int -> int)

> // ПРАВИЛЬНЫЙ способ мемоизации функции - fib вызывает
// мемоизованную версию.
let rec rightMemFib =
    let fib x =
        match x with
        | 0 | 1 -> 1
        | 2 -> 2
        | n -> rightMemFib (n - 1) + rightMemFib (n - 2)

    memoize fib;;

val rightMemFib : (int -> int)

> #time;;

--> Таймер включен

> wrongMemFib 45;;
Real: 00:00:21.553, CPU: 00:00:21.528, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 1836311903
> rightMemFib 45;;
Real: 00:00:00.001, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 1836311903
```

Функции как изменяемые значения

Функция как значение обеспечивает простой способ передачи реализации функции. Однако значение функции может также изменяться, что позволяет легко замещать одну реализацию функции другой. Практически так же, как изменяемое целое значение позволяет изменять его значение, изменяемые функции позволяют изменять свое значение.

В примере 7.25 демонстрируется, как можно на лету изменять реализацию функции generateWidget. Эта возможность позволяет изменять реализацию функции, не прибегая к таким тяжеловесным решениям, как полиморфизм и наследование.

Первоначально функция generateWidget возвращает экземпляры Sprockets. Но после изменения ее реализации функция generateWidget начинает возвращать экземпляры Cogs.

Пример 7.25. Использование функции как изменяемого значения

```
> // Код создает экземпляры типа Widget,
// основу любых приложений .NET
type Widget =
    | Sprocket of string * int
    | Cog of string * float

let mutable generateWidget =
    let count = ref 0
    (fun () -> incr count
               Sprocket("Model A1", !count));;

type Widget =
    | Sprocket of string * int
    | Cog of string * float
val mutable generateWidget : (unit -> Widget)

> // Создать Widget
generateWidget();;
val it : Widget = Sprocket ("Model A1",1)
> // ... и еще один
generateWidget();;
val it : Widget = Sprocket ("Model A1",2)

> // Теперь изменим реализацию функции...
generateWidget <-
    let count = ref 0
    (fun () -> incr count
               Cog( (sprintf "Version 0x%x" !count), 0.0));;
val it : unit = ()
> // Создать другой Widget
// с вызовом измененной версии функции
generateWidget();;
val it : Widget = Cog ("Version 0x1",0.0)
> generateWidget();;
val it : Widget = Cog ("Version 0x2",0.0)
```

Совершенно очевидно, что обращение к функции, когда нет 100% уверенности в том, что она делает, может привести к трудноуловимым ошибкам. Тем не менее возможность изменения реализации функции в некоторых ситуациях может быть очень удобной.

Отложенные вычисления

Об отложенных (или ленивых) вычислениях уже упоминалось в главе 3; это самостоятельный стиль программирования. В двух словах, под отложенными вычислениями подразумеваются вычисления, которые выполняются, только когда в этом возникает необходимость; противоположность им составляют *немедленные вычисления (eagerly)*, которые производятся, как только они будут объявлены.

При правильном использовании отложенные вычисления имеют ряд замечательных преимуществ.

Уменьшение потребления памяти

Выше мы познакомились с продолжениями как средством получения хвостовой рекурсии в операции обхода двоичного дерева. Но даже если забыть о проблеме переполнения стека, операции над очень большими деревьями будут занимать продолжительное время. Представьте, что вам потребовалось применить некоторую функцию к каждому узлу дерева, содержащему миллион элементов. Представьте также, что вам необходимо обратиться лишь к незначительному количеству узлов, получившихся в результате выполнения этой функции. Если воспользоваться отложенными вычислениями, функция будет применяться только к нужным узлам.

В примере 7.26 определяется двоичное дерево, при работе с узлами которого используются отложенные вычисления. То есть функция `map` применяет функцию `f` к дочерним узлам, только когда это действительно необходимо (при обращении к свойству `Value`).

При такой реализации функции `map` не происходит создание копии всего дерева. Новые узлы создаются, только когда к ним происходит обращение. Использование отложенных вычислений при работе с огромными деревьями, из которых требуется извлечь лишь несколько узлов, могут привести к существенной экономии памяти.

Пример 7.26. Тип Lazy

```
type LazyBinTree<'a> =
    | Node of 'a * LazyBinTree<'a> Lazy * LazyBinTree<'a> Lazy
    | Empty

let rec map f tree =
    match tree with
    | Empty -> Empty
    | Node(x, l, r) ->
        Node(
            f x,
            lazy (
                let lfNode = l.Value
                map f lfNode),
            lazy (
```

```
                let rtNode = r.Value
                map f rtNode)
        )
```

Абстрактный доступ к данным

Другим примером применения отложенных вычислений является абстрактный доступ к данным. В примере 7.27 выполняется загрузка содержимого всех файлов в формате CSV (Comma-Separated-Value – данные, разделенные запятыми) в указанном каталоге, а данные возвращаются в виде единой последовательности массивов строк. Однако сами файлы не открываются и не читаются, пока в этом не появится необходимость, то есть с внешней точки зрения результат вызова функции `processFile` выглядит как непрерывная последовательность данных, при этом за кулисами происходит фактическое открытие и закрытие отдельных файлов. Функции `Seq.map` и `Seq.concat` не открывают файл, возвращаемый функцией `processFile`, пока действительно не потребуются элементы последовательности. Говоря другими словами, последовательность создается отложенно.

Пример 7.27. Применение отложенных вычислений для обработки файлов CSV

```
open System
open System.IO

let processFile (filePath : string) =
    seq {
        use fileReader = new StreamReader(filePath)

        // пропустить строку заголовка
        fileReader.ReadLine() |> ignore

        while not fileReader.EndOfStream do
            let line = fileReader.ReadLine()
            yield line.Split( [| ',' |] )
    }

let rootPath = @"D:\DataFiles\"
let csvFiles = Directory.GetFiles(rootPath, "*.csv")

let allCsvData =
    csvFiles
    |> Seq.map processFile
    |> Seq.concat
```

8

Прикладное объектно-ориентированное программирование

Эту главу также можно было бы назвать «Функциональные приемы в объектно-ориентированном мире». Теперь вы владеете синтаксисом и стилем программирования на языке F#, но при этом еще многое нужно сказать об использовании F# в объектно-ориентированном мире платформы .NET.

В этой главе вы узнаете, как создавать новые типы в языке F#, которые лучше взаимодействуют с другими объектно-ориентированными языками программирования, такими как C#, например добавлять перегруженные операторы для возможности использовать ваши типы с символьными операторами, а также добавлять события, которые позволят получать уведомления о выполняемых действиях. Это поможет не только сделать ваш код на F# более выразительным, но и свести к минимуму разногласия с другими разработчиками, не использующими язык F# при разработке совместного проекта.

Операторы

Использование символьной нотации для работы с объектами может упростить код за счет устранения вызовов методов, таких как `Add` и `Remove`, и использования более понятных операторов, таких как + и -. Кроме того, если объект является коллекцией данных, возможность их индексирования с использованием таких операторов, как `.[ ]` или `.[`*выражение .. выражение*`]`, может существенно упростить доступ к данным, хранящимся в объекте.

Перегрузка операторов

Перегрузка операторов (operator overloading) используется для добавления нового значения существующим операторам, таким как + или -.

Вы можете перегрузить эти операторы, чтобы они работали не только с простыми типами данных, но и с пользовательскими типами. Перегрузка операторов в языке F# реализуется простым добавлением в тип статических методов. После этого вы сможете применять операторы, как если бы они изначально поддерживали ваш тип данных. Компилятор F# просто будет преобразовывать эти операторы в вызовы ваших статических методов.

Перегружая операторы, убедитесь, что при этом семантика их использования будет соответствовать этим операторам. Например, перегруженный оператор + не должен выполнять вычитание значений или удаление элементов коллекции.

При перегрузке оператора имя оператора необходимо заключить в круглые скобки, например (+) при перегрузке оператора сложения. Для перегрузки унарного оператора, нужно просто добавить перед ним символ тильда ~.

В примере 8.1 перегружаются операторы + и – для манипулирования бутылками (`Bottle`). Обратите внимание, что оператор ~– является унарным оператором отрицания.

Пример 8.1. Перегрузка операторов

```
[<Measure>]
type ml

type Bottle(capacity : float<ml>) =

    new() = new Bottle(0.0<ml>)

    member this.Volume = capacity

    static member (+) ((lhs : Bottle), rhs) =
        new Bottle(lhs.Volume + rhs)

    static member (-) ((lhs : Bottle), rhs) =
        new Bottle(lhs.Volume - rhs)

    static member (~-) (rhs : Bottle) =
        new Bottle(rhs.Volume * -1.0)

    override this.ToString() =
        sprintf "Bottle(%.1fml)" (float capacity)
```

После добавления операторов вы можете использовать свои классы с интуитивно понятными стандартными операторами. (Конечно, бутылка не может иметь отрицательный объем, но основную идею, я думаю, вы поняли.)

```
> let half = new Bottle(500.0<ml>);;
```

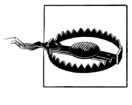
```
val half : Bottle = Bottle(500.0ml)

> half + 500.0<ml>;;
val it : Bottle = Bottle(1000.0ml) {Volume = 1000.0;}
> half - 500.0<ml>;;
val it : Bottle = Bottle(0.0ml) {Volume = 0.0;}
> -half;;
val it : Bottle = Bottle(-500.0ml) {Volume = -500.0;}
```

Вы можете также добавлять перегруженные операторы к размеченным объединениям и к записям таким же образом, что и для обычных классов:

```
type Person =
    | Boy of string
    | Girl of string
    | Couple of Person * Person
    static member (+) (lhs, rhs) =
        match lhs, rhs with
        | Couple(_), _
        | _, Couple(_)
            -> failwith "Three's a crowd!"
        | _ -> Couple(lhs, rhs)
```

Теперь с помощью оператора + можно преобразовать два объекта типа `Person` в вариант `Couple` (пара):

```
Boy("Dick") + Girl("Jane");;
val it : Person = Couple (Boy "Dick",Girl "Jane")
```



В языке F# вы можете определить любые символьные операторы. Однако только часть из них может быть использована в языках C# и VB.NET. Эти языки поддерживают только следующие операторы:

Бинарные операторы:

```
+, -, *, /, %, &, |, <<, >>
+=, -=, *=, /=, %=
```

Унарные операторы:

```
~+, ~-, ~!, ~++, ~--
```

Остальные операторы будут доступны в виде методов с удобными именами, описывающими выполняемые ими операции. Например, перегруженный оператор +-+ будет доступен в виде статического метода с именем `op_PlusMinusPlus`.

Индексаторы

Для типов, являющихся коллекциями значений, индексаторы являются естественным расширением семантики их использования, позволяя разработчикам обращаться к значениям в коллекции по их индексам.

Например, вы можете получить произвольный элемент массива с помощью оператора .[ ].

В языке F# индексаторы реализуются простым добавлением свойства с именем Item. Обращение к этому свойству будет выполняться всякий раз при вызове метода .[ ].

В примере 8.2 создается класс Year, в который добавлен индексатор, позволяющий получать информацию об *n*-ом дне года.

Пример 8.2. Добавление индексатора в класс

```
open System

type Year(year : int) =

    member this.Item (idx : int) =
        if idx < 1 || idx > 365 then
            failwith "Invalid day range"

        let dateStr = sprintf "1-1-%d" year
        DateTime.Parse(dateStr).AddDays(float (idx - 1))
```

Следующий сеанс FSI демонстрирует использование этого индексатора:

```
> // Использование собственного индексатора
let eightyTwo = new Year(1982)
let specialDay = eightyTwo.[171];;

val eightyTwo : Year
val specialDay : DateTime = 6/20/1982 12:00:00 AM

> specialDay.Month, specialDay.Day, specialDay.DayOfWeek;;
val it : int * int * DayOfWeek = (6, 20, Sunday)
```

Но что если вам потребуется не только читать значения с помощью индексатора, но и записывать их? Кроме того, могут также потребоваться нечисловые индексы. В языке F# предусмотрена возможность, удовлетворяющая обоим этим требованиям.

В примере 8.3 определяется индексатор, который принимает нечисловые индексы. В отличие от предыдущего примера класса Year, следующая реализация индексатора способна принимать кортеж с названием месяца и с номером дня месяца.

Пример 8.3. Нечисловые индексы

```
open System

type Year2(year : int) =

    member this.Item (month : string, day : int) =
        let monthIdx =
            match month.ToUpper() with
```

```
        | "JANUARY" -> 1  | "FEBRUARY" -> 2  | "MARCH"     -> 3
        | "APRIL"   -> 4  | "MAY"      -> 5  | "JUNE"      -> 6
        | "JULY"    -> 7  | "AUGUST"   -> 8  | "SEPTEMBER" -> 9
        | "OCTOBER" -> 10 | "NOVEMBER" -> 11 | "DECEMBER"  -> 12
        | _ -> failwithf "Invalid month [%s]" month

    let dateStr = sprintf "1-1-%d" year
    DateTime.Parse(dateStr).AddMonths(monthIdx - 1).AddDays(float (day-1))
```

В классе `Year2` реализован более понятный индексатор, упрощающий работу с этим типом:

```
> // Использование нечисловых индексов
let O'seven = new Year2(2007)
let randomDay = O'seven.["April", 7];;

val O'seven : Year2
val randomDay : DateTime = 4/7/2007 12:00:00 AM

> randomDay.Month, randomDay.Day, randomDay.DayOfWeek;;
val it : int * int * DayOfWeek = (4, 7, Saturday)
```

Индексаторы только для чтения, которые рассматривались до сих пор, подходят не всегда. Иногда возникает необходимость в индексаторах не только для получения элементов, но и для обновления их значений. Для реализации индексатора для чтения/записи достаточно явно определить методы чтения и записи свойства `Item`.

В примере 8.4 определяется тип `WordBuilder`, позволяющий получать и изменять символы в слове по их индексам.

Пример 8.4. Индексатор, обеспечивающий доступ для чтения/записи

```
open System.Collections.Generic

type WordBuilder(startingLetters : string) =
    let m_letters = new List<char>(startingLetters)

    member this.Item
        with get idx = m_letters.[idx]
        and set idx c = m_letters.[idx] <- c

    member this.Word = new string (m_letters.ToArray())
```

Для операции записи нового значения используется тот же синтаксис, что и для изменения значения элемента массива:

```
> let wb = new WordBuilder("Jurassic Park");;

val wb : WordBuilder

> wb.Word;;
val it : string = "Jurassic Park"
> wb.[10];;
```

```
val it : char = 'a'
> wb.[10] <- 'o';;
val it : unit = ()
> wb.Word;;
val it : string = "Jurassic Pork"
```

Добавление срезов

Подобно индексаторам, срезы обеспечивают доступ к элементам коллекции по их индексам. Главное отличие их состоит в том, что индексаторы возвращают единственный элемент, тогда как срезы – новую коллекцию данных. Синтаксис получения срезов позволяет указывать необязательную нижнюю и необязательную верхнюю границы среза.

Для добавления одномерного среза следует добавить метод с именем `GetSlice`, который принимает два параметра типа `option`. В примере 8.5 определяется оператор получения срезов для типа `TextBlock`, который возвращает фрагмент текста.

Пример 8.5. Добавление в класс оператора получения срезов

```
type TextBlock(text : string) =
    let words = text.Split([| ' ' |])

    member this.AverageWordLength =
        words |> Array.map float |> Array.average

    member this.GetSlice(lowerBound : int option, upperBound : int option) =
        let words =
            match lowerBound, upperBound with
            // Указаны нижняя и верхняя границы
            | Some(lb), Some(ub) -> words.[lb..ub]
            // Указана только одна граница
            | Some(lb), None -> words.[lb..]
            | None, Some(ub) -> words.[..ub]
            // Не указана ни одна из границ
            | None, None -> words.[*]
        words
```

Использование класса `TextBlock` следующее:

```
> // Использование собственного оператора получения срезов
let text = "The quick brown fox jumped over the lazy dog"
let tb = new TextBlock(text);;

val text : string = "The quick brown fox jumped over the lazy dog"
val tb : TextBlock

> tb.[..5];;
val it : string [] = [|"The"; "quick"; "brown"; "fox"; "jumped"; "over"|]
> tb.[4..7];;
val it : string [] = [|"jumped"; "over"; "the"; "lazy"|]
```

Если вам потребуется более тонкое управление срезами, можно реализовать получение двухмерных срезов. Это позволит указывать два диапазона данных. Для определения двухмерных срезов достаточно снова добавить метод `GetSlice`, который будет принимать четыре параметра в следующем порядке: нижняя граница для первого измерения, верхняя граница для первого измерения, нижняя граница для второго измерения и верхняя граница для второго измерения.

В примере 8.6 определяется класс, инкапсулирующий последовательность точек. Этот класс реализует операцию извлечения двухмерных срезов, возвращающую только точки, координаты которых попадают в указанный диапазон.

Пример 8.6. Реализация двухмерного среза

```
open System

type DataPoints(points : seq<float * float>) =
    member this.GetSlice(xlb, xub, ylb, yub) =

        let getValue optType defaultValue =
            match optType with
            | Some(x) -> x
            | None -> defaultValue

        let minX = getValue xlb Double.MinValue
        let maxX = getValue xub Double.MaxValue

        let minY = getValue ylb Double.MinValue
        let maxY = getValue yub Double.MaxValue

        // Вернуть, если данный кортеж представляет точку
        // с координатами в заданном диапазоне
        let inRange (x, y) =
            (minX < x && x < maxX &&
             minY < y && y < maxY)

        Seq.filter inRange points
```

Следующая сессия FSI демонстрирует использование двухмерного среза. Срез позволяет получить точки, координаты которых находятся в указанном диапазоне, что дает возможность получать определенные фрагменты данных для последующего анализа:

```
> // Создание 1000 случайных точек со значениями координат от 0.0 до 1.0
let points =
    seq {
        let rng = new Random()
        for i = 0 to 1000 do
            let x = rng.NextDouble()
            let y = rng.NextDouble()
```

```
            yield (x, y)
    };;

val points : seq<float * float>

> points;;
val it : seq<float * float> =
  seq
    [(0.6313018304, 0.8639167859); (0.3367836328, 0.8642307673);
     (0.7682324386, 0.1454097885); (0.6862907925, 0.8801649925); ...]

> let d = new DataPoints(points);;

val d : DataPoints

> // Получить точки, координаты x и y которых больше 0.5.
d.[0.5.., 0.5..];;

val it : seq<float * float> =
  seq
    [(0.7774719744, 0.8764193784); (0.927527673, 0.6275711235);
     (0.5618227448, 0.5956258004); (0.9415076491, 0.8703156262); ...]
> // Получить точки, координата x которых находится в диапазоне от 0.90..0.99,
// не ограничивая значение координаты.
d.[0.90 .. 0.99, *];;

val it : seq<float * float> =
  seq
    [(0.9286256581, 0.08974070246); (0.9780231351, 0.5617941956);
     (0.9649922103, 0.1342076446); (0.9498346559, 0.2927135142); ...]
```

Срезы обеспечивают выразительный синтаксис извлечения данных из коллекций. Однако вы не сможете получать срезы с размерностью больше двух.

Ограничения обобщенных типов

Добавление обобщенных параметров в объявление класса позволяет использовать этот класс совместно с любыми другими типами. Но как быть, если необходимо, чтобы класс фактически выполнял какие-либо операции с данными обобщенного типа? Не просто хранить ссылку типа `'a`, но выполнять некоторые разумные операции с этим экземпляром. Поскольку `'a` может быть экземпляром любого типа, вы можете подумать, что это невозможно.

К счастью, можно добавить *ограничение обобщенного типа (generic type constraint)*, что позволит вашему классу быть обобщенным, но ограничит возможные типы обобщенных параметров. Например, можно определить обобщенный класс, который будет принимать только типы, реализующие определенный интерфейс. Это даст вам возможность опреде-

лить, какие типы могут передаваться в виде параметров обобщенного типа, чтобы обеспечить дополнительные операции с их экземплярами.

Ограничение обобщенного типа — это просто форма аннотации типа. Всякий раз, когда вы определяете параметр обобщенного типа `'a`, вы можете добавить ограничение типа с помощью ключевого слова `when`.

В примере 8.7 определяется обобщенный класс `GreaterThanList`, параметр типа которого должен реализовывать интерфейс `IComparable`. Это позволит приводить экземпляры параметров к интерфейсу `IComparable` и сравнивать их с первым элементом списка.

Пример 8.7. Ограничения обобщенного типа

```
open System
open System.Collections.Generic

exception NotGreaterThanHead

/// Список, в котором каждый последующий элемент больше первого.
type GreaterThanList< 'a when 'a :> IComparable<'a> >(minValue : 'a) =

    let m_head = minValue

    let m_list = new List<'a>()
    do m_list.Add(minValue)

    member this.Add(newItem : 'a) =
        // приведение к интерфейсу IComparable было бы невозможно,
        // если бы мы не ограничили тип 'a
        let ic = newItem :> IComparable<'a>

        if ic.CompareTo(m_head) >= 0 then
            m_list.Add(newItem)
        else
            raise NotGreaterThanHead

    member this.Items = m_list :> seq<'a>
```

В языке F# существует шесть способов ограничения типов, а возможность использования их комбинаций позволяет точно определить допустимые типы параметров. Объединить несколько ограничений можно с помощью ключевого слова `and`:

```
open System

let compareWithNew (x : 'a when 'a : (new : unit -> 'a) and
                         'a :> IComparable<'a>) =

// Мы можем создать новый экземпляр типа 'a, потому что
// ограничение типа требует такой конструктор.
let clone = new 'a()
```

```
// Сравнение x с новым экземпляром возможно, так как заранее известно,
// что тип 'a реализует интерфейс IComparable
let ic = x :> IComparable<'a>
ic.CompareTo(clone)
```

Ограничение подтипа

Ограничение подтипа (subtype constraint) требует, чтобы параметр типа соответствовал указанному типу или являлся одним из его подтипов. (То же относится и к интерфейсам.) Это позволит приводить тип `'a` к известному типу. Синтаксис ограничения подтипа следующий:

```
'a :> тип
```

Ограничение на конструктор по умолчанию

Ограничение на конструктор по умолчанию (default constructor constraint) требует, чтобы указанный параметр типа имел конструктор по умолчанию (то есть конструктор без параметров). Это позволит создавать экземпляры обобщенного типа. Синтаксис ограничения на конструктор по умолчанию следующий:

```
'a : (new : unit -> 'a)
```

Ограничение по делегату

Ограничение по делегату (delegate constraint) требует, чтобы указанный параметр типа был делегатом с заданной сигнатурой и типом возвращаемого значения. (Делегаты будут рассматриваться в следующем разделе.) Синтаксис ограничения по делегату следующий:

```
'a : delegate<кортеж-типов-аргументов, тип-возвращаемого-значения>
```

Ограничение по значимому типу

Ограничение по значимому типу (struct constraint) требует, чтобы указанный параметр типа был значимым типом. Синтаксис ограничения по значимому типу следующий:

```
'a : struct
```

Ограничение по ссылочному типу

Ограничение по ссылочному типу (reference constraint) требует, чтобы указанный параметр типа был ссылочным типом. Синтаксис ограничения по ссылочному типу следующий:

```
'a : not struct
```

Ограничение по перечислению

Ограничение по перечислению (enumeration constraint) требует, чтобы указанный в параметре тип был перечислением с указанным базовым типом (таким как `int`). Синтаксис ограничения по перечислению следующий:

```
'a : enum<базовый-тип>
```

Кроме того, в языке F# имеется возможность определить ограничения по *сравнению* (comparison constraint) и по *равенству* (equality constraint). Но на практике они редко используются или бывают необходимы.

Делегаты и события

До сих пор мы определяли полезные типы и классы, но на них необходимо было как-то воздействовать, чтобы что-то произошло. Однако иногда бывает удобно иметь не *реактивные (reactive)*, а *активные (proactive)* типы, то есть типы, экземпляры которых могут воздействовать на экземпляры других типов, а не наоборот.[1]

Рассмотрим пример 8.8, в котором определяется тип `CoffeeCup` (чашка кофе), уведомляющий все заинтересованные стороны, когда чашка опустошается. Это удобно, когда необходимо реализовать автоматическое наполнение чашки и не лишиться возможности выпить еще чашечку вкусного кофе. Это реализуется за счет сохранения списка функций, которые необходимо вызвать при опустошении чашки.

Пример 8.8. Создание активных типов

```
open System.Collections.Generic

[<Measure>]
type ml

type CoffeeCup(amount : float<ml>) =
    let mutable m_amountLeft = amount
    let mutable m_interestedParties = List<(CoffeeCup -> unit)>()

    member this.Drink(amount) =
        printfn "Drinking %.1f..." (float amount)
        m_amountLeft <- max (m_amountLeft - amount) 0.0<ml>
        if m_amountLeft <= 0.0<ml> then
            this.LetPeopleKnowI'mEmpty()
```

[1] Здесь возникла некоторая путаница в терминах и понятиях. Так, автор называет модель взаимодействия «реактивной» (reactive), когда экземпляры одного типа взаимодействуют с экземплярами другого типа. Однако обычно такая модель программирования называется «интерактивной» (interactive), от слова interact – взаимодействовать. А реактивной моделью программирования обычно называют такую модель, когда один объект «реагирует» на события, происходящие в другой части системы. И хотя термин «активный» (proactive) также является устойчивым и применяется в данном случае корректно, термин «реактивный» здесь применяется не в общепринятом смысле. Подробнее об этом см. выступление Эрика Мейера (Eric Mejier) на Channel9: *http://channel9.msdn.com/Shows/Going+Deep/E2E-Erik-Meijer-and-Wes-Dyer-Reactive-Framework-Rx-Under-the-Hood-1-of-2. – Прим. науч. ред.*

```
    member this.Refil(amountAdded) =
        printfn "Coffee Cup refilled with %.1f" (float amountAdded)
        m_amountLeft <- m_amountLeft + amountAdded

    member this.WhenYou'reEmptyCall(func) =
        m_interestedParties.Add(func)

    member private this.LetPeopleKnowI'mEmpty() =
        printfn "Uh oh, I'm empty! Letting people know..."
        for interestedParty in m_interestedParties do
            interestedParty(this)
```

Следующий сеанс FSI показывает, как выполняется повторное наполнение чашки:

```
> let cup = new CoffeeCup(100.0<ml>);;

val cup : CoffeeCup

> // Следующая функция будет вызвана, когда чашка опустеет
cup.WhenYou'reEmptyCall(
    (fun cup ->
        printfn "Thanks for letting me know..."
        cup.Refil(50.0<ml>) ));;

val it : unit = ()

> cup.Drink(75.0<ml>);;

Drinking 75.0...
val it : unit = ()

> cup.Drink(75.0<ml>);;
Drinking 75.0...
Uh oh, I'm empty! Letting people know...
Thanks for letting me know...
Coffee Cup refilled with 50.0
val it : unit = ()
```

Говоря абстрактно, мы можем описать класс `CoffeeCup` как шаблон, позволяющий *потребителям* – людям, у которых есть экземпляр нашего класса, – подписаться на получение уведомления о *событии* опустошения чашки. Эта возможность оказывается очень полезной и является основой программирования приложений с графическим интерфейсом (так называемое событийно-ориентированное программирование, event-driven programming). Например, оконные системы позволяют определять функции, которые будут вызываться по щелчку мышью на кнопке или в результате изменения состояния флажка (элемента управления Check Box). Однако нашу реализацию, основанную на хранении списка функций, можно признать в лучшем случае неуклюжей.

К счастью, платформа .NET обеспечивает самую широкую поддержку такого шаблона с помощью *делегатов* и *событийно-ориентированного программирования (event-driven programming)*. *Делегаты* очень похожи на функции-значения в языке F#. А *события* – это просто механизм вызова зарегистрированных функций.

Делегаты и события формируют контракт. Один класс *публикует* событие, что дает возможность другим классам *подписаться* на их получение. Когда в классе возникает событие, вызываются все делегаты, предоставленные подписчиками.

Делегаты являются одной из разновидностей функций-значений. Они представляют собой указатель на метод и могут передаваться в виде параметров или вызываться как обычные функции. Но кроме этого делегаты имеют несколько преимуществ перед обычными функциями языка F#, с которыми вы познакомитесь ниже.

Определение делегатов

Для определения делегата используется следующий синтаксис. Первый тип (*тип1*) представляет параметры, передаваемые делегату, а второй тип (*тип2*) определяет тип возвращаемого значения:

```
type идентификатор = delegate of тип1 -> тип2
```

В примере 8.9 демонстрируется, как создать простой делегат и насколько он напоминает обычную функцию. Обратите внимание, что экземпляр делегата создается с помощью ключевого слова `new`, при этом тело делегата передается в виде параметра, а чтобы выполнить делегат, необходимо вызвать его метод `Invoke`.

Пример 8.9. Определение и использование делегатов

```
let functionValue x y =
    printfn "x = %d, y = %d" x y
    x + y

// Определение делегата
type DelegateType = delegate of int * int -> int

// Создание значения делегата
let delegateValue1 =
    new DelegateType(
        fun x y ->
            printfn "x = %d, y = %d" x y
            x + y
    )

// Вызов функции и делегата
let functionResult = functionValue 1 2
let delegateResult = delegateValue1.Invoke(1, 2)
```

Благодаря некоторой магии, которой обладает компилятор F#, отпадает необходимость создавать экземпляры некоторых типов делегатов. Если член класса принимает делегат в виде параметра, то вместо экземпляра делегата указанного типа можно передать лямбда-выражение, при условии, что оно имеет ту же сигнатуру, что и делегат.

В следующем фрагменте кода приводится определение типа делегата `IntDelegate` и метода `ApplyDelegate`. Метод `ApplyDelegate` вызывается дважды – в первом вызове методу передается явно созданный экземпляр делегата `IntDelegate`, а во втором используется неявное создание делегата. Лямбда-выражение соответствует сигнатуре экземпляра `IntDelegate`, и поэтому компилятор F# автоматически создает экземпляр делегата.

Эта возможность значительно повышает читабельность кода на языке F#, взаимодействующего с компонентами .NET, принимающими делегаты, такими как библиотека Parallel Extensions для .NET Framework, которая будет рассматриваться в главе 11:

```
type IntDelegate = delegate of int -> unit

type ListHelper =
    /// Вызывает делегат для каждого элемента списка
    static member ApplyDelegate (l : int list, d : IntDelegate) =
        l |> List.iter (fun x -> d.Invoke(x))

// Явное создание делегата
ListHelper.ApplyDelegate([1 .. 10], new IntDelegate(fun x -> printfn "%d" x))

// Неявное создание делегата
ListHelper.ApplyDelegate([1 .. 10], (fun x -> printfn "%d" x))
```

Объединение делегатов

Несмотря на то, что делегаты очень похожи на функции, между ними существуют два важных отличия. Во-первых, делегаты могут объединяться. Объединение нескольких делегатов позволяет вызвать несколько функций одним вызовом метода `Invoke`. Объединение делегатов осуществляется с помощью статических методов `Combine` и `Remove` типа `System.Delegate` – базового класса всех делегатов.

В примере 8.10 показано объединение делегатов, благодаря которому единственный вызов метода `Invoke` приводит к выполнению сразу нескольких делегатов.

Пример 8.10. Объединение делегатов

```
open System.IO

type LogMessage = delegate of string -> unit

let printToConsole =
```

```
    LogMessage(fun msg -> printfn "Logging to console: %s..." msg)

let appendToLogFile =
    LogMessage(fun msg -> printfn "Logging to file: %s..." msg
                          use file = new StreamWriter("Log.txt", true)
                          file.WriteLine(msg))

let doBoth = LogMessage.Combine(printToConsole, appendToLogFile)
let typedDoBoth = doBoth :?> LogMessage
```

Ниже приводится результат выполнения кода в окне FSI. Вызов делегата `typedDoBoth` приводит к выполнению обоих делегатов, `printToConsole` и `appendToLogFile`. Функции F# не могут объединяться таким способом.

```
> typedDoBoth.Invoke("[some important message]");;
Logging to console: [some important message]...
Logging to file: [some important message]...
val it : unit = ()
```

Второе важное отличие делегатов от функций заключается в том, что делегаты могут вызываться асинхронно, то есть делегат может выполняться в другом потоке, а программа продолжит работу. Чтобы вызвать делегата асинхронно, следует воспользоваться методами `BeginInvoke` и `EndInvoke`.

К сожалению, делегаты, созданные в языке F#, не содержат методов `BeginInvoke` и `EndInvoke`, поэтому делегат, который может вызываться асинхронно, придется создавать на языке VB.NET или C#. В главе 11 мы будем рассматривать приемы параллельного и асинхронного программирования на языке F# и познакомимся с простыми способами решения этой проблемы.

События

Теперь, когда вы познакомились с делегатами, давайте рассмотрим, как воспользоваться их преимуществами при создании событий. События – это просто синтаксический сахар, используемый для определения свойств классов, являющихся делегатами. То есть при возбуждении некоторого события просто происходит вызов объединения делегатов, связанных с этим событием.

Создание событий

Для начала рассмотрим простой пример, от которого мы сможем отталкиваться. В отличие от C# или VB.NET, в языке F# отсутствует ключевое слово `event`.

В примере 8.11 создается тип `NoisySet`, который возбуждает событие при добавлении и удалении элементов множества. Вместо того чтобы вручную сохранять ссылки на подписчиков событий, здесь используется тип `Event<'Del, 'Arg>`, который обсуждается чуть ниже.

Пример 8.11. События и тип Event<_,_>

```
type SetAction = Added | Removed

type SetOperationEventArgs<'a>(value : 'a, action : SetAction) =
    inherit System.EventArgs()

    member this.Action = action
    member this.Value = value

type SetOperationDelegate<'a> =
    delegate of obj * SetOperationEventArgs<'a> -> unit

// Содержит множество элементов, которое возбуждает событие
// при добавлении новых элементов.
type NoisySet<'a when 'a : comparison>() =
    let mutable m_set = Set.empty : Set<'a>

    let m_itemAdded =
        new Event<SetOperationDelegate<'a>, SetOperationEventArgs<'a>>()

    let m_itemRemoved =
        new Event<SetOperationDelegate<'a>, SetOperationEventArgs<'a>>()

    member this.Add(x) =
        m_set <- m_set.Add(x)
        // Возбудить событие 'Add'
        m_itemAdded.Trigger(this, new SetOperationEventArgs<_>(x, Added))

    member this.Remove(x) =
        m_set <- m_set.Remove(x)
        // Возбудить событие 'Remove'
        m_itemRemoved.Trigger(this, new SetOperationEventArgs<_>(x, Removed))

    // Опубликовать события, чтобы другие классы могли подписаться на них
    member this.ItemAddedEvent = m_itemAdded.Publish
    member this.ItemRemovedEvent = m_itemRemoved.Publish
```

Делегаты для обработки событий

Первое, что следует отметить в этом примере, – это то, что для обработки событий используются делегаты. Как правило, в .NET для событий используются делегаты, возвращающие значение типа `unit` и принимающие два параметра. Первый параметр – источник события, то есть объект, возбудивший событие. Второй параметр – аргументы делегата, которые передаются в виде экземпляра класса, производного от `System.EventArgs`.

Тип `SetOperationEventArgs` в примере обеспечивает сохранение всей необходимой информации о событии, такой как значение добавляемого или удаляемого элемента. Однако во многих случаях события не требу-

ют передачи дополнительной информации, и тогда не нужно создавать классы, наследующие EventArgs.

Создание и возбуждение событий

После создания делегата следующий шаг – создание самого события. В языке F# это делается за счет создания нового экземпляра типа Event<_,_>:

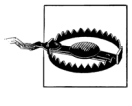
```
let m_itemAdded =
    new Event<SetOperationDelegate<'a>, SetOperationEventArgs<'a>>()
```

Мы вскоре познакомимся с типом Event<_,_>, а пока обратите внимание, что этот тип должен использоваться всякий раз, когда возникает необходимость возбудить событие или подписаться на него. Возбуждение события производится вызовом метода Trigger, который принимает параметры для вызываемых делегатов:

```
// Возбудить событие 'Add'
m_itemAdded.Trigger(this, new SetOperationEventArgs<_>(x, Added))
```

Подписка на события

Наконец, чтобы подписаться на событие, необходимо вызвать метод AddHandler события. После этого всякий раз, когда будет возбуждаться данное событие, будет вызываться делегат, переданный методу AddHandler. Если необходимость в вызове обработчика события отпала, вызовите метод RemoveHandler.



Не забывайте удалять обработчики событий, когда в уведомлениях отпадает необходимость. В противном случае ссылка на ваш объект будет сохраняться в объекте, возбуждающем события, что будет препятствовать утилизации вашего объекта сборщиком мусора.

В примере 8.12 демонстрируется использование типа NoisySet<_>.

Пример 8.12. Добавление и удаление обработчиков событий

```
> // Использование событий
let s = new NoisySet<int>()

let setOperationHandler =
    new SetOperationDelegate<int>(
        fun sender args ->
            printfn "%d was %A" args.Value args.Action
    )

s.ItemAddedEvent.AddHandler(setOperationHandler)
s.ItemRemovedEvent.AddHandler(setOperationHandler);;
```

```
val s : NoisySet<int>
val setOperationHandler : SetOperationDelegate<int>

> s.Add(9);;
9 was Added
val it : unit = ()
> s.Remove(9);;
9 was Removed
val it : unit = ()
```

Как видно из примера 8.12, все необходимое для объединения делегатов и в конечном счете для возбуждения события выполняется классом `Event<'Del, 'Arg>`. Но что этот класс делает на самом деле? И зачем нужны два обобщенных параметра?

Класс Event<_,_>

Тип `Event<'Del, 'Arg>` сохраняет ссылку на делегат, связанный с событием, и упрощает публикацию и возбуждение исключений. При создании экземпляра этого типа конструктору передаются два параметра обобщенного типа. Первый определяет тип делегата, связанного с событием, – в предыдущем примере это был тип `SetOperationDelegate`. Второй определяет тип аргумента делегата – тип `SetOperationEventArgs` в предыдущем примере. (Обратите внимание, что здесь отсутствует фактический первый параметр делегата, объект-источник события.)

Если при создании своих событий вы не следуете этому шаблону, в котором первым параметром делегату передается объект-источник события, то вам следует использовать тип `DelegateEvent<'del>`. Этот тип используется точно так же, за исключением того, что аргументы передаются в виде массива объектов `obj`.

В примере 8.13 определяется тип часов с одним событием, которое возбуждается каждую секунду и извещает подписчиков о текущем времени, передавая часы, минуты и секунды. Обратите внимание, что здесь используется событие типа `DelegateEvent<_>`, поэтому методу `Trigger`, возбуждающему событие, все параметры делегата должны передаваться в виде массива объектов `obj`. Напомню, что функция `box` преобразует свой параметр в экземпляр типа `obj`.

Пример 8.13. Тип DelegateEvent

```
open System

type ClockUpdateDelegate = delegate of int * int * int -> unit

type Clock() =

    let m_event = new DelegateEvent<ClockUpdateDelegate>()
```

```
    member this.Start() =
        printfn "Started..."
        while true do
            // Приостановить выполнение на одну секунду...
            Threading.Thread.Sleep(1000)
            let hour = DateTime.Now.Hour
            let minute = DateTime.Now.Minute
            let second = DateTime.Now.Second

            m_event.Trigger( [| box hour; box minute; box second |] )

    member this.ClockUpdate = m_event.Publish
```

Ниже приводится пример использования класса из примера 8.13:

```
> // Нестандартные типы событий
let c = new Clock();;

val c : Clock

> // Добавить обработчик события
c.ClockUpdate.AddHandler(
    new ClockUpdateDelegate(
        fun h m s -> printfn "[%d:%d:%d]" h m s
    )
);;
val it : unit = ()

> c.Start();;
Started...
[14:48:27]
[14:48:28]
[14:48:29]
```

Модуль Observable

События – это полезный механизм, который позволяет создавать активные классы, извещающие другие объекты о происходящих событиях. Однако события в языке F# не обязательно должны быть ассоциированы с каким-то классом, также как и функции в языке F# не обязательно должны быть членами какого-либо класса.

Модуль `Observable` содержит ряд функций для создания экземпляров интерфейса `IObservable<_>`, что дает возможность работать с событиями как с полноправными элементами языка, такими как объекты и функции. Это позволяет использовать приемы функционального программирования, такие как композиция функций и трансформация данных совместно с событиями.

В примере 8.14 определяется тип `JukeBox` (музыкальная шкатулка), возбуждающий событие всякий раз, когда начинается проигрывание но-

вой мелодии. (Назначение загадочного атрибута CLIEvent будет описано ниже.)

Пример 8.14. Композиционные события

```
[<Measure>]
type minute

[<Measure>]
type bpm = 1/minute

type MusicGenre = Classical | Pop | HipHop | Rock | Latin | Country

type Song = { Title : string; Genre : MusicGenre; BPM : int<bpm> }

type SongChangeArgs(title : string, genre : MusicGenre, bpm : int<bpm>) =
    inherit System.EventArgs()

    member this.Title = title
    member this.Genre = genre
    member this.BeatsPerMinute = bpm

type SongChangeDelegate = delegate of obj * SongChangeArgs -> unit

type JukeBox() =
    let m_songStartedEvent = new Event<SongChangeDelegate, SongChangeArgs>()

    member this.PlaySong(song) =
        m_songStartedEvent.Trigger(
            this,
            new SongChangeArgs(song.Title, song.Genre, song.BPM)
        )

