Сошников Д. В.

Функциональное программирование на F#

Электронное издание

Профобразование
Саратов • 2017

УДК 004.438F#
ББК 32.973.26-018.1
С54

Сошников Д. В.
С54 Функциональное программирование на F#. – Эл. изд. – Саратов: Профобразование, 2017. – 191 с.: ил.

ISBN 978-5-4488-0131-0

Автор этой книги имеет богатый опыт преподавания курсов функционального программирования на базе F# в ведущих российских университетах, в то же время, будучи технологическим евангелистом Майкрософт, он умеет доходчиво объяснить концепции функционального программирования даже начинающему разработчику ПО, не прибегая к сложным поняти-ям лямбда-исчисления.

Книга содержит много полезных примеров использования F# для решения практических задач: доступа к реляционным или слабоструктурированным XML-данным, использование F# для веб-разработки и веб-майнинга, визуализация данных и построение диаграмм, написание сервисов для облачных вычислений и асинхронных приложений для Windows Phone 7. Используя фрагменты кода, рассмотренные в книге, читатели могут немедленно приступить к решению своих практических задач на F#.

УДК 004.438F#
ББК 32.973.26-018.1

ISBN 978-5-4488-0131-0

Содержание

Предисловие Дона Сайма

F# является достаточно молодым языком программирования – его разработку мы начали в MicrosoftResearch в Кембридже 7 лет назад. С тех пор язык стал богатым и выразительным инструментом индустриального программирования, использующимся в различных областях, в особенности после того, как он был включен в стандартную поставку VisualStudio 2010.

В прошлом году мне довелось выступать с несколькими виртуальными докладами на конференциях по разработке ПО в России, и я убедился, что здесь есть значительный интерес к F#, как среди разработчиков ПО, так и в кругу любителей функционального программирования. Поэтому мне особенно приятно, что Дмитрий взял на себя тяжелый труд по написанию первой русскоязычной книги по F#.

F# – это интересный язык, поскольку он является не только эффективным инструментом для разработки коммерческого ПО на платформе .NET, но и очень удобным академическим языком для преподавания функционального программирования в университетах (использование для этой цели в качестве базового языка, который имеет индустриальную поддержку, имеет много преимуществ), а также исследовательским языком (на самом деле многие идеи для следующего поколения .NET-языков приходят из F#, взять хотя бы асинхронные блоки, агенты и типо-образующие классы). Дмитрий Сошников, автор этой книги, имеет богатый опыт преподавания курсов функционального программирования на базе F# в ведущих российских университетах, в то же время, будучи технологическим евангелистом Майкрософт, он умеет доходчиво объяснить концепции функционального программирования даже начинающему разработчику ПО, не прибегая к сложным понятиям лямбда-исчисления.

Эта книга в первую очередь ориентирована на практикующих программистов, которые найдут в ней хорошее введение в функциональное программирование. Затем в книге следует более глубокое изложение приемов функционального программирования, полезные главы по объектно-ориентированному, асинхронному и параллельному программированию на F#. Книга также содержит много полезных примеров использования F# для решения практических задач: доступа к реляционным или слабоструктурированным XML-данным, использование F# для веб-разработки (как на ASP.NETWebForms/MVC, так и с применением WebSharper) и веб-майнинга, визуализация данных и построение диаграмм, написание сервисов для облачных вычислений и асинхронных приложений для WindowsPhone 7. Используя фрагменты кода, рассмотренные в книге (а также доступные для скачивания со странички автора из Интернет), читатели могут немедленно приступить к решению своих практических задач на F#.

Я искренне надеюсь, что эта книга поможет раскрыть красоту, богатство и мощь языка F# для многих разработчиков ПО из России и других русско-говорящих стран.

Дон Сайм,
ведущий исследователь, MicrosoftResearch Cambridge,
создатель языка F#

Предисловие автора

Решение взяться за написание книги по F# было очень непростым. Во-первых, не было понятно, насколько такая книга будет востребована. Во-вторых, сложно конкурировать с уже появившейся полудюжиной англоязычных книг по этому языку. И наконец, написание книги – это очень трудоемкий процесс, учитывая, что он является далеко не единственным делом моей жизни.

Однако, встречаясь со многими студентами и разработчиками и выступая в вузах, я понял, что интерес к языку огромен, а необходимость в русскоязычных книгах имеется, поскольку не все начинающие изучать программирование хорошо владеют английским языком, – а хотелось показать прелести функционального программирования не только профессионалам, но и **всем**. Поэтому в какой-то момент решение было принято, а результатом стало то, что вы держите в руках.

Тем не менее мне не хотелось дублировать существующие англоязычные книги. Основной целью было создать небольшую книгу, которая позволит практикующим программистам и начинающим за короткий срок овладеть как основами функционального программирования, так и базовым синтаксисом языка F#. Для удобства в книге сначала рассматриваются базовые понятия, которые позволят вам быстро (после прочтения первых трех глав) начать писать на F# весьма нетривиальные программы и понимать чужой код. Последняя, седьмая глава содержит в себе множество коротких и лаконичных примеров и фрагментов кода, многие из которых вы можете использовать с минимальными изменениями для решения своих задач. Таким образом, после прочтения книги вы не только «расширите свое сознание», изучив еще один подход к программированию, но и пополните свой арсенал чрезвычайно мощным средством для решения различных задач обработки данных.

Создание этой книги было бы невозможно без участия многих моих друзей и коллег. Первоначальный интерес к функциональному программированию появился у меня в результате бесед с профессором В. Э. Вольфенгагеном и другими моими знакомыми и коллегами из «кибернетической школы» МИФИ, в частности А. В. Гавриловым и С. В. Зыковым; в дальнейшем он был подкреплен знакомством с Саймоном Пейтоном-Джонсом, одним из создателей языка Haskell, ныне работающим в Microsoft Research в Кембридже. После того как благодаря в первую очередь усилиям Дона Сайма F# был включен в состав Visual Studio 2010, я стал более плотно заниматься вопросами продвижения функционального программирования в обучение в рамках своих обязанностей как технологического евангелиста Майкрософт. Я хотел бы поблагодарить коллег из Новосибирска (ИСИ СО РАН, НГУ и НГТУ): А. Г. Марчука, Л. И. Городнюю и Н. В. Шилова за

плодотворную дискуссию на тему использования F# для преподавания в рамках семинара ИСИ СО РАН, окончательно убедившую меня в том, что F# может помочь решить благую задачу внедрения в учебный процесс курсов функционального программирования, которые при этом будут иметь значительную практическую направленность.

Мне посчастливилось поставить и прочитать такие курсы на базе F# в ведущих московских вузах: МФТИ и ГУ ВШЭ – за эту возможность я благодарен В. Е. Кривцову и С. М. Авдошину. Многие материалы книги основаны на этих курсах, которые я вел совместно с С. Лаптевым и С. В. Косиковым. Некоторые примеры были разработаны студентами Т. Абаевым (ГУ ВШЭ), А. Брагиным (МАИ), А. Мыльцевым (МФТИ). Благодаря заинтересованности и поддержке А. В. Шкреда видеокурс доступен в рамках интернет-университета информационных технологий ИНТУИТ.РУ.

За идею и за возможность издать книгу по языку F# я благодарен Д. А. Мовчану и сотрудникам «ДМК Пресс», причастным к подготовке книги. Я также благодарен моим друзьям и коллегам: С. Звездину (ЮУрГУ), В. Юневу, Ю. Трухину (ТвГТУ), которые любезно согласились прочитать рукопись и высказать свои пожелания и дополнения. Многие вопросы о целесообразности издания такой книги мы обсуждали с В. Е. Зайцевым (МАИ), а проблемы изложения основ функционального подхода – с моим другом Р. В. Шаминым (РУДН). Мне очень важна была также идеологическая поддержка создателя языка F# Дона Сайма (Microsoft Research Cambridge), который любезно согласился написать предисловие.

Наконец, хочу поблагодарить мою дочь Вики, которая регулярно терпеливо недополучала внимание отца, уходящее на написание этой книги. Очень хотел бы надеяться, что потраченные на книгу усилия того стоят и помогут зародить любовь к функциональному программированию и к языку F# в сердцах многих начинающих и уже профессионально практикующих разработчиков и архитекторов.

0. Введение

0.1. Зачем изучать функциональное программирование

Вы держите в руках книгу по новому языку программирования F#, которая также для многих будет путеводителем в новый мир функционального программирования. Как вы узнаете из этой книги, функциональное программирование – это вовсе не стиль программирования, в котором используются много функций, это – другая парадигма программирования, где нет переменных, где не может возникнуть побочный эффект и в которой можно писать более короткие программы, требующие меньшей отладки.

Для начала хотелось бы немного пояснить, зачем же изучать F# и функциональное программирование вообще. До недавнего времени считалось, что функциональные языки используются в основном в исследовательских проектах, поскольку для реальных задач они недостаточно производительны. Действительно, в таких языках в обилии используются сложные структуры данных с динамическим распределением памяти, применяется сборка мусора, реализован более сложный (но и более гибкий) механизм вызова функций и т. д. Кроме того, есть мнение, что только специалисты с ученой степенью способны в них разобраться.

Действительно, функциональные языки представляют из себя очень удобный аппарат для научных исследований в области теоретического программирования, а также инструмент быстрого прототипирования систем, связанных с обработкой данных. Однако можно привести и несколько больших и известных программных систем широкого назначения, реализованных на функциональных языках: среди них графические системы компании Autodesk (использующие диалект языка LISP), текстовый редактор GNU emacs и др. Однако подавляющее большинство промышленных программных систем остаются написанными на «классических» императивных языках типа C#, Java или C++.

Однако в последнее время наблюдается тенденция все большего проникновения функционального подхода в область индустриального программирования. Современные функциональные языки – такие как Haskell, Ocaml, Scheme, Erlang – приобретают все большую популярность. В довершение всего в недрах Microsoft Research на базе OCaml был разработан язык F# для платформы .NET, который было решено включить в базовую поставку Visual Studio 2010 наравне с традиционными языками C# и Visual Basic.NET. Это беспрецедентное решение открывает возможности функционального программирования для большого круга разработ-

чиков на платформе .NET, позволяя им разрабатывать фрагменты программных систем на разных языках в зависимости от решаемой задачи. В этой книге мы надеемся убедить наших читателей, что для многих задач обработки данных F# окажется более удобным языком. Аналогично появляется семейство «в значительной степени функциональных» языков на платформе Java: речь идет о Scala и Clojure.