[<CLIEvent>]
member this.SongStartedEvent = m_songStartedEvent.Publish
```

Вместо того чтобы просто добавлять обработчики событий непосредственно в экземпляр класса JukeBox, мы можем воспользоваться модулем Observable и с его помощью создать новые, более конкретные обработчики событий. В примере 8.15 с помощью методов Observable.filter и Observable.partition создаются два новых события, slowSongEvent и fastSongEvent, которые возбуждаются, когда экземпляр JukeBox начинает проигрывание мелодии, отвечающей определенным критериям.

Преимущество этой возможности заключается в том, что вы можете на основе существующего события создавать новые события, которые могут передаваться как обычные значения.

Пример 8.15. Использование модуля Observable

```
> // Использование модуля Observable для подписки
// на более конкретные события
```

```
let jb = new JukeBox()

let fastSongEvent, slowSongEvent =
    jb.SongStartedEvent
    // Оставить только события, относящиеся к танцевальной музыке
    |> Observable.filter(fun songArgs ->
        match songArgs.Genre with
        | Pop | HipHop | Latin | Country -> true
        | _ -> false)
    // Разбить событие на 'быструю мелодию' и 'медленную мелодию'
    |> Observable.partition(fun songChangeArgs ->
        songChangeArgs.BeatsPerMinute >= 120<bpm>);;

val jb : JukeBox
val slowSongEvent : IObservable<SongChangeArgs>
val fastSongEvent : IObservable<SongChangeArgs>

> // Добавить обработчики событий для событий IObservable
slowSongEvent.Add(fun args -> printfn
                              "You hear '%s' and start to dance slowly..."
                              args.Title)

fastSongEvent.Add(fun args -> printfn
                              "You hear '%s' and start to dance fast!"
                              args.Title);;
> jb.PlaySong( { Title = "Burnin Love"; Genre = Pop; BPM = 120<bpm> } );;
You hear 'Burnin Love' and start to dance fast!
val it : unit = ()
```

Однако `Observable.filter` и `Observable.partition` — это не единственные полезные методы в модуле.

Observable.add

Метод `Observable.add` просто подписывается на событие. Обычно этот метод используется в конце последовательности прямых конвейерных операторов и является более простым способом подписки на события, чем вызов метода `AddHandler`.

Метод `Observable.add` имеет следующую сигнатуру:

```
val add : ('a -> unit) -> IObservable<'a> -> unit
```

Пример 8.16 выводит окно с сообщением, когда указатель мыши оказывается в нижней половине формы.

Пример 8.16. Подписка на события с помощью метода Observable.add

```
open System.Windows.Forms

let form = new Form(Text="Keep out of the bottom!", TopMost=true)
```

```
form.MouseMove
|> Observable.filter (fun moveArgs -> moveArgs.Y > form.Height / 2)
|> Observable.add (fun moveArgs -> MessageBox.Show("Moved into bottom half!")
                                   |> ignore)
form.ShowDialog()
```

Observable.merge

Метод `Observable.merge` принимает два события и возвращает одно событие, которое будет возбуждаться, когда наступит любое из исходных событий.

Метод `Observable.merge` имеет следующую сигнатуру:

```
val merge: IObservable<'a> -> IObservable<'a> -> IObservable<'a>
```

Его удобно использовать для объединения и упрощения событий. Например, если бы в предыдущем примере нас не интересовали различия между быстрыми и медленными танцевальными мелодиями, мы могли бы объединить оба события в одно – `justDance`:

```
> // Объединение двух событий начала мелодий
let justDanceEvent = Observable.merge slowSongEvent fastSongEvent
justDanceEvent.Add(fun args ->
                      printfn "You start dancing, regardless of tempo!");;

val justDanceEvent : System.IObservable<SongChangeArgs>

> // Добавить в очередь еще одну мелодию
jb.PlaySong(
    { Title = "Escape(The Pina Colada Song)"; Genre = Pop; BPM = 70<bpm> } );;
You hear 'Escape (The Pina Colada Song)' and start to dance slowly...
You start dancing, regardless of tempo!
val it : unit = ()
```

Observable.map

Иногда при работе с событием возникает потребность в преобразовании его к некоторой другой функции, с которой было бы проще работать. Метод `Observable.map` позволяет преобразовать одно событие с аргументом определенного типа в другое событие. Этот метод имеет следующую сигнатуру:

```
val map: ('a -> 'b) -> IObservable<'a> -> IObservable<'b>
```

При использовании Windows Forms во всех событиях щелчка мышью передаются координаты указателя в пикселах относительно верхнего левого угла формы. Так, для любой формы `f` левый верхний угол имеет координаты (0, 0), а правый нижний угол – координаты (`f.Width`, `f.Height`). В примере 8.17 с помощью метода `Observable.map` создается новое событие `Click`, которое отображает координаты указателя мыши в координаты относительно центра формы.

Пример 8.17. Преобразование аргументов события с помощью метода Observable.map

```
> // Создание формы
open System.Windows.Forms

let form = new Form(Text="Relative Clicking", TopMost=true)

form.MouseClick.AddHandler(
    new MouseEventHandler(
        fun sender clickArgs ->
            printfn "MouseClickEvent    @ [%d, %d]" clickArgs.X clickArgs.Y
    )
);;

val form : System.Windows.Forms.Form =
  System.Windows.Forms.Form, Text: Relative Clicking

> // Создание нового события, предоставляющего координаты
// относительно центра формы
let centeredClickEvent =
    form.MouseClick
    |> Observable.map (fun clickArgs -> clickArgs.X - (form.Width  / 2),
                                        clickArgs.Y - (form.Height / 2))
// Подписка на событие
centeredClickEvent
|> Observable.add (fun (x, y) ->
                       printfn "CenteredClickEvent @ [%d, %d]" x y);;

val centeredClickEvent : System.IObservable<int * int>

> // Для каждого щелчка на форме выводятся два сообщения:
// первое - с координатами относительно верхнего левого угла,
// и второе - с координатами относительно центра формы.
form.ShowDialog();;
MouseClickEvent    @ [4, 8]
CenteredClickEvent @ [-146, -142]
MouseClickEvent    @ [150, 123]
CenteredClickEvent @ [0, -27]
val it : DialogResult = Cancel
```

Использование модуля `Observable` дает возможность рассматривать события как обычные функции. Он привносит дополнительную выразительность и обеспечивает более широкие возможности управления событиями в F#, чем при работе с другими языками на платформе .NET.

Создание событий .NET

Для создания событий на языке F# требуется всего один дополнительный шаг – достаточно создать свойство, возвращающее `IEvent<_, _>`. Однако чтобы создать событие, которое сможет использоваться в других

языках .NET, необходимо добавить атрибут [<CLIEvent>]. Это подсказка компилятору F#, которая сообщает ему, как должен генерироваться код класса. При использовании этого класса в языке F# он будет вести себя точно так же, как и без атрибута [<CLIEvent>], однако для других языков .NET вместо свойства типа IEvent<_,_> появится событие .NET.

В примере 8.18 приводится переработанная версия примера с чашкой кофе, но теперь в нем вместо списка функций, которые должны вызываться для уведомления об опустошении чашки, используется обычное событие .NET.

Пример 8.18. Создание событий, совместимых с другими языками .NET

```
open System

[<Measure>]
type ml

type EmptyCoffeeCupDelegate = delegate of obj * EventArgs -> unit

type EventfulCoffeeCup(amount : float<ml>) =
    let mutable m_amountLeft = amount
    let m_emptyCupEvent = new Event<EmptyCoffeeCupDelegate, EventArgs>()

    member this.Drink(amount) =
        printfn "Drinking %.1f..." (float amount)
        m_amountLeft <- min (m_amountLeft - amount) 0.0<ml>
        if m_amountLeft <= 0.0<ml> then
            m_emptyCupEvent.Trigger(this, new EventArgs())

    member this.Refil(amountAdded) =
        printfn "Coffee Cup refilled with %.1f" (float amountAdded)
        m_amountLeft <- m_amountLeft + amountAdded

    [<CLIEvent>]
    member this.EmptyCup = m_emptyCupEvent.Publish
```

II

Программирование на языке F#

9

Сценарии

В предыдущих главах вы видели, насколько язык F# эффективен с различными парадигмами программирования. Но F# отлично подходит не только для программирования в разных *стилях*, он точно так же прекрасно подходит для создания программ других *типов*. Язык F# удобен не только для разработки клиентских приложений, но и для создания сценариев.

Под *сценариями* (*scripting*) обычно подразумеваются программы, не имеющие пользовательского интерфейса и предназначенные для решения конкретной задачи. В противоположность выполняемым программам, в большинстве скриптовых языков сценарии полностью состоят из кода и интерпретируются во время исполнения.

При программировании в этом стиле обычно приходится платить скоростью выполнения за возможность легко изменять и развертывать программу. Вместо того чтобы создавать сложное решение, способное работать где угодно, вы можете просто переносить сценарий с компьютера на компьютер и изменять его по мере необходимости.

Язык F#, возможно, лучше всего подходит для разработки клиентских приложений, однако с точки зрения разработки сценариев у этого языка имеется еще одно важное преимущество: вы уже знаете этот язык. Вам не придется изучать еще один язык программирования для разработки сценариев и вы сможете повторно использовать существующий код. Кроме того, код F# никогда не интерпретируется – он всегда сначала компилируется. Благодаря этому сценарии на языке F# могут выполняться быстрее, чем сценарии, написанные на чистых скриптовых языках.

Если вам необходимо решить простую задачу и не требуется создавать сложный пользовательский интерфейс, подумайте о возможности создания сценария. В этой главе вы узнаете о некоторых конструкциях,

доступных в файлах сценариев, а также познакомитесь с некоторыми рецептами по созданию сценариев на языке F#, которым вы быстро найдете применение.

Файлы сценариев на языке F#

После установки F# в системе автоматически регистрируется новый тип файлов с расширением *.fsx* как файлов сценариев на языке F#. Для этих файлов специально предусмотрен простой способ их исполнения. Двойной щелчок мышью на файле сценария F# запустит Visual Studio и откроет этот файл для редактирования. Если щелкнуть правой кнопкой мыши на таком файле в Проводнике Windows (Windows Explorer), его можно запустить на выполнение, выбрав пункт Run with F# Interactive (Выполнить в интерактивной оболочке F#) контекстного меню, как показано на рис. 9.1.

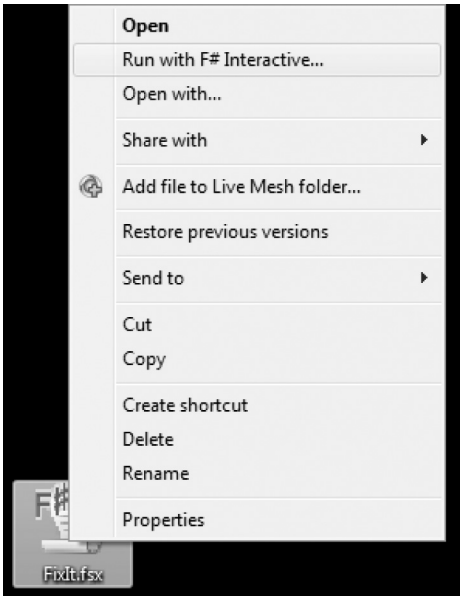


Рис. 9.1. *Интеграция файлов сценариев F# в оболочку Windows*

При выполнении сценария на языке F# его содержимое просто передается консольной версии интерактивной оболочки F# – `fsi.exe`.

Файл сценария выполняется точно так же, как если бы он был открыт в среде Visual Studio и передан непосредственно в окно FSI. (В действительности именно так все и происходит.) Исходный код сценария будет скомпилирован и выполнен и произведет все действия, предусмотренные сценарием. В файлах сценариев на языке F# вы можете исполь-

зовать все возможности языка F#, а также некоторые дополнительные возможности, упрощающие разработку сценариев.

Директивы

Наиболее заметным отличием сценариев от обычных файлов с исходным кодом на языке F# является возможность использования директив компилятора – специальных подсказок и команд компилятору F#.

Общие директивы

Следующие директивы допускаются в любых исходных текстах на языке F#, но чаще всего они используются именно в сценариях.

__SOURCE_DIRECTORY__

Директива __SOURCE_DIRECTORY__ возвращает текущий каталог, в котором находится выполняемый код. Эту директиву особенно удобно использовать в сценариях для определения его местоположения. Во время компиляции все экземпляры директивы __SOURCE_DIRECTORY__ замещаются строковыми литералами:

```
C:\Program Files\Microsoft F#\v4.0>fsi.exe

Microsoft F# Interactive, (c) Microsoft Corporation, All Rights Reserved
F# Version 1.9.8.0, compiling for .NET Framework Version v4.0.20620

Please send bug reports to fsbugs@microsoft.com
For help type #help;;

> __SOURCE_DIRECTORY__;;
val it : string = "C:\Program Files\Microsoft F#\v4.0"
> #q;;

C:\Program Files\Microsoft F#\v4.0>
```

__SOURCE_FILE__

Директива __SOURCE_FILE__ возвращает имя текущего файла. Объединив ее с директивой __SOURCE_DIRECTORY__, можно определить, какой сценарий в действительности выполняется.

В примере 9.1 демонстрируется сценарий, который выводит на консоль значение, возвращаемое директивой __SOURCE_FILE__. Директивы __SOURCE_DIRECTORY__ и __SOURCE_FILE__ заменяются строковыми литералами на этапе компиляции.

Пример 9.1. Использование директив __SOURCE_FILE__ и __SOURCE_DIRECTORY__

```
C:\Program Files\Microsoft F#\v4.0>type AweomestScriptEver.fsx
printfn "__SOURCE_DIRECTORY__ = %s" __SOURCE_DIRECTORY__
```

```
printfn "__SOURCE_FILE__ = %s" __SOURCE_FILE__

C:\Program Files\Microsoft F#\v4.0>fsi AweomestScriptEver.fsx
__SOURCE_DIRECTORY__ = C:\Program Files\Microsoft F#\v4.0
__SOURCE_FILE__ = AweomestScriptEver.fsx
```

Директивы сценариев

Ниже перечислены директивы, которые не могут использоваться в обычных файлах с кодом на языке F#. Они предназначены исключительно для использования в сценариях F# и позволяют рассматривать сценарии как однофайловые проекты Visual Studio. Эти директивы также могут использоваться в окне FSI в среде Visual Studio.

#r

Директива `#r` определяет ссылку на сборку. Это позволяет использовать в файлах сценариев F# типы, определенные в другой сборке .NET, возможно даже написанной на другом языке программирования, таком как C#. Директиве `#r` может передаваться простая строка с именем файла сборки или полное имя сборки, включающее имя, номер версии, культуру и так далее.

В примере 9.2 показано, как загрузить библиотеки Windows Forms для отображения списка файлов, находящихся в папке *Pictures*, в виде таблицы.

Пример 9.2. Использование ссылок на сборки в сценариях

```
#r "System.Windows.Forms.dll"
#r "System.Drawing.dll"

open System
open System.IO
open System.Collections.Generic
open System.Drawing
open System.Windows.Forms

// val images : seq<string * System.Drawing.Image>
let images =
    let myPictures =
        Environment.GetFolderPath(Environment.SpecialFolder.MyPictures)

    Directory.GetFiles(myPictures, "*.JPG")
    |> Seq.map (fun filePath ->
            Path.GetFileName(filePath),
            Bitmap.FromFile(filePath))

// Создание таблицы и добавление ее на форму
let dg = new DataGridView(Dock = DockStyle.Fill)
dg.DataSource <- new List<_>(images)
```

```
let f = new Form()
f.Controls.Add(dg)

f.ShowDialog()
```

Порядок поиска сборок:

1. Путь поиска (Include Path) (обсуждается ниже)
2. Текущий каталог сценария
3. Глобальный кэш сборок .NET (Global Assembly Cache)

#I

Директива `#I` добавляет новый каталог в путь поиска сборок. Если вы хотите хранить сценарии и сборки, от которых они зависят, в отдельных папках, то с помощью директивы `#I` вы можете добавить папку в путь поиска, чтобы компилятор имел возможность находить сборки, указанные в директивах `#r`.

В следующем примере показано добавление нескольких каталогов в путь поиска сборок, чтобы при их загрузке с помощью директив `#r` компилятор смог их найти:

```
// Подключить любую установленную версию Managed DirectX...
#I @"C:\WINDOWS\Microsoft.NET\DirectX for Managed Code\1.0.2904.0"
#I @"C:\WINDOWS\Microsoft.NET\DirectX for Managed Code\1.0.2905.0"
#I @"C:\WINDOWS\Microsoft.NET\DirectX for Managed Code\1.0.2906.0"
#I @"C:\WINDOWS\Microsoft.NET\DirectX for Managed Code\1.0.2907.0"

#r "Microsoft.DirectX.dll"
#r "Microsoft.DirectX.Direct3D.dll"
#r "Microsoft.DirectX.Direct3Dx.dll"

// ...
```

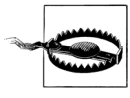
#load

Директива `#load` открывает другой файл F# по относительному пути и выполняет его. Эту директиву можно использовать, когда необходимо разбить сценарий на несколько файлов или когда нужно повторно использовать существующий код. Загрузка файла выполняется так, как если бы вы открыли файл и все его содержимое скопировали непосредственно в окно FSI. В примере 9.3 показана загрузка файла, содержащего диаграммы и графики, в файл сценария, который выполняет только вычисления.

Пример 9.3. Загрузка файла с кодом в сценарий F#

```
#load "Plotting.fs"

open System
```

```
let data =
    seq {
        let rng = new Random()
        for i in 1 .. 10000 do
            yield (rng.NextDouble(), rng.NextDouble())
    }

// Функция plot определена в файле 'Plotting.fs'
plot "Random plot" data
```



Не злоупотребляйте директивами `#load` в сценариях F#. Они усложняют сопровождение и развертывание таких сценариев. Если в сценарии необходимо задействовать несколько файлов с кодом, подумайте о создании настоящего проекта.

Рецепты по созданию сценариев

В этом разделе рассказывается о нескольких простых утилитах, которые сделают ваши сценарии более эффективными, например использование выделения цветом при выводе данных в консоль или автоматизация ввода данных в Microsoft Excel.

Выделение цветом

Создание пользовательского интерфейса не является основной задачей сценариев – обычно сценарии используются для решения задач, которые не нуждаются в данных, получаемых от пользователя. Однако возможность выделения цветом при выводе в консоль может существенно повысить читаемость выводимых данных. В примере 9.4 определяется функция `cprintfn`, которая ведет себя точно так же, как функция `printfn`, принимая строку формата и дополнительные аргументы, но помимо этого она принимает дополнительный параметр, определяющий цвет.

Пример 9.4. Вывод данных с указанным цветом

```
/// Выделение цветом для повышения читаемости
let cprintfn c fmt =
    Printf.kprintf
        (fun s ->
            let orig = System.Console.ForegroundColor
            System.Console.ForegroundColor <- c;
            System.Console.WriteLine(s)
            System.Console.ForegroundColor <- orig)
        fmt
```

Функция `cprintfn` использует функцию `Printf.kprintf` из стандартной библиотеки F#, которая выполняет все преобразования, предусмотренные спецификаторами формата, такими как `"%d %f %s"`, а также принимает дополнительные аргументы и вызывает указанную функцию для

получения окончательного результата. Затем функция `cprintfn` просто изменяет цвет шрифта в консоли и выводит данные.

Следующий код выводит на консоль текст различным цветом. В последнем блоке этого кода используются некоторые хитрости функционального программирования для вывода строки, в которой каждый символ выводится своим цветом. Сможете ли вы понять, как это делается? (Подсказка: экземпляр типа `string` может интерпретироваться как экземпляр типа `seq<char>`.)

```
open System

cprintfn ConsoleColor.Blue "Hello, World! in blue!"
cprintfn ConsoleColor.Red "... and in red!"
cprintfn ConsoleColor.Green "... and in green!"

let rotatingColors =
    seq {
        let i = ref 0
        let possibleColors = Enum.GetValues(typeof<ConsoleColor>)
        while true do
            yield (enum (!i) : ConsoleColor)
            i := (!i + 1) % possibleColors.Length
    }

"Experience the rainbow of possibility!"
|> Seq.zip rotatingColors
|> Seq.iter (fun (color, letter) -> cprintfn color "%c" letter)
```

Воспроизведение звука

Помимо возможности вывода данных на консоль различным цветом, для улучшения своих сценариев можно также использовать возможность воспроизведения звуковых сигналов. Реализовать это можно с помощью метода `System.Console.Beep`, который принимает в виде параметров частоту и длительность сигнала. При чрезмерном использовании звуковые сигналы могут раздражать пользователя, однако подача звукового оповещения по окончании работы сценария или в случае наступления определенного события может оказаться весьма полезной возможностью.

Ниже приводятся определения двух функций, `victory` и `defeat`, которые предназначены для подачи сигнала в случае успеха или ошибки. Функция `victory` подает серию сигналов, тон которых постепенно повышается, чтобы передать радость успеха, а функция `defeat` подает серию сигналов все более и более низких тонов:

```
open System

/// Успех!
let victory() =
```

```
    for frequency in [100 .. 50 .. 2000] do
        Console.Beep(frequency, 25)

/// Неудача :(
let defeat() =
    for frequency in [2000 .. -50 .. 37] do
        Console.Beep(frequency, 25)
```

Обход дерева каталогов

В сценариях часто возникает необходимость обработать все файлы в определенном каталоге. Наиболее простой и эффективный способ обработки всех файлов заключается в том, чтобы использовать выражение последовательности, определенное в примере 9.5. Напомню, что в выражениях последовательности инструкция `yield!` объединяет последовательность элементов в единую последовательность, как это делает функция `Seq.concat`.

Пример 9.5. Получение списка всех подкаталогов, начиная с указанного каталога

```
open System.IO

let rec filesUnder basePath =
    seq {
        yield! Directory.GetFiles(basePath)
        for subDir in Directory.GetDirectories(basePath) do
            yield! filesUnder subDir
    }
```

Этот фрагмент может использоваться для обхода дерева каталогов и выбора определенных файлов, которые затем будут обрабатываться вашим сценарием. Примечательно, что использование директивы `__SOURCE_DIRECTORY__` позволяет организовать поиск файлов в каталоге, в котором находится сценарий.

Следующий фрагмент использует функцию `filesUnder` для копирования всех файлов в формате JPEG из каталога со сценарием в новую папку:

```
open System.IO

let rec filesUnder basePath =
    seq {
        yield! Directory.GetFiles(basePath)
        for subDir in Directory.GetDirectories(basePath) do
            yield! filesUnder subDir
    }

let NewImageFolder = @"D:\NewImagesFolder\"

__SOURCE_DIRECTORY__
|> filesUnder
```

```
|> Seq.filter(fun filePath -> filePath.ToUpper().EndsWith("JPG"))
|> Seq.iter(fun filePath ->
                        let fileName = Path.GetFileName(filePath)
                        let destPath = Path.Combine(NewImageFolder, fileName)
                        File.Copy(filePath, destPath))
```

Простой запуск процессов

Другой распространенной задачей, решаемой с помощью сценариев, является запуск дополнительных инструментов. Возможность запуска новых процессов встроена в библиотеку базовых классов .NET, а самый простой способ состоит в том, чтобы вызвать метод `Process.Start` и просто дождаться завершения процесса:

```
open System.Diagnostics
Process.Start("notepad.exe", "file.txt").WaitForExit()
```

В более сложных случаях может потребоваться проверить результаты работы процесса – код его завершения или его вывод на консоль.

В примере 9.6 определяется функция с именем `shellExecute`, которая запускает программу с указанными аргументами и возвращает кортеж с кодом ее завершения и с выводом программы на консоль.

Пример 9.6. Запуск нового процесса

```
open System.Text
open System.Diagnostics

/// Запускаем новый процесс. Получаем кортеж (код_завершения, stdout)
let shellExecute program arguments =

    let startInfo = new ProcessStartInfo()
    startInfo.FileName  <- program
    startInfo.Arguments <- arguments

    startInfo.UseShellExecute        <- false
    startInfo.RedirectStandardOutput <- true

    let proc = new Process()
    proc.EnableRaisingEvents <- true

    // Добавить обработчик события 'OutputDataRecieved', чтобы можно было
    // сохранить поток STDOUT процесса.
    let driverOutput = new StringBuilder()
    proc.OutputDataReceived.AddHandler(
        DataReceivedEventHandler(
            (fun sender args -> driverOutput.AppendLine(args.Data) |> ignore)
        )
    )

    proc.StartInfo <- startInfo
    proc.Start() |> ignore
```

```
proc.BeginOutputReadLine()

proc.WaitForExit()
(proc.ExitCode, driverOutput.ToString())
```

Автоматизация операций в Microsoft Office

Еще одна область применения сценариев F# – автоматизация или программное выполнение рутинных задач. Следующий сценарий демонстрирует, как можно автоматизировать выполнение простых задач в Microsoft Office.

Пример 9.7 автоматизирует выполнение операций в Microsoft Word. Он отправляет на печать все документы, находящиеся в той же папке, что и сценарий, и изменяет настройки печати так, чтобы на одном листе выводилось по четыре страницы.

API для взаимодействия с Microsoft Word достаточно прост – единственная сложность в примере 9.7 связана с тем, что Word COM API ожидает параметры в виде `byref<obj>`, поэтому пришлось добавить функцию `comarg`, преобразующую значение в изменяемый экземпляр типа `obj`. (Более подробно тип `byref<_>` обсуждается в приложении B.)

Пример 9.7. Вывод на печать всех документов в папке

```
#r "stdole.dll"
#r "Microsoft.Office.Interop.Word"

open Microsoft.Office.Interop.Word

let private m_word : ApplicationClass option ref = ref None

let openWord()        = m_word := Some(new ApplicationClass())
let getWordInstance() = Option.get !m_word
let closeWord()       = (getWordInstance()).Quit()

// COM-объекты ожидают тип byref<obj>, ссылочные ячейки будут
// преобразованы компилятором в значения типа byref<obj>.
let comarg x = ref (box x)

let openDocument filePath =
    printfn "Opening %s..." filePath
    getWordInstance().Documents.Open(comarg filePath)

let printDocument (doc : Document) =

    printfn "Printing %s..." doc.Name

    doc.PrintOut(
        Background        = comarg true,
        Range             = comarg WdPrintOutRange.wdPrintAllDocument,
        Copies            = comarg 1,
```

```
        PageType          = comarg WdPrintOutPages.wdPrintAllPages,
        PrintToFile       = comarg false,
        Collate           = comarg true,
        ManualDuplexPrint = comarg false,
        PrintZoomColumn   = comarg 2, // Страниц поперек листа
        PrintZoomRow      = comarg 2) // Страниц по высоте листа

let closeDocument (doc : Document) =
    printfn "Closing %s..." doc.Name
    doc.Close(SaveChanges = comarg false)

// ------------------------------------------------------------

open System
open System.IO

try
    openWord()

    printfn "Printing all files in [%s]..." __SOURCE_DIRECTORY__

    Directory.GetFiles(__SOURCE_DIRECTORY__, "*.docx")
    |> Array.iter
        (fun filePath ->
            let doc = openDocument filePath
            printDocument doc
            closeDocument doc)
finally
    closeWord()

printfn "Press any key..."
Console.ReadKey(true) |> ignore
```

В примере 9.8 создается новая книга Microsoft Excel и в таблицу записывается все содержимое папки *Pictures*. С той же легкостью можно было бы сохранить результаты каких-либо вычислений.

Сначала сценарий запускает приложение Excel, создавая новую книгу, и записывает заголовки столбцов с помощью функции `setCellText`. Затем сценарий выполняет обход всех файлов в папке *My Pictures* и записывает в таблицу имя файла, его размер и другие данные.

Пример 9.8. Создание листов в книге Excel

```
#r "Microsoft.Office.Interop.Excel"

open System
open System.IO
open System.Reflection
open Microsoft.Office.Interop.Excel

let app = ApplicationClass(Visible = true)
```

```
let sheet = app.Workbooks
              .Add()
              .Worksheets.[1] :?> _Worksheet

let setCellText (x : int) (y : int) (text : string) =
    let range = sprintf "%c%d" (char (x + int 'A')) (y+1)
    sheet.Range(range).Value(Missing.Value) <- text

let printCsvToExcel rowIdx (csvText : string) =
    csvText.Split([| ',' |])
    |> Array.iteri (fun partIdx partText ->
                        setCellText partIdx rowIdx partText)

let rec filesUnderFolder basePath =
    seq {
        yield! Directory.GetFiles(basePath)
        for subFolder in Directory.GetDirectories(basePath) do
            yield! filesUnderFolder subFolder
    }

// Вывод заголовков столбцов
printCsvToExcel 0 "Directory, Filename, Size, Creation Time"

// Вывод данных
filesUnderFolder (Environment.GetFolderPath(Environment.SpecialFolder.
MyPictures))
|> Seq.map (fun filename -> new FileInfo(filename))
|> Seq.map (fun fileInfo -> sprintf "%s, %s, %d, %s"
                                fileInfo.DirectoryName
                                fileInfo.Name
                                fileInfo.Length
                                (fileInfo.CreationTime.ToShortDateString()))
|> Seq.iteri (fun idx str -> printCsvToExcel (idx + 1) str)
```

Сценарии F# позволяют использовать код на языке F# за рамками обычных «проектов и решений» (projects and solutions) Visual Studio. Сценарии F# могут использоваться для решения многих задач, автоматизировать которые на других языках, таких как C#, было бы слишком утомительно, а на таких, как Python, так же просто, но потребовало бы установить и настроить дополнительное программное обеспечение.

Используя сценарии F#, вы можете автоматизировать такие задачи, как загрузка файлов на веб-сервер, извлечение и анализ данных в лог-файлах и так далее. Использование языка F# для разработки сценариев – это всего лишь один из способов применения языка для повышения эффективности труда.

10

Вычислительные выражения

В главах 2, 3 и 4 мы рассматривали генераторы списков, последовательностей и массивов, представляющие собой способ получения коллекции упорядоченных элементов в виде списков, последовательностей и массивов соответственно. Все эти генераторы – не просто синтаксический сахар, встроенный в язык, скорее это особая разновидность инструкций, которые называются *вычислительными выражениями* (*computation expressions*) (иногда их менее формально называют *потоками операций (workflows)*).

Фактически та же самая техника, которая используется для упрощенного объявления последовательностей, может применяться для асинхронного программирования или для создания предметно-ориентированных языков (Domain-specific Languages, DSL).

Проще говоря, вычислительные выражения позволяют определять, *как* будет выполняться код F#. Использование вычислительных выражений, выполняющих все рутинные операции за кулисами, может существенно уменьшить избыточность кода и до определенной степени даже расширить сам язык F#.

На пути к вычислительным выражениям

Давайте кратко вспомним, для чего нужны выражения последовательностей, – они позволяют определять последовательности за счет встраивания кода F# в вычислительное выражение `seq`. В примере 10.1 показано вычислительное выражение, генерирующее дни года.

Обратите внимание, что внутри блока `seq { }` допускается использовать практически любые конструкции языка F#, а результатом такого выражения является последовательность элементов, возвращаемых с помощью ключевого слова `yield`.

Пример 10.1. Выражение последовательности, перечисляющее дни года

```
> // Последовательность для воспроизведения кортежей (месяц, день)
let daysOfTheYear =
    seq {
        let months =
            [
                "Jan"; "Feb"; "Mar"; "Apr"; "May"; "Jun";
                "Jul"; "Aug"; "Sep"; "Oct"; "Nov"; "Dec"
            ]

        let daysInMonth month =
            match month with
            | "Feb"
                -> 28
            | "Apr" | "Jun" | "Sep" | "Nov"
                -> 30
            | _ -> 31

        for month in months do
            for day = 1 to daysInMonth month do
                yield (month, day)
    };;

val daysOfTheYear : seq<string * int>

> daysOfTheYear;;
val it : seq<string * int> =
  seq [("Jan", 1); ("Jan", 2); ("Jan", 3); ("Jan", 4); ...]
> Seq.length daysOfTheYear;;
val it : int = 365
```

Понятно, что с помощью кода, который можно поместить внутрь выражений последовательностей и соответственно внутрь вычислительных выражений, можно выполнять достаточно сложные вычисления. Фактически к коду, который может использоваться внутри вычислительных выражений, предъявляются всего два требования:

- Внутри вычислительных выражений не могут объявляться новые типы.
- Внутри вычислительных выражений не могут использоваться изменяемые значения – вместо них следует использовать ссылочные ячейки (ref cells).

Как уже заявлялось, вычислительные выражения могут существенно упростить код. В качестве доказательства рассмотрим пример 10.2, в котором делается попытка выполнить серию простых арифметических операций, но, чтобы избежать появления исключений `DivideByZeroException` при делении, каждая операция деления обернута собственным типом, возвращающим признак успеха или неудачи.

В этом примере вычисляется общее сопротивление электрической цепи, содержащей три резистора, включенных параллельно, для чего требуется выполнить четыре операции деления. После каждой операции деления требуется проверить результат деления на равенство значению Success или DivByZero, поэтому код получается чересчур сложным, так как после каждого деления нам нужно убедиться в том, можно ли продолжать вычисления или нужно прекратить их в случае получения DivByZero.

Пример 10.2. Определение состояния вычислительного процесса

```
type Result = Success of float | DivByZero

let divide x y =
    match y with
    | 0.0 -> DivByZero
    | _   -> Success(x / y)

// Общее сопротивление трех резисторов, включенных параллельно,
// определяется по формуле: 1/R_e = 1/R_1 + 1/R_2 + 1/R_3
let totalResistance r1 r2 r3 =
    let r1Result = divide 1.0 r1
    match r1Result with
    | DivByZero
        -> DivByZero
    | Success(x)
        -> let r2Result = divide 1.0 r2
        match r2Result with
        | DivByZero
            -> DivByZero
        | Success(y)
            -> let r3Result = divide 1.0 r3
            match r3Result with
            | DivByZero
                -> DivByZero
            | Success(z)
                -> let finalResult = divide 1.0 (x + y + z)
                   finalResult
```

Этот пример работает именно так, как и ожидается, но совершенно очевидно, что программирование в таком стиле утомительно и может приводить к ошибкам. В идеале можно было бы определить функцию с именем let_with_check, которая выполняла бы проверки автоматически. Она могла бы проверять результат вызова функции divide и в случае получения значения Success выполнять следующую строку кода, в противном случае возвращала бы DivByZero:

```
let totalResistance r1 r2 r3 =
    let_with_check x = divide 1.0 r1
    let_with_check y = divide 1.0 r2
```

```
    let_with_check z = divide 1.0 r3
    return_with_check 1.0 (x + y + z)
```

Однако функция `let_with_check` не может быть реализована на языке F#, потому что нарушит нормальный порядок выполнения кода. То есть в языке F# нет способа прервать выполнение функции преждевременно, кроме генерации исключения или использования каскада выражений `if/match`.

Единственный способ решить эту проблему заключается в разделении этой функции на две части так, чтобы функция `let_with_check` принимала два параметра: первый – результат деления, а второй – функция, выполняющая оставшуюся часть вычислений. (Как вы помните из главы 7, это называется *продолжением*.)

В примере 10.3 определяется функция `let_with_check`, которая действует именно так. Она принимает два параметра: результат операции деления и функцию, представляющую оставшуюся часть вычислений. Если операция деления возвращает `DivByZero`, то оставшаяся часть вычислений не выполняется.

Пример 10.3. Определение функции let_with_check

```
let let_with_check result restOfComputation =
    match result with
    | DivByZero  -> DivByZero
    | Success(x) -> restOfComputation x

let totalResistance r1 r2 r3 =
    let_with_check
        (divide 1.0 r1)
        (fun x ->
            let_with_check
                (divide 1.0 r2)
                (fun y ->
                    let_with_check
                        (divide 1.0 r3)
                        (fun z -> divide 1.0 (x + y + z))
                )
        )
```

Из-за дополнительных отступов и лямбда-выражений этот вариант выглядит хуже предыдущего. Однако давайте попробуем переписать функцию `totalResistance` еще раз, на этот раз поместив лямбда-выражения в одну строку с вызовом функции.

В этом случае код выглядит практически так, как мы хотели, наглядно демонстрируя, что выполнение функции будет немедленно остановлено, как только будет получено первое значение `DivByZero`:

```
let totalResistance r1 r2 r3 =
    let_with_check (divide 1.0 r1) (fun x ->
    let_with_check (divide 1.0 r2) (fun y ->
```

```
    let_with_check (divide 1.0 r3) (fun z ->
    divide 1.0 (x + y + z) ) ) )
```

Построители вычислительных выражений

Вычислительные выражения по сути являются тем же преобразованием кода, который был показан в предыдущем примере с функцией `let_with_check`. Давайте посмотрим, как можно использовать вычислительные выражения в языке F# для упрощения кода.

В примере 10.4 показано, как можно создать собственный поток операций (workflow). О том, что происходит за кулисами, мы поговорим ниже, а пока просто обратите внимание на реализацию метода `Bind`, на блок `defined {}` и на использование ключевого слова `let!`.

Пример 10.4. Вычислительное выражение, которое может завершиться успехом/неудачей

```
type Result = Success of float | DivByZero

let divide x y =
    match y with
    | 0.0 -> DivByZero
    | _   -> Success(x / y)

type DefinedBuilder() =

    member this.Bind ((x : Result), (rest : float -> Result)) =
        // Если результатом является Success(_), выполнить функцию rest.
        // В противном случае завершить работу преждевременно.
        match x with
        | Success(x) -> rest x
        | DivByZero  -> DivByZero

    member this.Return (x : 'a) = x

// Создание построителя вычислительного выражения
let defined = DefinedBuilder()

let totalResistance r1 r2 r3 =
    defined {
        let! x = divide 1.0 r1
        let! y = divide 1.0 r2
        let! z = divide 1.0 r3
        return divide 1.0 (x + y + z)
    }
```

Вы можете проверить этот код в окне FSI и убедиться, что эта версия возвращает тот же результат, при этом количество строк кода сократилось на треть!

```
> totalResistance 0.01 0.75 0.33;;
val it : Result = Success 0.009581881533
> totalResistance 0.00 0.55 0.75;;
val it : Result = DivByZero
```

Все, что делают вычислительные выражения, – это разбивают выражение на несколько вызовов функций *построителя вычислительного выражения (computation expression builder)*. В предыдущем примере это был построитель типа `DefinedBuilder`. Для использования построителя вычислительного выражения достаточно указать имя его экземпляра перед фигурными скобками {}, которые окружают код, предназначенный для выполнения. В данном примере использовался построитель `defined`, являющийся экземпляром `DefinedBuilder`.

Каждое выражение `let!` заменяется вызовом метода `Bind` построителя, которому в первом параметре передается результат выражения, стоящего справа от знака равенства, а во втором параметре – функция, представляющая оставшуюся часть вычислений. Это объясняет, почему работа вычислительного выражения прекращается при получении первого же значения `DivByZero`. Реализация метода `Bind` построителя вычислительного выражения не выполняет оставшуюся часть вычислений, если операция деления возвращает `DivByZero`. Оставшаяся часть вычислений выполняется, только если в результате деления получается `Success`.

В примере 10.5 приводится вычислительное выражение `defined` в развернутом виде, в котором оно поразительно похоже на код примера 10.3.

Пример 10.5. Вычислительное выражение в развернутой форме

```
// Развернутая форма
let totalResistance r1 r2 r3 =
    defined.Bind(
        (divide 1.0 r1),
        (fun x ->
            defined.Bind(
                (divide 1.0 r2),
                (fun y ->
                    defined.Bind(
                        (divide 1.0 r3),
                        (fun z ->
                            defined.Return(
                                divide 1.0 (x + y + z)
                            )
                        )
                    )
                )
            )
        )
    )
```

Внутри собственных построителей вычислительных выражений можно использовать обычный код на языке F#, но, кроме того, появляется возможность применять новые ключевые слова, такие как `let!`, `return` и `yield`. С помощью этих новых примитивов можно определять, как будет обрабатываться код в теле вычислительного выражения. (Например, предусмотреть расширение семантики ключевого слова `yield`.)

Предоставляя реализацию функции `Bind`, вы определяете, как будет вычисляться выражение `let!`. Аналогично, предоставляя реализацию функции `Return`, вы определяете, как будет действовать ключевое слово `return`. Однако построители вычислительных выражений могут иметь и другие методы, определяющие порядок выполнения кода. Полный перечень методов приводится в табл. 10.1.

Вычислительные выражения можно также рассматривать как способ вставки кода между различными этапами вычислений для выполнения промежуточных операций без явного определения их в вашем коде.

Таблица 10.1. Методы вычислительных выражений

Сигнатура метода	Описание
`member For: seq<'a> * ('a -> Result<unit>) -> Result<unit>`	Разрешает применение циклов `for`. В качестве параметров передаются значения, по которым выполняется цикл, и тело цикла `for`.
`member Zero: unit -> Result<unit>`	Разрешает применение выражений, возвращающих `unit`, таких как выражения `if` без ветки `else`, когда условие возвращает `false`.
`member Combine: Result<unit> * Result<'a> -> Result<'a>`	Разрешает объединение частей вычислительного выражения, например для последовательности из двух циклов `for`.
`member While: (unit -> bool) * Result<unit> -> Result<unit>`	Разрешает применение циклов `while`. В качестве параметров передается функция, которая определяет, продолжать или нет выполнение цикла `while`, и тело цикла.
`member Return: 'a -> Result<'a>`	Разрешает применение ключевого слова `return`.
`member ReturnFrom: 'a -> Result<'a>`	Разрешает применение ключевого слова `return!`.
`member Yield: 'a -> Result<'a>`	Разрешает применение ключевого слова `yield`.
`member YieldFrom: seq<'a> -> Result<'a>`	Разрешает применение ключевого слова `yield!`.

Таблица 10.1 (продолжение)

Сигнатура метода	Описание
`member Delay: (unit -> Result<'a>) -> Result<'a>`	Эта операция обычно используется в паре с `Combine`, чтобы обеспечить правильный порядок выполнения последовательности операций (когда они имеют побочные эффекты).
`member Run: Result<'a> -> Result<'a>`	Этот метод, если он определен, вызывается в самом начале вычислительного выражения.
`member Using: 'a * ('a -> Result<'b>) -> Result<'b> when 'a :> System.IDisposable`	Разрешает применение ключевых слов `use` и `use!`. Реализация метода `Using` отвечает за вызов метода `IDisposable.Dispose`.
`member Bind: Result<'a> * ('a -> Result<'b>) -> Result<'b>`	Разрешает применение ключевых слова `let!` и `do!`. (`do!` – это специализированная форма `let!`, когда происходит связывание с типом `Result<unit>`).
`member TryFinally: Result<'a> * (unit -> unit) -> Result<'a>`	Разрешает применение выражений `try/finally`. В качестве параметров передаются результат блока `try` и функция, представляющая блок `finally`.
`member TryWith: Result<'a> * (exn -> Result<'a>) -> Result<'a>`	Разрешает применение `try/with`. В качестве параметров передаются результат блока `try` и функция, представляющая блок `with`.

Ту или иную синтаксическую конструкцию, указанную выше, допускается использовать в вычислительных выражениях, только если построитель вычислительных выражений содержит реализацию соответствующих методов. Так, если бы в предыдущем примере отсутствовала реализация метода `Return`, попытка использовать ключевое слово `return` в последней строке привела бы к ошибке.