Растущую популярность функционального подхода можно объяснить двумя факторами. Во-первых, препятствующие ранее распространению функциональных языков проблемы с производительностью перестают иметь важное значение. Действительно, сейчас подавляющее большинство современных языков используют сборку мусора, и это не вызывает нареканий. В современном мире проще слегка пожертвовать производительностью, но сэкономить на высокооплачиваемом труде программиста. Функциональный подход, как мы увидим далее, способствует более высокому уровню абстракции при написании программ, что ведет к большему уровню повторного использования кода, экономя время, идущее на разработку и отладку. Благодаря отсутствию побочных эффектов отладка еще больше упрощается.

Во-вторых, растет актуальность параллельного и асинхронного программирования. Поскольку закон Мура в его упрощенном понимании – скорость вычислений (частота процессора) удваивается каждые 18 месяцев – перестал действовать и увеличивается не частота, а количество доступных вычислительных ядер, возрастает актуальность написания распараллеливаемого кода. Однако на традиционных императивных языках – из-за наличия общей памяти – написание такого кода сопряжено со значительными сложностями, и одно из решений кроется именно в переходе к более функциональному стилю программирования с неизменяемыми данными.

Движение в сторону функционального стиля подтверждается не только появлением нового языка F# в инструментарии программиста. На самом деле множество функциональных особенностей появилось еще в C# 3.0 (а следом и в Java. next, и в новом стандарте C++) – это и вывод типов, и лямбда-выражения, и целое функциональное ядро LINQ внутри языка, и анонимные классы. Многие уже испытали на себе возможности по эффективному распараллеливанию функциональных LINQ-запросов, когда почти вся работа берется на себя инфраструктурой Parallel LINQ и добавление одного вызова `.AsParallel` приводит к автоматическому ускорению работы программы на многоядерном процессоре.

Подытоживая – какую практическую пользу может извлечь для себя разработчик, изучив F#? Одна из особенностей F# и функциональных языков в целом, которую мы надеемся продемонстрировать на протяжении книги, состоит в том, что они позволяют выражать свои мысли короче. Иными словами, функциональному программисту приходится больше думать, но меньше писать кода и меньше отлаживать. Мне как автору этой книги нравится думать, и я хочу показать вам, как можно думать «по-другому», в функциональном стиле. Если вы тоже разделяете мои пристрастия – добро пожаловать в мир функционального программирования!

Конечно, не для всех задач F# окажется удобным инструментом. Visual Studio не будет поддерживать F# вместе с визуальными инструментами создания приложе-

ний Windows Forms или веб-приложений ASP.NET – то есть при визуальном создании приложений по-прежнему будут доступны лишь классические императивные языки. Однако благодаря прозрачной интероперабельности F# с другими языками платформы .NET в функциональном стиле будет удобно реализовывать многие задачи обработки данных, построение синтаксических анализаторов, различные вычислительные алгоритмы и конечно же параллельный и асинхронный код.

Книга будет полезна вам даже в том случае, если вы решите не использовать F# в своих разработках или у вас не будет такой возможности. Функциональный подход – это другой, отличный от привычного нам императивного, подход к программированию, он в некотором смысле «расширяет сознание» и учит смотреть на какие-то вещи по-новому. Хотел бы надеяться, что, обогатив свое сознание функциональным подходом, при написании обычных программ вы будете делать это немного по-другому, в функциональном стиле.

0.2. О чем и для кого эта книга

С момента объявления о том, что F# войдет в состав Visual Studio 2010, интерес к этому языку только увеличивается. По сути дела, F# перестал быть чисто академическим языком «для ученых», им начинают интересоваться практикующие разработчики. В то время как англоязычная литература по F# уже несколько лет доступна, русскоязычных ресурсов катастрофически не хватает.

Основная цель этой книги – доступно изложить основы фукнционального программирования для разработчиков, одновременно знакомясь с базовым синтаксисом языка F#, что позволяет в результате прочтения книги сделать этот язык своим рабочим инструментом для решения ряда практических задач. В этой книге мы постарались, с одной стороны, не вдаваться слишком глубоко в теоретические основы функционального программирования (лямбда-исчисление, теорию категорий, системы типов и т. д.), а с другой – не ставили целью исчерпывающим образом изложить все конструкции и тонкости F#. Мы надеемся, что читатель, вдохновленный нашей книгой, начнет самостоятельно экспериментировать с языком и в случае необходимости, хорошо понимая базовые понятия, сможет разобраться с деталями по соответствующим англоязычным источникам (в первую очередь мы намекаем на книги Дона Сайма Expert F# 2.0 [1] и Криса Смита Programming F# [2] (которая, кстати, совсем скоро будет доступна в русском переводе). С другой стороны, тех из вас, кого заинтересовал сам предмет функционального программирования, нам бы хотелось отослать к более классическому учебнику Филда и Харрисона [11], издававшемуся на русском языке издательством «Мир», либо же к видеокурсу функционального программирования в интернет-университете ИНТУИТ.РУ [7], который автор читал для студентов ФИВТ МФТИ.

Для повышения практической привлекательности книги мы также постарались в конце привести несколько типовых примеров использования F# для решения практических задач на платформе .NET. Вы можете применять содержащийся в примерах код как отправную точку для реализации собственных проектов обработки данных на F#.

0.3. Как установить и начать использовать F#

Самый лучший способ изучать F# – это начать им пользоваться. Установите себе систему программирования на F#, начните (продолжите) читать эту книгу и попробуйте параллельно порешать на F# несколько простых задач, например из проекта Эйлера: http://projecteuler.net – на этом сайте приводится целый список задач от простых к более сложным, которые предлагается использовать для постепенного овладевания навыками программирования. Что приятно – поискав в Интернете, вы найдете множество решений задач проекта Эйлера на F# и сможете сравнить их с тем, что получается у вас. Другим хорошим источником фрагментов кода (code snippets) на F# будет сайт http://fssnip.net – небольшие кусочки кода там разбиты по категориям, причем вы не только сможете смотреть и использовать готовые фрагменты, но и помещать на сайт свои достижения по мере того, как будете овладевать языком.

Итак, поговорим о том, как установить F#. Поскольку способы установки, версии и ссылки со временем меняются, мы рекомендуем вам следовать инструкции, расположенной в Интернете на странице автора по адресу http://www.soshnikov.com/fsharp. Здесь же мы рассмотрим только краткие особенности установки.

Наверное, самый правильный способ – это использовать последнюю версию Visual Studio (на момент выхода книги это версия Visual Studio 2010), которая уже содержит в себе F#. Возможно также установить F# поверх Visual Studio 2008, скачав самый последний Community Technology Preview (CTP). Если же вы не обладаете лицензией на Visual Studio 2008 или 2010 (хочу отметить, что все студенты могут получить такую лицензию в рамках программы DreamSpark, www.DreamSpark.ru), то вы можете установить свободно распространяемую оболочку Visual Studio Shell и поставить поверх нее F# CTP для Visual Studio 2008.

Для работы большинства примеров, рассматриваемых в этой книге, вам потребуется так называемый F# Power Pack – это свободно распространяемый набор дополнительных библиотек к F#, доступный в исходных кодах на сайте http://fsharppowerpack.codeplex.com.

Существуют два способа использования F# в Visual Studio:

- создав отдельный проект на F#: F# Library (библиотеку) или F# Application (приложение). Если в Visual Studio установлен F#, то при создании нового проекта вам будет доступна соответствующая опция (см. рис. 0.1). В этом случае при компиляции проекта будет создана соответствующая сборка или выполняемый файл. Все файлы проекта при этом имеют расширение `.fs`;
- в интерактивном режиме вы можете вводить текст на F# и немедленно выполнять его в специальном окне F# Interactive в Visual Studio (см. нижнее окно на рис. 0.2). Такой режим интерпретации удобно использовать при первоначальном создании алгоритма, чтобы немедленно видеть результаты своей работы и по ходу дела менять алгоритм. В этом случае компиляция и

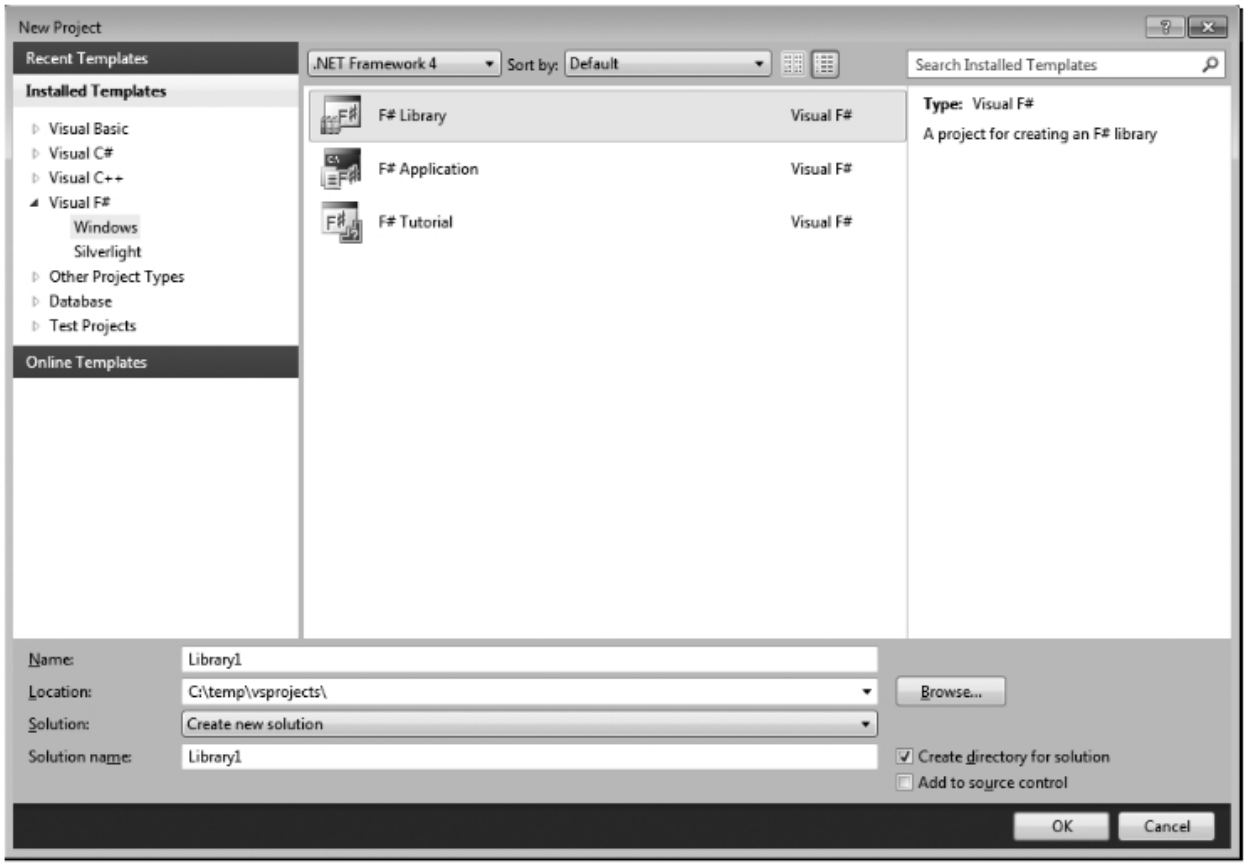

Рис. 0.1. Создание F#-проекта в Visual Studio 2010

создание промежуточной сборки происходят «на лету», а программист работает в интерактивном режиме как бы в рамках одного сеанса. Если окно F# Interactive при запуске отсутствует, надо открыть его, выбрав пункты меню **View ⇨ Other Windows ⇨ F# Interactive**.

Поскольку текст в окне F# Interactive не сохраняется в файл, то обычно удобно использовать отдельный файл с текстом программы, так называемый F# Script файл с расширением `.fsx`, открытый в основном окне кода Visual Studio (см. рис. 0.2). В этом случае для выполнения фрагмента кода в окне F# Interactive нужно этот фрагмент выделить и нажать Alt-Enter – результат выполнения появится в нижнем окне F# Interactive.

Для режима интерпретации (а если быть строгим – то псевдоинтерпретации) существует также отдельная утилита командной строки `fsi.exe`, позволяющая запускать F# вне Visual Studio. В этом случае вам недоступны многие полезные возможности по редактированию кода (подсветка кода, автоматическое дополнение IntelliSense и т. д.), но возможности языка от этого не меняются. Также существует компилятор командной строки `fsc.exe`, предназначенный для автоматической компиляции из командного или make-файла.

Возможно также использовать F# в UNIX-подобных системах, в которых поддерживается среда Mono. Загрузив F# для Visual Studio 2008 и разархивировав дистрибутив на диск, вы найдете там файл `install-mono.sh`, который необходимо

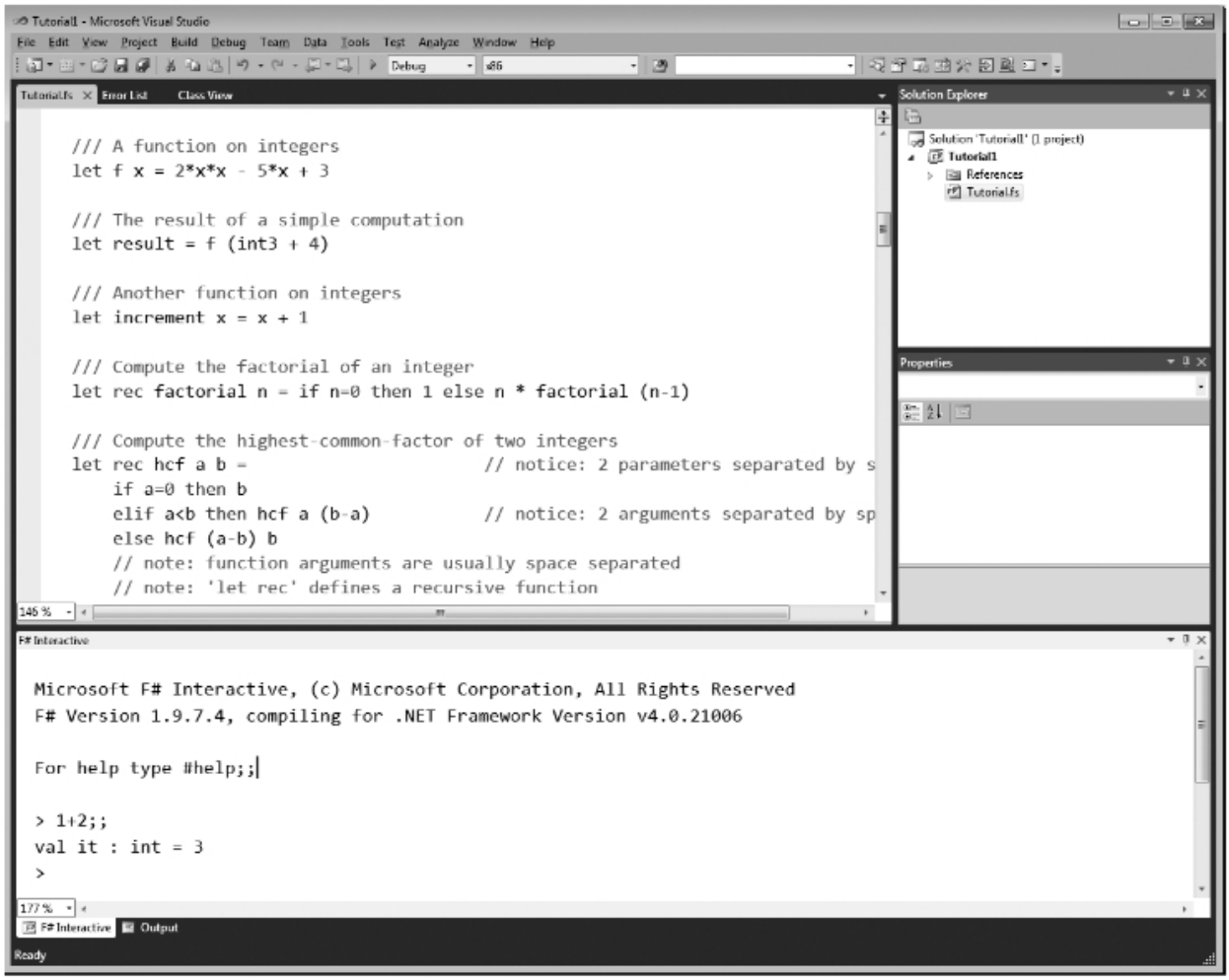

Рис. 0.2. Окно псевдо-интерпретатора F# Interactive

запустить для проведения установки. После этого вам станут доступны командные утилиты `fsc.exe` и `fsi.exe`, описанные ранее.

Большинство примеров, приведенных в книге, вам будет проще всего запускать из скриптовых файлов `.fsx` в режиме интерпретации непосредственно из среды Visual Studio. Исходный код всех примеров, рассмотренных в книге, вы сможете скачать с сайта автора по адресу http://www.soshnikov.com/fsharp.

В режиме интерпретации F# позволяет вам ввести выражение, которое затем он пытается вычислить и выдать результат. Вот пример простого диалога, вычисляющего арифметическое выражение:

```
> 1+2;;
val it : int = 3
```

Здесь жирным выделен текст, вводимый пользователем (после знака >), остальное – приглашение и ответ интерпретатора. Вот чуть более сложный пример решения квадратного уравнения:

```
> let a,b,c = 1.0, 2.0, -3.0;;
val c : float = -3.0
val b : float = 2.0
val a : float = 1.0
> let d = b*b-4.*a*c;;
val d : float = 16.0
> let x1,x2 = (-b+sqrt(d))/2./a,(-b-sqrt(d))/2./a;;
val x2 : float = -3.0
val x1 : float = 1.0
```

Что означает этот код, как правильно его понимать и как научиться писать такой же, мы с вами поговорим в следующей главе.

1. Основы функционального программирования

В этой главе мы ставим себе амбициозные задачи – изложить основные идеи функционального программирования, одновременно познакомив вас с базовыми конструкциями F#.

1.1. Применение функций vs. Присваивание

Вот что пишет одна авторитетная веб-энциклопедия про функциональное программирование:

> *Функциональное программирование – это раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательность изменения состояний. Функциональное программирование не предполагает изменяемость данных (в отличие от императивного, где одной из базовых концепций является переменная).*

Это определение напоминает случай, который произошел с автором, когда он был молодым и преподавал программирование на первом курсе факультета прикладной математики МАИ. Один из студентов никак не мог понять, что значит X:=X+1. «Как же так, как X может быть равен X+1?» Пришлось объяснить ему, как такое возможно, и в этот момент в нем умер функциональный программист.

Таким образом, поскольку большинство наших читателей наверняка владеют навыками традиционного, императивного программирования, то придется решать обратную задачу – объяснять, почему X не может быть равен X+1. Точнее, почему в функциональном программировании отсутствует присваивание и как возможно записывать алгоритмы без него. Попробуем разобраться!

Императивные языки программирования произошли из необходимости записывать в более удобном виде инструкции для ЭВМ. Обратимся к архитектуре компьютера, которая превалировала в 50-е годы прошлого века (архитектуре фон Неймана) и которая используется до сих пор. Основными компонентами компьютера являются память (разбитая на пронумерованные ячейки), содержащая как программу, так и данные, и центральный процессор, способный выполнять при-

митивные команды вроде арифметических операций и переходов. Таким образом, основным шагом работы программы является некоторое действие (команда, оператор), которое определенным образом изменяет состояние памяти. Например, команда сложения может взять содержимое одной ячейки, сложить его с содержимым другой ячейки и поместить результат в третью ячейку[1] – на языке высокого уровня[2] это запишется как X:=Y+Z. Здесь понятие переменной (обозначаемой некоторыми буквенными идентификаторами), по сути дела, соответствует понятию ячейки памяти (или нескольким ячейкам, необходимым для хранения значения какого-то типа данных).

Соответственно, в записи X:=X+1, с такой точки зрения, нет ничего странного – мы берем содержимое некоторой ячейки, увеличиваем на единицу и сохраняем получившееся значение в той же ячейке памяти. Такое последовательное увеличение значений некоторой переменной является типичным приемом императивного программирования, называемым инкрементом, и используемым, например, в цикле со счетчиком.

Подобный стиль программирования, основанный на присваиваниях и последовательном изменении состояния, является естественным для ЭВМ и благодаря заложенным нам с юных лет основам программирования стал естественным и для нас. Однако возможны и другие подходы к программированию, изначально более естественные для человека, обладающие большей математической строгостью и красотой. К ним относится функциональное программирование.

Представим себе математика, которому нужно решить некоторую задачу. Обычно задача формулируется как необходимость вычислить некоторый результат по имеющимся входным данным. В самом простейшем случае такое вычисление может задаваться обычным арифметическим выражением, например для нахождения одного корня квадратного уравнения $x^2 + 2x - 3$ существует явная формула, которую можно записать на F# следующим образом:

```
(-2.0+sqrt(2.0*2.0-4.0*(-3.0))) / 2.0 ;;
```

Если такое выражение ввести в ответ на приглашение интерпретатора F#, то мы получим искомый результат:

```
> (-2.0+sqrt(2.0*2.0-4.0*(-3.0))) / 2.0 ;;
val it : float = 1.0
```

Двойная точка с запятой в конце свидетельствует о том, что набранный текст можно передавать на исполнение интерпретатору. Отдельные же выражения можно разделять точкой с запятой или переходом на новую строку.