```
        return divide 1.0 (x + y + z)
--------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
stdin(259,9): error FS0039: The field, constructor or member 'Return' is not defined.
```

(Поле, конструктор или элемент "Tastiness" не определен.)

Поиск методов вычислительного выражения только по их именам имеет пару интересных следствий. Во-первых, методы построителей вычислительных выражений могут добавляться как методы расширения, что позволяет использовать вычислительные выражения с непредусмотренными типами. Во-вторых, благодаря перегрузке методов вы можете получить несколько реализаций методов вычислительных выражений, таких как `Bind`.

Собственные построители вычислительных выражений

Теперь вы знаете, как изменять выполнение различных языковых конструкций, определяя дополнительные действия, которые должны производиться при их выполнении. Но в каких случаях это может пригодиться? Очевидно, что вычислительные выражения в языке F# удобно использовать для определения успеха или неудачи вычислений, но кому-то может показаться сомнительным, что тот же самый прием может упростить асинхронное программирование.

Асинхронные вычислительные выражения

Представьте себе самые утомительные аспекты параллельного и асинхронного программирования – работу с потоками. Если вам необходимо, чтобы программа одновременно решала две задачи, вам потребуется создать новые потоки, дать им задание и затем получить обратно их результаты. Это также сложно, как и асинхронные операции ввода-вывода. Платформа .NET Framework содержит асинхронные операции ввода-вывода в виде методов `FileStream.BeginRead` и `FileStream.BeginWrite`. Однако для работы с этими конструкциями необходимо использовать жуткую смесь из интерфейсов `IAsyncResult`, делегатов `AsyncCallback` и огромного количества однотипного кода.

Даже если вас не волнуют все сложности, связанные с работой потоков и управлением незавершенными операциями вручную, все равно остается проблема обработки исключений. Кроме того, не существует достаточно удобного способа прервать выполнение операции, после того как она будет запущена.

Если бы была возможность упростить код управления потоками.... К счастью, такая библиотека существует, и она встроена в язык F#. В языке F# *асинхронные потоки операций* (*asynchronous workflow*) представляют собой тип построителей вычислительных выражений, специально предназначенный для выполнения асинхронных операций.

В примере 10.6 демонстрируется, как можно использовать асинхронные операции чтения и записи данных в файл с помощью вычислительного выражения. Обратите внимание на использование ключевого слова `let!` для выполнения асинхронных операций; все управление потоками выполнения осуществляется реализацией метода `Bind` построителя `async`. Подробнее о том, как действуют построитель вычислительных выражений `async` и метод `Async.Start`, будет рассказываться в главе 11, которая полностью посвящена параллельному и асинхронному программированию.

Пример 10.6. Асинхронные вычислительные выражения в языке F#

```
open System.IO

let asyncProcessFile (filePath : string) (processBytes : byte[] -> byte[]) =
    async {

        printfn "Processing file [%s]" (Path.GetFileName(filePath))

        let fileStream = new FileStream(filePath, FileMode.Open)
        let bytesToRead = int fileStream.Length

        let! data = fileStream.AsyncRead(bytesToRead)

        printfn
            "Opened [%s], read [%d] bytes"
            (Path.GetFileName(filePath))
            data.Length

        let data' = processBytes data

        let resultFile = new FileStream(filePath + ".results",FileMode.Create)
        do! resultFile.AsyncWrite(data', 0, data'.Length)
        printfn "Finished processing file [%s]" <| Path.GetFileName(filePath)
    } |> Async.Start
```

При добавлении вычислительных выражений в язык F# не предполагалось, что они будут использоваться для асинхронных операций, скорее они рассчитаны на реализацию некоторых полезных типичных задач программирования.

Если вам нравится быть оригиналом, то можете называть вычислительные выражения разновидностью *моноидов из категории эндофункторов*, но я не рекомендую использовать такое название.

Вычислительное выражение округления

При выполнении некоторых типов математических вычислений может возникнуть потребность в ограничении точности расчетов, например при вычислении длин, имеющих смысл в реальном мире. (Если, конечно, вы не собираетесь производить линейки, позволяющие измерять длины с точностью до пикометров.)

В примере 10.7 определяется вычислительное выражение, выполняющее округление. Обратите внимание, что метод `Bind` изменяет значение, которое ему передается. Кроме того, так как метод `Bind` может принимать только значения типа `float`, компилятор будет выдавать сообщение об ошибке при использовании `let!` с другими типами.

Пример 10.7. Реализация вычислительного выражения, выполняющего округление

```
open System

// Построитель вычислительного выражения, выполняющий округление результата,
// для ограничения количества значимых десятичных разрядов.
type RoundingWorkflow(sigDigs : int) =

    let round (x : float) = Math.Round(float x, sigDigs)

    // Так как результатом может быть только значение типа float, вы сможете
    // использовать let! только с типом float.
    // (В противном случае компилятор генерирует сообщение об ошибке.)
    member this.Bind(result : float, rest : float -> float) =
        let result' = round result
        rest result'

    member this.Return (x : float) = round x

let withPrecision sigDigs = new RoundingWorkflow(sigDigs)
```

Ниже показано применение вычислительного выражения, выполняющего округление:

```
> // Проверка вычислительного выражения, выполняющего округление
let test =
    withPrecision 3 {
        let! x = 2.0 / 12.0
        let! y = 3.5
        return x / y
    };;

val test : float = 0.048
```

Вычислительное выражение, сохраняющее состояние

В качестве более практичного примера вычислительного выражения рассмотрим поток операций для отслеживания состояния. Представим, что мы разрабатываем библиотеку для веб-скриптов. Каждое действие, такое как щелчок мышью по ссылке или кнопке, изменяет текущее состояние – текущий URL, HTML, cookies-сессии и так далее, т. е. данные, которые должны передаваться следующему этапу вычислений.

В примере 10.8 приводится некоторый гипотетический код сценария веб-сессии, который становится слишком громоздким, как только возникает необходимость передачи текущего состояния каждому этапу вычислений. (На первый взгляд проще было бы использовать прямой конвейерный оператор |>. Однако в этом случае необходимо, чтобы каждый метод в последовательности возвращал один и тот же объект

с информацией о состоянии, что может быть бессмысленным для некоторых функций.)

Пример 10.8. Передача объекта с информацией о состоянии

```
let reviewBook() =
    let state1 = openPage "www.fsharp.net"
    let state2 = clickLink "F# Books" state1
    let (table, state3) = getHtmlTableFromHeader "Programming F#" (state2)
    let state4 = clickButton table "Five Star Rating" state3
    ...
```

С помощью вычислительного выражения мы можем избавиться от передачи объекта с состоянием через все вычисления. Как мы уже знаем, вычислительные выражения позволяют выполнять «дополнительную работу» за кулисами обычного кода F#, поэтому мы, скорее всего, сможем реализовать дополнительные действия по сохранению состояния:

```
let reviewBook() =
    state {
        do! OpenPage "www.fsharp.net"
        do! ClickLink "F# Books"
        let! table = GetHtmlTableWithHeader "Programming F#"
        do! ClickButton table "Five Star Rating"
    }
```

Конструирование вычислительного выражения

Добиться иллюзии сохранения состояния можно с помощью размеченного объединения типа `StatefulFunc`, состоящего из одного варианта, представляющего собой функцию, которая принимает начальное состояние и возвращает кортеж из результата и нового состояния:

```
type StatefulFunc<'state, 'result> = StatefulFunc of ('state -> 'result *
'state)
```

Каждый вызов метода `Bind` принимает существующий объект типа `StatefulFunc` и возвращает новый объект `StatefulFunc`, вызывая функцию, прикрепленную к существующему объекту `StatefulFunc`, и затем выполняя остальную часть вычислений (продолжение, переданное методу `Bind`).

Итак, допустим, что нам требуется написать вычислительное выражение, содержащее следующий код:

```
state {
    do! OpenWebpage "www.bing.com" // Изменяет информацию о состоянии
    do! EnterText "Jellyfish"      // Использует информацию о состоянии
    do! ClickButton "Search"   // Использует и изменяет информацию о состоянии
}
```

Использование типа `StatefulFunc` можно представить следующим образом:

```
// do! OpenWebpage "www.Bing.com"
let step1 =
    StatefulFunc(fun initialState ->
        let result, updatedState = OpenWebpage "www.bing.com" initialState
        result, updatedState)

// do! EnterText
let step2 =
    StatefulFunc(fun initialState ->
        let result, updatedState = EnterText "Jellyfish" initialState
        result, updatedState)

// do! ClickButton "Search"
let step3 =
    StatefulFunc(fun initialState ->
        let result, updatedState = ClickButton "Search" initialState
        result, initialState)
```

Результатом вычислительного выражения `state` является объект типа `StatefulFunc`, связанный с функцией, выполняющей все вычисления, передаваемый с объектом с состоянием, который создается в методе `Bind` внутри потока операций.

После создания объекта `StatefulFunc` весь процесс вычислений будет выглядеть примерно так:

```
StatefulFunc(fun initialState ->

    let result1, updatedState1 = OpenWebPage "www.bing.com" initialState

    updatedState1 |> (fun initialState ->
        let result2, updatedState2 = EnterText "Jellyfish" initialState

        updatedState3 |> (fun initialState ->
            let result3, updatedState3 = ClickButton "Search" initialState

            result3, updatedState3
        )
    )
```

Реализация

В примере 10.9 приводится полная реализация вычислительного выражения, сохраняющего информацию о состоянии. Благодаря реализации дополнительных методов, таких как `Combine` и `TryFinally`, внутри вычислительного выражения допускается использовать весь диапазон возможностей языка F#.

В сигнатуру каждого метода были добавлены явные аннотации типов, чтобы сделать их более понятными, однако на практике эти аннотации можно опустить.

Пример 10.9. Реализация вычислительного выражения, сохраняющего информацию о состоянии

```
open System

// Тип, представляющий функцию с состоянием. Функция принимает входное
// состояние и возвращает результат вместе обновленным состоянием.
type StatefulFunc<'state, 'result> =
    StatefulFunc of ('state -> 'result * 'state)

// Вызываем функцию, сохраняющую информацию о состоянии
let Run (StatefulFunc f) initialState = f initialState

type StateBuilder() =

    member this.Bind(
                        result : StatefulFunc<'state, 'a>,
                        restOfComputation : 'a -> StatefulFunc<'state, 'b>
                    ) =

        StatefulFunc(fun initialState ->
            let result, updatedState = Run result initialState
            Run (restOfComputation result) updatedState
        )

    member this.Combine(
                        partOne : StatefulFunc<'state, unit>,
                        partTwo : StatefulFunc<'state, 'a>
                       ) =

        StatefulFunc(fun initialState ->
            let (), updatedState = Run partOne initialState
            Run partTwo updatedState
        )

    member this.Delay(
                        restOfComputation : unit -> StatefulFunc<'state, 'a>
                     ) =

        StatefulFunc (fun initialState ->
            Run ( restOfComputation() ) initialState
        )

    member this.For(
                    elements : seq<'a>,
                    forBody : ('a -> StatefulFunc<'state, unit>)
                   ) =

        StatefulFunc(fun initialState ->
            let state = ref initialState
```

```
        for e in elements do
            let (), updatedState = Run (forBody e) (!state)
            state := updatedState

        // Возвращает unit * finalState
        (), !state
    )

member this.Return(x : 'a) =
    StatefulFunc(fun initialState -> x, initialState)

member this.Using<'a, 'state,
                  'b when 'a :> IDisposable>(
                  x : 'a,
                  restOfComputation : 'a -> StatefulFunc<'state, 'b>
                 ) =

    StatefulFunc(fun initialState ->
        try
            Run (restOfComputation x) initialState
        finally
            x.Dispose()
    )

member this.TryFinally(
                       tryBlock : StatefulFunc<'state, 'a>,
                       finallyBlock : unit -> unit
                      ) =
    StatefulFunc(fun initialState ->
        try
            Run tryBlock initialState
        finally
            finallyBlock()
    )

member this.TryWith(
                    tryBlock : StatefulFunc<'state, 'a>,
                    exnHandler : exn -> StatefulFunc<'state, 'a>
                   ) =

    StatefulFunc(fun initialState ->
        try
            Run tryBlock initialState
        with
        | e ->
            Run (exnHandler e) initialState
    )

member this.While(
                    predicate : unit -> bool,
                    body : StatefulFunc<'state, unit>
                  ) =
```

```
        StatefulFunc(fun initialState ->

            let state = ref initialState
            while predicate() = true do
                let (), updatedState = Run body (!state)
                state := updatedState

            // Возвращает unit * finalState
            (), !state
        )

    member this.Zero() =
        StatefulFunc(fun initialState -> (), initialState)

// Объявляем построителя вычислительных выражений состояния
let state = StateBuilder()

// Функции для получения и изменения информации о состоянии
let GetState          = StatefulFunc (fun state -> state, state)
let SetState newState = StatefulFunc (fun prevState -> (), newState)
```

В примере 10.10 показано использование вычислительного выражения `state` для реализации четырех функций калькулятора. (То же самое вычислительное выражение `state` можно использовать для реализации поискового веб-робота, однако код реализации такого робота легко перекрыл бы по объему эту главу.)

В состоянии сохраняется текущий результат и история операций калькулятора. Функции `Add`, `Subtract`, `Multiply` и `Divide` принимают состояние, изменяют его и возвращают новый объект типа `StatefulFunc`.

Пример 10.10. Использование вычислительного выражения для реализации калькулятора

```
let Add x =
    state {
        let! currentTotal, history = GetState
        do! SetState (currentTotal + x, (sprintf "Added %d" x) :: history)
    }

let Subtract x =
    state {
        let! currentTotal, history = GetState
        do! SetState (currentTotal - x,
                      (sprintf "Subtracted %d" x) :: history)
    }

let Multiply x =
    state {
        let! currentTotal, history = GetState
        do! SetState (currentTotal * x,
                      (sprintf "Multiplied by %d" x) :: history)
```

```
    }

let Divide x =
    state {
        let! currentTotal, history = GetState
        do! SetState (currentTotal / x,
                      (sprintf "Divided by %d" x) :: history)
    }
```

Использование функций, таких как `Add` и `Subtract`, с вычислительным выражением типа `state` создает впечатление, что состояние не передается, однако в действительности создается объект типа `StatefulFunc`, который может быть выполнен при передаче его функции `Run`:

```
> // Определение объекта типа StatefulFunc.
// При этом передавать параметр с состоянием для каждой функции не нужно.
let calculatorActions =
    state {
        do! Add 2
        do! Multiply 10
        do! Divide 5
        do! Subtract 8

        return "Finished"
    }

// Теперь выполняем объект StatefulFunc, передав ему начальное состояние
let sfResult, finalState = Run calculatorActions (0, []);;

val calculatorActions : StatefulFunc<(int * string list),string> =
  StatefulFunc <fun:Delay@324>
val sfResult : string = "Finished"
val finalState : int * string list =
  (-4, ["Subtracted 8"; "Divided by 5"; "Multiplied by 10"; "Added 2"])
```

Вычислительные выражения в языке F# являются настолько сложным понятием, что могут вызывать трудности даже у опытных разработчиков. Тем не менее вычислительные выражения могут использоваться для упрощения кода, позволяя выполнять дополнительные действия между вычислениями. Если вы поймаете себя на том, что пишете слишком много избыточного, типового кода, посмотрите, нельзя ли обернуть эту функциональность в вычислительное выражение.

11

Асинхронное и параллельное программирование

Основная причина необходимости овладения асинхронным и параллельным программированием заключается в том, что без этого вы не сможете полностью использовать потенциал аппаратных средств. Это особенно верно для программ, выполняющих большое количество вычислений (такие операции еще называют CPU-bound-операции). Раньше отказ от использования параллельных и асинхронных вычислений можно было оправдать сложностью и высокой вероятностью ошибок. Однако с появлением замечательных библиотек в составе Visual Studio 2010 совместно с возможностью писать программы в функциональном стиле эта проблема существенно утратила свою актуальность.

В этой главе мы сосредоточим свое внимание на проблеме повышения скорости выполнения программ на языке F# с помощью асинхронного и параллельного программирования. К концу этой главы вы научитесь выполнять код в различных контекстах (потоках выполнения), освоите асинхронное программирование с использованием библиотеки асинхронных вычислительных выражений, а также познакомитесь с библиотекой Parallel Extensions для .NET.

Но перед этим давайте сначала дадим определение асинхронному и параллельному программированию и выясним, почему это так важно.

Асинхронное программирование

Под *асинхронным программированием* (*asynchronous programming*) понимаются программы и операции, которые после запуска выполняются в фоновом режиме и завершаются «когда-нибудь потом». Например, большинство клиентских программ электронной почты получают новые письма асинхронно, в фоновом режиме, благодаря чему пользователю не приходится вручную проверять наличие новой почты.

Параллельное программирование

Под *параллельным программированием* (*parallel programming*) понимается разделение операций по обработке некоторых ресурсов с целью ускорения выполнения программы. Например, преобразование песни в формат MP3 можно распараллелить – разделить исходный файл на фрагменты и выполнять преобразование фрагментов одновременно.

Асинхронное и параллельное программирование увеличивают сложность приложений, однако их использование имеет ряд очевидных преимуществ:

Максимальное использование потенциала многопроцессорных систем и многоядерных процессоров

Современные компьютеры содержат несколько процессоров, каждый из которых имеет несколько ядер. Благодаря этому открывается возможность решать сразу несколько задач. Поэтому вместо того чтобы писать программы, которые выполняются последовательно, вы можете воспользоваться преимуществами дополнительных процессоров и ядер и разделить свои программы на несколько мелких задач, которые могут выполняться параллельно. Например, компьютер с двумя четырехъядерными процессорами способен одновременно выполнять восемь операций.

Асинхронный ввод-вывод

Исторически операции ввода-вывода, такие как чтение и запись данных на диск, были самым узким местом в высокопроизводительных приложениях. Современное оборудование обеспечивает возможность выполнения асинхронных операций ввода-вывода, что может существенно повысить общую производительность.

Отзывчивость приложений

Улучшение отзывчивости приложений – это, пожалуй, самая важная причина использования асинхронного программирования. Если длительные вычисления выполняются асинхронно, приложение сможет взаимодействовать с пользователями, что позволит им продолжить свою работу или даже отменить уже запущенные операции.

Работа с потоками

Асинхронное и параллельное программирование осуществляется с помощью *потоков (threads)*, которые являются отдельной вычислительной единицей (computational unit). После запуска операционная система будет выделять каждому потоку определенное время работы, прежде чем передать управление другому потоку.

От *процессов* (*process*) потоки отличаются тем, что процессы выполняются изолированно друг от друга. Потоки же принадлежат создавшему

их процессу и обладают доступом к общей памяти, поэтому взаимодействие между потоками легко можно организовать в виде операций чтения и записи данных в общую память. Процессы, напротив, не могут обращаться к памяти других процессов, поэтому взаимодействия между ними намного сложнее.

В каждый конкретный момент времени операционная система точно знает, какой поток выполняется. Те потоки, которые не выполняются, находятся в приостановленном состоянии и просто ожидают, когда операционная система выделит им следующий квант процессорного времени.

Запуск нескольких потоков в однопроцессорной системе не принесет особого вреда, но в каждый конкретный момент времени процессор сможет выполнять только один из этих потоков и просто будет переключаться на выполнение различных потоков. Та же программа в многопроцессорной или многоядерной системе, напротив, получит возможность выполнять несколько потоков одновременно, поэтому она будет выполняться намного быстрее.

Недостаток многопоточных программ состоит в том, что при одновременном выполнении нескольких потоков бывает сложно согласовать операции чтения и записи данных. Не окажутся ли устаревшими данные, прочитанные потоком *alpha*, потому что поток *beta* только что изменил их. Как быть, если потоку *alpha* потребовалось получить доступ к файлу, который был захвачен потоком *beta*? В многопоточных программах могут возникать различные проблемы синхронизации, которые мы рассмотрим ниже.

Запуск потоков

Для запуска нового потока достаточно просто создать новый экземпляр класса `System.Threading.Thread`, передав конструктору делегат типа `ThreadStart`, и затем вызвать метод `Start` созданного объекта.

В примере 11.1 демонстрируется запуск двух потоков, каждый из которых считает до пяти.

Помните, что в языке F#, если функция принимает делегат в виде параметра, вы можете передать ей лямбда-выражение или значение функции, а компилятор автоматически создаст экземпляр делегата. В данном примере мы передаем конструктору класса `Thread` простую функцию `threadBody`, но в действительности конструктор получит вновь созданный экземпляр класса `ThreadStart`.

Пример 11.1. Создание потоков

```
// Создание новых потоков
open System
open System.Threading
```

```
// Эту функцию будет выполнять каждый поток
let threadBody() =
    for i in 1 .. 5 do
        // Ждать 1/10 секунды
        Thread.Sleep(100)
        printfn "[Thread %d] %d..."
            Thread.CurrentThread.ManagedThreadId
            i

let spawnThread() =
    let thread = new Thread(threadBody)
    thread.Start()

// Запустить сразу два потока
spawnThread()
spawnThread()
```

Если запустить пример 11.1, он выведет следующее (или что-то похожее – нельзя заранее сказать, в каком порядке операционная система передаст управление потокам и какие идентификаторы им присвоит):

```
[Thread 5] 1...
[Thread 6] 1...
[Thread 5] 2...
[Thread 6] 2...
[Thread 5] 3...
[Thread 6] 3...
[Thread 5] 4...
[Thread 6] 4...
[Thread 5] 5...
[Thread 6] 5...
```

Нам удалось запустить два потока, которые считают до пяти. Неплохое начало, но представьте теперь, что функция `threadBody` вычисляет страховые выплаты или производит выравнивание генных последовательностей. Запуск нескольких потоков реализуется очень просто, для этого достаточно создать новый экземпляр класса `Thread` и вызвать метод `Start`.

Кроме метода `Start` класс типа `Thread` обладает еще двумя методами, о которых вам следует знать: `Sleep` и `Abort`.

Thread.Sleep

`Thread.Sleep` – статический метод, который приостанавливает выполнение текущего потока на указанное количество миллисекунд. В примере 11.1 оба потока перед выводом в консоль приостанавливаются на 100 миллисекунд, или одну десятую секунды.

Обычно метод `Thread.Sleep` используется, чтобы выполнить задержку на определенное время или просто уступить квант времени, выделенный

потоку, чтобы операционная система могла отдать освободившееся процессорное время другому потоку.

Thread.Abort

Метод `Thread.Abort` прерывает выполнение потока, генерируя исключение `ThreadAbortException` в процессе работы потока. (За некоторыми исключениями поток может быть прерван в любом положении указателя инструкций.) Учитывая особенности модели памяти в .NET, неожиданное прерывание работы потока не должно вызывать утечек памяти, тем не менее вам нужно избегать использования метода `Thread.Abort`.

Далее в этой главе мы познакомимся с более мощными библиотеками, построенными на основе класса `System.Thread`, которые реализуют такие понятия, как отмена выполнения, что позволяет избежать необходимости использовать метод `Thread.Abort`.

Пул потоков .NET

Создание новых потоков может быть дорогостоящим. Например, каждый поток имеет собственный стек, размер которого может составлять несколько мегабайт.[1] Для выполнения коротких задач, чтобы постоянно не выделять память для новых потоков, рекомендуется использовать *пул потоков .NET*, который является коллекцией потоков, ожидающих, пока им будет дано задание.

Вы можете добавлять новые задания в пул потоков .NET с помощью метода `ThreadPool.QueueUserWorkItem`, который принимает делегат типа `WaitCallback`. Следующий пример точно так же выводит числа в консоль, но при этом в нем не создаются и не запускаются потоки вручную.

Делегат `WaitCallback` принимает параметр типа `obj`, потому что в одной из перегруженных версий метода `QueueUserWorkItem` передается параметр типа `obj`, который будет затем передан функции обратного вызова. Если в вызове `QueueUserWorkItem` параметр отсутствует, параметр функции обратного вызова будет равен `null`:

```
open System.Threading

ThreadPool.QueueUserWorkItem(fun _ -> for i = 1 to 5 do printfn "%d" i)

// Задание, которое будет передано в пул потоков. Обратите внимание,
// что параметр делегата имеет тип obj
let printNumbers (max : obj) =
    for i = 1 to (max :?> int) do
        printfn "%d" i

ThreadPool.QueueUserWorkItem(new WaitCallback(printNumbers), box 5)
```

[1] По умолчанию размер стека потока в операционных системах семейства Windows составляет 1 Мбайт. – *Прим. науч. ред.*

Разделяемые данные

При использовании нескольких потоков может возникать необходимость совместного использования одних и тех же данных для координации работы и сохранения промежуточных результатов. При этом возникают две проблемы: состояние гонок (race conditions) и взаимоблокировки (deadlocks).

Состояние гонок

Состояние гонок возникает, когда два потока пытаются одновременно прочитать или изменить одно и то же значение. Если два потока изменяют значение, сохранится значение, которое будет записано последним, а результат первой записи будет утрачен навсегда. Точно так же, если один поток пытается прочитать значение, а другой в это время изменяет его, первый поток может прочитать устаревшие или вообще испорченные данные.

В примере 11.2 выполняется обход массива двумя потоками и производится вычисление суммы его элементов. Оба потока постоянно изменяют значение ссылочной ячейки `total`, что приводит к появлению гонки.

Пример 11.2. Гонка за ресурсами

```
open System.Threading

let sumArray (arr : int[]) =
    let total = ref 0

    // Вычисление суммы элементов первой половины массива
    let thread1Finished = ref false

    ThreadPool.QueueUserWorkItem(
        fun _ -> for i = 0 to arr.Length / 2 - 1 do
                    total := arr.[i] + !total
                 thread1Finished := true
        ) |> ignore

    // Вычисление суммы элементов второй половины массива
    let thread2Finished = ref false

    ThreadPool.QueueUserWorkItem(
        fun _ -> for i = arr.Length / 2 to arr.Length - 1 do
                    total := arr.[i] + !total
                 thread2Finished := true
        ) |> ignore

    // Дождаться, пока оба потока завершат работу
    while !thread1Finished = false ||
          !thread2Finished = false do

        Thread.Sleep(0)

    !total
```

Результаты вызова функции sumArray в окне FSI выглядят достаточно странными. Мало того что эта функция работает неправильно, так еще и результат каждый раз получается разным! Для массива, содержащего миллион элементов со значением 1, сумма должна составлять точно один миллион, но вместо этого функция sumArray возвращает примерно половину этого значения:

```
> // Массив, содержащий миллион единиц
let millionOnes = Array.create 1000000 1;;

val millionOnes : int []=
[|1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  ...|]

> sumArray millionOnes;;
val it : int = 547392
> sumArray millionOnes;;
val it : int = 579574
> sumArray millionOnes;;
val it : int = 574109
> sumArray millionOnes;;
val it : int = 573321
> sumArray millionOnes;;
val it : int = 550012
```

Проблема заключается в том, что изменение значения total выполняется в три этапа: чтение значения, его увеличение и запись измененного значения обратно. В идеале изменение total должно было бы производиться в рамках одной атомарной операции.

Решение проблемы гонок состоит в том, чтобы *захватывать* совместно используемые данные, то есть запрещать доступ к данным из других потоков, пока не закончится операция их изменения. Например, функция sumArray не содержала бы проблем, если бы в каждый момент времени только один поток мог читать и изменять значение total.

В языке F# имеется возможность заблокировать доступ к значению с помощью функции lock, которая имеет приведенную ниже сигнатуру. Она принимает ссылочное значение, доступ к которому требуется заблокировать, и устанавливает блокировку до тех пор, пока указанная функция не завершит работу.

```
val lock : ('a -> (unit -> 'b) -> 'b) when 'a : not struct
```

В примере 11.3 приводится измененная версия примера 11.2, где весь доступ к значению total выполняется только в пределах вызова функции lock. То есть теперь все другие потоки будут вынуждены ждать своей очереди доступа к этому значению, пока указанная функция не завершит работу.

Пример 11.3. Вычисление суммы элементов массива с применением блокировки

```
let lockedSumArray (arr : int[]) =
    let total = ref 0

    // Вычисление суммы элементов первой половины массива
    let thread1Finished = ref false
    ThreadPool.QueueUserWorkItem(
        fun _ -> for i = 0 to arr.Length / 2 - 1 do
                    lock (total) (fun () -> total := arr.[i] + !total)
                 thread1Finished := true
        ) |> ignore

    // Вычисление суммы элементов второй половины массива
    let thread2Finished = ref false
    ThreadPool.QueueUserWorkItem(
        fun _ -> for i = arr.Length / 2 to arr.Length - 1 do
                    lock (total) (fun () -> total := arr.[i] + !total)
                 thread2Finished := true
        ) |> ignore

    // Дождаться, пока оба потока завершат работу
    while !thread1Finished = false ||
          !thread2Finished = false do

        Thread.Sleep(0)

    !total
```

Теперь функция работает именно так, как требовалось, – для массива, содержащего миллион единиц, возвращается сумма, равная точно одному миллиону. Однако с применением блокировки пропало повышение быстродействия за счет суммирования элементов массива двумя потоками, потому что фактически подсчет выполняется по одному элементу за раз – из-за того, что как только блокировка освобождается одним потоком, она тут же захватывается другим снова и снова.

Далее в этой главе вы узнаете, как использовать многопоточные структуры данных, которые обеспечивают высокую производительность, но при этом используют блокировку внутри себя для обеспечения целостности данных:

```
> lockedSumArray millionOnes;;
val it : int = 1000000
> lockedSumArray millionOnes;;
val it : int = 1000000
> lockedSumArray millionOnes;;
val it : int = 1000000
> lockedSumArray millionOnes;;
val it : int = 1000000
```

Взаимоблокировка

Блокировки позволяют решить проблему гонки. Однако применение блокировок порождает другие проблемы. Представьте ситуацию: вам потребовалось заблокировать доступ к одному ресурсу, одновременно удерживая блокировку на доступ к другому ресурсу. Такие ситуации могут приводить к *взаимоблокировке*.

Взаимоблокировки возникают, когда необходимо получить доступ к ресурсу, который уже захвачен другим потоком, но при этом другой поток не освободит ресурс, пока первый поток в свою очередь не освободит ресурс, который уже был захвачен им. В результате каждый из потоков ожидает, пока другой освободит захваченный им ресурс.

Всякий раз, захватывая какой-либо объект, вы рискуете оказаться в состоянии взаимоблокировки, если другому потоку также потребуется получить к нему же доступ.

В примере 11.4 показана простая реализация банковского счета и функции перевода средств с одного счета на другой, использующей блокировку.

Пример 11.4. Взаимоблокировки

```
type BankAccount = { AccountID:int; OwnerName:string; mutable Balance:int}

/// Перевод средств со счета на счет
let transferFunds amount fromAcct toAcct =

    printfn "Locking %s's account to deposit funds..." toAcct.OwnerName
    lock fromAcct
        (fun () ->
            printfn "Locking %s's account to withdraw funds..."
                    fromAcct.OwnerName
            lock toAcct
                (fun () ->
                    fromAcct.Balance <- fromAcct.Balance - amount
                    toAcct.Balance <- toAcct.Balance + amount
                )
        )
```

Предыдущий фрагмент выглядит достаточно простым, но если вдруг произойдет перевод средств между этими же счетами в одно и то же время, произойдет взаимоблокировка.

В следующем примере первый вызов `transferFunds` захватит доступ к счету клиента с именем John Smith, а второй вызов `transferFunds` захватит доступ к счету клиента с именем Jane Doe. Затем, когда `transferFunds` попытается захватить доступ к другому счету, оба потока перейдут в состояние бесконечного ожидания:

```
let john = { AccountID = 1; OwnerName = "John Smith"; Balance = 1000 }
let jane = { AccountID = 2; OwnerName = "Jane Doe"; Balance = 2000 }
```

```
ThreadPool.QueueUserWorkItem(fun _ -> transferFunds 100 john jane)
ThreadPool.QueueUserWorkItem(fun _ -> transferFunds 100 jane john)
```

Возможность взаимоблокировки не означает, что совместное использование одних и тех же ресурсов разными потоками невозможно. Достаточно всего лишь проявить благоразумие при использовании блокировок и помнить о зависимостях между потоками. Существует множество способов написания кода, использующего блокировки, который не приводит к взаимоблокировкам. Например, если бы в предыдущем примере функция `transferFunds` всегда старалась сначала захватить доступ к счету с наименьшим значением `AccountID`, взаимоблокировка бы не возникла.

Асинхронное программирование

Теперь, после знакомства с потоками, можно рассмотреть использование асинхронных операций в программах на .NET.

Под асинхронными операциями понимаются такие операции, которые запускаются для выполнения в фоновом потоке и завершаются позднее, а завершение операции либо игнорируется, либо периодически проверяется ее состояние.

Современное оборудование может выполнять параллельные операции чтения и записи, поэтому использование асинхронных операций ввода-вывода может повысить производительность приложения. Это особенно верно для операций записи на диск данных, которые не потребуются приложению позднее. В этом случае нет необходимости «блокировать» выполнение приложения, ожидая, пока операция записи завершится, вместо этого можно просто запустить асинхронную операцию и продолжить выполнение приложения.

Исторически асинхронные приложения для .NET пишутся с использованием *асинхронной модели программирования (Asyncronous Programming Model, APM)*.

APM – это некоторый шаблон, согласно которому выполнение асинхронной операции реализуется в виде двух методов, `Begin`*`Operation`* и `End`*`Operation`*. Метод `Begin`*`Operation`* запускает асинхронную операцию и по ее завершении вызывает переданный ему делегат. Реализация делегата должна вызвать метод `End`*`Operation`*, который вернет результаты асинхронной операции. Координация действий, таких как периодическая проверка окончания операции, выполняется с использованием интерфейса `IAsyncResult`.

Применение APM показано в примере 11.5, который асинхронно открывает файл, выполняет некоторые действия над данными из файла и затем асинхронно записывает измененные данные на диск. При асинхронном выполнении операций ввода-вывода они фактически ставятся в очередь, что позволяет контроллеру диска работать с максимально

возможной скоростью. Что особенно важно – текущий поток не блокируется в ожидании завершения операции.

Не волнуйтесь, если делегат типа `AsyncCallback` и интерфейс `IAsyncResult` покажутся вам слишком сложными. Модель асинхронного программирования действительно сложна в использовании, но в следующем разделе вы увидите способ, позволяющий отказаться от явного использования этой модели.

Пример 11.5. Асинхронный ввод-вывод с использованием APM

```
open System
open System.IO

let processFileAsync (filePath : string) (processBytes : byte[] -> byte[]) =

    // Функция обратного вызова, которая вызывается по завершении AsyncWrite
    let asyncWriteCallback =
        new AsyncCallback(fun (iar : IAsyncResult) ->
            // Получить результаты асинхронной операции
            let writeStream = iar.AsyncState :?> FileStream

            // Завершить асинхронную операцию записи вызовом EndWrite
            let bytesWritten = writeStream.EndWrite(iar)
            writeStream.Close()

            printfn
                "Finished processing file [%s]"
                (Path.GetFileName(writeStream.Name))
        )

    // Функция обратного вызова, которая вызывается по завершении AsyncRead
    let asyncReadCallback =
        new AsyncCallback(fun (iar : IAsyncResult) ->
            // Получить результаты асинхронной операции
            let readStream, data = iar.AsyncState :?> (FileStream * byte[])

            // Завершить асинхронную операцию чтения вызовом EndRead
            let bytesRead = readStream.EndRead(iar)
            readStream.Close()

            // Обработать результаты
            printfn
                "Processing file [%s], read [%d] bytes"
                (Path.GetFileName(readStream.Name))
                bytesRead

            let updatedBytes = processBytes data

            let resultFile = new FileStream(readStream.Name + ".result",
                                            FileMode.Create)
```

```
            let _ =
                resultFile.BeginWrite(
                    updatedBytes,
                    0, updatedBytes.Length,
                    asyncWriteCallback,
                    resultFile)

            ()
        )

    // Запустить асинхронную операцию чтения и передать ей функцию
    // обратного вызова, которая запустит асинхронную операцию записи
    let fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read,
                                    FileShare.Read, 2048,
                                    FileOptions.Asynchronous)

    let fileLength = int fileStream.Length
    let buffer = Array.zeroCreate fileLength

    // Состояние, передаваемое асинхронной операции чтения
    let state = (fileStream, buffer)

    printfn "Processing file [%s]" (Path.GetFileName(filePath))
    let _ = fileStream.BeginRead(buffer, 0, buffer.Length,
                                 asyncReadCallback, state)
     ()
```

Пример 11.5 наглядно демонстрирует простым смертным программистам, насколько сложной является APM и насколько просто допустить ошибку при ее использовании. Этой модели свойственны следующие проблемы:

- Если в программе друг за другом следуют сразу нескольких асинхронных операций, становится очень сложно разобраться в потоке ее выполнения, так как он разбрасывается по множеству функций обратного вызова.
- Если забыть вызвать метод End*Operation*, это может привести к нежелательным эффектам, начиная от утечек памяти и заканчивая необработанными исключениями и зависанием программы.
- Приведение типа, хранящегося в свойстве `IAsyncResult.AsyncState`, может привести к исключению приведения типа во время выполнения.
- Как только вы начинаете использовать асинхронные вызовы, вы тут же сталкиваетесь с кошмарным спагетти из функций обратного вызова.

Но это далеко не все проблемы, которые поджидают программиста, использующего APM. Попробуйте представить себе корректную обработку исключений в последовательности асинхронных операций – вам не удастся обернуть все обращения к функциям обратного вызова един-

ственным выражением `try-with` и придется предусматривать обработку исключений внутри каждой функции обратного вызова.

Целью знакомства с APM было не отпугнуть вас от использования асинхронных операций, а подготовить к тем улучшениям, которые имеются в языке F#.

В идеале было бы неплохо уйти от необходимости явно разбивать код на то, что должно происходить *перед* асинхронной операцией (код, вызывающий `Begin`*`Operation`*), и то, что должно происходить *после* асинхронной операции (функция обратного вызова, которая вызывает `End`*`Operation`*).

Было бы очень удобно, если бы код в синхронном стиле мог использовать какую-нибудь вспомогательную библиотеку, которая взяла бы на себя проблему «возни с потоками» и «передачи результатов операции вашей функции обратного вызова». Было бы идеально, если бы можно было написать такой код:

```
printfn "Opening file..."
let_wait_until_async_is_done data = openFileAsync filePath

let result = processFile data filePath

printfn "Writing result to disk..."
do_wait_until_async_is_done writeFileAsync (filePath + ".result") result

printfn "Finished!"
```

Не напоминает ли это проблему, с которой мы уже сталкивались в главе 10? Тогда мы определили классы задач, при решении которых нам хотелось бы, чтобы некоторые вспомогательные операции выполнялись неявно, и при этом у нас сохранялась бы возможность писать обычный код F#. А может, этот прием поможет нам и в данной ситуации....

Правильно, нам помогут вычислительные выражения!

Асинхронные вычислительные выражения

Асинхронные вычислительные выражения (asynchronous workflows) позволяют выполнять асинхронные операции без явного использования функций обратного вызова. Благодаря этому вы можете писать код, который выглядит так, как будто он выполняется синхронно, но в действительности он выполняется асинхронно, приостанавливаясь и возобновляя работу по завершении асинхронных операций.

Асинхронные вычислительные выражения не вводят новые примитивы в платформу .NET. Скорее синтаксис вычислительных выражений упрощает использование существующих библиотек, предназначенных для работы с потоками.

В примере 11.6 представлена еще одна реализация той же задачи асинхронного чтения и записи в файл, только на этот раз вместо APM для выполнения асинхронных операций используется библиотека асинхронных вычислительных выражений. В результате код стал существенно проще.

Волшебство, стоящее за асинхронными вычислительными выражениями, мы рассмотрим чуть позже, а пока просто обратите внимание, что код обернут построителем асинхронных выражений `async`, а асинхронные операции начинаются с ключевых слов `let!` и `do!`. Кроме того, следует также обратить внимание, что результат построителя `async` передается таинственному методу `Async.Start`.

Пример 11.6. Асинхронные операции ввода-вывода с использованием асинхронных вычислительных выражений

```
open System.IO

let asyncProcessFile (filePath : string) (processBytes : byte[] -> byte[]) =
    async {

        printfn "Processing file [%s]" (Path.GetFileName(filePath))

        use fileStream = new FileStream(filePath, FileMode.Open)
        let bytesToRead = int fileStream.Length

        let! data = fileStream.AsyncRead(bytesToRead)

        printfn
            "Opened [%s], read [%d] bytes"
            (Path.GetFileName(filePath))
            data.Length

        let data' = processBytes data

        use resultFile = new FileStream(filePath + ".results", FileMode.Create)
        do! resultFile.AsyncWrite(data', 0, data'.Length)

        printfn "Finished processing file [%s]" <| Path.GetFileName(filePath)
    } |> Async.Start
```

Библиотека Async

Секрет асинхронных вычислительных выражений состоит в том, что код, заключенный в блок `async`, не выполняется немедленно. Вместо этого вычисления, выполняемые этим кодом, возвращаются в виде объекта типа `Async<'T>`, который можно представить как асинхронную операцию, возвращающую экземпляр типа `'T`. Все заботы по извлечению экземпляра `'T` из объекта `Async<'T>` берут на себя модуль `Async` и построитель вычислительных выражений `async`.

Всякий раз, встречая let!, do! или похожую инструкцию, построитель async запускает ее асинхронно и по завершении выполняет оставшуюся часть вычислений.

В предыдущем примере функция AsyncRead возвращает экземпляр Async<byte[]>. То есть если бы код был записан как:

```
let data = fileStream.AsyncRead(bytesToRead)
```

значение data имело бы тип Async<byte[]>.

Однако в действительности используется инструкция let! (произносится как *лет-банг*). Это приводит к тому, что построитель async запускает вычисления асинхронно и возвращает в результате массив байтов, как только операция завершится:

```
let! data : byte[] = fileStream.AsyncRead(bytesToRead)
```

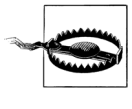
За дополнительной информацией о том, как инструкции let! и do! преобразуются в вызовы методов построителя вычислительных выражений, обращайтесь к главе 10.

Запуск асинхронных операций

Существует несколько методов запуска асинхронных вычислительных выражений. Самый простой из них – метод Async.Start. Метод Async.Start принимает параметр типа Async<unit> и просто запускает его в асинхронном режиме. Если необходимо получить результат выполнения асинхронной операции, следует дождаться ее завершения вызовом метода Async.RunSynchronously.

В примере 11.7 определяется функция getHtml, которая получает адрес URL и возвращает содержимое веб-страницы в виде экземпляра типа Async<string>.

Обратите внимание, что результат возвращается из асинхронного вычислительного выражения с помощью ключевого слова return!. Асинхронное вычислительное выражение также может возвращать результат с помощью ключевого слова return, но в этом случае результатом будет объект типа Async<'T>, а не 'T.



В примере 11.7 используется метод расширения AsyncReadToEnd из сборки FSharp.PowerPack.dll. Дополнительные сведения о том, как получить сборку PowerPack для F#, приводятся в приложении А.

Пример 11.7. Получение значений из асинхронного вычислительного выражения

```
#r "FSharp.PowerPack.dll"

open System.IO
```

```
open System.Net
open System.Microsoft.FSharp.Control.WebExtensions

let getHtml (url : string) =
    async {

        let req = WebRequest.Create(url)
        let! rsp = req.AsyncGetResponse()

        use stream = rsp.GetResponseStream()
        use reader = new StreamReader(stream)

        return! reader.AsyncReadToEnd()
    }

let html =
    getHtml "http://en.wikipedia.org/wiki/F_Sharp_programming_language"
    |> Async.RunSynchronously
```

Метод `Async.RunSynchronously` не имеет большого самостоятельного значения, потому что он приостанавливает работу текущего потока до окончания операции, что являет собой полную противоположность асинхронному выполнению операций. Обычно этот метод вызывается сразу после вызова метода `Async.Parallel`, который принимает экземпляр типа `seq< Async<'T> >` и запускает все асинхронные операции в последовательности параллельно. Результаты этих операций объединяются в экземпляре `Async<'T[]>`.

Следующий фрагмент извлекает сразу несколько веб-страниц, используя метод `getHtml`:

```
let webPages : string[] =
[ "http://www.google.com"; "http://www.bing.com"; "http://www.yahoo.com" ]
|> List.map getHtml
|> Async.Parallel
|> Async.RunSynchronously
```

Помимо упрощения кода использование асинхронных вычислительных выражений также упрощает обработку исключений и обеспечивает отмену операций, что достаточно сложно реализовать, применяя APM.

Исключения

Выше, когда мы рассматривали асинхронные операции и методы `Begin`*`Operation`* и `End`*`Operation`*, уже отмечалась сложность корректной обработки исключений. Асинхронные вычислительные выражения позволяют поместить обработчик исключения непосредственно в блок `async`, вместо того чтобы создавать множество обработчиков – по одному в каждой функции обратного вызова.

Даже если в блоке try выполняется несколько асинхронных операций (инструкций let! и do!), исключение все равно будет перехвачено, как и ожидается.

```
let asncOperation =
    async {
        try
            // ...
        with
        | :? IOException as ioe ->
            printfn "IOException: %s" ioe.Message
        | :? ArgumentException as ae ->
            printfn "ArgumentException: %s" ae.Message
    }
```

Но что произойдет, если исключение не будет перехвачено в вычислительном выражении async? Продолжит ли это исключение свое распространение и не приведет ли оно к аварийному завершению процесса? К сожалению, да.

Исключения, возникшие и необработанные в асинхронных вычислительных выражениях, приводят к аварийному завершению всего процесса. Для перехвата необработанного исключения, возникшего в асинхронном вычислительном выражении, можно использовать метод Async.StartWithContinuations, который будет рассматриваться ниже, или комбинатор Async.Catch.

Метод Async.Catch принимает асинхронное вычислительное выражение и возвращает результат его выполнения в виде вариантов «успех» или «исключение». Он имеет следующую сигнатуру:

```
val Catch : Async<'T> -> Async< Choice<'T,exn> >
```

Тип Choice может иметь одно из нескольких значений, то есть за кулисами тип Choice компилируется как многовариантный активный шаблон.

Следующий пример использует метод Async.Catch для запуска асинхронного вычислительного выражения. Если операция завершается успехом, результат выводится на экран. В противном случае выводится сообщение об исключении:

```
asyncTaskX
|> Async.Catch
|> Async.RunSynchronously
|> function
    | Choice1Of2 result     -> printfn "Async operation completed: %A" result
    | Choice2Of2 (ex : exn) -> printfn "Exception thrown: %s" ex.Message
```

Отмена асинхронных операций

Когда есть некоторый код, который выполняется асинхронно, важно иметь механизм отмены его выполнения – на случай, если пользователь заметит, что что-то пошло не так или потеряет терпение. При работе с обычными потоками единственным способом прервать выполнение операции является метод `Thread.Abort`[1], который грубо уничтожает поток, что влечет за собой множество дополнительных проблем.

Выполнение асинхронных вычислительных выражений также может быть прервано, но, в отличие от метода `Thread.Abort`, механизм отмены асинхронного выражения основан на отправке запросов. Операция не завершается немедленно – отмена выполнения происходит при выполнении ближайшей инструкции `let!`, `do!` и других; в этом случае вместо функции, представляющей остаток вычислений, выполняется обработчик отмены операции.

Самый простой способ добавить свой обработчик отмены операции – использовать метод `Async.TryCancelled`. Подобно методу `Async.Catch`, он принимает объект вычислительного выражения типа `Async<'T>` и возвращает измененный объект `Async<'T>`, который вызовет указанную функцию в случае отмены операции.

В примере 11.8 определяется простое задание, которое ведет счет до 10, но прерывается в следующем ниже коде. Обратите внимание, что отмена операции выполняется с помощью метода `Async.CancelDefaultToken`.

Пример 11.8. Прерывание асинхронных операций

```
open System
open System.Threading

let cancelableTask =
    async {
        printfn "Waiting 10 seconds..."
        for i = 1 to 10 do
            printfn "%d..." i
            do! Async.Sleep(1000)
        printfn "Finished!"
    }
```

[1] Здесь автор не совсем прав. Работая с потоками в любой среде и с любой библиотекой, вы можете реализовать так называемый «паттерн кооперативной отмены» (cooperative cancellation pattern), когда в процессе выполнения длительной операции рабочий поток проверяет некоторый флаг, и если он выставляется, то он считает, что внешний код запросил отмену операции. Однако этот способ утомителен и может приводить к ошибкам, поскольку требует однотипных операций. – *Прим. науч. ред.*

```
// Функция обратного вызова, которая используется в случае прерывания операции
let cancelHandler (ex : OperationCanceledException) =
    printfn "The task has been canceled."

Async.TryCancelled(cancelableTask, cancelHandler)
|> Async.Start

// ...

Async.CancelDefaultToken()
```

Отмена асинхронного вычислительного выражения в языке F# выполняется с использованием того же самого механизма Parallel Extensions с помощью типа `CancellationToken`. Тип `CancellationToken` следит за тем, прервана операция или нет. (Точнее, был ли отправлен запрос на прерывание операции.) То есть всякий раз, когда встречается инструкция `let!` или `do!`, построитель `async` проверяет `CancellationToken`, ассоциированный с вычислительным выражением, чтобы определить, следует ли продолжить выполнение операции или необходимо завершить ее раньше.

Метод `Async.CancelDefaultToken` прерывает работу асинхронного вычислительного выражения, запущенного последним. Это подходит в большинстве случаев, но может вызывать затруднения при одновременном запуске сразу нескольких асинхронных вычислительных выражений.

Чтобы получить возможность прерывать работу любых асинхронных вычислительных выражений, необходимо создать и сохранить объект типа `CancellationTokenSource`. Объект `CancellationTokenSource` как раз и является средством, позволяющим отправить запрос на отмену, который в свою очередь вызывает изменение всех связанных с ним объектов `CancellationToken`.

Следующий фрагмент демонстрирует, как можно запустить асинхронное вычислительное выражение и прерывать его выполнение с помощью объекта `CancellationTokenSource`:

```
open System.Threading

let computation = Async.TryCancelled(cancelableTask, cancelHandler)
let cancellationSource = new CancellationTokenSource()

Async.Start(computation, cancellationSource.Token)

// ...

cancellationSource.Cancel()
```

Из-за такого обилия разных способов запуска и манипулирования асинхронными вычислительными выражениями модуль `Async` может показаться немного сложным. Однако, скорее всего, вы будете пользоваться всего одним методом, позволяющим перехватывать необработанные ис-

ключения и отмену выполнения – методом Async.StartWithContinuations. По сути, это Async.Catch, Async.TryCancelled и Async.Start в одном флаконе. Данный метод принимает четыре параметра:

- Асинхронное вычислительное выражение, которое требуется выполнить
- Функцию, которая вызывается в случае успешного выполнения операции
- Функцию, которая вызывается в случае исключения
- Функцию, которая вызывается в случае, если выполнение операции было прервано

Следующий фрагмент показывает использование метода Async.StartWithContinuations:

```
Async.StartWithContinuations(
    superAwesomeAsyncTask,
    (fun result -> printfn "Task completed with result %A" result),
    (fun exn -> printfn "Task threw an exception with Message: %s" exn.Message),
    (fun oce -> printfn "Task was cancelled. Message: %s" oce.Message)
)
```

Асинхронные операции

Асинхронные вычислительные выражения показывают простоту, с которой можно писать код, выполняемый асинхронно, но как получить сам экземпляр Async<'T>? Можно создавать собственные объекты async, как будет показано ниже, однако, скорее всего, вы предпочтете использовать методы расширения, определенные в стандартной библиотеке F#.

Тип System.IO.Stream обладает рядом дополнительных методов, реализующих асинхронные операции чтения и записи. Вы уже видели примеры использования методов AsyncRead и AsyncWrite ранее.

Класс System.Net.WebRequest расширен методом AsyncGetResponse, позволяющим получать веб-страницы асинхронно. Однако прежде чем использовать этот метод расширения, необходимо открыть пространство имен Microsoft.FSharp.Control.WebExtensions:

```
open System.IO
open System.Net

open Microsoft.FSharp.Control.WebExtensions

let getHtml (url : string) =

async {

    let req = WebRequest.Create(url)
    let! rsp = req.AsyncGetResponse()
```

```
    use stream = rsp.GetResponseStream()
    use reader = new StreamReader(stream)

    return reader.ReadToEnd()
}
```

Метод `Async.Sleep` позволяет приостановить выполнение асинхронной операции точно так же, как это делает метод `Thread.Sleep`, за исключением того, что он не блокирует поток выполнения асинхронного вычислительного выражения. Вместо этого текущий рабочий поток будет использован для выполнения какого-нибудь другого задания, а когда метод `Sleep` завершится, для завершения этой операции будет использован другой поток.

Следующий фрагмент приостанавливает и возобновляет выполнение асинхронного вычислительного выражения каждую секунду для обновления индикатора хода выполнения операции, присутствующего на форме:

```
#r "System.Windows.Forms.dll"
#r "System.Drawing.dll"

open System.Threading
open System.Windows.Forms

let form = new Form(TopMost = true)

let pb = new ProgressBar(Minimum = 0, Maximum = 15, Dock = DockStyle.Fill)
form.Controls.Add(pb)

form.Show()

async {
    for i = 0 to 15 do
        do! Async.Sleep(1000)
        pb.Value <- i
} |> Async.Start
```

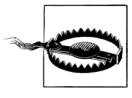
Создание собственных асинхронных примитивов

В языке F# существует возможность создавать собственные примитивы асинхронных вычислительных выражений с помощью функции `Async.FromBeginEnd`. В качестве параметров она принимает методы `Begin`*`Operation`* и `End`*`Operation`* из модели APM, но оборачивает их так, что они могут использоваться в асинхронных вычислительных выражениях. (Функция `Async.FromBeginEnd` имеет перегруженные версии, принимающие различные типы операций APM.)

В примере 11.9 определяются два метода расширения для получения списка файлов в каталоге и асинхронного копирования файлов. Работа обоих примитивов основана на использовании типа `Func<_,_>`, который

предназначен для создания значений функций, но использует для этого делегаты. Напомню, что делегат любого типа, определенный на языке C# или VB.NET, содержит методы BeginInvoke и EndInvoke, которые могут служить основой APM.

После создания двух асинхронных примитивов AsyncGetFiles и AsyncCopy они используются для копирования всех файлов из одного каталога в другой.



Не торопитесь с созданием собственных асинхронных примитивов. Если только нет веских причин для выполнения операций в асинхронном режиме, например создание ловушки операционной системы для повышения эффективности ввода-вывода, вы не получите увеличения производительности, так как обычно сначала одна операция завершается синхронно, а только потом происходит переключение в поток асинхронного вычислительного выражения.

Пример 11.9. Создание собственных асинхронных примитивов

```
open System
open System.IO

type System.IO.Directory with
    /// Асинхронно получить список всех файлов в каталоге
    static member AsyncGetFiles(path : string, searchPattern : string) =
        let dele = new Func<string * string, string[]>(Directory.GetFiles)
        Async.FromBeginEnd(
            (path, searchPattern),
            dele.BeginInvoke,
            dele.EndInvoke)

type System.IO.File with
    /// Копировать файлы асинхронно
    static member AsyncCopy(source : string, dest : string) =
        let dele = new Func<string * string, unit>(File.Copy)
        Async.FromBeginEnd((source, dest), dele.BeginInvoke, dele.EndInvoke)

let asyncBackup path searchPattern destPath =
    async {
        let! files = Directory.AsyncGetFiles(path, searchPattern)

        for file in files do
            let filename = Path.GetFileName(file)
            do! File.AsyncCopy(file, Path.Combine(destPath, filename))
    }
```

Ограничения

Асинхронные вычислительные выражения в языке F# отлично подходят для асинхронного ввода-вывода. Однако из-за того, что библиотека является всего лишь тонкой оберткой вокруг пула потоков, их исполь-

зование не гарантирует максимальную производительность. Существует множество факторов, оказывающих влияние на производительность параллельных операций, которые необходимо принимать во внимание:

- Количество ядер в процессоре
- Когерентность кэша процессора
- Существующая загрузка процессора

Асинхронные вычислительные выражения упрощают параллельное выполнение нескольких операций, однако у нас нет никаких инструментов управления порядком выполнения потоков для достижения наиболее оптимального их использования. Например, при наличии восьми потоков, конкурирующих за процессор, когда машина находится под высокой нагрузкой, масса времени будет тратиться впустую на переключение контекста между потоками. Идеальной была бы возможность выделять некоторым потокам больше процессорного времени, чтобы уменьшить потери на переключение контекста.

Асинхронные вычислительные выражения следует использовать только для выполнения асинхронных операций ввода-вывода (так называемых IO-bound-операций), а для параллельного выполнения процессороемких операций (так называемых CPU-bound-операций) желательно использовать библиотеку .NET Parallel Extensions, которая обсуждается ниже.

Параллельное программирование

Под параллельным программированием (parallel programming) подразумевается разделение вычислений на *n* частей с целью увеличения, как мы надеемся, производительности в *n* раз (хотя на практике такое увеличение редко достижимо).

Самый простой способ параллельного программирования в .NET заключается в использовании библиотеки Parallel Extensions for .NET Framework (PFX), которая, как заявляет Microsoft, реализует новые примитивы, упрощающие параллельное программирование. При использовании PFX у вас не должно возникать необходимости вручную управлять потоками или пулом потоков.

Библиотека Parallel Framework присутствует только в CLR 4.0. Если вы пишете на языке F# приложения, которые должны выполняться на платформе .NET 2.0, 3.0 или 3.5, вы не сможете использовать библиотеку PFX. Кроме того, чтобы использовать библиотеку PFX в сеансе FSI, необходимо добавить ссылку на `System.Core.dll`:

```
#r "System.Core.dll"
```

Однако асинхронные вычислительные выражения F# доступны в более старых версиях платформы.

Parallel.For

Первое, что можно сделать для распараллеливания операций в своих приложениях, – заменить циклы `for` вызовами методов `Parallel.For` и `Parallel.ForEach`, которые определены в пространстве имен `System.Threading`. Как следует из названий, эти функции могут использоваться в качестве замены циклов `for` и выполняют итерации цикла параллельно.

В примере 11.10 показано использование метода `Parallel.For` для умножения двух матриц. Операция получения одной строки результирующей матрицы реализована в виде функции `rowTask`; благодаря этому строки результирующей матрицы могут вычисляться параллельно.

Пример 11.10. Использование метода Parallel.For

```
open System
open System.Threading.Tasks

/// Умножение двух матриц с помощью PFX
let matrixMultiply (a : float[,]) (b : float[,]) =

    let aRow, aCol = Array2D.length1 a, Array2D.length2 a
    let bRow, bCol = Array2D.length1 b, Array2D.length2 b
    if aCol <> bRow then failwith "Array dimension mismatch."

    // Выделить память для результирующей матрицы c
    let c = Array2D.create aCol bRow 0.0
    let cRow, cCol = aCol, bRow

    // Вычислить одну строку результирующей матрицы
    let rowTask rowIdx =
        for colIdx = 0 to cCol - 1 do
            for x = 0 to aRow - 1 do
                c.[colIdx, rowIdx] <-
                    c.[colIdx, rowIdx] + a.[x, colIdx] * b.[rowIdx, x]
        ()

    let _ = Parallel.For(0, cRow, new Action<int>(rowTask))

    // Вернуть полученную матрицу
    c
```

Простая замена циклов `for` вызовом метода `Parallel.For` приведет к ошибкам, если код, выполняемый в теле цикла `for`, зависит от результатов предыдущей итерации, например при использовании цикла `for` для выполнения операции свертки. В предыдущем примере порядок вычисления строк результирующей матрицы не имеет значения, поэтому эти операции можно безопасно выполнять параллельно.

Вы можете начать замечать преимущества ограничения изменяемости и отсутствие разделяемых глобальных состояний. В языке F# данные

по умолчанию неизменяемые, благодаря чему по умолчанию получается потокобезопасный код.

Модуль Array.Parallel

Разработчики на языке F# могут использовать модуль `Array.Parallel`, построенный поверх библиотеки PFX и содержащий некоторые из методов, которые можно найти в модуле `Array`, такие как `map`, `mapi` и `partition`. Единственное их отличие состоит в том, что эти методы выполняют операции параллельно.

В примере 11.11 показано использование модуля `Array.Parallel` для одновременного открытия нескольких файлов и поиска в них. Так же как и при использовании метода `Parallel.For`, если операции, выполняемые над каждым элементом массива, не зависят друг от друга, вы можете безопасно заменить синхронные операции параллельными.

Код в примере получает список файлов с расширением *.classified*, параллельно расшифровывает их, затем параллельно производит поиск ключевых слов в нескольких файлах и наконец зашифровывает их, причем опять параллельно.

Пример 11.11. Использование модуля Array.Parallel

```
open System.IO

let getSecretData keyword =

    let secretFiles = Directory.GetFiles(@"D:\TopSecret\", "*.classified")

    Array.Parallel.iter File.Decrypt secretFiles

    let secretData =
        Directory.GetFiles(@"D:\TopSecret\", "*.classified")
        |> Array.Parallel.map (fun filePath -> File.ReadAllText(filePath))
        |> Array.Parallel.choose (fun contents ->
            if contents.Contains(keyword)
            then Some(contents)
            else None)

    Array.Parallel.iter File.Encrypt secretFiles

    secretData
```

Библиотека PFX

В предыдущем разделе мы узнали, как выполнять параллельные операции над массивами и в циклах `for`. Однако далеко не все задачи так легко поддаются декомпозиции. Наиболее вероятно вам будут встре-

чаться задачи, которые достаточно просто разбиваются на составляющие, но на неравные составляющие. Или, возможно, какую-то часть задачи потребуется разбивать на все более и более мелкие части с помощью подхода «разделяй и властвуй».

К счастью, с помощью библиотеки PFX вы можете дробить ваши операции на части, настолько мелкие, насколько вам нужно, а библиотека позаботится о параллельном их выполнении.

Основой библиотеки PFX является класс Task. Подобно типу Async<'T>, тип Task представляет операцию, выполнение которой будет закончено позднее. Чтобы увидеть, что может предложить объект Task, давайте коротко рассмотрим, как на языке F# реализуются параллельные операции.

Допустим, что у нас имеются два независимых задания, которые выполняются достаточно продолжительное время. Самый простой вариант реализации – выполнить их последовательно:

```
let result1 = longTask1()
let result2 = longTask2()
```

Вы можете увеличить скорость выполнения, задействовав пул потоков, с которым мы уже познакомились выше. Однако это снизит читаемость кода, а кроме того, это не позволит прерывать выполнение операций и не обеспечит простого способа обработки исключений:

```
let mutable completed = false
let mutable result1 = null

ThreadPool.QueueUserWorkItem(fun _ ->
    result1 <- longTask1()
    completed <- true
) |> ignore

let result2 = longTask2()

// Дождаться завершения задания №1
while not completed do
    Thread.Sleep(0)
```

Ниже показано, как можно было бы решить ту же самую проблему с помощью PFX:

```
open System.Threading.Tasks

let taskBody = new Func<string>(longTask1)
let task = Task.Factory.StartNew<string>(taskBody)

let result2 = longTask2()
let result1 = task.Result
```

Примитивы

Новые задания могут создаваться с помощью одной из перегруженных версий метода `Task.Factory.StartNew`. После того как задание будет создано, оно будет выполнено параллельно. При этом библиотека PFX автоматически определит, сколько заданий запускать одновременно, в зависимости от таких показателей, как количество доступных вычислительных ресурсов, количество процессоров и так далее.

То есть вам не нужно управлять этим самостоятельно – библиотека PFX сама сделает это за вас. Запуск большого количества заданий с использованием асинхронных вычислительных выражений мог бы привести к переполнению пула потоков и перегрузке процессора, но библиотека PFX корректно регулирует количество заданий, выполняемых параллельно, чтобы обеспечить наиболее оптимальную производительность.

Чтобы получить результат, возвращаемый заданием, достаточно просто обратиться к свойству `Result`.

```
let x = task.Result
```

При обращении к свойству `Result` может произойти следующее:

- Немедленно будет возвращен результат выполнения задания, если это задание уже выполнено.
- Текущий поток будет приостановлен до завершения задания, если задание уже выполняется.
- Если выполнение задания еще не начато, задание начнет выполняться и будет выполнено в текущем потоке.

Библиотека PFX избавит вас от необходимости следить за отдельными заданиями. Для объединения нескольких заданий вы можете использовать такие примитивы синхронизации, как `Task.WaitAll` и `Task.WaitAny`. Эти функции принимают массив заданий и приостанавливают выполнение потока, пока не завершатся все или какое-нибудь одно задание.

В примере 11.12 показан запуск множества заданий, изменяющих размеры изображений, хранящихся в каталоге, и использование метода `WaitAll`, приостанавливающего выполнение текущего потока, пока не будут выполнены все задания.

Пример 11.12. Использование Task.WaitAll для ожидания выполнения всех заданий

```
open System
open System.IO
open System.Drawing
open System.Drawing.Imaging
open System.Drawing.Drawing2D
open System.Threading.Tasks
```

```
/// Изменение размеров изображений до нужной ширины и высоты
let resizeImage (newWidth : int, newHeight : int) (filePath : string) =
    let originalImage = Bitmap.FromFile(filePath)

    let resizedImage = new Bitmap(newWidth, newHeight)
    use g = Graphics.FromImage(resizedImage)
    g.InterpolationMode <- InterpolationMode.HighQualityBicubic
    g.DrawImage(originalImage, 0, 0, newWidth, newHeight)

    let fileName = Path.GetFileNameWithoutExtension(filePath)
    let fileFolder = Path.GetDirectoryName(filePath)
    resizedImage.Save(
        Path.Combine(fileFolder, fileName + ".Resized.jpg"),
        ImageFormat.Jpeg)

    // Запуск нового задания PFX, изменяющего размер изображения
    let spawnTask filePath =
        let taskBody = new Action(fun () -> resizeImage (640, 480) filePath)
        Task.Factory.StartNew(taskBody)

    let imageFiles = Directory.GetFiles(@"C:\2009-03-16 iPhone\", "*.jpg")

    // Запуск заданий, изменяющих размеры изображений
    let resizeTasks = imageFiles |> Array.map spawnTask

    // Дождаться завершения всех заданий
    Task.WaitAll(resizeTasks)
```

На первый взгляд методу `WaitAny` будет трудно найти применение на практике – когда может потребоваться запускать множество заданий и ждать завершения одного из них? Однако существуют две ситуации, где этот примитив может пригодиться.

Во-первых, если у вас есть сразу несколько алгоритмов решения поставленной задачи с разной степенью эффективности в зависимости от исходных данных, то имеет смысл запустить их все параллельно и воспользоваться результатом первого завершившегося задания. (При этом вам может потребоваться прервать выполнение остальных заданий, если заведомо известно, что их результаты не потребуются.)

Во-вторых, это может пригодиться для создания более отзывчивых приложений. Например, если бы пример 11.12 имел пользовательский интерфейс, пользователь мог бы захотеть видеть результаты изменения размеров изображений по мере выполнения заданий, а не только после того, как все задания будут выполнены.

Отмена заданий

Еще одно преимущество класса `Task` перед ручным управлением потоками заключается в возможности отменять выполнение заданий. Подобно асинхронным вычислительным выражениям, операция отмены

заключается в отправке запроса и не приводит к немедленному завершению задания. Поэтому чтобы убедиться, что выполнение задания было прервано, необходимо явно проверить это.

Как упоминалось выше, библиотека Parallel Extensions for .NET использует тот же механизм отмены, что и асинхронные вычислительные выражения. В частности, каждое задание может быть связано с объектом типа `CancellationToken`, который управляет отменой его выполнения. Объект типа `CancellationToken` создается объектом `CancellationTokenSource`, который фактически отправляет запрос отмены.

В примере 11.13 показан запуск задания, выполняющегося в течение 10 секунд, если оно не будет прервано. С заданием связан экземпляр класса `CancellationToken`, который создается с помощью `longTaskCTS CancellationTokenSource`. После получения запроса на отмену задание может либо завершиться, выполнив все необходимые для этого операции, либо просто сгенерировать исключение `OperationCancelledException`.

Пример 11.13. Отмена выполнения заданий

```
open System
open System.Threading
open System.Threading.Tasks

let longTaskCTS = new CancellationTokenSource()

let longRunningTask() =
    let mutable i = 1
    let mutable loop = true

    while i <= 10 && loop do
        printfn "%d..." i
        i <- i + 1
        Thread.Sleep(1000)

        // Проверить, не был ли получен запрос на прерывание
        if longTaskCTS.IsCancellationRequested then
            printfn "Cancelled; stopping early."
            loop <- false

    printfn "Complete!"

let startLongRunningTask() =
    Task.Factory.StartNew(longRunningTask, longTaskCTS.Token)

let t = startLongRunningTask()

// ...

longTaskCTS.Cancel()
```

Исключения

Обработка исключений при использовании PFX требует приложить немного больше усилий. Если задание сгенерирует исключение, оно вернет исключение типа `System.AggregateException`. Может показаться странным, почему задания не возвращают исключения оригинального типа, но не забывайте, что задания выполняются в контексте, совершено отличном от того, где исключение будет перехвачено. А что случится, если задание запустит дочернее задание, которое сгенерирует исключение?

Тип `AggregateException` объединяет все исключения, генерируемые заданиями, которые будут доступны позднее через коллекцию `InnerExceptions`.

В примере 11.14 показано использование библиотеки PFX для реализации поиска в графе, который в данном примере представляет волшебную пещеру, наполненную сокровищами и йети. Всякий раз, когда встречается новая развилка, запускается новое задание, чтобы выполнить обход левой и правой ветвей.

Пример 11.14. Использование библиотеки PFX для поиска в графе

```
open System
open System.Threading.Tasks

type CaveNode =
    | ManEatingYeti of string
    | BottomlessPit
    | Treasure
    | DeadEnd
    | Fork of CaveNode * CaveNode

let rec findAllTreasure node =
    match node with

    // Найти сокровища в волшебной пещере совсем непросто...
    | ManEatingYeti(n) -> failwithf "Eaten by a yeti named %s" n
    | BottomlessPit -> failwith "Fell into a bottomless pit"

    // Лист дерева
    | DeadEnd -> 0
    | Treasure -> 1

    // Ветви. Отправить половину экспедиции влево,
    // оставшуюся половину - вправо...
    // ... Надеемся, что никто не будет съеден Йети
    | Fork(left, right) ->
        let goLeft = Task.Factory.StartNew<int>(fun () -> findAllTreasure left)
```

```
        let goRight = Task.Factory.StartNew<int>(fun () -> findAllTreasure right)
        goLeft.Result + goRight.Result
```

Следующий фрагмент вызывает функцию `findAllTreasure` **и обрабатывает исключение** `AggregateException`**. Обратите внимание, что для извлечения исключений, отличных от** `AggregateExceptions`**, используется функция** `decomposeAggregEx`**:**

```
let colossalCave =
    Fork(
        Fork(
            DeadEnd,
            Fork(Treasure, BottomlessPit)
        ),
        Fork(
            ManEatingYeti("Umaro"),
            Fork(
                ManEatingYeti("Choko"),
                Treasure
            )
        )
    )

try
    let treasureFindingTask =
        Task.Factory.StartNew<int>(fun () -> findAllTreasure colossalCave)

    printfn "Found %d treasure!" treasureFindingTask.Result

with
| :? AggregateException as ae ->

    // Получить все исключения, хранящиеся в агрегатном исключении.
    let rec decomposeAggregEx (ae : AggregateException) =
        seq {
            for innerEx in ae.InnerExceptions do
                match innerEx with
                | :? AggregateException as ae -> yield! decomposeAggregEx ae
                | _ -> yield innerEx
        }

    printfn "AggregateException:"

    // Как вариант, чтобы поместить все вложенные исключения в
    // единое исключение AggregateException, можно использовать ae.Flatten().
    decomposeAggregEx ae
    |> Seq.iter (fun ex -> printfn "\tMessage: %s" ex.Message)
```

Параллельные структуры данных

Безусловно, умение работать с типом Task совершенно необходимо для работы с библиотекой PFX и разработки программ, выполняющих параллельные вычисления. Однако во многих случаях одного только свойства Task.Result недостаточно для сохранения результатов. Чаще всего вам придется работать с заданиями, взаимодействующими друг с другом и использующими общие структуры данных.

Но совместное использование структур данных несколькими заданиями может вызывать проблемы гонок, о чем рассказывалось ранее в этой главе. Можно было бы блокировать доступ к данным при каждом обращении к ним, но захват блокировки – это достаточно дорогостоящая операция, которая может оказать отрицательное влияние на производительность, если она выполняется напрасно. (То есть если блокировка захватывается тогда, когда никакие другие задания не обращаются к данным.) Кроме того, действительно ли необходимо захватывать блокировку при операции чтения?

Библиотека PFX вводит несколько новых типов коллекций, решающих эту проблему. Пространство имен System.Collections.Concurrent содержит версии стандартных типов коллекций, которые могут безопасно совместно использоваться несколькими потоками. Каждый тип коллекции предусматривает блокирование доступа к данным, гарантируя их целостность, но блокировки реализованы так, чтобы обеспечить максимальную производительность. (Или, если это возможно, используют алгоритмы доступа без блокировок.)

ConcurrentQueue

В параллельном программировании часто используется шаблон, когда программа создает список заданий и распределяет их выполнение по нескольким потокам. Однако при использовании простого списка может возникнуть состояние гонки. К счастью, тип ConcurrentQueue позволяет решить эту проблему.

Эта коллекция представляет собой очередь типа «первый пришел, первый вышел», которая может безопасно использоваться несколькими потоками. Основными методами этой коллекции являются методы TryDequeue и Enqueue. Приставка Try указывает, что функция возвращает true, если операция выполнена успешно, и false – в противном случае. (В контексте параллельных структур данных чаще всего это может быть вызвано гонкой.)

Подобно методу Dictionary.TryGetValue, метод TryDequeue возвращает кортеж, первый элемент которого является признаком успешности операции:

```
open System.Collections.Concurrent

let cq = new ConcurrentQueue<int>()
[1 .. 10] |> List.iter cq.Enqueue
```

```
let success, firstItem = cq.TryDequeue()
if success then
    printfn "%d was successfully dequeued" firstItem
else
    printfn "Data race, try again later"
```

ConcurrentDictionary

Тип ConcurrentDictionary ведет себя точно так же, как обычные словари. В примере 11.15 показано использование ConcurrentDictionary для подсчета количества вхождений разных слов, встречающихся в множестве файлов.

В этом примере используется метод AddOrUpdate. Тип ConcurrentDictionary имеет различные методы добавления новых элементов в словарь, и метод AddOrUpdate является одним из них. Он принимает ключ и добавляет новое значение, если ключ отсутствует в словаре, или вызывает указанную функцию, изменяющую значение ключа, если ключ уже присутствует. Данный метод упрощает добавление новых элементов и позволяет обойтись без вызова метода TryAdd и проверки его результата.

Пример 11.15. Использование ConcurrentDictionary

```
open System
open System.IO
open System.Collections.Concurrent

// Очередь обрабатываемых файлов
let filesToProcess =
    Directory.GetFiles(@"D:\Book\", "*.txt")
    |> (fun files -> new ConcurrentQueue<_>(files))

// Словарь для хранения информации о количестве вхождений различных слов
let wordUsage = new ConcurrentDictionary<string, int>()

let processFile filePath =
    let text = File.ReadAllText(filePath)
    let words =
        text.Split([| ' '; '\r'; '\n' |], StringSplitOptions.RemoveEmptyEntries)
        |> Array.map (fun word -> word.Trim())

    // Добавить слово в словарь. Вставить значение '1' или, если ключ уже
    // присутствует, изменить значение, прибавив к нему '1'.
    Array.Parallel.iter(fun word ->
        wordUsage.AddOrUpdate(
            word,
            1,
            new Func<string,int,int>(fun _ count -> count + 1)
        ) |> ignore
    )
```

```
// Приступить к заполнению словаря wordUsage
let fillDictionary() =
    let mutable continueWorking = true

    while continueWorking do

        let dequeueSuccessful, filePath = filesToProcess.TryDequeue()

        if not dequeueSuccessful then
            // Если очередь опустела, завершить работу
            if filesToProcess.IsEmpty then
                continueWorking <- false
            // ... в противном случае неудача обусловлена тем, что сразу два
            // задания попытались одновременно обратиться к очереди.
            // Повторить попытку.
            else
                continueWorking <- true
        // Обработать файл
        processFile filePath

// Запустить три задания подсчета частоты встречаемости слов
fillDictionary()
fillDictionary()
fillDictionary()
```

ConcurrentBag

Класс `ConcurrentBag` действует практически так же, как тип `HashSet<_>`, о котором рассказывалось в главе 4, за исключением того, что он позволяет добавлять в коллекцию дубликаты. Это небольшая проблема, так как сразу после заполнения коллекции вы сможете отфильтровать дубликаты самостоятельно.

Тип `ConcurrentBag` идеально подходит для объединения результатов работы множества заданий.

В примере 11.16 выполняется запуск нескольких заданий, отыскивающих множество простых чисел, меньших указанного максимального значения. Результаты работы этих заданий сохраняются в коллекции типа `ConcurrentBag`. Если бы в этом примере использовалась коллекция типа `HashSet<_>`, это могло бы привести к «потере» некоторых значений из-за гонки при попытке обновления внутренних структур данных `HashSet`. (Или хуже того – сами данные могли бы быть повреждены.)

Функция `computePrimes` возвращает коллекцию типа `ConcurrentBag` как экземпляр `seq<int>` и затем передает ее конструктору `HashSet<_>`. (Класс `ConcurrentBag`, подобно большинству коллекций, реализует интерфейс `IEnumerable<_>`, благодаря чему может приводиться к типу `seq<_>`.)

Пример 11.16. Использование коллекции типа ConcurrentBag для хранения результатов заданий, выполняющихся параллельно

```
open System
open System.Threading.Tasks
```

```
open System.Collections.Concurrent
open System.Collections.Generic

/// Проверяет, является ли число простым
let isPrime x =
    let rec primeCheck count =
        // Если счетчик достиг значения x, следовательно, число x - простое
        if count = x       then true
        // Если x делится на значение счетчика, число x – не простое
        elif x % count = 0 then false
        else primeCheck (count + 1)
    // Специальный случай: число 1 - простое
    if x = 1
    then true
    else primeCheck 2

let computePrimes tasksToSpawn maxValue =
    let range = maxValue / tasksToSpawn
    let primes = new ConcurrentBag<int>()