[1] На самом деле команды процессора обычно более примитивные и оперируют только одним операндом в памяти, но для понимания материала это в данный момент не существенно.

[2] В данном случае мы используем синтаксис, похожий на язык Паскаль, чтобы подчеркнуть отличие оператора присваивания := от равенства =.

Если вы параллельно с чтением книги экспериментируете на компьютере – поздравляю, вы только что написали свою первую функциональную программу!

Обычно, конечно, задача не может быть решена одним лишь выражением. На деле при рассмотрении решения квадратного уравнения сначала вычисляют дескриминант D = b^2 – 4ac (используя для его обозначения некоторую букву или **имя**, D) и затем уже – сами корни. В математических терминах пишут:

$x_1 = (-b + \sqrt{D})/(2a)$, где $D = b^2 - 4ac$,

или

пусть $D = b^2 - 4ac$, тогда $x_1 = (-b + \sqrt{D})/(2a)$.

На языке F# соответствующая запись примет следующий вид:

```
let D = 2.0*2.0-4.0*(-3.0) in (-2.0+sqrt(D)) / 2.0 ;;
```

Здесь `let` обозначает введение именованного обозначения – в следующем за `in` выражении буква D будет обозначать соответствующую формулу. Изменить значение D (в той же области видимости) уже невозможно.

С использованием let можно описать решение уравнения следующим образом:

```
let a = 1.0 in
 let b = 2.0 in
  let c = -3.0 in
   let D = b*b-4.*a*c in
    (-b+sqrt(D)) / (2.*a) ;;
```

Безусловно, не все задачи решаются «в одну строчку» выписыванием формулы с ответом. Однако ключевым здесь является сам подход к решению задачи – вместо переменных и присваиваний мы пытаемся выписать некоторое выражение (применение функции к исходным данным) для решения задачи, используя по мере необходимости другие выражения (и функции), определенные в программе. По мере прочтения этой главы вы поймете, что с таким подходом можно решать весьма сложные задачи!

1.2. Упорядоченные кортежи, списки и вывод типов

Приведенный выше пример позволял нам вычислить лишь один корень квадратного уравнения. Для вычисления второго корня при таком подходе нам пришлось бы выписать аналогичное выражение, заменив в одном месте «+» на «–». Безусловно, такое дублирование кода не является допустимым.

В данном случае проблема легко решается использованием *пары значений*, или, более строго, *упорядоченного кортежа* (tuple) как результата вычислений. Упорядоченный набор значений является базовым элементом функционального

языка, и с его использованием выражение для нахождения обоих корней уравнения запишется так:

```
let a = 1.0 in
 let b = 2.0 in
  let c = -3.0 in
   let D = b*b-4.*a*c in
    ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a)) ;;
```

В результате мы получим такой ответ системы:

```
val it : float * float = (1.0, -3.0)
```

Здесь `it` – это специальная переменная, содержащая в себе результат последнего вычисленного выражения, а `float*float` – тип данных результата, в данном случае декартово произведение float на float, то есть пара значений вещественного типа.

Мы видим, что компилятор способен самостоятельно определить тип выражения – это называется *автоматическим выводом типов*. Вывод типов – это одна из причин, по которой программы на функциональных языках выглядят так компактно – ведь практически никогда не приходится в явном виде указывать типы значений для вновь описываемых имен и функций.

Помимо упорядоченных кортежей, F# содержит также встроенный синтаксис для работы со *списками* – последовательностями значений одного типа. Мы могли бы вернуть список решений (вместо пары решений), используя следующий синтаксис:

```
let a = 1.0 in
 let b = 2.0 in
  let c = -3.0 in
   let D = b*b-4.*a*c in
    [(-b+sqrt(D))/(2.*a);(-b-sqrt(D))/(2.*a)];;
```

Результат в этом случае выглядел бы так:

```
val it : float list = [1.0; -3.0]
```

Здесь `float list` – это список значений типа `float`. Суффикс `list` применим к любому типу и представляет собой описание полиморфного типа данных. Подробнее о списках мы расскажем позднее в главе 2.

1.3. Функциональные типы и описание функций

Операция решения квадратного уравнения является достаточно типовой и вполне может пригодиться нам в дальнейшем при написании довольно сложной программы. Поэтому было бы естественно иметь возможность описать процесс ре-

шения квадратного уравнения как самостоятельную функцию. Наверное, вы уже догадались, что на вход она будет принимать тройку аргументов – коэффициенты уравнения, а на выходе генерировать пару чисел – два корня. Описание функции и ее применение для решения конкретного уравнения будут выглядеть следующим образом:

```
let solve (a,b,c) =
   let D = b*b-4.*a*c in
    ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a))
 in solve (1.0,2.0,-3.0);;
```

Здесь сначала определяется функция solve, внутри нее определяется локальное имя D (локальное – это значит, что вне функции оно недоступно), а затем эта функция применяется для решения исходного уравнения с коэффициентами 1, 2 и –3.

Обратите внимание, что для описания функции используется тот же самый оператор `let`, что и для определения имен. На самом деле в функциональном программировании функции являются базовым типом данных (как еще говорят – *first-class citizens*), и вообще говоря, различия между данными и функциями делаются минимальные[1]. В частности, функции можно передавать в качестве параметров другим функциям и возвращать в качестве результата, можно описывать функциональные константы и т. д.

Для удобства в F# применяется также специальный синтаксис (так называемый #light-синтаксис), в котором можно опускать конструкцию `in`, просто записывая описания функций последовательно друг за другом. Вложение конструкций в этом случае будет определяться отступами – если выражение записано с отступом по сравнению с предыдущей строчкой, то оно является вложенным по отношению к нему, локальным. В таком синтаксисе приведенный пример запишется так:

```
let solve (a,b,c) =
   let D = b*b-4.*a*c
   ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a));
solve (1.0,2.0,-3.0);;
```

Интересно посмотреть, какой тип данных в этом случае будет иметь функция `solve`. Как вы, наверное, догадались, она отображает тройки значений `float` в пары решений, что в нашем случае запишется как `float*float*float -> float*float`. Стрелка означает так называемый *функциональный тип* – то есть функцию, отображающую одно множество значений в другое.

В F# также существует конструкция для описания константы функционального типа, или так называемое *лямбда-выражение*. Свое название оно получило от

[1] В чистом λ-исчислении, которое лежит в основе функционального программирования, вообще нет различий между данными и функциями.

лямбда-исчисления, математической теории, лежащей в основе функционального программирования. В лямбда-исчислении, чтобы описать функцию, вычисляющую выражение x^2+1, используется нотация $\lambda x.x^2+1$. Аналогичная запись на F# будет выглядеть так:

```
fun x -> x*x+1
function x -> x*x+1
```

В данном случае обе эти записи эквивалентны, хотя в будущем мы расскажем о некоторых различиях между `fun` и `function`. С использованием приведенной нотации наш пример можно также переписать следующим образом:

```
let solve = fun (a,b,c) ->
   let D = b*b-4.*a*c
   ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a));
```

1.4. Каррирование

Часто, как в нашем прошлом примере, бывает необходимо описать функцию с несколькими аргументами. В лямбда-исчислении и в функциональном программировании мы всегда оперируем функциями от одного аргумента, который, однако, может иметь сложную природу. Как в прошлом примере, всегда можно передать в функцию в качестве аргумента кортеж, тем самым передав множество значений входных параметров.

Однако в функциональном программировании распространен и другой прием, называемый *каррированием*. Рассмотрим функцию от двух аргументов, например сложение. Ее можно описать на F# двумя способами:

```
let plus (x,y) = x+y
let cplus x y = x+y
```

Первый случай похож на рассмотренный ранее пример, и функция `plus` будет иметь тип `int*int -> int`. Второй случай – это как раз каррированное описание функции, и cplus будет иметь тип `int -> int -> int`, что на самом деле, используя соглашение о расстановке скобок в записи функционального типа, означает `int -> (int -> int)`.

Смысл каррированного описания – в том, что функция сложения применяется к своим аргументам «по очереди». Предположим, нам надо вычислить `cplus 1 2` (применение функции к аргументам в F# записывается как и в лямбда-исчислении, без скобок, простым указанием аргументов вслед за именем функции). Применяя cplus к первому аргументу, мы получаем значение функционального типа `int->int` – функцию, которая прибавляет единицу к своему аргументу. Применяя затем эту функцию к числу 2, мы получаем искомый результат 3 – целого типа. Запись plus 1 2, таким образом, рассматривается как (plus 1) 2, то есть сначала мы

получим функцию инкремента, а потом применим ее к числу 2, получив требуемый результат. В частности, все стандартные операции могут быть использованы в каррированной форме путем заключения операции в скобки и использования префиксной записи, например:

```
(+) 1 2;;
let incr = (+)1;;
```

В нашем примере с квадратным уравнением мы также могли бы описать каррированный вариант функции solve:

```
let solve a b c =
   let D = b*b-4.*a*c
   ((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a));;
solve 1.0 2.0 -3.0;;
```

Такой подход имеет как минимум одно преимущество – с его помощью можно легко описать функцию решения линейных уравнений как частный случай решения квадратных при a = 0:

```
let solve_lin = solve 0.0;;
```

Правда, вдумчивый читатель сразу заметит, что наша функция решения не предназначена для использования в ситуациях, когда a = 0, – в этом случае будет происходить деление на 0. Как расширить функцию solve для правильной обработки различных ситуаций, мы узнаем в следующем разделе.

1.5. Условный оператор и опциональный тип

Что будет, если использовать описанную нами функцию для решения уравнения, у которого нет корней? Программист на языках типа C#, наверное, ожидает, что будет сгенерировано исключение, возникающее при попытке извлечь корень из отрицательного числа. На F# в данном случае все несколько иначе – функция корректно работает, но возвращает результат `(nan,nan)`, то есть пару значений *not-a-number*, свидетельствующих об ошибке в арифметической операции с типом `float`.

Конечно, программисту было бы правильнее отдельно обрабатывать такие случаи и возвращать некоторое осмысленное значение, которое позволяет определить, что же произошло внутри функции. В нашем примере для правильного описания функции `solve` необходимо отдельно рассмотреть случай D < 0, при котором корней нет. Для этого уместно воспользоваться условным выражением, которое имеет вид:

```
if <логическое выражение> then <выражение-1> else <выражение-2>
```

Обратите внимание, что речь идет именно о *выражении*, а не об операторе: приведенное выражение возвращает значение выражения-1, если логическое выражение истинно, и значение выражения-2 в противном случае. Отсюда следует, что if-выражение не может употребляться без else-ветки[1], так как в этом случае не очень понятно, что возвращать в качестве результата, а также что типы данных обоих выражений должны совпадать. Знатоки Си-подобных языков (куда входят также C++, C#, Java и др.), наверное, уже поняли, что условный оператор в F# больше всего напоминает тернарный условный оператор ?:.

В нашем случае не очень понятно, какое значение возвращать из функции solve в том случае, когда решений нет. Можно, конечно, придумать какое-то выделенное значение (–9999), которое будет означать отсутствие решений, но такой прием по нескольким причинам не является хорошим. В идеале нам хотелось бы иметь возможность строить такой полиморфный тип данных, который в одном случае позволял бы возвращать значения базового типа, а в другом – специальный флаг, говорящий о том, что возвращать нечего (или что возвращается некоторое «пустое» значение).

Поскольку такая ситуация возникает достаточно часто, то соответствующий тип данных присутствует в языке и называется *опциональным типом* (option type). Например, значения типа `int option` могут содержать в себе либо конструкцию `Some(…)` от некоторого целого числа, либо специальную константу `None`. В нашем случае функция решения уравнения, возвращающая опициональный тип, будет описываться так:

```
let solve a b c =
   let D = b*b-4.*a*c
   if D<0. then None
   else Some(((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a)));;
```

Сама функция в этом случае будет иметь тип `solve : float -> float -> float -> (float * float) option` – этот тип будет выведен компилятором автоматически. Работать с опциональным типом можно примерно следующим образом:

```
let res = solve 1.0 2.0 3.0 in
 if res = None
   then "Нет решений"
   else Option.get(res).ToString();;
```

[1] Строго говоря, существует случай, когда в if-выражении можно опускать ветку else, – когда выражение имеет тип `unit`.

1.6. Типы данных, размеченное объединение и сопоставление с образцом

На самом деле опциональный тип представляет собой частный случай типа данных, называемого *размеченным объединением* (discriminated union). Он мог бы быть описан на F# следующим образом:

```
type 'a option = Some of 'a | None
```

В нашем примере, чтобы описать более общий случай решения как квадратных, так и линейных уравнений, мы опишем решение в виде объединения трех различных случаев: отсутствие решений, два корня квадратного уравнения и один корень линейного уравнения:

```
type SolveResult =
   None
 | Linear of float
 | Quadratic of float*float
```

В данном случае мы описываем тип данных, который может содержать либо значение `None`, либо `Linear(...)` с одним аргументом типа `float`, либо `Quadratic(...)` с двумя аргументами. Сама функция решения уравнения в общем случае будет иметь такой вид:

```
let solve a b c =
   let D = b*b-4.*a*c
   if a=0. then
     if b=0. then None
     else Linear(-c/b)
   else
     if D<0. then None
     else Quadratic(((-b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a)))
```

Для определения того, какой именно результат вернула функция `solve`, необходимо воспользоваться специальной конструкцией *сопоставления с образцом* (pattern matching):

```
let res = solve 1.0 2.0 3.0
match res with
  None -> printf "Нет решений"
| Linear(x) -> printf "Линейное уравнение, корень: %f" x
| Quadratic(x1,x2) -> printf "Квадратное уравнение, корни: %f %f" x1 x2
```

Операция match осуществляет последовательное сопоставление значения выражения с указанными шаблонами, при этом при первом совпадении вычисляется и возвращается соответствующее выражение, указанное после стрелки. В процессе сопоставления также происходит связывание имен переменных в шаблоне с соответствующими значениями. Возможно также указание более сложных условных выражений после шаблона, например:

```
match res with
  None -> printf "Нет решений"
| Linear(x) -> printf "Линейное уравнение, корень: %f" x
| Quadratic(x1,x2) when x1=x2 -> printf "Квадр.уравнение,1 корень: %f" x1
| Quadratic(x1,x2) -> printf "Квадр. уравнение,2 корня: %f %f" x1 x2
```

Следует отметить, что сопоставление с образцом в F# может производиться не только в рамках конструкции match, но и при сопоставлении имен let и при описании функциональной константы с помощью ключевого слова function. В частности, следующие два описания функции получения текстового результата решения уравнения text_res эквивалентны:

```
let text_res x = match x with
  None -> "Нет решений"
| Linear(x) -> "Линейное уравнение, корень: "+x.ToString()
| Quadratic(x1,x2) when x1=x2 ->
"Квад.уравнение, один корень: "+x1.ToString()
| Quadratic(x1,x2) ->
"Квадратное уравнение, два корня:"+x1.ToString()+x2.ToString()

let text_res = function
  None -> "Нет решений"
| Linear(x) -> "Линейное уравнение, корень: "+x.ToString()
| Quadratic(x1,x2) when x1=x2 ->
"Квадратное уравнение, один корень: "+x1.ToString()
| Quadratic(x1,x2) ->
"Квадратное уравнение, два корня:"+x1.ToString()+x2.ToString()
```

Наиболее часто распространенным примером использования конструкции сопоставления с образцом внутри let является одновременное сопоставление нескольких имен, например:

```
let x1,x2 = -b+sqrt(D))/(2.*a),(-b-sqrt(D))/(2.*a)
```

В данном случае на самом деле происходит сопоставление одной упорядоченной пары типа float с другой упорядоченной парой, что приводит к попарному сопоставлению обоих имен.

1.7. Рекурсия, функции-параметры и цикл for

В любом языке программирования одна из важнейших задач – выполнение повторяющихся действий. В императивных языках программирования для этого используются циклы (с предусловием, со счетчиком и т. д.), однако циклы основаны на изменении некоторого значения (счетчика цикла, или некоторого условия) при каждом проходе, и поэтому не могут быть напрямую использованы в функциональном подходе. Здесь нам на помощь приходит понятие *рекурсии*.

Например, рассмотрим простейшую задачу – печать всех целых чисел от A до B. Для решения задачи при помощи рекурсии мы думаем, как на каждом шаге выполнить одно действие (печать первого числа, A), после чего свести задачу к применению такой же функции (печать всех чисел от A + 1 до B). В данном случае получится такое решение:

```
let rec print_ab A B =
  if A>=B then printf "%d " A
  else
    printf "%d " A
    print_ab (A+1) B
```

Здесь ключевое слово `rec` указывает на то, что производится описание рекурсивных действий. Это позволяет правильно проинтерпретировать ссылку на функцию с тем же именем `print_ab`, расположенную в правой части определения. Без ключевого слова `rec` компилятор пытался бы найти имя `print_ab`, определенное в более высокой области видимости, и связать новое имя `print_ab` с другим выражением для более узкого фрагмента кода.

Очевидно, что решать каждый раз задачу выполнения повторяющихся действий с помощью рекурсии, описывая отдельную функцию, неудобно. Поэтому мы можем выделить идею итерации как отдельную абстракцию, а выполняемое действие передавать в функцию в качестве параметра. В этом случае мы получим следующее описание функции итерации:

```
let rec for_loop f A B =
  if A>=B then f A
  else
    f A
    for_loop f (A+1) B
```

Такое абстрактное описание понятия итерации мы теперь можем применить для печати значений от 1 до 10 следующим образом:

```
for_loop (fun x -> printf "%d " x) 1 10
```

Здесь мы передаем в тело цикла функциональную константу, описанную здесь же при помощи лямбда-выражения. В результате получившаяся конструкция напоминает обычный цикл со счетчиком, однако важно понимать отличия: здесь тело цикла представляет собой функцию, вызываемую с различными последовательными значениями счетчика, что, например, исключает возможную модификацию счетчика внутри тела цикла.

На самом деле цикл со счетчиком в такой интерпретации достаточно часто используется, поэтому в F# для его реализации есть специальная встроенная конструкция `for`. Например, для печати чисел от 1 до 10 ее можно использовать следующим образом:

```
for x=1 to 10 do printf "%d " x
for x in 1..10 do printf "%d " x
```

В качестве еще одного примера использования рекурсии рассмотрим определение функции `rpt`, которая будет возводить заданную функцию $f(x)$ в указанную степень n, то есть строить вычисление n-кратного применения функции f к аргументу x:

$$rpt\ n\ f\ x = f(f(...f(x)..)$$

Для описания такой функции вспомним, что $f^0(x) = x$ и $f^n(x) = f(f^{n-1}(x))$, тогда рекурсивное определение получится естественным образом:

```
let rec rpt n f x =
    if n=0 then x
    else f (rpt (n-1) f x)
```

1.8. Конструкции >>, |>

В приведенном выше определении мы рассматривали функцию `rpt` применительно к некоторому аргументу x. Однако мы могли рассуждать в терминах функций, не опускаясь до уровня применения функции к конкретному аргументу. Заметим, что исходное рекуррентное определение можно записать так:

$f^0 = Id$

$f^n = f \circ f^{n-1}$

Здесь Id обозначает тождественную функцию, а знак ∘ – композицию функций. Такое рекуррентное соотношение на F# может быть записано следующим образом:

```
let rec rpt n f =
    if n=0 then fun x->x
    else f >> (rpt (n-1) f)
```

В этом определении знак >> описывает композицию функций. Хотя эта операция является встроенной в библиотеку F#, она может быть определена следующим образом:

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

Помимо композиции, есть еще одна аналогичная конструкция |>, которая называется *конвейером* (pipeline) и определяется следующим образом:

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

С помощью конвейера можно последовательно передавать результаты вычисления одной функции на вход другой, например (возвращаясь к решению квадратного уравнения):

```
solve 1.0 2.0 3.0 |> text_res |> System.Console.Write
```

В этом случае результат решения типа `SolveResult` подается на вход функции `text_res`, которая преобразует его в строку, выводимую на экран системным вызовом `Console.Write`. Такой же пример мог бы быть записан без использования конвейера следующим образом:

```
System.Console.Write(text_res(solve 1.0 2.0 3.0))
```

Очевидно, что в случае последовательного применения значительного количества функций синтаксис конвейера оказывается более удобным. Следует отметить, что в F# для удобства также предусмотрены обратные операторы конвейера <| и композиции <<.

1.9. Пример – построение множества Мандельброта

В качестве примера использования всех изученных конструкций F# рассмотрим более сложную задачу – построение фрактального изображения, знаменитого множества Мандельброта. Математически это множество определяется следующим образом: рассмотрим последовательность комплексных чисел $z_{n+1} = z_n^2 + c$, $z_0 = 0$. Для различных c эта последовательность либо сходится, либо расходится. Например, для $c = 0$ все элементы последовательности $z_i = 0$, а для $c = 2$ имеем расходящуюся последовательность. Множество Мандельброта – это множество тех c, для которых последовательность сходится.