    // Запустить сразу несколько заданий, добавляющих простые числа
    // в коллекцию типа ConcurrentBag
    let tasks =
        [|
        for i in 0 .. tasksToSpawn - 1 do
            yield Task.Factory.StartNew(
                Action(fun () ->
                    for x = i * range to (i + 1) * range - 1 do
                        if isPrime x then primes.Add(x)
                )
            )
        |]
    Task.WaitAll(tasks)

    new HashSet<_>(primes :> seq<int>)
```

Теперь вы можете писать на языке F# программы, выполняющие асинхронные и параллельные операции, используя преимущества замечательных библиотек. Кроме того, вы можете видеть, почему так выгодно использовать функциональный стиль при выполнении параллельных операций. Создание функций без побочных эффектов упрощает декомпозицию задач на более мелкие операции, которые могут выполняться независимо друг от друга, и снижает вероятность появления ошибок.

Не забывайте, что добавление параллелизма в приложения не всегда приводит к повышению их производительности. Всякий раз, внося какие-либо архитектурные изменения с целью повысить производительность, обязательно тестируйте скорость выполнения своего приложения.

12

Рефлексия

Мы познакомились практически со всеми синтаксическими конструкциями языка F#, а также с парадигмами программирования, которые он поддерживает. В этой главе вы не будете изучать новые особенности языка F# как такового – вместо этого вы узнаете, как использовать *.NET Reflection API*.

Рефлексия позволяет во время выполнения получить доступ к метаданным с информацией о выполняемом коде. Метаданные могут быть простой информацией о типе, такой как сведения о методах класса, или принимать форму *атрибутов*, которые являются аннотациями кода, созданными программистом.

Механизм рефлексии в .NET обычно используется в *метапрограммировании*, то есть при разработке программ, которые могут анализировать или изменять себя, например, чтобы загрузить плагины (plug-ins) для добавления новых функциональных возможностей прямо во время выполнения. Другими словами, механизм рефлексии позволяет другим разработчикам расширять ваше приложение без доступа к его исходным текстам.

Но прежде чем вы сможете приступить к метапрограммированию, вам необходимо узнать, как аннотировать код для добавления в него необходимых метаданных.

Атрибуты

Атрибуты – это форма метаданных, которые присоединяются к сборкам, методам, параметрам и так далее. Эти аннотации могут быть получены во время выполнения с помощью механизма рефлексии, с которым мы познакомимся ниже в этой главе. Компилятор F# также распознает эти атрибуты. Фактически вам уже приходилось использовать атрибуты, чтобы подсказать компилятору, как компилировать ваш код.

Вспомните, как в главе 5 мы добавляли атрибут [<Abstract>] в объявление типа, чтобы пометить класс как абстрактный. Когда компилятор F# встречает этот атрибут, он по-иному генерирует код, чем в случае отсутствия атрибута. То же относится к атрибутам [<Struct>], [<ReferenceEquality>] и другим:

```
[<AbstractClass>]
type Animal =
    abstract Speak : unit -> unit
    abstract NumberOfLegs : int with get
```

При программировании для .NET используется множество различных атрибутов, которые также распознаются компилятором F#. Например, System.ObsoleteAttribute используется для указания устаревших методов и классов. При использовании программных элементов, помеченных атрибутом [<Obsolete>], компилятор F# генерирует сообщение об ошибке или предупреждение в зависимости от способа использования.

В примере 12.1 приводится код на языке F#, который впоследствии был переработан. В обновленную версию был добавлен атрибут [<Obsolete>], чтобы сообщить разработчикам, использующим класс, о необходимости обновить свой код.

Как вы уже видели раньше, для добавления атрибута достаточно просто вставить его имя, заключив в скобки [< >], перед определением класса или метода, к которому требуется его применить. Если имя атрибута оканчивается суффиксом *Attribute*, то его можно опустить. Например, для использования атрибута System.ObsoleteAttribute достаточно просто добавить в код [<Obsolete>], если пространство имен System было открыто.

Пример 12.1. Атрибут Obsolete

```
open System

type SafetyLevel =
    | RecreationalUse = 1
    | HaveMcCoyOnDuty = 3
    | SetPhasersToKill = 4

type HoloDeck(safetyLevel : SafetyLevel) =

  let mutable m_safetyLevel = safetyLevel

  member this.SafetyLevel with get () = m_safetyLevel

  [<Obsolete("Deprecated. Cannot update protocols once initialized.", true)>]
  member this.SafetyLevel with set x = m_safetyLevel <- x
```

На рис. 12.1 показано окно редактора Visual Studio 2010 после попытки вызова метода или свойства, помеченного атрибутом [<Obsolete>].

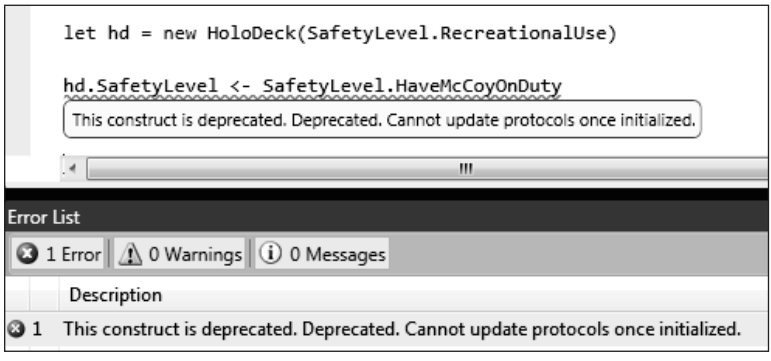


Рис. 12.1. *Предупреждения и сообщения об ошибках в редакторе Visual Studio*

Применение атрибутов

Далее в этой главе вы узнаете, как анализировать информацию о типе и проверять имеющиеся атрибуты. Но прежде чем анализировать метаданные, их необходимо получить, поэтому давайте поближе посмотрим на то, как добавляются атрибуты в ваш код.

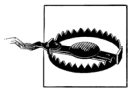
Цели атрибутов

Чаще всего атрибуты располагаются непосредственно перед типом или методом, к которому требуется его применить. Однако иногда цель атрибута бывает не так однозначна. Чтобы применить атрибут к какой-то определенной конструкции, можно явно указать *цель атрибута (atribute target)*, то есть ключевое слово, указывающее на то, к чему применяется атрибут.

Например, атрибуты уровня сборки имеют глобальный характер, в том смысле, что они принадлежат не какому-то определенному классу или методу, а применяются ко всей сборке в целом. Поэтому для таких атрибутов требуется указывать цель, чтобы показать, что эти атрибуты применяются к сборке, а не к следующему за ним объявлению типа или значения.

К сборкам чаще всего применяются атрибуты, определяющие версию сборки и содержащие информацию об авторских правах. Обратите внимание, что в следующем примере используются коды символов \169 и \174 для обозначения символов © и ‘‘ соответственно:

```
open System.Reflection

[<assembly:AssemblyDescription("RemoteCombobulator.exe")>]
[<assembly:AssemblyCompany("Initech Corporation")>]
[<assembly:AssemblyCopyright("\169 Initech Corporation. All rights reserved.")>]
[<assembly:AssemblyProduct("Initech \174 RemoteCombobulator")>]
do()
```



В языке F# атрибуты сборки должны помещаться внутрь модуля непосредственно перед инструкцией связывания `do`.

В табл. 12.1 приводится перечень наиболее часто используемых целей атрибутов.

Таблица 12.1. Цели атрибутов

Имя цели	Описание
`assembly`	Применяется к сборке
`field`	Применяется к полю класса
`parameter`	Применяется к параметру метода
`return`	Применяется к возвращаемому значению метода или свойства

Применение нескольких атрибутов

Когда нужно применить сразу несколько атрибутов, их можно добавить по отдельности или сгруппировать в список, разделяя их имена точкой с запятой.

В примере 12.2 показаны те же атрибуты сборки, что и в предыдущем фрагменте, но записанные в более понятной форме.

Пример 12.2. Применение нескольких атрибутов

```
open System.Reflection
[<
 assembly:AssemblyDescription("RemoteCombobulator.exe");
 assembly:AssemblyCompany("Initech Corporation");
 assembly:AssemblyCopyright("\169 Initech Corporation. All rights reserved.");
 assembly:AssemblyProduct("Initech \174 RemoteCombobulator")
>]
do()
```

Определение новых атрибутов

Для определения новых собственных атрибутов достаточно просто создать класс, наследующий класс `System.Attribute`. Определение собственных атрибутов позволяет создавать свои аннотации типов.

В примере 12.3 определяется пара новых атрибутов, позволяющих добавлять описание классов и методов. Комментарии XML-документации недоступны во время выполнения, тогда как новые атрибуты `ClassDescription` и `MethodDescription` будут доступны.

Обратите внимание, что к определениям новых атрибутов применяется необязательный атрибут `AttributeUsage`, который определяет, к чему могут применяться эти вновь созданные атрибуты. (Самое настоящее мета-метапрограммирование!) В примере 12.3 атрибут `AttributeUsage`

указывает, что атрибут MethodDescriptionAttribute может применяться только к методам, иначе компилятор F# выдаст сообщение об ошибке. Аналогично атрибут ClassDescriptionAttribute может применяться только к классам.

Пример 12.3. Определение собственных атрибутов

```
open System

/// Содержит описание заданного класса
[<AttributeUsage(AttributeTargets.Class)>]
type ClassDescriptionAttribute(desc) =
    inherit Attribute()
    member this.Description = desc

/// Содержит описание заданного метода
[<AttributeUsage(AttributeTargets.Method)>]
type MethodDescriptionAttribute(desc) =
    inherit Attribute()
    member this.Description = desc

type Widget =
    | RedWidget
    | GreenWidget
    | BlueWidget

/// Комментарии XML-документирования, подобные этому, отлично подходят
/// для описания классов, но они доступны только на этапе компиляции.
/// Метаданные, представленные в виде атрибутов, доступны во время выполнения.
[<ClassDescription("Represents a stack of Widgets.")>]
type WidgetStack() =

    let mutable m_widgets : Widget list = []

    [<MethodDescription("Pushes a new Widget onto the stack.")>]
    member this.Push(x) = m_widgets <- x :: m_widgets

    [<MethodDescription("Access the top of the Widget stack.")>]
    member this.Peek() = List.head m_widgets

    [<MethodDescription("Pops the top Widget off the stack.")>]
    member this.Pop() = let top = List.head m_widgets
                        m_widgets <- List.tail m_widgets
                        top
```

Рефлексия типов

Итак, теперь вы знаете, как добавлять атрибуты в код, но как можно проверить наличие и содержимое этих атрибутов? Получить доступ к атрибутам в коде можно с помощью механизма *рефлексии типов (type reflection)*.

Напомню, что в .NET все программные элементы прямо или косвенно наследуют класс System.Object, и поэтому любое значение в .NET имеет метод GetType, который возвращает экземпляр класса System.Type. Класс Type описывает все, что он знает о типе, включая и атрибуты.

Получение информации о типе

Ниже приводится листинг сеанса окна FSI, в котором определяется новый тип с именем Sprocket, и затем используются два способа получения информации о типе. В первом случае используется обобщенная функция typeof<_>, принимающая параметр обобщенного типа, которая возвращает тип указанного параметра. Во втором случае вызывается метод GetType экземпляра Sprocket, унаследованный от типа obj:

```
> // Определение простого типа
type Sprocket() =
    member this.Gears = 16
    member this.SerialNumber = "WJM-0520";;

type Sprocket =
  class
    new : unit -> Sprocket
    member Gears : int
    member SerialNumber : string
  end

> // Получить информацию о типе с помощью функции typeof<_>
let type1 = typeof<Sprocket>;;

val type1 : System.Type = FSI_0002+Sprocket

> // Получить информацию о типе с помощью метода GetType
let aSprocket = new Sprocket()
let type2 = aSprocket.GetType();;

val aSprocket : Sprocket
val type2 : System.Type = FSI_0002+Sprocket

> type1 = type2;;
val it : bool = true
> type1.Name;;
val it : string = "Sprocket"
```

Получение информации об обобщенных типах

Функцию typeof<_> достаточно удобно использовать для получения информации о типе, однако когда ей передается параметр обобщенного типа, она, будучи не в состоянии точно определить его тип, возвращает тип obj. Если вам необходимо, чтобы возвращаемый тип был полностью обобщенным, используйте функцию typedefof<_>.

Следующий фрагмент демонстрирует получение информации о типах последовательности. Обратите внимание, что функция typedefof<_> возвращает обобщенный тип, тогда как функция typeof<_> сообщает, что последовательность содержит типы obj, что, вообще говоря, не совсем правильно:

```
> let fixedSeq = typeof< seq<float> >;;

val fixedSeq : Type = System.Collections.Generic.IEnumerable`1[System.Double]

> let genSeq = typeof< seq<'a> >;;

val genSeq : Type = System.Collections.Generic.IEnumerable`1[System.Object]

> let fullyGenSeq = typedefof< seq<'a> >;;

val fullyGenSeq : Type = System.Collections.Generic.IEnumerable`1[T]
```

Изучение методов

Получив экземпляр класса Type, можно приступать к изучению методов и свойств типа путем вызова методов GetMethods и GetProperties и так далее.

В примере 12.4 определяется функция с именем describeType, которая выводит все методы, свойства и поля типа.

Перечисление BindingFlags используется для фильтрации результатов. Например, флаг BindingFlags.DeclaredOnly используется для того, чтобы метод GetMethods не возвращал унаследованные методы типа, такие как GetHashCode и ToString.

Пример 12.4. Изучение и анализ типов

```
open System
open System.Reflection

/// Выводит методы, свойства и поля типа
let describeType (ty : Type) =

    let bindingFlags =
        BindingFlags.Public   ||| BindingFlags.NonPublic |||
        BindingFlags.Instance ||| BindingFlags.Static    |||
        BindingFlags.DeclaredOnly

    let methods =
        ty.GetMethods(bindingFlags)
        |> Array.fold (fun desc meth -> desc + sprintf " %s" meth.Name) ""

    let props =
        ty.GetProperties(bindingFlags)
        |> Array.fold (fun desc prop -> desc + sprintf " %s" prop.Name) ""
```

```
    let fields =
        ty.GetFields(bindingFlags)
        |> Array.fold (fun desc field -> desc + sprintf " %s" field.Name) ""

    printfn "Name: %s" ty.Name
    printfn "Methods:    \n\t%s\n" methods
    printfn "Properties: \n\t%s\n" props
    printfn "Fields:     \n\t%s" fields
```

Ниже приводится листинг в окне FSI, демонстрирующий использование функции `describeType` **для описания типа** `string`**:**

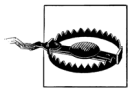
```
Name: String
Methods:
     Join get_FirstChar Join nativeCompareOrdinal nativeCompareOrdinalEx
nativeCompareOrdinalWC SmallCharToUpper EqualsHelper CompareOrdinalHelper
Equals Equals Equals Equals Equals op_Equality op_Inequality get_Chars CopyTo
To CharArray ToCharArray IsNullOrEmpty GetHashCode get_Length get_ArrayLength
get_Capacity Split Split Split Split Split Split InternalSplitKeepEmptyEntries
InternalSplitOmitEmptyEntries MakeSeparatorList MakeSeparatorList Substring
Substring InternalSubStringWithChecks InternalSubString Trim TrimStart TrimEnd
CreateString

...пропущено...

Properties:
     FirstChar Chars Length ArrayLength Capacity

Fields:
    m_arrayLength m_stringLength m_firstChar Empty WhitespaceChars TrimHead
                                             TrimTail TrimBoth
charPtrAlignConst alignConst
```

А теперь попробуем применить функцию `describeType` **к нашему собственному классу. Для начала определим тип с именем** `Cog`**. Обратите внимание на наличие модификатора доступа** `private` **у поля и свойства класса:**

```
/// Единица измерения - миллилитр
[<Measure>]
type ml

type Cog =

    val mutable private m_oilLevel : float<ml>

    new() = { m_oilLevel = 100.0<ml> }

    member private this.Gears = 16
    member this.SerialNumber = "1138-THX"

    member this.Rotate() =
```

```
        // При вращении механизма на каждом обороте теряется капля масла...
        this.m_oilLevel <- this.m_oilLevel - 0.01<ml>
```

Теперь вызовем функцию `describeType` и передадим ей тип `Cog`. Обратите внимание, что выводится вся информация о типе, включая члены, помеченные модификатором `private`.



Модификаторы доступа на уровне языка предусмотрены для помощи программистам, однако объявление какого-либо компонента закрытым (`private`) или внутренним (`internal`) не обеспечивает его безопасность во время выполнения. Используя механизм рефлексии в .NET, можно получить доступ к любым полям и членам объекта. Имейте это в виду при сохранении секретной информации в памяти.

```
> describeType (typeof<Cog>);;
Name: Cog
Methods:
     get_Gears get_SerialNumber Rotate

Properties:
     SerialNumber Gears

Fields:
     m_oilLevel
val it : unit = ()
```

Изучение атрибутов

Изучение атрибутов типа производится практически так же, как изучение методов и свойств. Единственное отличие состоит в том, что атрибуты могут быть размещены в различных местах.

В примере 12.5 показано изучение атрибутов `ClassDescription` и `MethodDescription`, реализованных нами выше. Обратите внимание, как выполняется фильтрация полного списка нестандартных атрибутов, чтобы оставить только атрибуты `ClassDescription` и `MethodDescription`.

Пример 12.5. Исследование атрибутов типа

```
open System

/// Отображает элементы, отмеченные атрибутами MethodDesc и ClassDesc
let printDocumentation(ty : Type) =
    // Возвращает true, если объект имеет указанный тип
    let objHasType ty obj = (obj.GetType() = ty)

    let classDescription : string option =
        ty.GetCustomAttributes(false)
        |> Seq.tryFind(objHasType typeof<ClassDescriptionAttribute>)
        |> Option.map(fun attr -> (attr :?> ClassDescriptionAttribute))
        |> Option.map(fun cda -> cda.Description)
```

```
let methodDescriptions : seq<string * string option> =
    ty.GetMethods()
    |> Seq.map(fun mi -> mi, mi.GetCustomAttributes(false))
    |> Seq.map(fun (methodInfo, methodAttributes) ->
        let attributeDescription =
            methodAttributes
            |> Seq.tryFind(objHasType typeof<MethodDescriptionAttribute>)
            |> Option.map(fun atr -> (atr :?> MethodDescriptionAttribute))
            |> Option.map(fun mda -> mda.Description)
        methodInfo.Name, attributeDescription)

let getDescription = function
                     | Some(desc) -> desc
                     | None -> "(no description provided)"

printfn "Info for class: %s" ty.Name
printfn "Class Description:\n\t%s" (getDescription classDescription)
printfn "Method Descriptions:"

methodDescriptions
|> Seq.iter(fun (methName, desc) -> printfn
                                        "\t%15s - %s"
                                        methName
                                        (getDescription desc))
```

Если применить функцию `printDocumentation` к типу `WidgetStack` из примера 12.3, она выведет следующие результаты:

```
> printDocumentation (typeof<WidgetStack>);;
Info for class: WidgetStack
Class Description:
    Represents a stack of Widgets.
Method Descriptions:
               Push - Pushes a new Widget onto the stack.
               Peek - Access the top of the Widget stack.
                Pop - Pops the top Widget off the stack.
           ToString - (no description provided)
             Equals - (no description provided)
         GetHashCode - (no description provided)
            GetType - (no description provided)
val it : unit = ()
```

Рефлексия типов языка F#

API рефлексии в .NET достаточно прост в использовании, однако он был реализован задолго до появления языка F# и потому не имеет никакого представления об особых его типах. По этой причине при изучении таких элементов, как записи или размеченные объединения, можно получить довольно неожиданные результаты.

Рассмотрим следующее размеченное объединение, описывающее игральные карты. Фактически тип `PlayingCard` имеет очень простую структуру:

```
type Suit =
  | Club
  | Diamond
  | Heart
  | Spade

type PlayingCard =
  | Ace   of Suit
  | King  of Suit
  | Queen of Suit
  | Jack  of Suit
  | ValueCard of int * Suit
  | Joker
```

Однако если применить к нему нашу функцию `describeType`, можно увидеть, как компилятор F# отображает размеченное объединение на объектно-ориентированную среду выполнения .NET:

```
> describeType (typeof<PlayingCard>);;
Name: PlayingCard
Methods:
    get_Joker get_IsJoker NewValueCard get_IsValueCard NewJack get_IsJack
    NewQueen get_IsQueen NewKing get_IsKing NewAce get_IsAce get_Tag
    __DebugDisplay CompareTo CompareTo CompareTo GetHashCode GetHashCode
    Equals Equals Equals

Properties:
    Tag Joker IsJoker IsValueCard IsJack IsQueen IsKing IsAce

Fields:
    _tag _unique_Joker
```

К счастью, в стандартной библиотеке F# в пространстве имен `Microsoft.FSharp.Reflection` имеется несколько дополнительных функций, позволяющих выполнять рефлексию функциональных типов.

Кортежи

В основе кортежей лежит тип `System.Tuple<_>`. Чтобы проанализировать конкретный экземпляр кортежа и определить типы его элементов, можно воспользоваться методом `FSharpType.GetTupleElements`, который возвращает типы элементов кортежа в виде массива:

```
> let xenon = ("Xe", 54);;

val xenon : string * int = ("Xe", 54)

> // Определить типы элементов кортежа
open Microsoft.FSharp.Reflection
let tupleElementTypes = FSharpType.GetTupleElements (xenon.GetType());;

val tupleElementTypes : Type [] = [|System.String; System.Int32|]
```

Размеченные объединения

Перечень вариантов размеченного объединения можно получить с помощью метода FSharpType.GetUnionCases. Он возвращает массив элементов типа UnionCaseInfo, содержащих описания всех вариантов.

В следующем фрагменте демонстрируется пример исследования типа PlayingCard и вывод имен всех вариантов:

```
> // Исследование типа размеченного объединения PlayingCard
FSharpType.GetUnionCases typeof<PlayingCard>
|> Array.iter (fun unionCase -> printfn "%s" unionCase.Name);;
Ace
King
Queen
Jack
ValueCard
Joker
val it : unit = ()
```

Записи

Для получения перечня всех свойств записи можно использовать метод FSharpType.GetRecordFields:

```
> // Определения типов
[<Measure>]
type far // Градусы по Фаренгейту

type Outlook = Sunny | Cloudy | Rainy

type Weather = { Outlook : Outlook; High : float<far>; Low : float<far> };;

...пропущено...

> // Исследование записи типа Weather
FSharpType.GetRecordFields typeof<Weather>
|> Array.iter (fun propInfo -> printfn
                                  "Name [%s], Type [%s]"
                                  propInfo.Name propInfo.PropertyType.Name);;

Name [Outlook], Type [Outlook]
Name [High], Type [Double]
Name [Low], Type [Double]
val it : unit = ()
```

Динамическое создание экземпляров

В предыдущем разделе были представлены способы изучения типов и их метаданных. Однако истинная мощь механизма рефлексии заключается в возможности *динамического создания экземпляров (dynamic instantiation)*, что означает возможность «создавать экземпляры во вре-

мя выполнения, тип которых не был известен на этапе компиляции». Другими словами, эта возможность позволяет создавать новые экземпляры, имея экземпляр класса System.Type, описывающего тип.

Создание экземпляров типов

Динамическое создание экземпляров можно выполнить с помощью класса Activator. В примере 12.6 определяется тип PoliteWriter и с помощью метода Activator.CreateInstance динамически создается новый экземпляр этого типа.

Параметры для конструктора типа PoliteWriter передаются в виде массива объектов obj. Результатом вызова метода CreateInstance является экземпляр типа obj, который необходимо привести к требуемому типу.

Пример 12.6. Динамическое создание экземпляров

```
> // Определение типа PoliteWriter
open System.IO

type PoliteWriter(stream : TextWriter) =

    member this.WriteLine(msg : string) =
        sprintf "%s... please" msg
        |> stream.WriteLine;;

type PoliteWriter =
  class
    new : stream:IO.TextWriter -> PoliteWriter
    member WriteLine : msg:string -> unit
  end

> // Создать экземпляр этого класса динамически
let politeConsole =
    Activator.CreateInstance(typeof<PoliteWriter>, [| box Console.Out |]);;

val politeConsole : obj

> (politeConsole :?> PoliteWriter).WriteLine("Hello, World!");;
Hello, World!... please
val it : unit = ()
```

Теперь вы можете не только загружать типы, но и создавать новые экземпляры этих типов во время выполнения. Это открывает перед приложениями на языке F# новые возможности, такие как загрузка плагинов.

Создание экземпляров типов языка F#

Подобно дополнительным функциям, расширяющим механизм рефлексии и позволяющим изучать функциональный код, в языке F# имеются также специальные методы для динамического создания экземп-

ляров. В табл. 12.2 перечислены некоторые наиболее часто используемые методы в модуле FSharpValue, позволяющие динамически создавать экземпляры функциональных типов, таких как кортежи и размеченные объединения.

Таблица 12.2. Наиболее часто используемые методы в модуле FSharpValue

Сигнатура	Описание
MakeTuple: obj[] * Type -> obj	Создает новый экземпляр кортежа.
MakeRecord: Type * obj[] -> obj	Создает новый экземпляр записи.
MakeUnion: UnionCaseInfo * obj[] -> obj	Создает новый экземпляр размеченного объединения. Получить экземпляр типа UnionCaseInfo можно с помощью FSharpType.GetUnionCases.

Динамический вызов

Если существует возможность динамической загрузки типов и создания их экземпляров, то неужели это и все? В .NET имеется возможность производить *динамические вызовы (dynamic invocation)* (еще один интересный термин, обозначающий *позднее связывание (late binding)*), то есть «вызывать во время выполнения методы, о которых ничего не было известно на этапе компиляции». Используя механизм рефлексии, можно получить доступ к методу или свойству экземпляра и вызвать его.

В примере 12.7 определяется класс Book и динамически получаются метаданные свойства CurrentPage этого типа. Далее их можно использовать для изменения значения свойства не путем непосредственного обращения к экземпляру типа Book, а за счет вызова метода SetValue объекта PropertyInfo.

Пример 12.7. Динамический вызов

```
> // Тип, представляющий книгу
type Book(title, author) =
    // Текущая страница, если книга была открыта
    let mutable m_currentPage : int option = None

    member this.Title = title
    member this.Author = author
    member this.CurrentPage with get () = m_currentPage
                            and set x = m_currentPage <- x

    override this.ToString() =
        match m_currentPage with
```

```
        | Some(pg) -> sprintf "%s by %s, opened to page %d" title author pg
        | None     -> sprintf "%s by %s, not currently opened" title author;;

    ...пропущено...

> let afternoonReading = new Book("The Mythical Man Month", "Brooks");;

val afternoonReading : Book =
  The Mythical Man Month by Brooks, not currently opened

> let currentPagePropertyInfo = typeof<Book>.GetProperty("CurrentPage");;

val currentPagePropertyInfo : PropertyInfo =
  Microsoft.FSharp.Core.FSharpOption`1[System.Int32] CurrentPage

> currentPagePropertyInfo.SetValue(afternoonReading, Some(214), [| |]);;
val it : unit = ()
> afternoonReading.ToString();;
val it : string = "The Mythical Man Month by Brooks, opened to page 214"
```

Впрочем, позднее связывание обычно не используется на практике, потому что, когда вы получаете объект типа `obj`, вы уже знаете, какому типу он должен принадлежать, и можете использовать оператор динамического приведения типа (`:?>`), после чего уже вызвать требуемый метод. Позднее связывание часто используется в динамических языках программирования, допускающих возможность добавления в объекты методов и свойств во время выполнения. Поэтому в них нужно сначала определить наличие нужного метода или свойства и только потом вызывать его!

Оператор «знак вопроса»

Как вы уже видели в предыдущем примере, динамическое получение свойства и вызов его в языке F# выглядит несколько неуклюже. По этой причине в язык были введены операторы «знак вопроса» `?` и «знак вопроса с присваиванием» `?<-`. Функции с этими именами имеют специальное значение для компилятора F# и существенно упрощают позднее связывание.

Если символьный оператор `?` определен, то при его вызове в первом параметре передается выражение для левой стороны, а текст для правой стороны передается в параметре типа `string`.

В следующем примере определяется оператор «знак вопроса», который проверяет, присутствует ли искомое свойство в типе, и выводит результат на экран. При использовании оператора «точка» (`.`) для доступа к членам класса проверка их существования выполняется на этапе компиляции. Благодаря оператору «знак вопроса» вы можете проверить наличие члена класса во время выполнения, имея строку с его именем:

```
> // Использование оператора "знак вопроса" для проверки наличия в типе
// заданного свойства.
let (?) (thingey : obj) (propName : string) =
    let ty = thingey.GetType()

    match ty.GetProperty(propName) with
    | null -> false
    | _    -> true;;

val ( ? ) : obj -> string -> bool

> // Любая строка имеет свойство Length
"a string"?Length;;
val it : bool = true
> // Целые числа не имеют свойства IsPrime
42?IsPrime;;
val it : bool = false
> // Даже если экземпляр типа string привести к типу obj, проверка даст
// положительный результат, так как она выполняется динамически
("a string" :> obj) ? Length;;
val it : bool = true
```

Используя оператор «знак вопроса», мы можем сделать код из примера 12.7 намного более понятным. В следующем фрагменте определяются оба оператора, «знак вопроса» и «знак вопроса с присваиванием», обеспечивающие возможность динамического доступа и изменения значения свойства класса:

```
> // Возвращает значение свойства. Обратите внимание, что возвращается
// значение обобщенного типа.
let (?) (thingey : obj) (propName: string) : 'a =
    let propInfo = thingey.GetType().GetProperty(propName)
    propInfo.GetValue(thingey, null) :?> 'a

// Устанавливает значение свойства.
let (?<-) (thingey : obj) (propName : string) (newValue : 'a) =
    let propInfo = thingey.GetType().GetProperty(propName)
    propInfo.SetValue(thingey, newValue, null);;

val ( ? ) : obj -> string -> 'a
val ( ?<- ) : obj -> string -> 'a -> unit

> let book = new Book("Foundation", "Asimov");;

val book : Book = Foundation by Asimov, not currently opened

> book?CurrentPage <- Some(14);;
val it : unit = ()
> let currentPage : int option = book?CurrentPage;;

val currentPage : int option = Some 14
```

На практике вы редко будете сталкиваться с ситуациями, когда вам будет известно имя свойства, но не известен его тип. Поэтому операторы «знак вопроса» имеют весьма ограниченное применение. Однако вполне можно представить ситуацию, когда проверяемое свойство может содержать дополнительную информацию, например `book?CurrentPageTextInSpanish`. Операторы «знак вопроса» обеспечивают дополнительную гибкость, расширяя границы возможностей кода на языке F#.

Использование рефлексии

Теперь, когда вы поближе познакомились с механизмом рефлексии, можно перейти к рассмотрению его *возможностей*. Другими словами, перейти к рассмотрению метапрограммирования.

Далее мы рассмотрим две специфические области применения механизма рефлексии в .NET: декларативное программирование и разработка расширяемых систем.

Декларативное программирование

Пожалуй, самое простое, что позволяет делать механизм рефлексии, – это писать более декларативный код. Вместо того чтобы жестко определять данные в виде свойств и методов, эти данные можно определять в виде атрибутов. Это позволяет писать более простой и читабельный код.

Рассмотрим пример 12.8, в котором задается функция `determineBoxToUse`, определяющая тип упаковки для посылки некоторого товара. Каждый тип товара, который может быть отправлен, наследует класс `ShippingItem` и переопределяет свойства `Weight` и `Dimension`, используемые для определения вида упаковки.

В этом примере пока не использовались ни механизм рефлексии, ни декларативное программирование, чтобы позднее его можно было сравнить с улучшенной версией кода.

Пример 12.8. Простейшая реализация функции выбора упаковки

```
/// Фунты
[<Measure>]
type lb

[<Measure>]
type inches

type Container =
    | Envelope
    | Box
    | Crate

type Dimensions =
```

```
    { Length : float<inches>; Width : float<inches>; Height : float<inches> }

[<AbstractClass>]
type ShippingItem() =
    abstract Weight : float<lb>
    abstract Dimension : Dimensions

// Лист бумаги с описью содержимого посылки
type ShippingManifest() =
    inherit ShippingItem()

    override this.Weight = 0.01<lb>
    override this.Dimension =
                {
                    Length = 11.0<inches>
                    Width = 8.5<inches>
                    Height = 0.01<inches>
                }

// Вам нужен блендер?
type Blender() =
    inherit ShippingItem()

    override this.Weight = 14.00<lb>
    override this.Dimension =
                {
                    Length = 6.0<inches>;
                    Width = 5.0<inches>
                    Height = 12.0<inches>
                }

/// Частичный активный шаблон, который совпадает, только если значение input
/// больше параметра.
let (|GreaterThan|_|) (param : float<'a>) input =
    if param > input
    then Some()
    else None

let determineBoxToUse(item : ShippingItem) =

    match item.Weight, item.Dimension with
    // Очень тяжелые предметы всегда должны упаковываться в деревянные ящики
    | GreaterThan 10.0<lb>, _
        -> Crate

    // Очень крупные предметы всегда должны упаковываться в деревянные ящики
    | _, { Length = GreaterThan 24.0<inches>; Width = _; Height = _}
    | _, { Length = _; Width = GreaterThan 24.0<inches>; Height = _}
    | _, { Length = _; Width = _; Height = GreaterThan 24.0<inches>}
        -> Crate
```

```
// Крупные предметы должны упаковываться в коробки
| GreaterThan 2.0<lb> _, _
    -> Box

// Минимальные размеры предметов, упаковываемых в коробки
| _, { Length = GreaterThan 10.0<inches>; Width = _; Height = _}
| _, { Length = _; Width = GreaterThan 10.0<inches>; Height = _}
| _, { Length = _; Width = _; Height = GreaterThan 10.0<inches>}
    -> Box

// Для остальных предметов вполне подойдет конверт
| _ -> Envelope
```

Пример 12.8 выглядит достаточно простым, но такой код будет сложно расширить. Представьте, что вам потребовалось расширить его так, чтобы он позволял добавлять специальные инструкции по обращению с товарами. Например, допустим, что вам необходимо учесть тот факт, что некоторые товары отличаются повышенной хрупкостью или огнеопасностью.

При использовании исключительно объектно-ориентированного подхода эту проблему можно было решить, добавив в базовый класс `ShippingItem` новые свойства:

```
[<AbstractClass>]
type ShippingItem() =
    abstract Weight : float<lb>
    abstract Dimension : Dimensions
    abstract Fragile : bool
    abstract Flammable : bool
```

Но так как в базовый класс добавляются новые свойства, вам придется либо предусмотреть реализацию по умолчанию для свойств `Fragile` и `Flammable`, либо добавить их реализацию в каждый класс, производный от `ShippingItem`. Эта проблема будет всплывать всякий раз, когда будет возникать необходимость расширить класс `ShippingItem` – либо добавлять реализацию во все классы, либо предусматривать реализацию по умолчанию в базовом классе.

Реализация по умолчанию может показаться простым решением, но не забывайте, что она усложняет базовый тип. Если, к примеру, вы впервые видите класс `ShippingItem`, содержащий 15 с лишним абстрактных методов или свойств, вам потребуется некоторое время, чтобы понять, какие из них важны, а какие нет.

Для многих классов предметов, о которых идет речь, было бы намного проще, если бы можно было предусматривать особые условия доставки непосредственно перед их объявлениями, чем добавлять в базовый класс `ShippingItem` десятки свойств, которые требуются далеко не всегда. Для обозначения особых условий доставки можно помечать эти типы атрибутами.

В примере 12.9 определяется несколько атрибутов для аннотирования типов доставляемых товаров, благодаря чему позднее можно будет выяснить эти специальные условия их доставки. Это освобождает от необходимости изменять какие-либо специальные свойства и позволяет определять новые типы товаров более декларативным способом. (Фактически свойства `Weight` и `Dimension` также можно было бы заменить атрибутами.)

Пример 12.9. Использование атрибутов для программирования в декларативном стиле

```
open System

type FragileAttribute() =
    inherit System.Attribute()

type FlammableAttribute() =
    inherit System.Attribute()

type LiveAnimalAttribute() =
    inherit System.Attribute()

/// Настоящий живой вомбат будет доставлен прямо к вашим дверям!
[<LiveAnimal>]
type Wombat() =
    inherit ShippingItem()

    override this.Weight = 60.0<lb>
    override this.Dimension =
                {
                    Length = 39.0<inches>
                    Width = 10.0<inches>
                    Height = 13.0<inches>
                }

[<Fragile; Flamable>]
type Fireworks() =
    inherit ShippingItem()

    override this.Weight = 5.0<lb>
    override this.Dimension =
                {
                    Length = 10.0<inches>
                    Width = 8.0<inches>
                    Height = 5.0<inches>
                }
```

В следующем фрагменте определяется функция `getShippingRequirements`, которая использует вложенную функцию `containsAttribute`, чтобы проверить, помечен ли данный объект указанным атрибутом. Эта функция

будет использоваться для проверки высылаемых предметов и определения дополнительных условий доставки:

```
open System.Collections.Generic

type ShippingRequirements =
    | NeedInsurance of ShippingItem
    | NeedSignature of ShippingItem
    | NeedBubbleWrap of ShippingItem

/// Определить дополнительные условия доставки
let getShippingRequirements (contents : ShippingItem list) =

    let containsAttribute (targetAttrib : Type) x =
        x.GetType().GetCustomAttributes(false)
        |> Array.tryFind(fun attr -> attr.GetType() = targetAttrib)
        |> Option.isSome

    let itemsWithAttribute attr =
        contents |> List.filter (containsAttribute attr)

    let requirements = new HashSet<ShippingRequirements>()

    // Содержит хрупкие предметы
    itemsWithAttribute typeof<FragileAttribute>
    |> List.iter (fun item -> requirements.Add(NeedBubbleWrap(item)) |> ignore)

    // Содержит легковоспламеняющиеся предметы
    itemsWithAttribute typeof<FlammableAttribute>
    |> List.iter (fun item -> requirements.Add(NeedInsurance(item)) |> ignore)

    // Живые животные
    itemsWithAttribute typeof<LiveAnimalAttribute>
    |> List.iter (fun item -> requirements.Add(NeedSignature(item)) |> ignore)

    // Вернуть список особых условий доставки
    Seq.toList requirements
```

Расширяемая архитектура

Другая область применения механизма рефлексии – реализация расширяемой архитектуры, когда другие разработчики получают возможность расширять приложение за счет создания новых сборок .NET, удовлетворяющих определенным требованиям. Вы можете использовать механизм рефлексии для динамической загрузки этих сборок и вызова кода, написанного другими программистами.

Обычно этот процесс состоит из двух этапов. Во-первых, создается сборка, определяющая базовую инфраструктуру компонентов, таких

как интерфейс и/или атрибуты для определения новых плагинов. Затем приложение ищет другие сборки, содержащие плагины, и подключает их.

Следующие примеры продолжают реализацию нашей системы доставки, дополняя ее возможностью для других разработчиков создавать новые методы доставки.

Интерфейсы плагинов

В примере 12.10 приводится код сборки `DeliverySystem.Core.dll` — библиотеки, которую должны будут использовать будущие плагины. Он определяет интерфейс `IDeliveryMethod`, который они должны будут реализовать.

Пример 12.10. DeliverySystem.Core.dll

```
// DeliverySystem.Core.dll

namespace DeliverySystem.Core

/// Представление посылки
type Package =
    class
    (* ... *)
    end

/// Адрес доставки
type Destination =
    {
        Address : string
        City    : string
        State   : string
        Zip     : int
        Country : string
    }

/// Интерфейс для новых систем доставки, которые могут быть добавлены
type IDeliveryMethod =
    interface
        abstract MethodName : string
        abstract DeliverPackage : Package * Destination -> unit
    end
```

Теперь создадим простой плагин. В примере 12.11 определяется новая сборка, которая использует `DeliverySystem.Core.dll` и реализует новую систему экспресс-доставки с именем `CarrierPigeon`.

Пример 12.11. CarrierPigeonPlugin.dll

```
// CarrierPigeonPlugin.dll

// Использует DeliverySystem.Core.dll
```

```
open DeliverySystem.Core

type CarrierPigeon() =
    interface IDeliveryMethod with

        member this.MethodName = "Carrier Pigeon"

        member this.DeliverPackage (pkg, dest) =
            (* Код, использующий System.Animals.Pigeon API *)
            ()
```

Загрузка сборок

Последним компонентом нашей плагинной системы является реализация загрузки других сборок и любых типов, которые могут потребоваться во время выполнения.

Сборки могут загружаться динамически с помощью метода `Assembly.Load`, который возвращает экземпляр типа `System.Assembly`. Как только у вас появится объект типа `Assembly`, вы сможете с помощью механизма рефлексии изучить типы, заключенные в ней, и/или создавать их экземпляры.

В примере 12.12 определяется метод `loadAsm`, который открывает сборку по имени и затем выводит количество типов в некоторых наиболее часто используемых сборках.

В сборке `FSharp.Core` определено 1317 типов. Это, конечно, очень много, но не забывайте, что вам необязательно знать их все.

Пример 12.12. Загрузка сборок

```
> // Загрузка сборки
open System.Reflection
let loadAsm (name : string) = Assembly.Load(name)

// Вывести информацию о сборках
let printAssemblyInfo name =
    let asm = loadAsm name
    printfn "Assembly %s has %d types" name (asm.GetTypes().Length);;

val loadAsm : string -> Assembly
val printAssemblyInfo : string -> unit

> // Некоторые наиболее часто используемые сборки
[ "System"; "System.Core"; "FSharp.Core"; "System.Windows.Forms" ]
|> List.iter printAssemblyInfo;;

Assembly System has 2034 types
Assembly System.Core has 927 types
```

```
Assembly FSharp.Core has 1317 types
Assembly System.Windows.Forms has 2265 types
val it : unit = ()
```

Загрузка плагинов

Теперь, когда мы научились загружать сборки, можно перейти к заключительному этапу реализации нашего демонстрационного примера плагинной системы. В примере 12.13 определяется метод `loadPlugins`. Он отыскивает в текущем каталоге все файлы с расширением *.dll*, загружает сборки, извлекает из них информацию о типах и оставляет только те типы, которые реализуют интерфейс `IDeliveryMethod`.

В заключение он создает экземпляры плагинов с помощью класса `Activator`.

Пример 12.13. Загрузка расширений

```
// DeliverySystem.Application.dll

open System
open System.IO
open System.Reflection

open DeliverySystem.Core

let loadPlugins() =

    // Возвращает типы, реализующие указанный интерфейс
    let typeImplementsInterface (interfaceTy : Type) (ty : Type) =
        printfn "Checking %s" ty.Name
        match ty.GetInterface(interfaceTy.Name) with
        | null -> false
        | _    -> true

    Directory.GetFiles(Environment.CurrentDirectory, "*.dll")
    // Загрузить все сборки
    |> Array.map (fun file -> Assembly.LoadFile(file))
    |> Array.map(fun asm -> asm.GetTypes())
    |> Array.concat
    // Оставить только типы, реализующие интерфейс IDeliveryMethod
    |> Array.filter (typeImplementsInterface typeof<IDeliveryMethod>)
    // Создать экземпляры каждого расширения
    |> Array.map (fun plugin -> Activator.CreateInstance(plugin))
    // Выполнить приведение к типу IDeliveryMethod
    |> Array.map (fun plugin -> plugin :?> IDeliveryMethod)

[<EntryPoint>]
let main(_) =

    let plugins = loadPlugins()
```

```
plugins |> Array.iter (fun pi -> printfn "Loaded Plugin - %s" pi.MethodName)

(* Отправить посылки! *)

0
```

На первый взгляд, загрузка внешних сборок и создание экземпляров типов, определенных в этих сборках, требует слишком много дополнительной работы, однако не забывайте, что само приложение *ничего не знает* о типах, которые будут загружаться в будущем. Так, благодаря механизму рефлексии вы можете расширять и модифицировать поведение приложения, не изменяя его исходный код.

13

Цитирование

В предыдущей главе мы рассматривали механизм рефлексии в .NET как средство метапрограммирования – анализ статической информации о типах, позволяющий получить представление о коде. Метапрограммирование с использованием механизма рефлексии позволяет выполнять такие операции, как загрузка плагинов, но не дает возможности разобраться в том, *как действует* код. API рефлексии в .NET позволяет получить голые инструкции на языке MSIL, но понять, что делают эти инструкции, – чрезвычайно сложная задача.

Однако существует множество ситуаций, когда бывает полезно знать не только структуру программы, но и как действует ее код: например, чтобы взять функцию, написанную на языке F#, и преобразовать ее в форму, которая может быть выполнена процессором GPU графической карты. (Что позволило бы выполнить ее намного быстрее.)

В языке F# предусмотрен механизм под названием *цитируемые выражения (quotation expressions)*, используя который можно получить доступ не только к статической информации о типах в некотором блоке кода, но и увидеть, как этот код структурирован, то есть внутреннее представление кода компилятором F# (иногда это представление называют *абстрактным синтаксическим деревом*, или AST, Abstract Syntax Tree).

Цитирование, поддерживаемое языком F#, позволяет:

- Выполнять анализ и изучение кода
- Переносить (defer) вычисления на другие платформы (SQL, GPU и другие)
- Генерировать новый код

Мы приступим к изучению этих возможностей чуть позже, но сначала поближе познакомимся с самой особенностью языка. Цитирование –

это уже серьезная магия, не предназначенная для слабых духом. Эта глава представляет собой лишь краткий экскурс по этой теме. Чтобы по-настоящему овладеть цитированием, обращайтесь к ресурсам на сайте центра разработчиков F# *http://fsharp.net*.

Основы цитирования

Говоря простым языком, *цитируемое выражение* – это объект, содержащий представление некоторого кода на языке F#. Чтобы получить цитируемое выражение, достаточно заключить обычное выражение в символы <@ @> или <@@ @@>:

```
> // Простое сложение
<@ 1 + 1 @>;;
val it : Quotations.FSharpExpr<int> =
  Call (None, Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
      [Value (1), Value (1)])
    {CustomAttributes = [NewTuple (Value ("DebugRange"),
         NewTuple (Value ("stdin"), Value (7), Value (3), Value (7), Value (8)))];
     Raw = ...;
     Type = System.Int32;}

> // Лямбда-выражение
<@@ fun x -> "Hello, " + x @@>;;

val it : Quotations.FSharpExpr =
  Lambda (x,
       Call (None,
             System.String op_Addition[String,String,String]
             (System.String, System.String),
             [Value ("Hello, "), x]))
    {CustomAttributes = [NewTuple (Value ("DebugRange"),
         NewTuple (Value ("stdin"), Value (10), Value (4), Value (10),
                   Value (26)))];
     Type = Microsoft.FSharp.Core.FSharpFunc`2[System.String,System.String];}
```

В языке F# цитируемые выражения представлены типом Expr<_>, объявленным в пространстве имен Microsoft.FSharp.Quotations. Параметр обобщенного типа в Expr<_> – это результат выражения. Например, выражение <@ 1 + 1 @> будет представлено как Expr<int>, потому что результатом выражения 1 + 1 является целое число.

Использование скобок <@@ @@> создает нетипизированное цитируемое выражение типа Expr, которое не содержит тип возвращаемого значения. В большинстве случаев совершенно не важно, как осуществляется цитирование кода. Тем не менее обратите внимание, что выражение типа Expr<_> всегда можно преобразовать в выражение типа Expr, обратившись к его свойству Raw.

Вы можете цитировать любые выражения, допустимые в языке F#, единственное исключение составляют объекты выражений. (Которые, как вы помните, приводят к созданию анонимных классов.)

Декомпозиция цитируемых выражений

Получив цитируемое выражение, вы сможете выполнить анализ абстрактного синтаксического дерева с помощью активных шаблонов, имеющихся в стандартной библиотеке F#. То есть, несмотря на сложную древовидную структуру, представленную экземпляром `Expr<_>`, вы можете с помощью сопоставления с образцом связывать новые значения с результатами сопоставления.

В примере 13.1 определяется функция `describeCode`, которая принимает цитируемое выражение и выводит описание кода. Примечательно, что для определения типа выражения в выражении сопоставления не используется динамическая проверка типа. Вместо этого сопоставление производится с результатом того, что представляет собой операция. Благодаря этому вы можете не только проверить, является ли цитируемый код вызовом функции, но даже изменить параметры вызова.

Пример 13.1. Простые цитируемые выражения

```
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

let rec describeCode (expr : Expr) =
    match expr with

    // Литеральное значение
    | Int32(i)  -> printfn "Integer with value %d" i
    | Double(f) -> printfn "Floating-point with value %f" f
    | String(s) -> printfn "String with value %s" s

    // Вызов метода
    | Call(calledOnObject, methInfo, args)
        -> let calledOn = match calledOnObject with
                          | Some(x) -> sprintf "%A" x
                          | None    -> "(Called a static method)"

           printfn "Calling method '%s': \n\
                    On value: %s \n\
                    With args: %A" methInfo.Name calledOn args

    // Lambda-выражение
    | Lambda(var, lambdaBody) ->
        printfn
            "Lambda Expression - Introduced value %s with type %s"
            var.Name var.Type.Name
        printfn "Processing body of Lambda Expression..."
```

```
            describeCode lambdaBody

        | _ -> printfn "Unknown expression form:\n%A" expr
```

Ниже приводится результат сеанса в окне FSI, который показывает использование метода describeCode:

```
> describeCode <@ 27 @>;;
Integer literal with value 27
val it : unit = ()

> describeCode <@ 1.0 + 2.0 @>;;
Calling method 'op_Addition':
On value: (Called a static method)
With args : [Value (1.0); Value (2.0)]
val it : unit = ()
> let localVal = "a string";;

val localVal : string = "a string"

> describeCode <@ localVal.ToUpper() @>;;
Calling method 'ToUpper':
On value: PropertyGet (None, System.String localVal, [])
With args : []
val it : unit = ()

> describeCode <@ fun x y -> (x, y) @>;;
Lambda Expression-Introduced on value 'x' with type 'Object'
Processing body of Lambda Expression...
Lambda Expression-Introduced on value 'y' with type 'Object'
Processing body of Lambda Expression...
Unknown expression form:
NewTuple (x, y)
val it : unit = ()
```

В стандартной библиотеке языка F# имеется полный набор активных шаблонов, что позволяет распознать любую форму цитируемого выражения. Все активные шаблоны находятся в пространстве имен Microsoft.FSharp.Quotations.Patterns.

Прежде чем продолжить, рассмотрим внимательнее активные шаблоны, использованные в примере 13.1.

Литеральные значения

В стандартной библиотеке F# содержатся активные шаблоны для всех простых типов, от int и float до string и других. Сопоставление констант с этими типами дает возможность получать их значения:

```
// Литеральное значение
| Int32(i)  -> printfn "Integer with value %d" i
| Double(f) -> printfn "Floating-point with value %f" f
| String(s) -> printfn "String with value %s" s
```

Вызовы функций

Вызовы функций сопоставляются с активным шаблоном `Call`. При совпадении шаблон вводит три значения: объект, чей метод вызывается (или `None` при вызове статического метода); значение `MethodInfo`, содержащее информацию о методе, полученную механизмом рефлексии; список аргументов, переданных методу.

```
// Вызов метода
| Call(calledOnObject, methInfo, args)
    -> let calledOn = match calledOnObject with
                      | Some(x) -> sprintf "%A" x
                      | None -> "(Called a static method)"

       printfn "Calling method '%s': \n\
                On value: %s \n\
                With args: %A" methInfo.Name calledOn args
```

Однако в случае поиска вызова какой-то конкретной функции активный шаблон `Call` — плохой помощник. По этой причине был создан активный шаблон `SpecificCall`. Его использование в выражении сопоставления позволяет определить, что вызывается конкретная функция, и изменить значения входных параметров. (При этом также происходит привязка конкретных типов к обобщенным параметрам, но на это в большинстве случаев можно не обращать внимание.) То есть активный шаблон `SpecificCall` позволяет получить сопоставление не всех вызовов функций, а только определенных.

В примере 13.2 показано использование активного шаблона `SpecificCall` для описания арифметических операций. В этом примере связываются только значения параметров, а обобщенные параметры типов игнорируются.

Пример 13.2. Активный шаблон SpecificCall

```
let describeArithmatic operation =
    match operation with
    | SpecificCall <@ (+) @> (_, _, [lhs; rhs]) ->
        printfn "Addition."

    | SpecificCall <@ (-) @> (_, _, [lhs; rhs]) ->
        printfn "Subtraction."

    | SpecificCall <@ (*) @> (_, _, [Int32(0); _])
    | SpecificCall <@ (*) @> (_, _, [_; Int32(0)]) ->
        printfn "Multiplication by zero."

    | SpecificCall <@ (*) @> (_, _, [lhs; rhs]) ->
        printfn "Multiplication."

    | SpecificCall <@ (/) @> (_, _, [lhs; Int32(0)]) ->
        printfn "Division by zero."
```

```
| SpecificCall <@ (/) @> (_, _, [lhs; rhs]) ->
    printfn "Division."

| _ -> failwith "Unknown quotation form."
```

Функции значения

Сопоставление значений функции осуществляется с помощью активного шаблона `Lambda`. Этот активный шаблон выполняет связывание двух значений типов `Var` и `Expr`:

```
// Лямбда-выражения
| Lambda(var, lambdaBody) ->
    printfn
        "Lambda Expression - Introduced Value %s of type %s"
        var.Name var.Type.Name
    printfn "Processing body of Lambda Expression..."
    describeCode lambdaBody
```

Значение типа `Expr` — это всего лишь тело лямбда-выражения. `Var` представляет новое значение, созданное лямбда-выражением (его параметр).

При цитировании кода всякий раз при декомпозиции выражения, создающего новое значение, такого как инструкция связывания `let`, вновь созданное значение будет представлено значением `Var`.

Цитирование тела метода

Несмотря на то, что выражение в скобках `<@ @>` возвращает скомпилированное представление кода, этого может оказаться недостаточно, если это выражение вызывают функции, которые в свою очередь могут вызывать другие функции, и так далее. Для получения полного абстрактного синтаксического дерева цитируемого выражения необходимо углубляться в тело методов. По умолчанию компилятор F# не в состоянии вернуть цитированную форму тела функции или члена класса.

Для включения тел функций внутрь цитируемых выражений необходимо применить атрибут `[<ReflectedDefinition>]`. Методы, помеченные этим атрибутом, могут «разворачиваться» при обработке цитируемых выражений с помощью активного шаблона `(|MethodWithReflectedDefinition|_|)`.

В примере 13.3 приводится измененная версия функции `describeCode`, в которую добавлена обработка методов, помеченных атрибутом `[<ReflectedDefinition>]`. Функция проверяет объект `MethodInfo`, возвращаемый активным шаблоном `(|Call|_|)`, и определяет, доступно ли тело метода. Если тело вызываемого метода доступно, то оно обрабатывается.

Пример 13.3. Использование атрибута ReflectedDefinition

```
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns
```

```
let rec describeCode2 (expr : Expr) =
    match expr with

    // Литеральное значение
    | Int32(i) -> printfn "Integer literal with value %d" i
    | Double(f) -> printfn "Floating point literal with value %f" f
    | String(s) -> printfn "String literal with value %s" s

    // Вызов метода
    | Call(calledOnObject, methInfo, args)
        -> let calledOn = match calledOnObject with
                          | Some(x) -> sprintf "%A" x
                          | None -> "(static method)"

           printfn "Calling method '%s': \n\
                    On intance: %s \n\
                    With args : %A" methInfo.Name calledOn args

           match methInfo with
           | MethodWithReflectedDefinition(methBody) ->
                printfn
                    "Expanding method body of '%s'..." methInfo.Name
                describeCode2 methBody
           | _ ->
                printfn
                    "Unable to expand body of '%s'. Quotation stops here."
                    methInfo.Name

    // Лямбда-выражения
    | Lambda(var, lambdaBody) ->
        printfn
            "Lambda Expression on value '%s' with type '%s'"
            var.Name var.Type.Name

        printfn "Processing body of Lambda Expression..."
        describeCode2 lambdaBody

    | _ -> printfn "Unknown expression form:\n%A" expr
```

Ниже приводятся результаты в окне FSI, которые показывают обработку двух цитируемых выражений. Функция `describeCode2` **сумела обработать тело функции** `invertNumberReflected`**, потому что она была помечена атрибутом** [<ReflectedDefinition>]**:**

```
> // Определение функций с атрибутом и без атрибута ReflectedDefinition
let invertNumber x = -1 * x

[<ReflectedDefinition>]
let invertNumberReflected x = -1 * x;;

val invertNumber : int -> int
val invertNumberReflected : int -> int
```

```
> // Описание программного кода без атрибута ReflectedDefinition
describeCode2 <@ invertNumber 10 @>
;;
Calling method 'invertNumber':
On intance: (static method)
With args : [Value (10)]
Unable to expand body of 'invertNumber'. Quotation stops here.
val it : unit = ()

> // Описание программного кода с атрибутом ReflectedDefinition
describeCode2 <@ invertNumberReflected 10 @>
;;
Calling method 'invertNumberReflected':
On intance: (static method)
With args : [Value (10)]
Expanding method body of 'invertNumberReflected'...
Lambda Expression on value 'x' with type 'Int32'
Processing body of Lambda Expression...
Calling method 'op_Multiply':
On intance: (static method)
With args : [Value (-1); x]
Unable to expand body of 'op_Multiply'. Quotation stops here.
val it : unit = ()
```

Декомпозиция произвольного кода

Взяв на вооружение лишь несколько активных шаблонов, мы смогли в методе `describeCode` проанализировать множество различных программных конструкций. Однако всякий раз, когда этому методу не удается распознать какой-либо код, выражение сопоставления терпит неудачу и генерирует исключение.

В большинстве случаев активные шаблоны используются для декомпозиции абстрактного синтаксического дерева, анализируя только интересующий код и игнорируя все остальное. Однако порой бывает важно выполнить обход всего дерева. Прерывание обхода дерева в том или ином узле – это неправильная стратегия, потому что при этом игнорируется значительная часть дерева, которая может включать код, представляющий для вас интерес.

Однако с другой стороны не имеет смысла задействовать все имеющиеся активные шаблоны, от `ForIntegerRangeLoop` (циклы `for`) и `IfThenElse` (выражения `if`) до `LetRecursive` (рекурсивные инструкции связывания `let`) и других.

К счастью, существует активный шаблон «на все случаи жизни», который соответствует большинству возможных выражений F#. Этот активный шаблон определен в модуле `ExprShape` и преобразует любые цитируемые выражения F# в экземпляры `ShapeVar`, `ShapeLambda` или `ShapeCombination`.

В примере 13.4 реализована новая функция generalizedDescribeCode, выполняющая декомпозицию любого цитируемого выражения. Цитируемое выражение совпадает с ShapeVar, если встречено значение, и с ShapeLambda, если встречено определение функции. Большинство цитируемых выражений F# соответствуют третьему варианту активного шаблона, ShapeCombination. Вариант ShapeCombination не предназначен для получения описания кода – он просто возвращает список подвыражений, из которых сконструирован код. (То есть вы не сможете определить представление этого кода – вы получите лишь список подвыражений, составляющих его.)

Пример 13.4. Обобщенный обход цитируемого выражения

```
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.ExprShape

let rec generalizedDescribeCode indentation expr =

    let indentedMore = indentation + " "

    match expr with
    // Переменная
    | ShapeVar(var) ->
        printfn "%s Looking up value '%s'" indentation var.Name

    // Новое лямбда-выражение
    | ShapeLambda(var, lambdaBody) ->
        printfn
            "%s Lambda expression, introducing var '%s'"
            indentation var.Name
        generalizedDecribeCode indentedMore lambdaBody

    // Прочие конструкции
    | ShapeCombination(_, exprs) ->
        printfn "%s ShapeCombination:" indentation
        exprs |> List.iter (generalizedDecribeCode indentedMore)
```

Следующий сеанс в окне FSI показывает использование метода generalizedDescribeCode:

```
> generalizedDescribeCode "" <@ (fun x y z -> x + y + z) @>;;
Lambda expression, introducing var 'x'
    Lambda expression, introducing var 'y'
        Lambda expression, introducing var 'z'
            ShapeCombination:
                ShapeCombination:
                    Looking up value 'x'
                    Looking up value 'y'
                Looking up value 'z'
val it : unit = () >
```

Теперь вы можете добавить в выражение сопоставления новые правила, показанные выше, чтобы «перехватывать» совпадения с конструкциями, представляющие для вас особый интерес, как это было сделано в предыдущих версиях метода `describeCode`.

Практическое применение: перенос вычислений на другие платформы

Итак, вы узнали, как выполнять декомпозицию цитируемых выражений F# для анализа сгенерированного компилятором кода, но что дальше делать с полученными результатами? Самое, пожалуй, интересное применение цитируемых выражений в языке F# – это перенос вычислений на другие платформы. Обрабатывая цитируемые выражения, вы можете передать полученный код для выполнения другой платформе, например процессору GPU графической карты или серверу баз данных SQL. Выполняя обход абстрактного синтаксического дерева, вы можете преобразовывать код F# в формат, поддерживаемый другой вычислительной средой. Этот прием позволяет избежать использования второго языка программирования и задействовать уже имеющийся комплект инструментов F# (IntelliSense, проверку типов и так далее).

Например, вместо того чтобы писать инструкции для графического процессора на языке ассемблера, оформляя их в виде строк, вы можете писать инструкции на языке F# в виде цитируемого выражения и затем преобразовывать их в соответствующие инструкции языка ассемблера.

В качестве простого примера мы с помощью цитируемых выражений реализуем преобразование арифметических выражений в *обратную польскую нотацию*, или ОПН.

ОПН – это форма представления арифметических выражений без использования круглых скобок. Она предполагает использование гипотетического стека чисел, а арифметические операции, такие как сложение, снимают со стека два операнда и помещают результат обратно в стек.

Например, выражение `2 * 3 + 4` в ОПН было бы представлено операциями, перечисленными в табл. 13.1.

*Таблица 13.1. Вычисление выражения 2 * 3 + 4 в ОПН*

Операция	Стек
Push 2	2
Push 3	3, 2
Call (*)	6
Push 4	4, 6
Call (+)	10

Преобразование функций в ОПН может оказаться достаточно сложным делом и чревато ошибками. Поэтому мы будем использовать цитируемые выражения для создания дерева выражений с сохранением порядка операций и затем преобразуем полученный код в ОПН.

В примере 13.5 определяется функция с именем `fsharpToRPN`, которая принимает цитируемое выражение и анализирует его, конструируя список операций в ОПН.

Эта функция находит арифметические операции и рекурсивно генерирует список операций ОПН для вычисления выражений слева и справа от оператора. Затем она просто добавляет соответствующий оператор в список операций ОПН в нужном порядке.

Пример 13.5. Преобразование выражений на языке F# в ОПН

```
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

let rec fsharpToRPN code stackOperations =
    match code with

    | Int32(n)  -> (sprintf "Push %d" n) :: stackOperations
    | Double(f) -> (sprintf "Push %f" f) :: stackOperations

    | SpecificCall <@ (+) @> (_, _, [lhs; rhs]) ->
        let lhs = fsharpToRPN lhs stackOperations
        let rhs = fsharpToRPN rhs stackOperations
        lhs @ rhs @ ["Call (+)"]

    | SpecificCall <@ (-) @> (_, _, [lhs; rhs]) ->
        let lhs = fsharpToRPN lhs stackOperations
        let rhs = fsharpToRPN rhs stackOperations
        lhs @ rhs @ ["Call (-)"]

    | SpecificCall <@ (*) @> (_, _, [lhs; rhs]) ->
        let lhs = fsharpToRPN lhs stackOperations
        let rhs = fsharpToRPN rhs stackOperations
        lhs @ rhs @ ["Call (*)"]

    | SpecificCall <@ (/) @> (_, _, [lhs; rhs]) ->
        let lhs = fsharpToRPN lhs stackOperations
        let rhs = fsharpToRPN rhs stackOperations
        lhs @ rhs @ ["Call (/)"]

    | expr -> failwithf "Unknown Expr:\n%A" expr
```

Ниже показано использование функции `fsharpToRPN`:

```
> fsharpToRPN <@ 1 + 2 @> [];;
val it : string list = ["Push 1"; "Push 2"; "Call (+)"]
> fsharpToRPN <@ 2 * 3 + 4 @> [];;
```

```
val it : string list = ["Push 2"; "Push 3"; "Call (*)"; "Push 4"; "Call (+)"]
> // Более сложное выражение
fsharpToRPN <@ (2 + 10) / (3 * (2 - 6 / 7 - 2)) @> []
|> List.iter (printfn "%s");;
Push 2
Push 10
Call (+)
Push 3
Push 2
Push 6
Push 7
Call (/)
Call (-)
Push 2
Call (-)
Call (*)
Call (/)
```

Создание цитируемых выражений

Возможность анализа цитируемых выражений позволяет не только исследовать абстрактное синтаксическое дерево и преобразовывать его, но и создавать цитируемые выражения динамически во время выполнения.

Создание цитируемых выражений особенно полезно для создания миниязыков программирования. Вместо того чтобы писать код на языке F# и обрабатывать абстрактное синтаксическое дерево, можно реализовать синтаксический анализ инструкций на другом, возможно, более простом языке, чтобы сгенерировать дерево выражения и затем его обработать.

Все активные шаблоны, применяемые для декомпозиции цитируемых выражений, имеют дополнительный статический метод класса `Expr`, который создает эквивалентное значение `Expr<_>`.

Ниже показано простое цитируемое выражение и пример создания эквивалентного ему выражения:

```
let organicQuotation =
    <@
        let x = (1, 2, 3)
        (x, x)
    @>

let syntheticQuotation =
    Expr.Let(
        new Var("x", typeof<int * int * int>),
        Expr.NewTuple( [ Expr.Value(1); Expr.Value(2); Expr.Value(3) ] ),
        Expr.NewTuple( [ Expr.GlobalVar("x").Raw; Expr.GlobalVar("x").Raw ] )
    )
```

Подстановка выражений

Методы типа Expr – это не единственный способ создания цитируемых выражений. Цитируемые выражения могут содержать *поля подстановки (expression holes)*, которые определяют местоположение для новых экземпляров Expr<_>, что позволяет создавать деревья выражений, просто вставляя в уже имеющиеся выражения.

Чтобы объявить поле подстановки, достаточно добавить знак процента % перед любым выражением внутри цитируемого выражения. В результате это выражение будет включено в дерево созданного выражения. (Аналогично в цитируемых выражениях вида <@@ @@> можно использовать поле подстановки вида %%, которое будет замещено экземпляром Expr вместо Expr<_>.)

В примере 13.6 определяется функция с именем addTwoQuotations, которая принимает два цитируемых выражения и создает новое выражение, соединяя два экземпляра Expr<_>. То есть если передать функции addTwoQuotations два целочисленных литерала в виде <@ 1 @> и <@ 2 @>, это будет равносильно цитируемому выражению <@ 1 + 2 @>. Кроме того, созданное выражение будет совершенно идентично его статической версии.

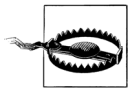
Пример 13.6. Подстановка цитируемых выражений

```
> let addTwoQuotations x y = <@ %x + %y @>;;

val addTwoQuotations : Expr<int> -> Expr<int> -> Expr<int>

> // Создать цитируемое выражение и получить его описание
addTwoQuotations <@ 1 @> <@ 2 @>
|> describeCode;;
Calling method 'op_Addition':
On value: (Called a static method)
With args: [Value (1); Value (2)]
val it : unit = ()

> // Создать более сложное цитируемое выражение
// и получить его описание
addTwoQuotations <@ "a string".Length @> <@ (2 * 2) @>
|> describeCode;;
Calling method 'op_Addition':
On value: (Called a static method)
With args: [PropGet (Some (Value ("a string")), Int32 Length, []);
 Call (None, Int32 op_Multiply[Int32,Int32,Int32](Int32, Int32),
      [Value (2), Value (2)])]
val it : unit = ()
```

Выполнение цитируемых выражений

Существует возможность не только анализировать цитируемые выражения, но и выполнять их. Пакет F# PowerPack содержит эксперименталь-

ную библиотеку, позволяющую преобразовывать цитируемые выражения F# в деревья выражений LINQ (Language Integrated Query – язык интегрированных запросов). Деревья выражений LINQ очень схожи с цитируемыми выражениями F# – и то и другое является представлением абстрактного синтаксического дерева. Цитируемые выражения F# могут отражать большее количество форм кода, чем деревья выражений, однако главное преимущество деревьев выражений состоит в том, что они могут быть скомпилированы и выполнены.



Пакет F# PowerPack не входит в состав Visual Studio 2010; его необходимо загружать отдельно с домашней страницы проекта F# PowerPack на сайте CodePlex *www.codeplex.com*. За дополнительной информацией о PowerPack обращайтесь к приложению А.

Чтобы выполнить или скомпилировать цитируемое выражение, необходимо загрузить и добавить ссылку на сборки `FSharp.PowerPack.dll` и `FSharp.PowerPack.Linq.dll` и открыть пространство имен `Microsoft.FSharp.Linq.QuoationEvaluation`. Это добавит методы расширения `Eval` и `Compile` к классам `Expr` и `Expr<_>`, позволяющие компилировать и выполнять цитируемые выражения.

Ниже показан простой пример компиляции и выполнения цитируемых выражений:

```
> // Добавить ссылки на необходимые библиотеки
#r "System.Core.dll"
#r "FSharp.PowerPack.dll"
#r "FSharp.PowerPack.Linq.dll"

// Добавить методы расширения к типу Expr<_>
open Microsoft.FSharp.Linq.QuotationEvaluation;;

--> Referenced 'C:\Program Files\Reference
Assemblies\Microsoft\Framework\v3.5\System.Core.dll'

--> Referenced 'C:\Program Files\MicrosoftF#\PowerPack\FSharp.PowerPack.dll'

--> Referenced 'C:\Program Files\MicrosoftF#\PowerPack\FSharp.PowerPack.Linq.dll'

> // Выполнить простое выражение
let x = <@ 1 + 2 * 3 @>

val x : Expr<int> =
  Call (None, Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
      [Value (1),
       Call (None, Int32 op_Multiply[Int32,Int32,Int32](Int32, Int32),
             [Value (2), Value (3)])])

> x.Eval();;
val it : int = 7
```

```
> // Скомпилировать функцию
let toUpperQuotation = <@ (fun (x : string) -> x.ToUpper()) @>
let toUpperFunc = toUpperQuotation.Compile() ();;

val toUpperQuotation : Expr<(string -> string)> =
  Lambda (x, Call (Some (x), System.String ToUpper(), []))
val toUpperFunc : (string -> string)

> toUpperFunc "don't panic";;
val it : string = "DON'T PANIC"
```

Практическое применение: создание производных

Возможность компилировать или выполнять цитируемые выражения добавляет новую важную особенность вашим приложениям на языке F#, а именно возможность генерировать новый код. В примере 13.7 анализируется обычное цитируемое выражение и затем генерируется новый экземпляр `Expr<_>`, содержащий производную от функции.

Производная от функции `f(x)` — это скорость изменения значения функции `f` в точке `x`. Если функция `f` возрастает при переходе от `f(x)` к `f(x+0.0001)`, производная будет иметь положительное значение. Аналогично если в точке `x` функция `f(x)` убывает, производная будет иметь отрицательное значение. Производные могут показаться сложной темой, однако они вычисляются очень просто, если следовать нескольким несложным правилам.

В примере 13.7 определяется функция `generateDerivative`, которая, как и пример 13.5, опирается на сопоставление с базовыми арифметическими функциями. Когда встречается арифметическая операция, находится правило соответствующей производной, так чтобы результирующее выражение `Expr` представляло производную для этой операции. Здесь вам нужно использовать пакет F# PowerPack для компиляции и выполнения сгенерированной функции.

Пример 13.7. Использование цитируемых выражений для создания производных функций

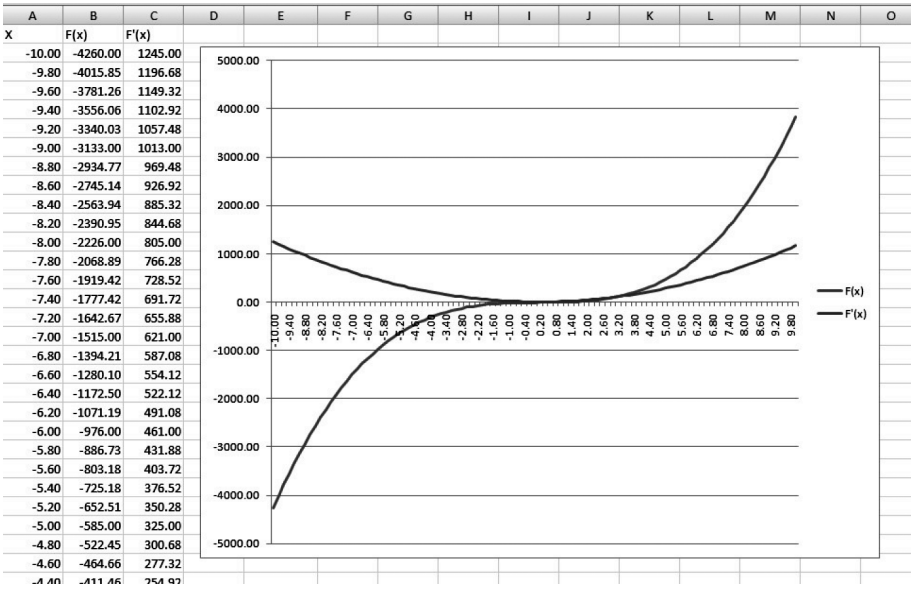
```
#r "FSharp.PowerPack.dll"
#r "FSharp.PowerPack.Linq.dll"

// Добавить методы расширения Expr<_>
open Microsoft.FSharp.Linq.QuotationEvaluation

open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

// Получить объект 'MethodInfo', соответствующий арифметической функции.
// Следующие методы будут использоваться для генерации нового экземпляра
// Expr<_>.
```

```
type Operations =
    static member Add (x, y) : float = x + y
    static member Sub (x, y) : float = x - y
    static member Mul (x, y) : float = x * y
    static member Div (x, y) : float = x / y

let addMi = (typeof<Operations>).GetMethod("Add")
let subMi = (typeof<Operations>).GetMethod("Sub")
let mulMi = (typeof<Operations>).GetMethod("Mul")
let divMi = (typeof<Operations>).GetMethod("Div")

let rec generateDerivative (equation : Expr) =

    match equation with

    // Лямбда-выражение – начало функции
    | Lambda(arg, body) ->
        Expr.Lambda(arg, generateDerivative body)

    // Вызов метода – начало функции
    | Call(None, MethodWithReflectedDefinition(methBody), [ arg ]) ->
        generateDerivative methBody

    // Метод чтения свойства – для свойств модуля
    | PropertyGet(None, PropertyGetterWithReflectedDefinition(body), []) ->
        generateDerivative body

    // Сложение
    // [d/dx] f(x) + g(x) = f'(x) + g'(x)
    | SpecificCall <@ (+) @> (_, _, [f; g]) ->
        let f' = generateDerivative f
        let g' = generateDerivative g
        Expr.Call(addMi, [f'; g'])

    // Вычитание
    // [d/dx] f(x) - g(x) = f'(x) - g'(x)
    | SpecificCall <@ (-) @> (_, _, [f; g]) ->
        let f' = generateDerivative f
        let g' = generateDerivative g
        Expr.Call(subMi, [f'; g'])

    // Умножение
    // [d/dx] f(x) * g(x) = (f'(x) * g(x)) + (f(x) * g'(x))
    | SpecificCall <@ (*) @> (_, _, [f; g]) ->
        let f' = generateDerivative f
        let g' = generateDerivative g
        Expr.Call(addMi,
            [ Expr.Call(mulMi, [f'; g]);
              Expr.Call(mulMi, [f; g']) ]
        )
```

```
    // Деление
    // [d/dx] f(x) / g(x) = ((f '(x) * g(x)) - (f(x) * g'(x))) / (g^2(x))
    | SpecificCall <@ (/) @> (_, _, [f; g]) ->
        let f' = generateDerivative f
        let g' = generateDerivative g

        let numerator =
            Expr.Call(subMi,
                [ Expr.Call(mulMi, [f'; g])
                  Expr.Call(mulMi, [f; g'])]
            )
        let denominator = Expr.Call(mulMi, [g; g])

        Expr.Call(divMi, [numerator; denominator])

    // Значение
    // [d/dx] x = 1
    | Var(x) ->
        Expr.Value(1.0, typeof<double>)

    // Константа
    // [d/dx] C = 0.0
    | Double(_) ->
        Expr.Value(0.0, typeof<double>)

    | _ -> failwithf "Unrecognized Expr form: %A" equation

let f = (fun x -> 1.5*x*x*x + 3.0*x*x + -80.0*x + 5.0)

let f' =
    let quote : Expr =
       generateDerivative <@ (fun x -> 1.5*x*x*x + 3.0*x*x + -80.0*x + 5.0) @>

    let typedQuote : Expr<float -> float> = Expr.Cast quote

    // Скомпилировать Expr<_> в фактический метод
    let compiledDerivative = typedQuote.Compile()
    compiledDerivative()
```

При наличии функции `f` и ее производной `f'` мы быстро можем построить графики их значений, как показано на рис. 13.1:

```
open System.IO

let generatePlot() =
    use csvFile = new StreamWriter("Plot.csv")
    csvFile.WriteLine("x, f(x), f'(x)")

    [-10.0 .. 0.1 .. 10.0]
    |> List.iter (fun x ->
                    csvFile.WriteLine(sprintf "%f, %f, %f" x (f x) (f' x)))

    csvFile.Close()
```



X	F(x)	F'(x)
-10.00	-4260.00	1245.00
-9.80	-4015.85	1196.68
-9.60	-3781.26	1149.32
-9.40	-3556.06	1102.92
-9.20	-3340.03	1057.48
-9.00	-3133.00	1013.00
-8.80	-2934.77	969.48
-8.60	-2745.14	926.92
-8.40	-2563.94	885.32
-8.20	-2390.95	844.68
-8.00	-2226.00	805.00
-7.80	-2068.89	766.28
-7.60	-1919.42	728.52
-7.40	-1777.42	691.72
-7.20	-1642.67	655.88
-7.00	-1515.00	621.00
-6.80	-1394.21	587.08
-6.60	-1280.10	554.12
-6.40	-1172.50	522.12
-6.20	-1071.19	491.08
-6.00	-976.00	461.00
-5.80	-886.73	431.88
-5.60	-803.18	403.72
-5.40	-725.18	376.52
-5.20	-652.51	350.28
-5.00	-585.00	325.00
-4.80	-522.45	300.68
-4.60	-464.66	277.32
-4.40	-411.46	254.92

Рис. 13.1. *График функции и ее производной*

Цитируемые выражения позволяют писать такие приложения, для реализации которых иным способом потребовалось бы разрабатывать собственный синтаксический анализатор, компилятор или и то и другое. Использование представления кода, созданного компилятором F#, позволяет сконцентрироваться на преобразовании абстрактного синтаксического дерева, вместо того чтобы выполнять синтаксический анализ инструкций на другом языке программирования и самостоятельно создавать их внутреннее представление.

За дополнительной информацией о цитируемых выражениях обращайтесь на сайт центра разработчиков F# *http://fsharp.net*.

A

Обзор библиотек .NET

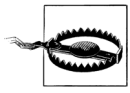
Экосистема .NET имеет невероятную широту – она позволяет выполнять код .NET на разных платформах, таких как Интернет, благодаря технологии Silverlight, или на мобильных устройствах, благодаря Compact Framework. Но она также имеет и невероятную глубину, благодаря богатому набору разнообразных библиотек – от визуализации до сетевого взаимодействия, работы с базами данных и других.

В этом приложении приводится краткий обзор некоторых существующих библиотек .NET, которые позволят вам перейти от простых примеров, приведенных в этой книге, к созданию приложений, предназначенных для решения реальных задач. Библиотеки, рассматриваемые здесь, будут разделены на три основные категории: визуализация, обработка данных и хранение данных. Это приложение мы закончим кратким знакомством с библиотекой языка F#, включая пакет F# PowerPack.

Визуализация

Язык F# – это отличный инструмент обработки данных, но при визуализации этих данных не следует ограничиваться командной строкой.

Для .NET существуют две основные библиотеки визуализации: Windows Forms (WinForms) и Windows Presentation Foundation (WPF). WinForms – это более старая библиотека; она представляет собой объектно-ориентированную обертку вокруг Windows API. Используя WinForms, можно легко создать функциональный пользовательский интерфейс с кнопками и другими типовыми элементами управления, но создать богатый и динамичный интерфейс с ее помощью будет достаточно сложно. Библиотека WPF, с другой стороны, обладает более широкими возможностями и позволяет создавать сложные пользовательские интерфейсы, но она сложнее и труднее в изучении.



Язык F# не поддерживает генерацию кода, поэтому вы не сможете воспользоваться визуальным редактором Visual Studio. Все примеры, которые вы увидите в этом приложении, создают пользовательский интерфейс программно. Однако можно порекомендовать писать части приложения, отвечающие за создание пользовательского интерфейса, на языке C# или VB.NET, чтобы воспользоваться преимуществами их разнообразных инструментальных средств.

Windows Forms

Библиотека WinForms концептуально очень проста. Каждое окно на экране – это экземпляр класса `Form`, содержащий коллекцию элементов управления пользовательского интерфейса типа `Control`. В процессе взаимодействия пользователей с элементами формы происходят события, которые могут обрабатываться программой. Например, элемент управления `Button` может генерировать событие `Click`, если на нем щелкнуть кнопкой мыши:

```
#r "System.Windows.Forms.dll"
#r "System.Drawing.dll"

open System.Windows.Forms

// Создать форму
let form = new Form()

// Создать кнопку
let btn = new Button(Text = "Click Me")
btn.Click.AddHandler(fun _ _ ->
    MessageBox.Show("Hello, World")
    |> ignore)

// Добавить кнопку на форму
form.Controls.Add(btn)

// Отобразить форму
form.ShowDialog()
```

В примере A.1 создается форма, которая содержит индикатор прогресса выполнения, отражающий процесс подсчета слов в полном собрании сочинений Шекспира. Форма содержит два элемента управления, `ProgressBar` и `Label` (элемент управления для отображения текста). Эти два элемента управления постоянно будут обновляться, отражая ход выполнения, благодаря тому, что файлы обрабатываются асинхронно.

Обратите внимание на наличие ссылок на сборки `System.Windows.Forms.dll` и `System.Drawing.dll` – они необходимы для работы с библиотекой WinForms.

Пример А.1. Использование библиотеки Windows Forms