Приступим к реализации алгоритма построения на F#. Для начала определим функцию `mandelf`, описывающую последовательность $z^2 + c$, – при этом необходимо в явном виде указать для аргументов тип `Complex`[1], поскольку по умолчанию для операции + полагается целый тип. Кроме того, чтобы тип `Complex` стал доступен, вначале придется указать преамбулу, открывающую соответствующие модули:

[1] Строго говоря, это достаточно сделать хотя бы для одной из переменных, но мы для симметрии сделали для двух.

```
open System
open Microsoft.FSharp.Math

let mandelf (c:Complex) (z:Complex) = z*z+c
```

Следующим этапом определим функцию `ismandel: Complex->bool`, которая будет по любой точке комплексной плоскости выдавать признак ее принадлежности множеству Мандельброта. Для простоты мы будем рассматривать слегка видоизмененное множество, похожее на множество Мандельброта – множество тех точек, для которых $z_{20}(0)$ является ограниченной величиной, то есть по модулю меньше 1.

Для вычисления $z_{20}(0)$ вспомним, что функция `mandelf` описана в каррированном представлении и при некотором фиксированном *c* представляет собой функцию из `Complex` в `Complex`. Таким образом, используя описанную ранее функцию *n*-кратного применения функции `rpt`, мы можем построить 20-кратное применение функции `mandelf`: `rpt 20 (mandelf c)`. Далее остается применить эту функцию к нулю и взять модуль значения:

```
let ismandel c = Complex.Abs(rpt 20 (mandelf c) (Complex.zero))<1.0
```

По сути дела, эти две строчки – описание функций `mandelf` и `ismandel` – определяют нам множество Мандельброта. Построить это множество – для начала в виде рисунка из звездочек на консоле – теперь дело техники и нескольких строк кода:

```
let scale (x:float,y:float) (u,v) n = float(n-u)/float(v-u)*(y-x)+x;;

for i=1 to 60 do
 for j=1 to 60 do
   let lscale = scale (-1.2,1.2) (1,60) in
   let t = complex (lscale j) (lscale i) in
   Console.Write(if ismandel t then "*" else " ")
 Console.WriteLine("")
```

Результат работы программы в консольном режиме можно наблюдать на рис. 1.1. Для получения такого результата мы преобразовали программу в самостоятельное F#-приложение – файл с расширением `.fs`, который затем можно откомпилировать из Visual Studio либо с помощью утилиты `fsc.exe` в независимое выполняемое приложение.

Таким образом, программа, отвечающая за построение множества Мандельброта, уместилась, по сути дела, на одном экране компактного кода. Если проанализировать причины, по которым программа получилась существенно компактнее возможных аналогов на C#, можно отметить следующее:

- компактный синтаксис для описания функций;
- вывод типов, благодаря которому не надо практически нигде указывать тип данных используемых значений. Обратите внимание, что при этом язык

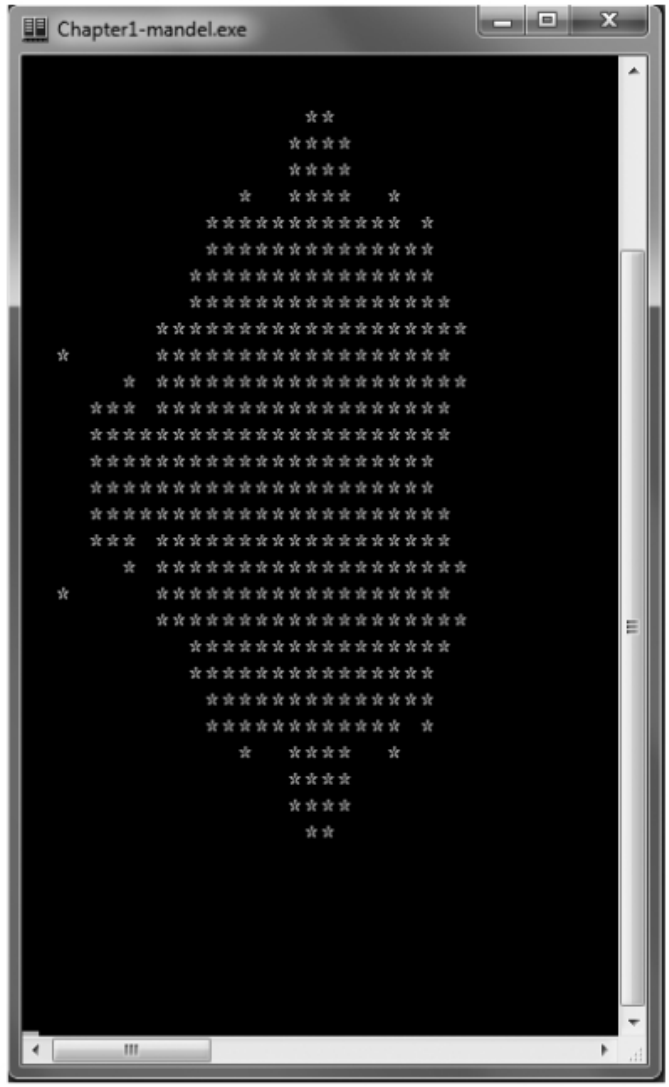

Рис. 1.1. Изображение множества Мандельброта на консоли

остается *статически типизируемым*, то есть проверка типов производится на этапе компиляции программы!

- использование каррированного вызова функций, благодаря чему очень просто можно оперировать понятием частичного применения функции;
- наличие удобных встроенных типов данных для упорядоченных кортежей и списков.

1.10. Интероперабельность с .NET

Безусловно, построение множества Мандельброта из звездочек впечатляет, но было бы еще интереснее построить графическое изображение с большим разрешением. К счастью, F# является полноценным языком семейства .NET и может использоваться совместно со всеми стандартными библиотеками .NET, такими как System.Drawing для манипулирования двумерными изображениями и даже библиотекой Windows Forms.

Код, строящий фрактальное изображение в отдельном окне, приведен ниже:

```
open System.Drawing
open System.Windows.Forms

let form =
   let image = new Bitmap(400, 400)
   let lscale = scale (-1.2,1.2) (0,image.Height-1)
   for i = 0 to (image.Height-1) do
     for j = 0 to (image.Width-1) do
       let t = complex (lscale i) (lscale j) in
       image.SetPixel(i,j,
if ismandel t then Color.Black else Color.White)
   let temp = new Form()
   temp.Paint.Add(fun e -> e.Graphics.DrawImage(image, 0, 0))
   temp.Show()
   temp
```

В начале программы мы подключаем библиотеки Windows Forms и System. Drawing. Основная функция – form – отвечает за создание основного окна с фрактальным изображением. В этой функции сначала создается объект `Bitmap` – двумерный пиксельный массив, который заполняется фрактальным изображением с помощью двойного цикла, похожего на использованные в предыдущем примере. После заполнения изображения создается форма и добавляется для нее функция перерисовки, которая при каждой перерисовке окна отрисовывает внутри единожды вычисленное фрактальное изображение (рис. 1.2).

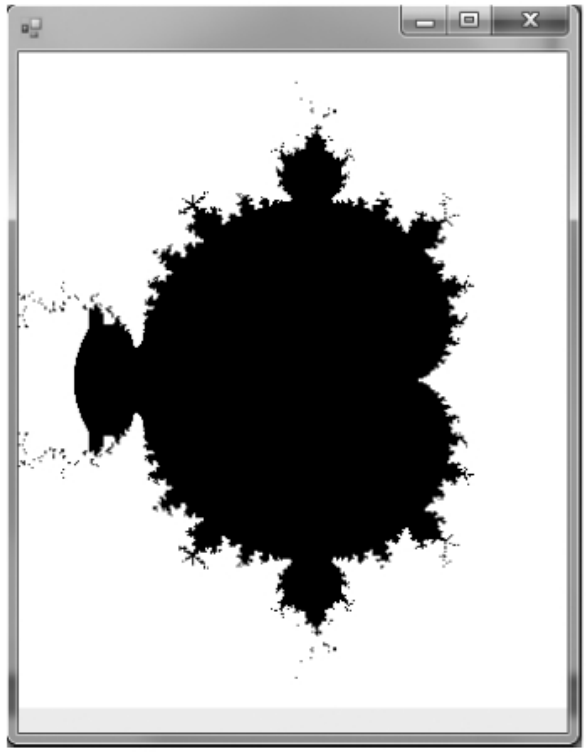

Рис. 1.2. Изображение множества Мандельброта в окне Windows Forms

Конечно, программа в таком виде имеет недостаточно богатый интерфейс, да и процесс построения формы через переопределение функции перерисовки не является самым правильным. Основная цель данного примера – показать, что F# может прозрачным образом работать со всем имеющимся многообразием функций платформы .NET, от графических примитивов до сетевого взаимодействия, от доступа к СУБД до построения Silverlight-приложений.

Важно, однако, понимать, что F# не является заменой традиционным языкам типа C# и Visual Basic. Предполагается, что для построения интерфейсов приложений с использованием визуальных дизайнеров будут по-прежнему использоваться императивные языки, а F# сможет эффективно применяться для решения задач, связанных с обработкой данных. Грамотное разделение кода и используемого языка программирования между отдельными модулями – это необходимое условие успешности и эффективности разработки программного проекта.

2. Рекурсивные структуры данных

Традиционное императивное программирование по своей идеологии близко к архитектуре современных ЭВМ. Поэтому для работы со значительными объемами одинаковых данных используется естественная структура данных – массив, аналог последовательной области памяти ЭВМ, адресация к которой производится указанием индекса массива. В функциональном программировании используется другой подход к оперированию структурами данных на основе некоторого «конструктора», позволяющего рекуррентным образом порождать последовательности элементов.

2.1. Списки и конструкторы списков

Простейшей структурой данных является список – конечная последовательность элементов одного типа. Вообще говоря, для построения списков не требуется специальная поддержка языка – их можно описать следующим образом:

```
type 't sequence = Nil | Cons of 't*'t sequence
```

Здесь `Cons` называется конструктором списка, `Nil` обозначает так называемый пустой список. С использованием такого конструктора список целых чисел 1,2,3 будет записываться как `Cons(1,Cons(2,Cons(3,Nil)))` и иметь тип `int sequence`.