```
#r "System.Windows.Forms.dll"
#r "System.Drawing.dll"

open System.IO
open System.Windows.Forms

// Подсчет количества слов в указанном текстовом файле.
let countWords filePath =
    System.Threading.Thread.Sleep(2000)
    let lines = File.ReadAllText(filePath)
    let words = lines.Split([| ' ' |])
    words.Length

// Полное собрание сочинений Шекспира
let filesToProcess = Directory.GetFiles(@"D:\CompleteWorksOfShakespeare\")

// Создание формы
let form = new Form(Text = "The Words of Shakespeare", TopMost=true, Height=130)

let wordCountText = new Label(Dock = DockStyle.Bottom)
let progress = new ProgressBar(Minimum = 0,
                               Maximum = filesToProcess.Length - 1,
                               Dock = DockStyle.Fill)

form.Controls.Add(progress)
form.Controls.Add(wordCountText)
form.Show()

// Начать асинхронную обработку файлов. По окончании обработки очередного
// файла обновить индикатор и метку.
async {

    let totalWords = ref 0

    for i in 0 .. filesToProcess.Length - 1 do
        totalWords := !totalWords + (countWords filesToProcess.[i])

    // Обновить индикатор и текст метки
    progress.Value <- i
    wordCountText.Text <- sprintf "%d words counted so far..." (!totalWords)

} |> Async.Start
```

Если запустить код из примера А.1, на экране появится форма, изображенная на рис. А.1.

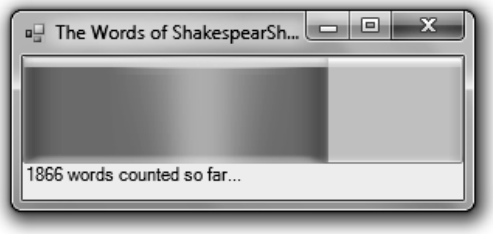


Рис. А.1. *Библиотека Windows Forms в действии*

Windows Presentation Foundation

Библиотека WinForms предоставляет простой API, позволяющий легко создавать функциональный пользовательский интерфейс, но, чтобы создать более современный пользовательский интерфейс «в стиле последних веяний», вам потребуется использовать иной подход. В состав .NET 3.0 была добавлена библиотека Windows Presentation Foundation, предоставляющая иной способ создания пользовательских интерфейсов.

Библиотека WPF построена на основе следующих принципов:

Разнообразие средств отображения информации

Она упрощает добавление в приложение проигрывания видеороликов и воспроизведения анимационных эффектов, а также других мультимедийных возможностей. Приложения, построенные на базе библиотеки WinForms, обычно используют достаточно ограниченный круг элементов управления, предоставляемых операционной системой, тогда как отличительными признаками приложения на базе WPF являются плавные закругления углов, полупрозрачность и четкость текста. Кроме того, приложения на базе WPF используют возможности аппаратного ускорения везде, где только возможно.

Декларативная модель программирования

Еще одна проблема приложений, построенных на базе библиотеки WinForms, заключается в неразрывной связи пользовательского интерфейса с кодом. В приложениях на базе WinForms просто невозможно изменить пользовательский интерфейс, не переписав значительную часть кода. Библиотека WPF позволяет отделить пользовательский интерфейс от кода, что упрощает дальнейшее развитие приложений.

Для создания простого приложения на базе WPF требуется приложить усилий ничуть не больше, чем при использовании библиотеки WinForms. В примере А.2 приводится реализация простого приложения «Hello, World» с использованием библиотеки WPF.

Пример А.2. Приложение «Hello, World» на основе библиотеки WPF

```
#r "WindowsBase.dll"
#r "PresentationCore.dll"
#r "PresentationFramework.dll"

open System
open System.Windows
open System.Windows.Controls

let win = new Window(Height = 128.0, Width = 360.0)

let label = new Label()
label.FontSize <- 62.0
label.Content <- "Hello, World"

win.Content <- label

let app = new Application()
app.Run(win)
```

На рис. А.2 приводится внешний вид нашего приложения «Hello, World». Обратите внимание, как гладко выглядит текст. Это обусловлено тем, что библиотека WPF использует механизмы сглаживания, то есть изображения, создаваемые с помощью WPF, могут увеличиваться в масштабе без появления эффекта пикселизации, что улучшает восприятие визуальной информации.

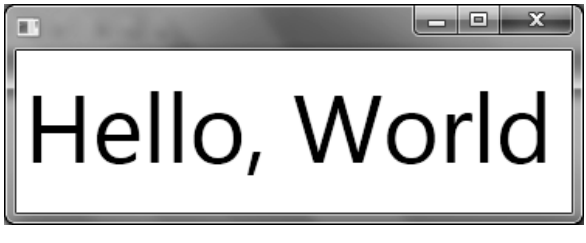


Рис. А.2. *Приложение «Hello, World» на основе библиотеки WPF*

XAML, расширяемый язык разметки приложений

Библиотека WPF позволяет создавать пользовательские интерфейсы программным путем, однако ключевой особенностью разработки приложений на базе WPF является возможность использования языка XAML (произносится как «зам-ель»), или eXtensible Application Markup Language (расширяемый язык разметки приложений). Язык XAML – это диалект языка XML, позволяющий создавать пользовательские интерфейсы декларативным способом. Он дает возможность гламурным дизайнерам создавать пользовательские интерфейсы, не вмешиваясь

в код, который пишут фанатичные программисты. Кроме возможности, позволяющей дизайнерам добавлять в приложения «модные штрихи», язык XAML решает проблемы, свойственные программированию на базе библиотеки WinForms и обусловленные тесной связью кода и пользовательского интерфейса.

В примере A.3 приводится реализация простого приложения на базе библиотеки WPF, которое принимает данные от пользователя и после щелчка на кнопке выводит текст "Hello, *Имя*". Описание пользовательского интерфейса приложения выполнено на языке XAML. Это описание анализируется во время выполнения с помощью класса `XamlReader`. (Особенности самого языка XAML здесь рассматриваться не будут.)

Пример A.3. Приложение SayHello

```
#r "WindowsBase.dll"
#r "PresentationCore.dll"
#r "PresentationFramework.dll"
#r "System.Xaml.dll" // Only on CLR 4.0

open System
open System.Windows
open System.Windows.Controls
open System.Windows.Markup

/// Описание пользовательского интерфейса на языке XAML
let windowXaml = "
<Window
    xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
    xmlns:sys='clr-namespace:System;assembly=mscorlib'
    xmlns:x='http://schemas.microsoft.com/winfx2006/xaml' >

    <StackPanel>
        <TextBlock>Who do you want to say hello to?</TextBlock>
        <TextBox Name='NameTextBox'> [здесь будет выведено имя] </TextBox>
        <Button  Name='SayHelloButton'>Say Hello!</Button>
    </StackPanel>
</Window>
" // Конец описания на языке XAML

// Загрузить разметку XAML
let getWindow() =
    let xamlObj = XamlReader.Parse(windowXaml)
    xamlObj :?> Window

let win = getWindow()

// Получить экземпляры визуальных элементов управления
// и подключить обработчики событий
let textBox = win.FindName("NameTextBox")   :?> TextBox
```

```
let button = win.FindName("SayHelloButton") :?> Button

button.Click.AddHandler(fun _ _ -> let msg = sprintf "Hello, %s" textBox.Text
                                   MessageBox.Show(msg) |> ignore)

let app = new Application()
app.Run(win)
```

В этом примере нам удалось описать пользовательский интерфейс на языке XAML, но код и интерфейс по-прежнему остаются тесно связанными между собой. Поскольку код получает элементы управления `TextBox` и `Button`, используя их свойство `Name`, разметка должна содержать кнопку и текстовое поле с определенными именами, в противном случае приложение не будет работать. В идеале было бы неплохо иметь возможность писать код, который не зависел бы от типов элементов интерфейса.

Связывание

Создатели библиотеки WPF стремились улучшить ситуацию и обеспечить возможность полного отделения пользовательского интерфейса от кода, управляющего им. Для этого в язык XAML была добавлена поддержка *связывания* (*binding*) объектов с элементами пользовательского интерфейса.

Теперь перепишем наше приложение «Say Hello», но основное внимание уделим коду, а интерфейс добавим позднее.

Основу нашего приложения «Say Hello» будет составлять класс с текстом предварительного сообщения и свойством `Name`, а также класс, реализующий вывод приветствия. В примере А.4 определяется простой класс `Greeter` и класс `GreetUserCommand`, который выводит окно с приветствием. Это весь код, необходимый для создания приложения на базе WPF. Обратите внимание на отсутствие каких-либо ссылок на элементы пользовательского интерфейса.

Пример А.4. Реализация функциональности приложения SayHello

```
// Greeter.fs

namespace GreetersNamespace

open System
open System.Windows
open System.Windows.Input

/// Класс, реализующий вывод приветствия
type GreetUserCommand() =

    let m_canExecuteChangedEvent = new Event<System.EventHandler, EventArgs>()

    interface ICommand with
        member this.CanExecute(param : obj) = true
```

```
        member this.Execute(param : obj) =
            MessageBox.Show(sprintf "Hello, %s" (string param))
            |> ignore

        [<CLIEvent>]
        member this.CanExecuteChanged with get() =
            m_canExecuteChangedEvent.Publish

/// Класс с информацией о пользователе - используется при создании
/// текста приветствия
type Greeter() =

    let m_greetCommand = new GreetUserCommand()
    let mutable m_name = " [name goes here] "

    /// Текст предварительного сообщения
    member this.Introduction = "Hello, what is your name?"

    member this.Name with get () = m_name
                     and set x = m_name <- x

    member this.GreetCommand = m_greetCommand
```

Теперь, когда у нас модель выполнения нашего кода, необходимо определить разметку XAML, которая будет использовать эту модель.

В первую очередь в разметке XAML необходимо указать пространство имен, где располагается наша реализация. В следующем фрагменте определяется новое пространство имен XML `g` со значением `GreetersNamespace` в текущей сборке (с именем `WPFinFSharp`):

```
<Window
    xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
    xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'
    xmlns:g='clr-namespace:GreetersNamespace;assembly=WPFinFSharp'
    Title='Greeter' Height='220' Width='450'>
```

Затем в окно типа `Greeter` мы добавляем ресурс с именем `TheGreeter`. При наличии такого объявления ресурса в разметке XAML библиотека WPF автоматически создаст экземпляр класса `Greeter` и сохранит его в ресурсах. Это позволит позднее связать его с элементами пользовательского интерфейса:

```
<!-- Добавить экземпляр 'Greeter' в ресурсы окна -->
<Window.Resources>
    <g:Greeter x:Key='TheGreeter' />
</Window.Resources>
```

Теперь мы можем ссылаться на этот объект, используя механизм привязки данных библиотеки WPF. Следующий фрагмент разметки XAML выполняет привязку объекта класса `Greeter` с элементом `StackPanel`, а за-

тем добавляет элемент типа TextBlock, который будет отображать содержимое свойства Introduction ресурса TheGreeter:

```
<StackPanel DataContext='{Binding Source={StaticResource TheGreeter}}'>

    <TextBlock Margin='7.5' FontSize='36.0'
        Text='{Binding Introduction}' />
```

Точно так же привяжем элемент TextBox со свойством Name ресурса TheGreeter. После этого текстовое поле будет автоматически обновляться при изменении значения свойства Name:

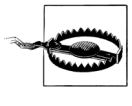
```
<TextBox Margin='7.5' FontSize='36.0'
    Text='{Binding Name}' />
```

Наконец, нам необходимо привязать событие щелчка на кнопке со свойством GreetCommand, значением которого является экземпляр класса GreetUserCommand. При возникновении этого события будет вызван метод Execute, который выведет приветствие пользователя:

```
<Button
    Margin='7.5' FontSize='36.0'
    Content='Say Hello' Command='{Binding GreetCommand}'
    CommandParameter='{Binding Name}' />
```

Введение дополнительного уровня абстракции – ресурса окна – на первый взгляд кажется дополнительным усложнением, но благодаря этому нам удалось ликвидировать явную связь кода с пользовательским интерфейсом.

В примере А.5 показана обновленная разметка XAML интерфейса приложения «Say Hello», в которой используется механизм привязки WPF. В разметку XAML был также включен элемент LinearGradientBrush, чтобы добавить в пользовательский интерфейс яркий красочный штрих.



Обратите внимание, что в данном примере реализация приложения находится в пространстве имен GreetersNamespace в текущей сборке WPFinFSharp. Вам необходимо будет заменить имя сборки именем своего приложения, если оно отличается от указанного, например *Application1*.

Пример А.5. Разметка XAML пользовательского интерфейса для приложения SayHello

```
// Program.fs
open System
open System.Windows
open System.Windows.Input
open System.Windows.Markup

module XamlCode =
    let xamlUI = "
        <Window
```

```
            xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
            xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'
            xmlns:g='clr-namespace:GreetersNamespace;assembly=WPFinFSharp'
            Title='Greeter' Height='240' Width='450'>

            <!-- Добавить экземпляр 'Greeter' в ресурсы окна -->
            <Window.Resources>
                <g:Greeter x:Key='TheGreeter' />
            </Window.Resources>

            <StackPanel DataContext='{Binding Source={StaticResource TheGreeter}}'>

                <!-- Добавить градиентную окраску -->
                <StackPanel.Background>
                    <LinearGradientBrush>
                        <GradientStop Color='Red'    Offset='0.00' />
                        <GradientStop Color='Orange' Offset='0.16' />
                        <GradientStop Color='Yellow' Offset='0.33' />
                        <GradientStop Color='Green'  Offset='0.50' />
                        <GradientStop Color='Blue'   Offset='0.66' />
                        <GradientStop Color='Indigo' Offset='0.83' />
                        <GradientStop Color='Violet' Offset='1.00' />
                    </LinearGradientBrush>
                </StackPanel.Background>

                <!-- Связать свойство Text элемента TextBlock со свойством
                     Introduction объекта Greeter -->
                <TextBlock Margin='7.5' FontSize='36.0'
                           Text='{Binding Introduction}' />

                <!-- Связать элемент TextBox со свойством
                     Name объекта Greeter -->
                <TextBox Margin='7.5' FontSize='36.0'
                         Text='{Binding Name}' />

                <!-- В случае щелчка мышью на кнопке выполнить команду,
                     предусмотренную экземпляром Greeter, передав ей
                     значение свойства Name в виде параметра -->
                <Button
                    Margin='7.5' FontSize='36.0'
                    Content='Say Hello' Command='{Binding GreetCommand}'
                    CommandParameter='{Binding Name}' />
            </StackPanel>
        </Window>
    "
[<EntryPoint; STAThread>]
let main(args) =

    let getWindow() =
        let o = XamlReader.Parse(XamlCode.xamlUI)
        o :?> Window
```

```
let win = getWindow()

let app2 = new Application()
let result = app2.Run(win)

result
```

После компиляции и запуска приложения «Say Hello» оно будет выглядеть, как показано на рис. А.3.

Рис. А.3. *Интерфейс XAML, связанный с данными*

В одном примере невозможно показать всю широту возможностей механизма привязки данных в библиотеке WPF. Тем не менее вы получили представление о том, как этот механизм позволяет отделять реализацию от представления и писать приложения с богатым пользовательским интерфейсом.

О библиотеке WPF можно говорить очень долго. Если вы соберетесь использовать ее в разработке своих приложений, я рекомендую прочитать книгу «WPF Unleashed»[1] Адама Натана (Adam Nathan), выпущенную издательством Sams.

Обработка данных

После того как будет создан привлекательный пользовательский интерфейс, следующим шагом будет реализация *выполнения каких-либо действий*. В .NET имеется множество замечательных библиотек обработки данных, способных удовлетворить любые ваши потребности.

[1] Адам Натан «WPF4. Подробное руководство». – Пер. с англ. – СПб.: Символ-Плюс, выход планируется в III квартале 2011 г.

Регулярные выражения

Работа с текстом – одна из наиболее типичных задач, с которыми вы можете столкнуться. Так как строки в .NET являются неизменяемыми, поиск и их изменение может оказаться достаточно трудным делом. К счастью, .NET обладает полной поддержкой регулярных выражений.

Регулярные выражения (*regular expressions*) – это формальный язык описания шаблонов, применяемых для поиска в тексте определенных слов или последовательностей символов. Вы можете использовать регулярные выражения в .NET для упрощения обработки текстовых данных и реализации таких операций, как:

- Проверка вводимых пользователем данных на соответствие некоторому шаблону
- Нахождение подстрок с определенной структурой
- Создание новых строк, которые являются модифицированными версиями исходных строк

Для начала рассмотрим следующий фрагмент. В нем определяется символьная функция `=~=`, которая принимает строку и регулярное выражение, формат которого мы рассмотрим немного ниже, и возвращает `true`, если строка соответствует указанному регулярному выражению. Регулярное выражение "`\d+`" проверяет наличие цифр в строке:

```
> // Простые регулярные выражения
open System.Text.RegularExpressions

let (=~=) str regex = Regex.IsMatch(str, regex);;

val ( =~= ) : string -> string -> bool

> "Does this contain a number?" =~= "\d+";;
val it : bool = false
> "Does this (42) contain a number?" =~= "\d+";;
val it : bool = true
```

Метасимволы

Регулярные выражения основаны на своем собственном языке метасимволов. В табл. А.1 приводится перечень метасимволов, которые могут использоваться в регулярных выражениях .NET.

Таблица А.1. Метасимволы регулярных выражений

Символ	Описание
Большинство символов	Символы, кроме `. $ { [ ( | ) ] } * + ?`, соответствуют сами себе.
.	Соответствует любому символу, кроме `\n`.

Таблица А.1 (продолжение)

Символ	Описание
`[aeiou0-9]`	Соответствует любому символу из указанной последовательности, включая символы, входящие в диапазон.
`[^aeiou0-9]`	Соответствует любому символу, кроме тех, что указаны.
`\w`	Соответствует любому символу слова `[a-zA-Z_0-9]`.
`\W`	Соответствует любому символу, не являющемуся символом слова `[^a-zA-Z_0-9]`.
`\d`	Соответствует любой десятичной цифре `[0-9]`.
`\D`	Соответствует любому символу, не являющемуся десятичной цифрой `[^0-9]`.
`\s`	Соответствует любому пробельному символу.
`\S`	Соответствует любому символу, не являющемуся пробельным символом.
`\`	Используется для экранирования метасимволов, таких как `. $ { [ ( | ) ] } * + ?`.
`+`	Соответствие предыдущему символу должно быть найдено один или более раз.
`*`	Соответствие предыдущему символу должно быть найдено ноль или более раз.
`?`	Соответствие предыдущему символу должно быть найдено ноль или один раз.

В предыдущем фрагменте было использовано регулярное выражение `\d`, соответствующее цифровым символам, и квантификатор `+`, который требует, чтобы проверяемая строка содержала не менее одного совпадения с регулярным выражением. То есть этому регулярному выражению будут соответствовать строки “1”, “42” и “888”, но не строки “foo” или “bar”.

Регулярные выражения могут с успехом применяться для проверки вводимых пользователем данных. Взгляните на следующее регулярное выражение, с помощью которого можно проверить, является ли введенная строка телефонным номером:

```
“(\(\d\d\d\))?\s*\d\d\d\s*-\s*\d\d\d\d”
```

При всех достоинствах регулярных выражений их синтаксис иногда может казаться совершенно непонятным. Рассмотрим это регулярное выражение по частям.

Во-первых, номер телефона может начинаться с трехсимвольного кода города. Такому коду соответствует регулярное выражение, совпадающее с тремя цифрами, заключенными в круглые скобки. Обратите внимание на необходимость экранирования круглых скобок:

```
\(\d\d\d\)
```

Чтобы показать, что необязательный код города может присутствовать ноль или один раз, соответствующее ему выражение необходимо заключить в круглые скобки и добавить квантификатор ?. Круглые скобки вокруг выражения \(\d\d\d\) заставляют интерпретировать последовательность из пяти символов как единый элемент, к которому применяется квантификатор ?:

```
(\(\d\d\d\))?
```

Далее могут следовать ноль или более пробельных символов \s*, три цифры \d\d\d, дефис с необязательными пробельными символами по обеим сторонам, отделяющий первые три цифры номера от остальных \s*-\s*, и последние четыре цифры номера \d\d\d\d.

В примере A.6 определяется функция isValidPhoneNumber, выполняющая проверку входных данных. Непосредственное сопоставление с регулярным выражением выполняется с помощью статического метода Regex.IsMatch.

Пример A.6. Проверка входных данных с помощью регулярных выражений

```
> // Сопоставление с шаблоном телефонного номера
open System.Text.RegularExpressions

let phoneNumberRegEx = "(\(\d\d\d\))?\s*\d\d\d\s*-\s*\d\d\d\d"
let isValidPhoneNumber text = Regex.IsMatch(text, phoneNumberRegEx);;

val phoneNumberRegEx : string = "(\(\d\d\d\))?\s*\d\d\d\s*-\s*\d\d\d\d"
val isValidPhoneNumber : string -> bool

> isValidPhoneNumber "(707) 827-7000";;
val it : bool = true
> isValidPhoneNumber "123-4567";;
val it : bool = true
> isValidPhoneNumber "123     -     4567";;
val it : bool = true
> isValidPhoneNumber "(this is not a phone number)";;
val it : bool = false
> isValidPhoneNumber "(123) 456-????";;
val it : bool = false
```

Манипулирование строками

Класс Regex обладает методами, помогающими создавать новые строки на основе существующих. Метод Split может использоваться для разбиения строки на части, разделенные фрагментами, совпадающими с регулярным выражением; метод Replace замещает все совпадения с регулярным выражением статической строкой.

С помощью этих двух методов можно реализовать синтаксический анализ частично структурированных текстовых данных.

На практике большое распространение получил формат файлов для сохранения шахматных партий – *Portable Game Notation* (переносимая шахматная нотация), или PGN. Файлы в этом формате начинаются парами ключ/значение, заключенными в квадратные скобки, могут содержать комментарии в фигурных скобках, а каждый ход начинается с его порядкового номера, за которым следует одна или три точки.

Ниже приводится пример файла в формате PGN, описывающего партию между Бобби Фишером и Борисом Спасским:

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia Yugoslavia|JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 {Открытое начало испанской партии, или дебют Рюи
Лопеса.} 3... a6 4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8
10. d4 Nbd7 11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5
17. dxe5 Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5 35.
Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6 Nf2
42. g4 Bd3 43. Re6 1/2-1/2
```

Прежде чем приступить к анализу ходов, необходимо удалить все комментарии и метаданные. Эту задачу можно легко решить с помощью метода `Regex.Replace`. Все комментарии и метаданные в этом формате заключены в скобки { } и [ ] соответственно. Поэтому с помощью двух регулярных выражений, совпадающих с любыми последовательностями символов, заключенными в скобки [ ] или { }, можно заменить комментарии и метаданные пустыми строками, то есть фактически удалить их:

```
// Удаление комментариев и метаданных из файла в формате PGN
let removeMarkup text =
    let tagPairs = new Regex(@"\[(^\])+\]")
    let noTagPairs = tagPairs.Replace(text, "")

    let comments = new Regex(@"\{(^\})+\}")
    let noComments = comments.Replace(noTagPairs, "")

    // Удалить все ведущие пробелы и преобразовать в одну строку
    noComments.Trim().Replace("\r", "").Replace("\n", " ")
```

Следующий шаг состоит в том, чтобы в блоке с описаниями ходов отделить ходы от их порядковых номеров. Это можно сделать с помощью метода `Regex.Split` и регулярного выражения, совпадающего с последовательностями цифр, за которыми следуют символы точки:

```
// Получить список ходов, каждый из которых начинается с порядкового номера,
// за которым следует одна или три точки
let getMoves text =
    let movePrefix = new Regex(@"\d+\.+")
    movePrefix.Split(text)
```

Использование методов `removeMarkup` **и** `getMoves` **позволит нам получить доступ к основной информации в файле формата PGN. Безусловно, здесь требуется написать более специфичный код, точнее учитывающий особенности формата, тем не менее в примере A.7 показано, как с помощью регулярных выражений можно быстро организовать преобразование текстовых данных в более простое для обработки представление.**

Пример A.7. Парсер файла PGN

```
> // Парсинг файлов в формате PGN
open System.Text.RegularExpressions

let pgnGameText = "
[Event \"F/S Return Match\"]
[Site \"Belgrade, Serbia Yugoslavia|JUG\"]
[Date \"1992.11.04\"]
[Round \"29\"]
[White \"Fischer, Robert J.\"]
[Black \"Spassky, Boris V.\"]
[Result \"1/2-1/2\"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 { Открытое начало испанской партии, или дебют Рюи
Лопеса.} 3... a6 4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8
10. d4 Nbd7 11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5
17. dxe5 Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5 35.
Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6 Nf2
42. g4 Bd3 43. Re6 1/2-1/2
"

// Удаляет комментарии и метаданные из файла PGN
let removeMarkup text =
    let tagPairs = new Regex(@"\[.*\]")
    let noTagPairs = tagPairs.Replace(text, "")

    let comments = new Regex(@"\{.*\}")
    let noComments = comments.Replace(noTagPairs, "")

    // Удалить все ведущие пробелы и преобразовать в одну строку
    noComments.Trim().Replace("\r", "").Replace("\n", " ")

// Возвращает список ходов, каждый из которых в исходном файле начинается
// с порядкового номера, за которым следует одна или три точки
let getMoves text =
```

```
    let factRegex = new Regex(@"\d+\.+", RegexOptions.Multiline)
    factRegex.Split(text)

let normalizedText = removeMarkup pgnGameText

let listMoves() =
    getMoves normalizedText
    |> Array.map (fun move -> move.Trim())
    |> Array.iter (printfn "%s");;

val pgnGameText : string =
  "
[Event "F/S Return Match"]
[Site "Belgrade, Serbia Yugoslavi"+[684 chars]
val removeMarkup : string -> string
val getMoves : string -> string []
val normalizedText : string =
  "1. e4 e5 2. Nf3 Nc6 3. Bb5 3... a6 4. Ba4 Nf6 5. O-O Be7 6. "+[464 chars]
val listMoves: unit -> unit

> listMoves ();;

e4 e5
Nf3 Nc6
Bb5
a6
Ba4 Nf6
O-O Be7
Re1 b5
Bb3 d6
c3 O-O
h3 Nb8
d4 Nbd7
c4 c6
...
```

Группировка

Регулярные выражения могут использоваться не только для проверки вводимых пользователем данных и манипулирования текстом. Они могут также применяться для извлечения текстовых данных, соответствующих заданному шаблону.

Если требуется применить квантификаторы, такие как `?`, `+` или `*`, к последовательности символов, можно сгруппировать их, заключив в круглые скобки. Например, выражение `"(foo)+"` совпадает с фразой `"foo"`, повторяющейся один или более раз. Без круглых скобок выражение `"foo+"` будет соответствовать символу `f`, `o`, затем еще одному символу `o`, повторяющемуся один или более раз.

Группировка может быть более полезной, помимо простого добавления выразительности в регулярные выражения. Метод `Regex.Match` возвращает объект типа `Match`, который можно использовать для получения строк, сохраненных в каждой сохраняющей группе.

В примере А.8 определяется регулярное выражение для извлечения слов-аналогий из стандартизованных тестов. Каждая часть регулярного выражения, соответствующая слову, заключена в круглые скобки, и поэтому является отдельной группой.

Код примера принимает объект `Match` и выводит содержимое каждой группы в консоль. Обратите внимание, что все регулярное выражение интерпретируется как «первая» группа, которая в данном примере представляет полную аналогию.

Пример А.8. Группы в регулярном выражении

```
> // Группы в регулярном выражении
open System
open System.Text.RegularExpressions

let analogyRegex = "(\w+):(\w+)::(\w+):(\w+)"[1]
let m = Regex.Match("dog:bark::cat:meow", analogyRegex);;

val analogyRegex : string = "(\w+):(\w+)::(\w+):(\w+)"
val m : Match = dog:bark::cat:meow

> // Вывод содержимого групп
printfn "Group captures..."
for i = 0 to m.Groups.Count - 1 do
    printfn "[%d] %s" i (m.Groups.[i].Value);;
Group captures...
[0] dog:bark::cat:meow
[1] dog
[2] bark
[3] cat
[4] meow
```

Ранее мы уже видели неплохой пример использования группировки в регулярных выражениях в соединении с активными шаблонами.

Пример А.9 взят из главы 7 (пример 7.7). Обратите внимание, что частичный активный шаблон возвращает кортеж из трех строк, которые

[1] Помимо обращения к группам по индексам, библиотека регулярных выражений в платформе .NET поддерживает так называемые именованные группы. Например, вы можете определить группу `(?<groupName>\w+)`, а затем получить значение не по индексу, а по имени: `m.Groups. ["groupName"]`. Это повышает читаемость кода и упрощает его последующее сопровождение. – *Прим. науч. ред.*

сохраняются тремя группами регулярного выражения. Метод `List.tail` используется, чтобы пропустить содержимое первой группы – совпадение со всем регулярным выражением – и вернуть значения отдельных групп.

В примере используется частичный активный шаблон (`|RegexMatch3|_|`), соответствующий различным формам представления даты, для связывания значений года, месяца и дня внутри правила.

Пример А.9. Регулярные выражения с группами в активных шаблонах

```
let (|RegexMatch3|_|) (pattern : string) (input : string) =
    let result = Regex.Match(input, pattern)

    if result.Success then
        match (List.tail [ for g in result.Groups -> g.Value ]) with
        | fst :: snd :: trd :: []
            -> Some (fst, snd, trd)
        | [] -> failwith "Match succeeded, but no groups found.\n \
                          Use '(.*)' to capture groups"
        | _ -> failwith "Match succeeded, but did not find exactly three groups."
    else
        None

let parseTime input =
    match input with
    // Сопоставить входные данные с формой представления "6/20/2008"
    | RegexMatch3 "(\d+)/(\d+)/(\d+)" (month, day, year)
    // Сопоставить входные данные с формой представления "2000-3-6"
    | RegexMatch3 "(\d+)-(\d+)-(\d+)" (year, month, day)
        -> Some( new DateTime(int year, int month, int day) )
    | _ -> None
```

Тем, кто захочет поближе познакомиться с регулярными выражениями, я рекомендую прочитать книгу «Mastering Regular Expressions»[1] Джеффри Е. Ф. Фридла (Jeffrey E. F. Friedl), выпущенную издательством O'Reilly.

Работа с XML

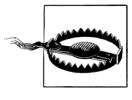
Использование XML позволяет легко создавать и сохранять структурированные данные без необходимости писать собственные парсеры. То есть для работы с документами XML вы также можете использовать библиотеки .NET.

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – Символ-Плюс, 2008.

В .NET существуют две основные библиотеки для работы с XML: более ранняя, менее удобная в использовании, версия в сборке System.Xml.dll и более новая и более дружественная библиотека в сборке System.XML.Linq.dll[1], которую мы и будем рассматривать далее.

Создание документов XML

Чтобы создать документ XML, достаточно просто создать иерархию объектов типа XElement. В примере A.10 определяется тип Book, экземпляры которого способны преобразовывать себя в документы XML с помощью метода ToXml.



Для работы с XML необходимо добавить в программу обе вышеупомянутые сборки, System.Xml.dll и System.Xml.Linq.dll.

Пример A.10. Создание документов XML

```
#r "System.Xml.dll"
#r "System.Xml.Linq.dll"

open System.Xml.Linq

// Преобразует строку в объект XName
let xname thing = XName.Get(thing.ToString())

type Book =
    | ComicBook of string * int
    | Novel of string * string
    | TechBook of string * string
    member this.ToXml() =
        match this with
        | ComicBook(series, issue) ->
            new XElement(xname "ComicBook",
                new XElement(xname "Series", series),
                new XElement(xname "Issue", issue))

        | Novel(author, title) ->
            new XElement(xname "Novel",
                new XElement(xname "Author", author),
                new XElement(xname "Title", title))

        | TechBook(author, title) ->
            new XElement(xname "TechBook",
                new XElement(xname "Author", author),
                new XElement(xname "Title", title))

/// Сгенерировать документ XML из списка книг
let bookshelfToXml (bookshelf : Book list) =
```

[1] Данная библиотека также известна под именем LINQ 2 XML. – *Прим. науч. ред.*

```
    let books = bookshelf
                |> List.map (fun book -> book.ToXml())
                |> Array.ofList
    new XElement(xname "Bookshelf", books)
```

Вы легко можете убедиться, насколько просто создаются документы XML, если использовать этот пример в сеансе FSI. Для сохранения созданного документа XML достаточно всего лишь вызвать метод `Save` объекта `XElement`:

```
> // Преобразование списка из объединений Book в документ XML
[
    ComicBook("IronMan", 1)
    ComicBook("IronMan", 2)
    TechBook("Machine Learning", "Mitchell")
    TechBook("Effective C++", "Meyers")
    Novel("World War Z", "Max Brooks")
    ComicBook("IronMan", 3)
]
|> bookshelfToXml
|> (printfn "%A");;

<Bookshelf>
  <ComicBook>
    <Series>IronMan</Series>
    <Issue>1</Issue>
  </ComicBook>
  <ComicBook>
    <Series>IronMan</Series>
    <Issue>2</Issue>
  </ComicBook>
  <TechBook>
    <Author>Machine Learning</Author>
    <Title>Mitchell</Title>
  </TechBook>
  <TechBook>
    <Author>Effective C++</Author>
    <Title>Meyers</Title>
  </TechBook>
  <Novel>
    <Author>World War Z</Author>
    <Title>Max Brooks</Title>
  </Novel>
  <ComicBook>
    <Series>IronMan</Series>
    <Issue>3</Issue>
  </ComicBook>
</Bookshelf>
val it : unit = ()
```

Чтение документов XML

Для открытия документов XML используются статические методы XElement.Parse, которые принимают строку с данными в формате XML и возвращают экземпляр XElement. Кроме того, существует метод Load, который принимает путь к файлу XML:[1]

```
> // Загрузка существующего документа XML с диска
do
    let shelf = XElement.Load("LordOfTheRings.xml")
    printfn "Loaded Document:\n%A" shelf;;

Loaded Document:
<Bookshelf>
  <Novel>
    <Author>The Two Towers</Author>
    <Title>Tolkien</Title>
  </Novel>
  <Novel>
    <Author>Fellowship of the Ring</Author>
    <Title>Tolkien</Title>
  </Novel>
  <Novel>
    <Author>The Return of the King</Author>
    <Title>Tolkien</Title>
  </Novel>
</Bookshelf>
val it : unit = ()

> // Парсинг документа XML из строки
do
    let shelf = XElement.Parse("<Bookshelf>
                                  <TechBook>
                                    <Author>Steve McConnell</Author>
                                    <Title>CodeComplete</Title>
                                  </TechBook><TechBook>
                                    <Author>Charles Petzold</Author>
                                    <Title>Code</Title>
                                  </TechBook>
                                </Bookshelf>")
    printfn "Parsed Document:\n%A" shelf;;

Parsed Document:
<Bookshelf>
  <TechBook>
```

[1] Существует несколько перегруженных версий метода Load, которые принимают Stream, TextReader, XmlReader, а также версии методов с параметром LoadOptions. Подробности см. в официальной документации класса XElement. – *Прим. науч. ред.*

```
    <Author>Steve McConnell</Author>
    <Title>CodeComplete</Title>
  </TechBook>
  <TechBook>
    <Author>Charles Petzold</Author>
    <Title>Code</Title>
  </TechBook>
</Bookshelf>
val it : unit = ()
```

Сохранение данных

После вычисления необходимых значений вам наверняка потребуется сохранить их на диске или в базе данных. В .NET существует огромное число способов сохранения данных, каждый из которых отличается скоростью выполнения и простотой использования.

Файл ввода-вывода

Большинство механизмов ввода-вывода в .NET опираются на использование абстрактного класса `System.IO.Stream`, который определяет простейший интерфейс между потоком ввода-вывода и нижележащим хранилищем. (Под нижележащим хранилищем понимается «место хранения данных».) Классы наследуют класс `Stream` и добавляют специализированные методы чтения и записи для работы с конкретным устройством хранения данных. Например, класс `MemoryStream` используется для чтения и записи данных в памяти, класс `NetworkStream` используется для чтения и записи данных из сетевых соединений, класс `FileStream` используется для чтения и записи данных из файлов и так далее.

Самый простой способ чтения или записи данных с диска заключается в использовании методов `File.ReadAllLines` и `File.WriteAllLines` соответственно. Метод `WriteAllLines` принимает имя файла и массив строк с текстовыми данными, которые должны быть записаны в файл, а метод `ReadAllLines` принимает имя файла и возвращает массив строк с содержимым файла:

```
> // Решение задачи FizzBuzz[1]
open System.IO

let computeFileContents() =
```

[1] Это математическая игра со следующими правилами. Все становятся в круг, и начинают считать от 1 до 99. Если число делится на 3, надо вместо него говорить «FIZZ» (по-русски можно применить междометие «ОЙ»). Если число делится на 5, надо говорить «BUZZ» (наверное, что-то вроде «АЙ»). То есть число 15 должно звучать как «FIZZ – BUZZ» («ОЙ – АЙ»), т.к. оно делится и на 3, и на 5. – *Прим. перев.*

```
    [| 1 .. 100 |]
    |> Array.map(function
                | x when x % 3 = 0 && x % 5 = 0 -> "FizzBuzz"
                | x when x % 3 = 0 -> "Fizz"
                | x when x % 5 = 0 -> "Buzz"
                | _ -> ".");;

val computeFileContents : unit -> string array

> File.WriteAllLines("FizzBuzzSln.txt", computeFileContents());;
val it : unit = ()

> let fileContents = File.ReadAllLines("FizzBuzzSln.txt");;

val fileContents : string [] =
  [|"."; "."; "Fizz"; "."; "Buzz"; "Fizz"; "."; "."; "Fizz"; "Buzz"; ".";
    "Fizz"; "."; "."; "FizzBuzz"; "."; "."; "Fizz"; "."; "Buzz"; "Fizz"; ".";
    "."; "Fizz"; "Buzz"; "."; "Fizz"; "."; "."; "FizzBuzz"; "."; "."; "Fizz";
    "."; "Buzz"; "Fizz"; "."; "."; "Fizz"; "Buzz"; "."; "Fizz"; "."; ".";
    "FizzBuzz"; "."; "."; "Fizz"; "."; "Buzz"; "Fizz"; "."; "."; "Fizz";
    "Buzz"; "."; "Fizz"; "."; "."; "FizzBuzz"; "."; "."; "Fizz"; "."; "Buzz";
    "Fizz"; "."; "."; "Fizz"; "Buzz"; "."; "Fizz"; "."; "."; "FizzBuzz"; ".";
    "."; "Fizz"; "."; "Buzz"; "Fizz"; "."; "."; "Fizz"; "Buzz"; "."; "Fizz";
    "."; "."; "FizzBuzz"; "."; "."; "Fizz"; "."; "Buzz"; "Fizz"; "."; ".";
    "Fizz"; "Buzz"|]
```

Сериализация данных

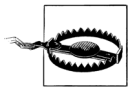
Сама по себе возможность читать и записывать данные в файл чрезвычайно полезна, однако в ней мало проку, если вы не сможете преобразовать свои классы в форму, пригодную для сохранения в файле. Для этого вам потребуется прибегнуть к *сериализации* (преобразованию данных в последовательную форму). Сериализация – это процесс преобразования объекта в двоичный формат,[1] в котором его можно будет записать в файл или передать по сети.

Конечно, вы можете создать собственный механизм сериализации, однако в .NET уже имеется встроенная поддержка сериализации. Чтобы воспользоваться механизмом сериализации, встроенным в .NET, достаточно просто пометить свой тип атрибутом `[<Serializable>]`[2].

---

[1] Конечно же, сериализация может быть не только бинарной, но и текстовой. Классическим примером текстовой сериализации является преобразование объекта в XML с помощью класса `XmlSerializer`. – *Прим. науч. ред.*

[2] Атрибут `[<Serializable>]` позволяет сериализовать/десериализовать объект с помощью класса `BinaryFormatter`, что также позволяет передавать экземпляры этих классов между доменами приложений (Application Domains). – *Прим. науч. ред.*



Наличие атрибута [<Serializable>] в объявлении типа вовсе не означает, что его экземпляры могут быть сериализованы произвольными способами. Различные формы сериализации накладывают свои ограничения.

Например, механизм сериализации объектов в формат XML требует, чтобы тип имел конструктор без параметров. Это ограничение лишает вас возможности использовать этот механизм с такими типами данных, как записи и размеченные объединения.