Таким образом, для присоединения каждого элемента к списку используется конструктор Cons. Первый элемент списка называется его головой (head), а весь оставшийся список – хвостом (tail). Для отделения головы и хвоста легко описать соответствующие функции:

```
let head (Cons(u,v)) = u
let tail (Cons(u,v)) = v
```

Структура данных называется рекурсивной, поскольку в ее описании используется сама же структура. Действительно, приведенное выше описание sequence может быть прочитано следующим образом: список типа T – это либо пустой список Nil, либо элемент типа T (голова) и присоединенный к нему список типа T (хвост).

Для обработки такой рекурсивной структуры вполне естественно использовать рекурсивные функции. Например, для вычисления длины списка (количества элементов) можно описать функцию `len` следующим образом:

```
let rec len l =
  if l = Nil then 0
  else 1+len(tail l)
```

На самом деле библиотека F# уже содержит определение списков, которое очень похоже на приведенное выше, только в качестве пустого списка используется константа [], а конструктор списков обозначается оператором (::):

```
let 't list = [] | (::) of 't * 't list
```

С использованием такого конструктора список из чисел 1,2,3 можно записать как 1::2::3::[], или [1;2;3], а определение функции `len` будет иметь вид:

```
let rec len l =
  if l = [] then 0
  else 1+len (List.tail l)
```

На самом деле модуль `List` содержит в себе определения множества полезных функций работы со списками, многие их которых мы рассмотрим в этой главе. В частности, там содержится определение функций `head` и `tail`, а также функции `length`.

2.2. Сопоставление с образцом

В соответствии с описанием списка каждый список может представлять из себя либо константу Nil/[], либо конструктор списка с двумя аргументами. Для распознавания того, что же представляет из себя список, мы использовали условный оператор `if`, однако еще удобнее использовать для этого сопоставление с образцом (*pattern matching*). Используя сопоставление с образцом, функция вычисления длины запишется следующим образом:

```
let rec len l =
   match l with
      [] -> 0
    | h::t -> 1+len t
```

После конструкции match следует один или более вариантов, разделенных |, на каждый из которых описывается свое поведение. В данном случае мы используем всего два варианта сопоставления, хотя их может быть больше. Напомним, что, помимо простого сопоставления, можно также использовать более сложные условные выражения, как в примере ниже, в котором мы описываем функцию суммирования положительных элементов списка:

```
let rec sum_positive l =
   match l with
      [] -> 0
```

```
    | h::t when h>0 -> h+sum_positive t
    | _::t -> sum_positive t
```

Этот пример демонстрирует также две особенности оператора match. Во-первых, шаблоны сопоставления проверяются в порядке следования, поэтому с последним шаблоном будут сопоставлены только случаи, в которых голова списка меньше или равна 0. Во-вторых, если значение какой-то части шаблона нам не важно, можно использовать символ подчеркивания _ для обозначения анонимной переменной.

Сопоставление с образцом работает не только в конструкции match, но и внутри сопоставления имен let и в конструкции описания функциональных констант function. Конструкция function аналогична fun, в отличие от нее, позволяет описывать только функции одного аргумента, но поддерживает сопоставление с образцом. При описании функций обработки рекурсивных структур данных часто удобно использовать function, например:

```
let rec len = function
   [] -> 0
 | _::t -> 1+len t
```

2.3. Простейшие функции обработки списков

Модуль List содержит основные функции для работы со списками, в частности описанную нами функцию определения длины списка List.length. Из других функций, заслуживающих внимания, стоит отметить функцию конкатенации списков List.append, которая также может обозначаться как @, например:

```
List.append [1;2] [3;4]
[1;2]@[3;4]
```

Традиционно функция конкатенации определяется следующим образом:

```
let rec append l r =
  match l with
    [] -> r
  | h::t -> h::(append t r)
```

Из этого определения видно, что функция является рекурсивной по первому аргументу, и, значит, для объединения списков длины L_1 и L_2 элементов потребуется $O(L_1)$ операций. Такая сложностная оценка операции конкатенации является следствием способа представления списков с помощью конструктора, в результате чего для составления списка-результата конкатенации мы вынуждены разбирать первый список поэлементно и затем присоединять эти элементы ко второму списку поочередно.

Для доступа к произвольному элементу списка по номеру может использоваться функция `List.nth`, которую также можно вызывать с помощью специального синтаксиса индексатора. Для доступа ко второму элементу списка (который имеет номер 1, поскольку нумерация идет с 0) можно использовать любое из следующих выражений:

```
List.nth [1;2;3] 1
[1;2;3].Item(1)
[1;2;3].[1]
```

Следует опять же помнить, что сложность такой операции – $O(n)$, где n – номер извлекаемого элемента.

2.4. Функции высших порядков

Рассмотрим основные операции, которые обычно применяются к спискам. Подавляющее большинство сложных операций обработки сводятся к трем базовым операциям: отображения, фильтрации и свертки. Поскольку такие функции в качестве аргументов принимают другие функции, работающие над каждым из элементов списка, то они называются *функционалами*, или функциями высших порядков.

2.4.1. Отображение

Операция отображения map применяет некоторую функцию к каждому элементу некоторого списка, возвращая список результирующих значений. Если функция-обработчик имеет тип 'a->'b, то map применяется к списку типа 'a list и возвращает 'b list. Соответственно, сама функция map имеет тип ('a->'b) -> 'a list -> 'b list. Определена она может быть[1] следующим образом (естественно, модуль `List` определяет соответствующую функцию `List.map`):

```
let rec map f = function
    [] -> []
  | h::t -> (f h)::(map f t)
```

Спектр использования функции map очень широк. Например, если необходимо умножить на 2 все элементы целочисленного списка, это можно сделать одним из следующих способов:

```
map (fun x -> x*2) [1;2;3]
map ((*)2) [1;2;3]
[ for x in [1;2;3] -> x*2 ]
```

[1] Применяемое в библиотеке определение несколько более сложное, поскольку использует хвостовую рекурсию. Для простоты мы приводим здесь более очевидное определение, а к вопросу использования хвостовой рекурсии вернемся позднее.

Последний способ использует синтаксис так называемого *спискового генератора* (list comprehension), который мы рассмотрим чуть ниже.

Другой пример – пусть нам необходимо загрузить содержимое нескольких веб-сайтов из Интернета, например с целью дальнейшего поиска. Предположим, у нас определена функция `http`, которая по адресу странички сайта (URL) возвращает ее содержимое[1]. Тогда осуществить загрузку всех сайтов из Интернета можно будет следующим образом:

```
["http://www.bing.com";"http://www.yandex.ru"] |> List.map http
```

Напоминаем, что здесь мы используем операцию последовательного применения функций |> (*pipeline*), которая позволяет последовательно применять цепочки

Иногда бывает полезно, чтобы функции обработки передавался номер обрабатываемого элемента списка. Для этого предусмотрена специальная функция `List.mapi`, которая принимает функцию обработки с двумя аргументами, один из которых – номер элемента в списке, начиная с 0. Например, если нам надо получить из списка строк пронумерованный список, это можно сделать так:

```
["Говорить";"Читать";"Писать"]
    |> List.mapi (fun i x -> (i+1).ToString()+". "+x)
```

В качестве чуть более сложного примера рассмотрим функцию «наивного» перевода списка цифр в некоторой системе счисления в число. Например, число 1000 можно представить как [1;0;0;0], и в двоичной системе оно будет обозначать $1\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0 = 8$. Таким образом, чтобы получить значение числа, нам надо умножать каждую цифру на основание системы счисления в степени, равной позиции цифры от конца числа. Для достижения этого проще всего сначала перевернуть (записать в обратном порядке) список цифр, после чего применить `mapi` для умножения цифр на возведенное в степень основание, далее сложить результат с помощью функции `List.sum`:

```
let conv_to_dec b l =
   List.rev l |>
   List.mapi (fun i x -> x*int(float(b)**float(i))) |>
   List.sum
```

Также в библиотеке определены функции попарного отображения двух списков с получением одного результата `List.map2`, на основе которых легко определить, например, сложение векторов, представленных списками:

```
List.map2 (fun u v -> u+v) [1;2;3] [4;5;6]
List.map2 (+) [1;2;3] [4;5;6]
```

[1] Описание функции http вы сможете найти далее в книге в соответствующей главе.

С помощью map2 также можно определить функцию conv_to_dec:

```
let conv_to_dec b l =
  [ for i = (List.length l)-1 downto 0 do yield int(float(b)**float(i)) ]
  |> List.map2 (*) l |> List.sum
```

В этом случае мы сначала явно порождаем список из степеней основания системы счисления, а потом попарно умножаем его на цифры числа, затем суммируя результат.

Также в библиотеке определены функции для «тройного» отображения List.map3, двойного отображения с индексацией List.mapi2 и др.

Бывают ситуации, когда для каждого элемента списка нам полезно рассмотреть несколько альтернатив. Например, пусть у нас есть список сайтов, для которых мы хотим получать странички contact.html и about.html. Первая попытка реализовать это будет выглядеть следующим образом:

```
[ "http://site1.com"; "http://site2.com"; "http://site3.com" ] |>
  List.map (fun url ->
[ http (url+"/about.html"); http (url+"/contact.html")])
```

Однако в этом случае в результате будет получен не список содержимого всех страничек (что было бы удобно для реализации поисковой системы), а список списков – для каждого сайта будет возвращен список из двух страничек. Чтобы объединить все результирующие списки в один, придется использовать в явном виде функцию конкатенации списка списков:

```
[ "http://site1.com"; "http://site2.com"; "http://site3.com" ] |>
  List.map (fun url ->
[ http (url+"/about.html"); http (url+"/contact.html")])
  |> List.concat
```

Однако намного более эффективно сразу использовать вместо map функцию collect, которая применяет заданную функцию к каждому элементу исходного списка и затем объединяет вместе возвращаемые этими функциями списки:

```
 [ "http://site1.com"; "http://site2.com"; "http://site3.com" ] |>
  List.collect (fun url ->
       [ http (url+"/about.html"); http (url+"/contact.html")])
```

2.4.2. Фильтрация

Фильтрация позволяет нам оставить в списке только элементы, удовлетворяющие заданной функции-фильтру. Например, для выделения только четных элементов списка чисел от 1 до 10 можно использовать:

```
[1..10] |> List.filter (fun x -> x%2=0)
```

Если `filter` применяется к списку типа `'t list`, то функция фильтрации должна иметь тип `'t -> bool`. В результирующий список типа `'t list` попадают только элементы, для которых функция принимает истинное значение. Функция фильтрации может быть реализована следующим образом:

```
let rec filter f = function
    [] -> []
  | h::t when (f h) -> h::(filter f t)
  | _::t -> filter f t
```

Вот как можно легко использовать `filter` для реализации простейшей поисковой системы в фиксированном множестве сайтов:

```
["http://www.bing.com";"http://www.yandex.ru"] |>
   List.map http |>
   List.filter (fun s -> s.IndexOf("bing")>0)
```

В качестве более сложного примера рассмотрим вычисление простых чисел в интервале от 2 до некоторого числа *N*. Для этого используется алгоритм, известный как **решето Эратосфена**. Он состоит в следующем: выписываем все числа от 2 до *N*, после чего применяем к ним многократно одинаковую процедуру: объявляем первое из написанных чисел простым, а из оставшихся вычеркиваем все числа, кратные данному. После чего процедура повторяется. В результате в списке у нас остаются только простые числа.