Рассмотрим следующие типы, определяющие бейсбольную команду в виде размеченного объединения, записи и класса, каждый из которых помечен атрибутом [<Serializable>]:

```
open System
open System.Collections.Generic

// Размеченное объединение
[<Serializable>]
type Position =
    // Бетери[1]
    | Pitcher   | Catcher
    // Инфилдеры
    | ShortStop | FirstBase   | SecondBase | ThirdBase
    // Аутфилдеры
    | LeftField | CenterField | RightField

// Запись
[<Serializable>]
type BaseballPlayer =
{
    Name : string
    Position : Position
    BattingAvg : float
}

// Класс
[<Serializable>]
type BaseballTeam(name : string ) =

    let m_roster = new List<BaseballPlayer>()

    member this.TeamName = name

    member this.DraftPlayer(player) = m_roster.Add(player)
    member this.BenchPlayer(player) = m_roster.Remove(player)

    member this.Roster = m_roster |> Seq.toArray
```

[1] Бейсбольная терминология взята с сайта *http://www.moldbaseball.com/rules/terminology/*. – *Прим. перев.*

Сериализация объекта в `byte[]` **выполняется простой передачей потока (stream) экземпляру класса** `BinaryFormatter`**. В примере A.11 приводится листинг в окне FSI, где метод** `serializeThing` **сериализует произвольный объект** `obj` **в экземпляр** `byte[]`**, а метод** `saveTeam` **сохраняет объект** `BaseballTeam` **в файл.**

Метод `serializeThing` **создает экземпляр класса** `MemoryStream`**, который затем передается экземпляру класса** `BinaryFormatter`**, записывающему сериализованную версию объекта** `obj` **в память, и возвращает данные из** `MemoryStream` **в виде массива** `byte[]`**.**

Пример A.11. Сериализация

```
> // Сериализация объекта
open System.IO
open System.Runtime.Serialization.Formatters.Binary

/// Преобразует объект в последовательную форму
let serializeThing(thing) =
    let bf = new BinaryFormatter()
    use mstream = new MemoryStream()
    bf.Serialize(mstream, thing)

    mstream.ToArray()

/// Сохраняет объект представления бейсбольной команды на диск
let saveTeam(team : BaseballTeam, filePath : string) =

    let byteArr = serializeThing team

    use fileStream = new FileStream(filePath, FileMode.CreateNew)
    fileStream.Write(byteArr, 0, byteArr.Length)
    fileStream.Close()

     ()
;;

val serializeThing : 'a -> byte []
val saveTeam : BaseballTeam * string -> unit

> // Создание бейсбольной команды
let M's = new BaseballTeam("Mariners")
let ichiro =
    {
        Name = "Ichiro Suzuki"
        Position = RightField
        BattingAvg = 0.330
    }
let griffeyJr =
   { Name = "Ken Griffey, Jr."; Position = LeftField; BattingAvg = 0.287 }
M's.DraftPlayer(ichiro)
M's.DraftPlayer(griffeyJr);;
```

```
val M's : BaseballTeam
val ichiro : BaseballPlayer = {Name = "Ichiro Suzuki";
                               Position = RightField;
                               BattingAvg = 0.33;}
val griffeyJr : BaseballPlayer = {Name = "Ken Griffey, Jr.";
                                  Position = LeftField;
                                  BattingAvg = 0.287;}

> // Преобразование объекта, представляющего игрока Ichiro, в массив byte[]
serializeThing ichiro;;
val it : byte [] =
  [|0uy; 1uy; 0uy; 0uy; 0uy; 255uy; 255uy; 255uy; 255uy; 1uy; 0uy; 0uy; 0uy;
    0uy; 0uy; 0uy; 0uy; 12uy; 2uy; 0uy; 0uy; 0uy; 67uy; 70uy; 83uy; 73uy;
    45uy; 65uy; 83uy; 83uy; 69uy; 77uy; 66uy; 76uy; 89uy; 44uy; 32uy; 86uy;
    101uy; 114uy; 115uy; 105uy; 111uy; 110uy; 61uy; 48uy; 46uy; 48uy; 46uy;
    48uy; 46uy; 48uy; 44uy; 32uy; 67uy; 117uy; 108uy; 116uy; 117uy; 114uy;
    101uy; 61uy; 110uy; 101uy; 117uy; 116uy; 114uy; 97uy; 108uy; 44uy; 32uy;
    80uy; 117uy; 98uy; 108uy; 105uy; 99uy; 75uy; 101uy; 121uy; 84uy; 111uy;
    107uy; 101uy; 110uy; 61uy; 110uy; 117uy; 108uy; 108uy; 5uy; 1uy; 0uy; 0uy;
    0uy; 23uy; 70uy; 83uy; 73uy; 95uy; ...|]
> saveTeam(M's, @"D:\MarinersTeam.bbt");;
val it : unit = ()
```

Десериализация

Десериализация данных выполняется практически так же, как сериализация, только вместо метода `Serialize` вызывается метод `Deserialize` класса `BinaryFormatter`. Метод `Deserialize` возвращает экземпляр типа `obj`, который перед использованием необходимо привести к необходимому типу.

В примере А.12 приводится реализация функции `loadTeam`, составляющей пару функции `saveTeam`. Она открывает объект `FileStream`, читает из него данные и десериализует их с помощью объекта `BinaryFormatter`, а затем приводит их к типу `BaseballTeam`.

Пример А.12. Десериализация

```
> // Десериализация
let loadTeam(filePath) =

    use fileStream = new FileStream(filePath, FileMode.Open)

    let bf = new BinaryFormatter()

    let result = bf.Deserialize(fileStream)

    (result :?> BaseballTeam);;

val loadTeam : string -> BaseballTeam
```

```
> let loadedM's = loadTeam @"D:\MarinersTeam.bbt";;

val loadedM's : BaseballTeam

> // Вывести загруженные данные
loadedM's.Roster
|> Array.iter (fun player -> printfn "%s - Batting %.3f"
                                     player.Name
                                     player.BattingAvg);;
Ichiro Suzuki - Batting 0.330
Ken Griffey, Jr. - Batting 0.287
val it : unit = ()
```

Стандартная библиотека языка F#

Подобно большинству современных языков программирования, F# поставляется вместе со своей собственной стандартной библиотекой. При этом стандартная библиотека F# делится на две основные части:

FSharp.Core.dll

: Библиотека FSharp.Core.dll неявно подключается всеми приложениями на языке F# и является стандартной библиотекой языка F#. Она содержит определения таких распространенных типов, как list, Async, option и других.

F# PowerPack

: F# PowerPack — это набор экспериментальных дополнений и дополнительных возможностей, расширяющих стандартную библиотеку F#. Например, в этой библиотеке присутствуют модули, позволяющие обеспечить совместимость приложений на языке F# с языком программирования OCaml на уровне исходных текстов.

FSharp.Core.dll

Изучение библиотеки FSharp.Core.dll поможет вам извлечь максимум пользы из своих проектов на языке F#. К этому моменту мы рассмотрели множество функциональных возможностей, имеющихся в стандартной библиотеке F#, однако в пространстве имен Microsoft.FSharp имеется еще несколько жемчужин, наиболее ценными из которых являются несколько исключительно функциональных (неизменяемых) структур данных.

В главе 4 были представлены изменяемые типы коллекций, являющиеся простыми в использовании альтернативами для типов list и seq. Однако желающие использовать исключительно функциональный стиль программирования могут вместо них применять функциональные типы коллекций.

Типы Set<_> и Map<_,_> действуют практически точно так же, как типы HashSet<_> и Dictionary<_,_>, представленные в главе 4, за исключением того, что их экземпляры не могут изменяться. То есть при добавлении в коллекцию нового элемента метод Add возвращает не значение типа unit, изменяя внутреннюю структуру данных, а новый экземпляр типа Set<_> или Map<_,_>.

В примере А.13 показано использование типа Set<_> для получения списка уникальных слов, содержащихся в веб-странице, роль которой в данном примере играет текст из книги «Франкенштейн», написанной Мэри Шелли (Mary Shelley).

Обратите внимание, что для создания экземпляра типа Set<_> используется метод Array.fold. Чтобы заполнить экземпляр коллекции функционального типа, необходимо использовать рекурсию. Кроме того, точно так же, как модули List и Seq предоставляют набор функций с экземплярами одноименных типов, модуль Set содержит методы, такие как Set.add и Set.count.

Пример А.13. Использование типа Set

```
> // Функциональный тип данных Set
open System
open System.IO
open System.Net

let getHtml (url : string) =
    let req = WebRequest.Create(url)
    let rsp = req.GetResponse()

    use stream = rsp.GetResponseStream()
    use reader = new StreamReader(stream)

    reader.ReadToEnd()

let uniqueWords (text : string) =
    let words = text.Split([| ' ' |], StringSplitOptions.RemoveEmptyEntries)

    let uniqueWords =
        Array.fold
            (fun (acc : Set<string>) (word : string) -> Set.add word acc)
            Set.empty
            words

    uniqueWords;;

val getHtml : string -> string
val uniqueWords : string -> Set<string>

> let urlToShelley'sFrankenstein = "http://www.gutenberg.org/files/84/84.txt";;
```

```
val urlToShelley'sFrankenstein : string =
"http://www.gutenberg.org/files/84/84.txt"

> // Создать множество Set уникальных слов
let wordsInBook =
    urlToShelley'sFrankenstein
    |> getHtml |> uniqueWords;;

val wordsInBook : Set<string> = ...

> Set.count wordsInBook;;
val it : int = 16406
```

В примере A.14 показано использование типа `Map<_,_>` для подсчета количества вхождений каждого слова в книгу. После подсчета экземпляр `Map<_,_>` преобразуется в отсортированную последовательность кортежей ключ-значение, из которой выбираются 20 наиболее часто встречающихся слов, и производится вывод их на экран.

Как и экземпляр типа `Set<_>` в предыдущем примере, экземпляр типа `Map<_,_>` также создается с помощью операции свертки.

Пример A.14. Использование типа Map

```
> // Функциональный тип Map
let wordUsage (text : string) =
    let words = text.Split([| ' ' |], StringSplitOptions.RemoveEmptyEntries)

    let wordFrequency =
        Array.fold
            (fun (acc : Map<string, int>) (word : string) ->
                if Map.containsKey word acc then
                    let timesUsed = acc.[word]
                    Map.add word (timesUsed + 1) acc
                else
                    Map.add word 1 acc)
            Map.empty
            words

    wordFrequency

let printMostFrequentWords (wordFrequency : Map<string, int>) =
    let top20Words =
        wordFrequency
        |> Map.toSeq
        |> Seq.sortBy (fun (word, timesUsed) -> -timesUsed)
        |> Seq.take 20

    printfn "Top Word Usage:"
    top20Words
    |> Seq.iteri(fun idx (word, timesUsed) ->
        printfn "%d\t '%s' was used %d times" idx word timesUsed);;
```

```
val wordUsage : string -> Map<string,int>
val printMostFrequentWords : Map<string,int> -> unit

> // Вывод наиболее часто встречающихся слов
urlToShelley'sFrankenstein
|> getHtml |> wordUsage |> printMostFrequentWords
;;
Top Word Usage:
0        'the' was used 3408 times
1         'and' was used 2569 times
2         'of' was used 2405 times
3         'I' was used 2333 times
4         'to' was used 1900 times
5         'my' was used 1389 times
6         'a' was used 1212 times
7         'in' was used 980 times
8         'that' was used 876 times
9         'was' was used 874 times
10        'with' was used 573 times
11        'had' was used 571 times
12        'but' was used 480 times
13        'me' was used 478 times
14        'which' was used 465 times
15        'as' was used 437 times
16        'not' was used 432 times
17        'his' was used 413 times
18        'by' was used 411 times
19        'you' was used 404 times
val it : unit = ()
```

F# PowerPack

Когда разработка языка F# еще находилась в стадии исследовательского проекта в подразделении Microsoft Research, первоначальная версия стандартной библиотеки F# была очень большой. Она включала в себя не только все функциональные возможности, которые ныне можно найти в библиотеке `FSharp.Core.dll`, но также содержала множество экспериментальных расширений. Когда в компании Microsoft было принято решение интегрировать язык F# в состав Visual Studio 2010, в библиотеке `FSharp.Core.dll` был оставлен только набор базовых возможностей, а все остальное было перенесено в библиотеку `F# PowerPack` (ее компоненты могут использоваться либо опытными разработчиками, либо со временем будут стабилизированы и перемещены в соответствующую библиотеку F#).

Внутри `F# PowerPack` вы найдете такие инструменты и утилиты, как:

- Функции и модули, которые можно найти в библиотеке языка OCaml
- Библиотеки для построения графиков и диаграмм
- Инструменты для создания парсеров

- Библиотеки, обеспечивающие поддержку LINQ-подобных механизмов в языке F#
- F# CodeDOM, или библиотека для генерирования исходного кода на языке F#

Библиотека `F# PowerPack` не распространяется вместе с Visual Studio 2010 – ее необходимо загружать самостоятельно с сайта CodePlex (*http://www.codeplex.com*).

Несмотря на то, что `F# PowerPack` содержит узкоспециализированные функциональные возможности, предназначенные для узкого круга разработчиков, тем не менее в ней имеются две особенности, которые могут представлять для вас интерес.

Единицы измерения СИ

В главе 7 вы узнали, как ликвидировать ошибки, вызванные использованием неправильных единиц измерения. Однако вместо того чтобы снова и снова определять стандартные единицы измерения, такие как секунды и метры, для каждого своего приложения на языке F#, можно использовать встроенные единицы измерения из международной системы единиц измерения (*Système International d'Unitès*), известной также как метрическая система мер, встроенные в библиотеку `F# PowerPack`.

Часть единиц измерения СИ из числа определенных в библиотеке F# перечислена в табл. А.2. Чтобы получить доступ к встроенным единицам измерения, необходимо открыть модуль `Microsoft.FSharp.Core.SI`. (Или просто `SI`, потому что пространство имен `Microsoft.FSharp.Core` открывается по умолчанию.)

```
> // Использование единиц измерения СИ, объявленных в F# PowerPack
#r "FSharp.PowerPack.dll"
open SI
let g = -9.8<m/s>;;

val g : float<m/s>
```

Таблица А.2. Наиболее часто используемые единицы измерения, объявленные в F# PowerPack

Единица измерения	Название	Описание
`m`	Метр	Единица измерения длины в СИ
`kg`	Килограмм	Единица измерения массы в СИ
`s`	Секунда	Единица измерения времени в СИ
`K`	Кельвин	Единица измерения температуры в СИ
`Hz`	Герц	Единица измерения частоты колебаний в СИ
`N`	Ньютон	Единица измерения силы в СИ

FSLex и FSYacc

Часто при создании предметно-ориентированных языков (Domain-Specific Languages) бывает необходимо создать парсер, или библиотеку, которая принимает текст (код) и преобразует его в древовидную структуру (абстрактное синтаксическое дерево, или AST – Abstract Syntax Tree). Существует множество способов реализации парсеров, из которых самый простой заключается в том, чтобы вообще отказаться от создания парсера и генерировать парсер на основе формальной грамматики.

В состав библиотеки F# PowerPack входят такие инструменты, как FSLex.exe и FSYacc.exe, которые используются для генерирования парсеров. Фактически даже парсер компилятора F# написан с использованием FSLex.exe и FSYacc.exe. Инструмент FSLex.exe используется для создания лексического анализатора, который принимает исходный код и генерирует поток лексем. А инструмент FSYacc.exe используется для создания парсера, который преобразует поток лексем в фактическое дерево AST.

С помощью FSLex.exe и FSYacc.exe вы сможете быстро разработать и отладить парсер, благодаря чему основное свое время вы будете тратить на реализацию своего языка, а не на манипулирование деревьями парсера.

Описание синтаксиса этих инструментов легко могло бы занять отдельную главу или даже целую книгу. Дополнительную информацию и обучающие руководства по этим инструментам вы найдете на домашней странице F# PowerPack на сайте CodePlex.

B

Взаимодействие программ на F# с кодом на других языках .NET

Эта книга представляет F# как замечательный язык программирования, позволяющий повысить производительность труда программиста, а также обеспечить бесшовную интеграцию с другими инструментами платформы .NET. Уже должно быть понятно, что F# является замечательным языком, однако выражение «бесшовная интеграция» требует некоторых пояснений.

Доминирующими языками программирования в .NET на сегодняшний день являются C# и Visual Basic .NET. Программируя на языке F#, вы можете писать и использовать объектно-ориентированные сборки для взаимодействия с кодом, написанным на языках C# и VB.NET, однако некоторые особенности языка F# при этом могут быть утрачены.

В этом приложении рассказывается о проблемах использования кода F# в других языках .NET. Прочитав материал, вы сможете самостоятельно определять проблемные области при программировании на нескольких языках. Кроме того, вы научитесь реализовывать взаимодействие программ на языке F# с неуправляемым (unmanaged или native) кодом с помощью таких механизмов, как Platform Invoke и COM.

Разрабатывая программы на языке F#, вы можете взаимодействовать с обоими языками, C# и Visual Basic .NET, однако ради удобства в этом приложении я буду рассматривать только язык C#. Всякий раз, когда вы будете видеть название C#, вы при желании можете мысленно подставлять вместо него «C# или Visual Basic .NET».

Взаимодействие с другими языками .NET

Ключом к взаимодействию языков программирования в .NET является доступность всех открытых типов и методов библиотеки классов во всех языках программирования. Совершенно ясно, что если библиотека экспортирует конструкции, которые не могут быть представлены на языке, в котором эти конструкции используются, то могут возникнуть проблемы.

Использование кода F# в программах на языке C#

При разработке кода на языке F# следует воздерживаться от прямого экспортирования таких функциональных типов, как размеченные объединения, записи и значения функций. (Используйте их только для внутренних нужд, а для экспортирования определяйте обычные классы и методы.)

Размеченные объединения

Размеченные объединения – это типы данных, которые определяют набор допустимых значений; в каждый конкретный момент времени экземпляр размеченного объединения может хранить только одно из этих значений. Для реализации этой возможности компилятор F# опирается на использование механизмов наследования и полиморфизма.

В примере B.1 определяется простое размеченное объединение на языке F#, описывающее служащих компании.

Пример B.1. Размеченное объединение, экспортируемое кодом на языке F#

```
// HumanResources.fs – программный код F#, объявляющий размеченное объединение

namespace Initech.HumanResources

type PersonalInfo = { Name : string; EmployeeID : int }

type Employee =
    | Manager of PersonalInfo * Employee list
    | Worker of PersonalInfo
    | Intern
```

Работа с размеченными объединениями не вызывает сложностей благодаря поддержке в языке F# их синтаксиса и возможности сопоставления с образцом. Однако в программах на языке C# работа с размеченными объединениями вызывает массу сложностей.

В примере B.2 показан код, реализующий то же самое размеченное объединение `Initech.HumanResources.Employee` на языке C# (и соответствующий интерфейс, который будет вынужден использовать программист на языке C#). В этом примере опущены интерфейсы и методы, сгенерированные компилятором.

Этот пример наглядно показывает, как сложно работать с таким типом в языке C#, а также позволяет почувствовать, какой большой объем работы выполняет компилятор F#, чтобы сделать выражения сопоставления с образцом такими элегантными.

Пример B.2. Размеченные объединения F# с точки зрения C#

```
using System;
using Microsoft.FSharp.Collections;

namespace Initech.HumanResources
{
    [Serializable, CompilationMapping(SourceConstructFlags.SumType)]
    public abstract class Employee
    {
        // Поля
        internal static readonly Employee _unique_Intern = new _Intern();

        // Методы
        internal Employee()
        {
        }

        [CompilationMapping(SourceConstructFlags.UnionCase, 0)]
        public static Employee NewManager(PersonalInfo item1,
        FSharpList<Employee> item2)
        {
            return new Manager(item1, item2);
        }

        [CompilationMapping(SourceConstructFlags.UnionCase, 1)]
        public static Employee NewWorker(PersonalInfo item)
        {
            return new Worker(item);
        }

        // Свойства
        public static Employee Intern
        {
            [CompilationMapping(SourceConstructFlags.UnionCase, 2)]
            get
            {
                return _unique_Intern;
            }
        }

        public bool IsIntern
        {
            [CompilerGenerated, DebuggerNonUserCode]
            get
            {
```

```
        return (this is _Intern);
    }
}

public bool IsManager
{
    [CompilerGenerated, DebuggerNonUserCode]
    get
    {
        return (this is Manager);
    }
}

public bool IsWorker
{
    [CompilerGenerated, DebuggerNonUserCode]
    get
    {
        return (this is Worker);
    }
}

public int Tag
{
    get
    {
        Employee employee = this;
        return (!(employee is _Intern) ? (!(employee is Worker)
        ? 0 : 1) : 2);
    }
}

// Вложенные типы
[Serializable]
internal class _Intern : Employee
{
    // Методы
    internal _Intern()
    {
    }
}

[Serializable]
public class Manager : Employee
{
    // Поля
    internal readonly PersonalInfo item1;
    internal readonly FSharpList<Employee> item2;

    // Методы
    internal Manager(PersonalInfo item1, FSharpList<Employee> item2)
```

```
        {
            this.item1 = item1;
            this.item2 = item2;
        }

        // Свойства
        [CompilationMapping(SourceConstructFlags.Field, 0, 0)]
        public PersonalInfo Item1
        {
            get
            {
                return this.item1;
            }
        }

        [CompilationMapping(SourceConstructFlags.Field, 0, 1)]
        public FSharpList<Employee> Item2
        {
            get
            {
                return this.item2;
            }
        }
    }

    public static class Tags
    {
        // Поля
        public const int Intern = 2;
        public const int Manager = 0;
        public const int Worker = 1;
    }

    [Serializable]
    public class Worker : Employee
    {
        // Поля
        internal readonly PersonalInfo item;

        // Методы
        internal Worker(PersonalInfo item)
        {
            this.item = item;
        }

        // Свойства
        [CompilationMapping(SourceConstructFlags.Field, 1, 0)]
        public PersonalInfo Item
        {
            get
            {
```

```
                return this.item;
            }
        }
    }
}
```

Функции как значения

Использование функций в качестве значений представляет похожую проблему при интеграции кода на языке F# с кодом на языке C#. Хотя язык C# также поддерживает лямбда-выражения, но компилятор C# обрабатывает их совершенно иначе.

В языке C# функции как значения представляются с помощью делегатов, а в версии C# 3.0 появились два новых типа обобщенных делегатов, чтобы помочь разработчикам на C# применять функциональный стиль программирования: `System.Action<_>` и `System.Func<_,_>`.

Тип `Action<_>` представляет функцию без возвращаемого значения, тогда как тип `Func<_,_>` представляет функцию, возвращающую некоторое значение. В обоих случаях параметры обобщенного типа представляют типы параметров функции.

В следующем фрагменте приводится код на языке F#, который создает значения типов `Action<_>` и `Func<_,_>`:

```
> // Использование типа System.Action
#r "System.Core.dll"

open System

let fsAction = new Action<string>(fun arg -> printfn "Hello, %s" arg);;

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.20620\System.Core.dll'

val fsAction : Action<string>

> fsAction.Invoke("World");;
Hello, World
val it : unit = ()
> // Использование типа System.Func
let fsFunc = new Func<string, int>(fun s -> Int32.Parse(s) * Int32.Parse(s));;

val fsFunc : Func<string,int>

> fsFunc.Invoke("4");;

val it : int = 16
```

Значения функций, создаваемые компилятором F#, являются экземплярами типа `FSharpFunc<_,_>`, а не `Action<_>` или `Func<_,_>`. В языке F#

функции как значения могут каррироваться, поэтому при передаче функции первого параметра возвращается новая функция, принимающая на один параметр меньше.

В примере B.3 показано определение типа `MathUtilities`, который содержит метод с именем `GetAdder`, возвращающий значение функции F#, принимающей три строковых параметра.

Пример B.3. Функции как значения в языке F#

```
namespace Initech.Math

open System

type MathUtilities =
    static member GetAdder() =
        (fun x y z -> Int32.Parse(x) + Int32.Parse(y) + Int32.Parse(z))
```

Метод `GetAdder` возвращает объект типа `FSharpFunc<_,_>`, который на языке F# может быть выражен как:

```
string -> string -> string -> int
```

Однако чтобы объявить функцию того же типа на языке C#, необходимо записать:

```
FSharpFunc <string, FSharpFunc <string, FSharpFunc <string, int>>>
```

А теперь представьте, на что будет похоже объявление функции, принимающей большее число параметров! Каррированные функции предлагают элегантно упростить код на функциональных языках программирования, но не следует переносить их на другие языки программирования, не поддерживающие каррирование.

Использование кода C# в программах на языке F#

Использование кода на языке C# в программах на языке F# обычно не вызывает проблем, потому что язык F# обладает практически всеми особенностями, которые можно найти в C# и VB.NET. Однако существует несколько вещей, которые можно делать в языке C#, но которые могут вызывать сложности в языке F#. Так, если вы предполагаете использовать библиотеки C# в программах на языке F#, имейте в виду следующие особенности.

Параметры ref и out

В языке C# можно объявлять параметры типа `out` и `ref`, которые позволяют функциям возвращать более одного значения за счет изменения их параметров. Спецификатор `ref` указывает, что значение параметра может изменяться в ходе выполнения функции, а спецификатор `out` указывает, что значение параметра будет установлено в ходе выполнения функции.

Язык F# поддерживает обе разновидности параметров, ref и out, и потому код F# может обращаться к коду C#, в котором используются такие параметры.

Взгляните на следующий фрагмент кода на языке C# – он определяет две функции, принимающие параметры out и ref:

```
public class CSharpFeatures
{
    public static bool ByrefParams(ref int x, ref int y)
    {
        x = x * x;
        y = y * y;
        return true;
    }

    public static bool OutParams(out int x, out int y)
    {
        x = 10;
        y = 32;
        return true;
    }
}
```

Передать параметры функции, принимающей аргументы типа ref, в языке F# можно двумя способами: либо передать значение ref<'a>, либо воспользоваться оператором получения адреса & изменяемого значения.

В следующем листинге сеанса FSI демонстрируется вызов метода CSharpFeatures.ByrefParams с использованием значения ref и оператора получения адреса. Обратите внимание, что после вызова функции оба параметра изменили свои значения:

```
> // Объявление параметров ref/out
let x = ref 3
let mutable y = 4;;

val x : int ref = {contents = 3;}
val mutable y : int = 4

> CSharpFeatures.ByrefParams(x, &y);;
val it : bool = true
> x;;
val it : int ref = {contents = 9;}
> y;;
val it : int = 16
```

В языке F# параметры типа out не являются обязательными, и, если они не передаются, параметры типа out возвращаются вместе с результатом функции в виде кортежа. При передаче параметров out в языке F# используется то же правило, что применяется к аргументам типа ref. Эти параметры должны быть экземплярами типа ref<'a> или из-

меняемыми значениями, к которым применяется оператор получения адреса.

В следующем листинге сеанса FSI показаны три вызова метода CSharpFeatures.OutParams. Обратите внимание, что тип возвращаемого значения функции изменяется в зависимости от того, были переданы методу параметры out или нет:

```
> // Объявление параметров ref/out
let x = ref 0
let mutable y = 0;;

val x : int ref = {contents = 0;}
val mutable y : int = 0

> // Передать оба параметра out, возвращаемое значение имеет тип bool
CSharpFeatures.OutParams(x, &y);;
val it : bool = true
> // Передать один параметр out, возвращаемое значение имеет тип bool * int
CSharpFeatures.OutParams(x);;
val it : bool * int = (true, 32)
> // Вызвать метод без параметров out,
// возвращаемое значение имеет тип bool * int * int
CSharpFeatures.OutParams();;
val it : bool * int * int = (true, 10, 32)
```

Чтобы экспортировать параметр метода, написанного на языке F#, как параметр ref, этот параметр должен иметь тип byref<'a>. Такой параметр можно интерпретировать как локальную изменяемую переменную. Чтобы экспортировать параметр out, он также должен иметь тип byref<'a>, но при этом к нему должен применяться атрибут System.Runtime.InteropServices.OutAttribute. В следующем фрагменте определяется функция на языке F#, которая принимает параметры ref и out соответственно:

```
open System.Runtime.InteropServices

let fsharpFunc (x : byref<int>, [<Out>] y : byref<int>) =
    x <- x + 1
    y <- y + 1
```

Обработка значения null

Еще одна проблема взаимодействия F#-C# заключается в обработке значения null. Экземпляры типов, созданных на языке F#, не могут иметь значение null, но в языке C# null считается допустимым значением для экземпляров ссылочных типов. Это может приводить к сложностям, когда в языке F# определяются типы, использующие типы из библиотек C# (потому что библиотечные функции на языке C# могут возвращать значение null, тогда как библиотеки на языке F# этого не предполагают).

Чтобы типы, определенные в сборке на языке F#, могли принимать значение null, можно воспользоваться атрибутом AllowNullLiteral. Этот атрибут может применяться только к классам и интерфейсам, то есть экземпляры записей, структур и размеченных объединений ни при каких условиях не могут принимать значение null.

В следующем листинге сеанса FSI показано использование атрибута AllowNullLiteral. Обратите внимание, что при попытке создать экземпляр типа Widget со значением null компилятор выдает сообщение об ошибке. Однако экземпляр типа Sprocket может принимать значение null благодаря тому, что объявление этого типа помечено атрибутом AllowNullLiteral:

```
> // Определение двух классов, один – с атрибутом [<AllowNullLiteral>],
// а другой – без атрибута
type Widget() =
    override this.ToString() = "A Widget"

[<AllowNullLiteral>]
type Sprocket() =
    override this.ToString() = "A Sprocket";;

type Widget =
  class
    new : unit -> Widget
    override ToString : unit -> string
  end
type Sprocket =
  class
    new : unit -> Sprocket
    override ToString : unit -> string
  end

> // ОШИБКА: экземпляр типа Widget не может иметь значение null
let x : Widget = null;;

  let x : Widget = null;;
  -----------------^^^^

stdin(12,18): error FS0043: The type 'Widget' does not have 'null' as a proper
value
```

(Тип "Widget" не содержит собственное значение "null".)

```
> // Нет ошибки
let x : Sprocket = null;;

val x : Sprocket = null
```

Взаимодействие с неуправляемым кодом

Платформа .NET предлагает богатые функциональные возможности и представляет собой отличную платформу для разработки новых приложений, тем не менее вам часто придется сталкиваться с необходимостью взаимодействовать с давно существующим кодом, например вызывать функции из библиотек, написанных на языке C или C++.

Язык F# поддерживает те же механизмы взаимодействий с неуправляемым кодом, что и язык C#.

Platform Invoke

Platform invoke, также называемый P/Invoke, – это механизм, который позволяет управляемым приложениям вызывать неуправляемый код, например функции Win32 API. Если у вас есть библиотека, реализованная не на платформе .NET, функции из которой вам требуется вызывать, эту возможность вам обеспечит механизм P/Invoke.

Механизм P/Invoke не ограничивается только операционной системой Windows. Любая реализация CLI, такая как Mono, может использовать P/Invoke для доступа к библиотекам, выполняющимся за пределами окружения CLI.

Чтобы в языке F# вызвать механизм P/Invoke, необходимо сначала объявить сигнатуру функции, чтобы платформа .NET смогла определить, как выполнить вызов функции (библиотека, где находится функция, количество и типы параметров и так далее).

В примере B.4 показан вызов функции `CopyFile` Win32 API, находящейся в библиотеке `kernel32.dll`. Атрибут `DllImport` определяет имя библиотеки и имя функции, с которой будет связан метод на языке F#. Вслед за атрибутом следует объявление функции – ключевое слово `extern` и определение сигнатуры функции, как она могла бы быть объявлена на C-подобном языке. (Имена типов предшествуют именам параметров.)

После того как будет объявлена сигнатура для механизма P/Invoke, можно вызывать фактическую функцию, как любую другую функцию на языке F#. Компилятор F# сам позаботится о передаче данных неуправляемой библиотеке. Процесс передачи данных между управляемым и неуправляемым выполняемыми кодами называется *маршаллингом (marshalling)*.

Пример B.4. Использование механизма Platform Invoke в языке F#

```
open System.Runtime.InteropServices

/// Сигнатура P/Invoke для функции CopyFile
[<DllImport("kernel32.dll", EntryPoint="CopyFile")>]
```

```
extern bool copyfile(char[] lpExistingFile,
                     char[] lpNewFile, bool bFailIfExists);

let file1 = @"D:\File.dat"
let file2 = @"D:\Backups\File.dat"

// Вызов функции CopyFile из библиотеки kernel32.dll
copyfile(file1.ToCharArray(), file2.ToCharArray(), false)
```

В более сложных случаях взаимодействия с помощью подсказок компилятору F# можно влиять на процесс маршаллинга данных.

В примере В.5 объявляются две структуры, `SequentialPoint` и `ExplicitRect`, для использования с функцией `PtInRect` Win32 API. Функция `PtInRect` проверяет, находится ли точка с указанными координатами внутри прямоугольной области. Однако эта функция ничего не знает о системе типов .NET и просто ожидает получить структуры с определенным набором полей.

Атрибут `StructLayout` определяет, как должны располагаться в памяти данные структуры, чтобы гарантировать порядок их размещения, который ожидается неуправляемой половиной вызова P/Invoke.

Например, функция `PtInRect` ожидает информацию о прямоугольной области в виде четырех 4-байтовых целых чисел в следующем порядке: левая, верхняя, правая и нижняя координаты. Тип `ExplicitRect` в примере содержит объявления полей в другом порядке. Поэтому мы использовали атрибуты `FieldOffset`, чтобы явно указать компилятору F#, в каком порядке должны следовать поля в типе `ExplicitRect`.

Пример В.5. Настройка маршаллинга

```
open System.Runtime.InteropServices

/// Определение структуры точки с последовательным расположением полей
[<Struct; StructLayout(LayoutKind.Sequential)>]
type SequentialPoint =

    new (x, y) = { X = x; Y = y }

    val X : int
    val Y : int

/// Определение структуры прямоугольника с явным описанием расположения полей
[<Struct; StructLayout(LayoutKind.Explicit)>]
type ExplicitRect =

    new (l, r, t, b) = { left = l; top = t; right = r; bottom = b }

    [<FieldOffset(12)>]
    val mutable bottom : int
    [<FieldOffset(0)>]
```

```
    val mutable left : int
    [<FieldOffset(8)>]
    val mutable right : int
    [<FieldOffset(4)>]
    val mutable top : int

/// Сигнатура P/Invoke для функции PtInRect
[<DllImport("User32.dll", EntryPoint="PtInRect")>]
extern bool PointInRect(ExplicitRect &rect, SequentialPoint pt);
```

Существует множество других способов настраивать маршаллинг данных между управляемым и неуправляемым кодами. За дополнительной информацией обращайтесь к документации в MSDN по пространству имен `System.Runtime.InteropServices`.

Взаимодействие с COM

Идея бесшовной интеграции множества языков программирования в мире .NET за счет использования одной и той же среды выполнения на самом деле не является такой уж новой. Фактически большая часть основных архитектурных решений .NET уходит корнями в другую технологию, которая называется *компонентной объектной моделью* (Component Object Model, COM). COM – это технология 1990-х годов обеспечения взаимодействия на двоичном уровне между различными программами. Например, используя эту технологию, приложения, написанные на языке C++ одним разработчиком, могут взаимодействовать с приложением, написанным на языке Visual Basic другим разработчиком.

Многие крупные неуправляемые приложения экспортируют интерфейсы COM, предоставляя возможность программного доступа к ним, загрузки расширений и так далее. Например, вместо того чтобы писать собственный веб-броузер, можно просто добавить в свое приложение COM элемент управления Internet Explorer.

Несмотря на то, что компоненты COM действуют за пределами среды выполнения .NET, как и механизм P/Invoke, тем не менее приложения .NET могут вызывать объекты COM.

Наиболее простой и наиболее типичный способ вызова объектов COM заключается в использовании обертки вызовов времени выполнения (Runtime Callable Wrapper, RCW). RCW – это сборка .NET, которая содержит классы-обертки вокруг интерфейсов COM. С точки зрения разработчика использование этой сборки мало чем отличается от использования любой другой сборки .NET, однако в действительности все функции в этой сборке и все экземпляры ее классов используют механизм COM, реализованный в среде выполнения .NET.

Сборка RCW генерируется с помощью инструмента `tlbimp.exe`, который поставляется вместе с Visual Studio как часть .NET Framework SDK. Следующий фрагмент показывает использование инструмента `tlbimp.`

exe для создания сборки RCW для интерфейсов Apple iTunes COM (объявляются в iTunes.exe). Интерфейсы iTunes COM позволяют реализовать управление программным обеспечением iTunes, например проигрыванием музыки или управлением библиотекой музыкальных произведений:

```
C:\Program Files\iTunes>tlbimp iTunes.exe /out:NETiTunesLibrary.dll
Microsoft (R) .NET Framework Type Library to Assembly Converter 3.5.21022.8
Copyright (C) Microsoft Corporation. All rights reserved.

Type library imported to C:\Program Files\iTunes\NETiTunesLibrary.dll
```

После создания сборки RCW ее можно использовать, как любую другую сборку .NET, и писать приложения, взаимодействующие с программами iTunes. В примере B.6 показан листинг сеанса FSI, где используется сборка RCW для iTunes API, чтобы вывести список альбомов из текущей библиотеки музыкальных произведений с наибольшим рейтингом.

Для этого создается экземпляр класса iTunesAppClass и выполняется обращение к его свойству LibrarySource. Все эти операции выполняются посредством интерфейсов COM, определенных в iTunes API, но благодаря сгенерированной сборке RCW они выглядят как обращения к любым другим сборкам .NET. Кроме того, если добавить ссылку на RCW в Visual Studio, вы получите полную поддержку механизма IntelliSense.

Пример B.6. Взаимодействие с COM

```
> // Необходимые ссылки на библиотеки
#r "System.Core.dll"
#r @"C:\Program Files\iTunes\NETiTunesLibrary.dll";;

--> Referenced 'C:\Program Files\Reference
Assemblies\Microsoft\Framework\v3.5\System.Core.dll'

--> Referenced 'C:\Program Files\iTunes\NETiTunesLibrary.dll'

> // Установить связь с приложением iTunes
open System.Linq
open NetiTunesLibrary

// Загрузить главную библиотеку iTunes
let iTunes = new iTunesAppClass()
let iTunesLibrary = iTunes.LibrarySource.Playlists.[1];;
Binding session to 'C:\Program Files\iTunes\NetiTunesLibrary.dll'...

val iTunes : NetiTunesLibrary.iTunesAppClass
val iTunesLibrary : NetiTunesLibrary.IITPlaylist

> // Выполнить поиск по библиотеке iTunes музыкальных произведений,
// отсортировать альбомы по средней величине рейтинга.
```

```
// Вернуть экземпляр типа seq< albumName * avgRating * tracksInAlbum>
let albumsByAverageRating : seq<string * float * seq<IITTrack>> =
    iTunesLibrary.Tracks.OfType<IITTrack>()
    |> Seq.groupBy(fun track -> track.Album)
    |> Seq.map(fun (albumName, tracks) ->
        // Получить средний рейтинг по альбому
        let trackRatings = tracks |> Seq.map (fun track -> float track.Rating)
        albumName, (Seq.average trackRatings), tracks);;

val albumsByAverageRating : seq<string * float * seq<IITTrack>>

> // Возвращает true, если число произведений в альбоме, имеющих оценку,
// превышает число произведений, не имеющих оценки
let mostTracksRated tracks =
    let rated, notRated =
        Seq.fold
            (fun (rated, notRated) (track : IITTrack) ->
                if track.Rating <> 0 then
                    (rated + 1, notRated)
                else
                    (rated, notRated + 1))
            (0, 0)
            tracks
    if rated > notRated then
        true
    else
        false

// Вывести альбомы с наибольшим рейтингом, отфильтровав альбомы, в которых
// оценено незначительное количество произведений
let printTopAlbums() =
    albumsByAverageRating
    |> Seq.filter(fun (_,_,tracks) -> mostTracksRated tracks)
    |> Seq.sortBy(fun (_, avgRating, _) -> -avgRating)
    |> Seq.iter(fun (albumName, avgRating, _) ->
        printfn
            "Album [%-18s] given an average rating of %.2f"
            albumName avgRating);;

val mostTracksRated : seq<IITTrack> -> bool
val printTopAlbums : unit -> unit

> printTopAlbums();;
Album [Aloha From Hawaii ] given an average rating of 100.00
Album [Sample This - EP  ] given an average rating of 95.00
Album [Dark Passion Play ] given an average rating of 83.08
Album [When I Pretend to Fall] given an average rating of 80.00
Album [Weezer (Red Album)] given an average rating of 78.30
Album [The Best Of The Alan Parsons Project] given an average rating of 76.36
Album [Karmacode         ] given an average rating of 64.00
val it : unit = ()
```

Взаимодействие с COM в .NET – это чрезвычайно сложная тема, и при его использовании требуется значительное внимание уделять таким деталям, как обеспечение безопасности, поддержка многопоточной модели выполнения и сборка мусора. К счастью, в языке F# отсутствуют какие-либо специфические требования к механизму COM, отличающиеся от требований в языке C#.

Если вам интересно поближе познакомиться с взаимодействием с COM, я рекомендую прочитать книгу «.NET and COM: The Complete Interoperability Guide» Адама Натана (Adam Nathan), выпущенную издательством Sams.

Алфавитный указатель

N

O

P

R

S

T

И

К

Л

М

Н

О

П

P

C

T