Для реализации этого алгоритма опишем функцию `primes`, которая применяется к списку от 2 до *N*. Эта функция будет рекурсивно реализовывать каждый шаг алгоритма Эратосфена:

```
let rec primes = function
    [] -> []
  | h::t -> h::primes(filter (fun x->x%h>0) t)
```

На каждом шаге первое число в списке h объявляется простым (то есть входит в результирующий список), а к остальным применяется функция фильтрации, которая вычеркивает из списка оставшихся чисел все, кратные h.

Еще один интересный пример – быстрая сортировка Хоара. Алгоритм быстрой сортировки состоит в том, что на каждом шаге из списка выбирается некоторый элемент и список разбивается на две части – элементы, меньшие или равные выбранному и большие выбранного. Затем сортировка рекурсивно применяется к обоим частям списка. С использованием `filter`, выбирая первый элемент списка в качестве элемента для сравнения, мы получим следующую реализацию:

```
let rec qsort = function
   [] -> []
```

```
 | h::t ->
     qsort(List.filter ((>)h) t) @ [h] @
     qsort(List.filter ((<=)h) t)
```

Эта же реализация может быть записана более наглядно, с использованием конструктора списков для фильтрации:

```
let rec qsort = function
   [] -> []
 | h::t ->
     qsort([for x in t do if x<=h then yield x]) @ [h]
   @ qsort([for x in t do if x>h then yield x])
```

Мы видим, что в данном случае мы, по сути, с помощью операции фильтрации разбиваем список на две части в соответствии с некоторым предикатом – при этом две операции фильтрации требуют двух проходов по списку. Чтобы сократить число проходом, можно воспользоваться функцией `partition`, возвращающей пару списков – из элементов, удовлетворяющих предикату фильтрации и всех остальных:

```
List.partition ((>)0) [1;-3;0;4;3]   => ([-3],[1;0;4;3])
```

С учетом этой функции быстрая сортировка запишется следующим образом:

```
let rec qsort = function
   [] -> []
 | h::t ->
     let (a,b) = List.partition ((>)h) t
     qsort(a) @ [h] @ qsort(b)
```

Еще одной альтернативой функции `filter`, объединяющей ее с отображением `map`, является функция `choose`, которая для каждого элемента возвращает опциональный тип и собирает только те результаты, которые не равны `None`. В частности:

```
let filter p = List.choose (fun x -> if p x then Some(x) else None)
let map f = List.choose (fun x -> Some(f x))
```

2.4.3. Свертка

Операция свертки применяется тогда, когда необходимо получить по списку некоторый интегральный показатель – минимальный или максимальный элемент, сумму или произведение элементов и т. д. Свертка является заменой циклической обработки списка, в которой используется некоторый аккумулятор, на каждом шаге обновляющийся в результате обработки очередного элемента.

Поскольку в функциональном программировании нет переменных, то традиционное решение с аккумулятором невозможно. Вместо этого используется в яв-

ном виде передаваемое через цепочку функций значение – состояние. Функция свертки будет принимать на вход это значение и очередной элемент списка, а возвращать – новое состояние. Таким образом, функция `fold`, примененная к списку $[a_1;...;a_n]$, будет вычислять $f(...f(f(s_0,a_1),a_2),...,a_n)$, где s_0 – начальное значение аккумулятора.

В качестве аккумулятора для вычисления суммы элементов списка будет выступать обычное числовое значение, которое мы будем складывать с очередным элементом:

```
let sum L = List.fold (fun s x -> s+x) 0 L
```

Поскольку функция, передаваемая `fold`, представляет собой обычное сложение, то мы можем записать то же самое короче:

```
let sum = List.fold (+) 0
let product = List.fold (*) 1
```

Здесь мы также определяем функцию произведения элементов списка. Для вычисления минимального и максимального элементов списка за один проход мы можем использовать состояние в виде пары:

```
let minmax L =
   let a0 = List.head L in
   List.fold (fun (mi,ma) x ->
               ((if mi>x then x else mi),
                (if ma<x then x else ma)))
             (a0,a0) L
let min L = fst (minmax L)
let max L = snd (minmax L)
```

Описанная нами операция свертки называется также левой сверткой, поскольку применяет операцию к элементам списка слева направо. Также имеется операция правой, или обратной, свертки `List.foldBack`, которая вычисляет $f(a_1,f(a_2,...f(a_n,s_0)...)$.

Наш пример с функцией-минимаксом с помощью обратной свертки запишется так:

```
let minmax L =
   let a0 = List.head L in
   List.foldBack (fun x (mi,ma) ->
               ((if mi>x then x else mi),
                (if ma<x then x else ma)))
             L (a0,a0)
```

Обратите внимание, что функция f в данном случае имеет тип 't→State→State (то есть сначала идет элемент списка, а затем – состояние) и что порядок аргументов у функции `foldBack` другой. Типы функций `fold` и `foldBack` следующие:

- fold: (State→'t →State) →State→'T list→State
- foldBack: ('t→State→State) →'T list→State→State

Для вычисления минимального и максимального элементов нам приходилось в явном виде использовать первый элемент списка в качестве начального состояния, из-за чего функция получилась несколько громоздкой. Альтернативно, если нам необходимо определить лишь функцию минимального или максимального элемента, мы можем воспользоваться редукцией списка `List.reduce`, которая применяет некоторую редуцирующую функцию f типа 'T→'T→'T попарно сначала к первым двум элементам списка, затем к результату и третьему элементу списка и так далее до конца, вычисляя $f(f(...f(a_3 f(a_1,a_2))...)$. С помощью редуцирования мы получим простое определение:

```
let min : int list -> int = List.reduce (fun a b -> Math.Min(a,b))
```

Здесь нам пришлось описать тип функции min в явном виде, поскольку иначе система вывода типов не может правильно выбрать необходимый вариант полиморфной функции Min из библиотеки .NET. Если же функция минимума определена как каррированная F#-функция, то определение будет еще проще:

```
let minimum a b = if a>b then b else a
let min L = List.reduce minimum L
```

2.4.4. Другие функции высших порядков

В библиотеке F# есть также множество функций, которые используются не так часто, как рассмотренные выше, но которые полезно упомянуть. Для простого итерирования по списку могут использоваться функция `iter` и ее разновидности `iteri` и `iter2`, например:

```
List.iteri (fun n x -> printf "%d. %s\n" (n+1) x) ["Раз";"Два";"Три"]
List.iter2 (fun n x -> printf "%d. %s\n" n x) [1;2;3] ["Раз";"Два";"Три"]
```

Для поиска элемента в списке по некоторому предикату используются функции `find` и `tryFind`. Первая из них возвращает найденный элемент и генерирует исключение, если элемент не найден; вторая возвращает опциональный тип, то есть `None`, в случае если элемент не найден. Аналогичные функции `findIndex` и `tryFindIndex` возвращают не сам элемент, а его порядковый номер.

Из других функций, работающих с предикатами, упомянем `exists` и `forall` (а также их варианты `exists2` и `forall2`), проверяющие, соответственно, истинность предиката на хотя бы одном или на всех элементах списка. Эти функции могут быть легко определены через свертку:

```
let exists p = List.fold (fun a x -> a || (p x)) false
let for_all p = List.fold (fun a x -> a && (p x)) true
```

Функции zip/unzip позволяют объединять два списка в список пар значений и, наоборот, из списка пар получать два списка. Есть их версии zip3/unzip3 для троек значений.

Имеется также целый спектр функций для сортировки списка. Обычная sort сортирует список в соответствии с операцией сравнения, определенной на его элементах. Если необходимо сортировать элементы в соответствии с некоторым другим критерием, то можно либо задать этот критерий явно (sortWith), либо задать функцию генерации по каждому элементу некоторого индекса, в соответствии с которым осуществлять сортировку (sortBy). Вот как с помощью этих функций можно отсортировать список слов по возрастанию длины:

```
["One";"Two";"Three";"Four"] |> List.sortBy(String.length)
["One";"Two";"Three";"Four"] |>
   List.sortWith(fun a b -> a.Length.CompareTo(b.Length))
```

Из арифметических операций над списками определены операции min/max (minBy/maxBy), sum/sumBy и average/averageBy – например, вот так мы могли бы найти строку с максимальной длиной и среднюю длину строк в списке:

```
["One";"Two";"Three";"Four"] |> List.maxBy(String.length)
["One";"Two";"Three";"Four"] |> List.averageBy(fun s -> float(s.Length))
```

В заключение упомянем функцию permute, которая умеет применять к списку перестановку, заданную целочисленной функцией:

```
List.permute (function 0->0 | 1->2 | 2->1) [1;2;3]
```

2.5. Генераторы списков

Мы уже ранее встречались с конструкциями, которые позволяли создавать список путем задания диапазона элементов, или некоторой функции генерации списка. Все эти конструкции укладываются в единообразный синтаксис генераторов списков, начинающийся с квадратной скобки. Списки могут задаваться:

- явным перечислением элементов: [1;2;3];
- заданием диапазона значений: [1..10]. В этом случае для создания списка на самом деле вызывается оператор диапазона (..). Его можно использовать и в явном виде, например:

```
let integers n = (..) 1
```

- заданием диапазона и шага инкремента: [1.1..0.1..1.9] или [10..-1..1];
- заданием генерирующей функции: [for x in 0..8 -> 2**float(x)]. Этот пример можно также записать в виде явного вызова функции List.init следующим образом: List.init 9 (fun x -> 2.0**float(x)), либо же в виде отображения [0..8] |> List.map (fun x -> 2.0**float(x));

- заданием более сложного алгоритма генерации элементов списка. В этом случае внутри скобок могут использоваться любые комбинации из операторов `for`, `let`, `if` и др., а для возврата элементов используется конструкция `yield`:

```
[ for a in -3.0..3.0 do
    for b in -3.0..3.0 do
      for c in -3.0..3.0 do
        let d = b*b-4.*a*c
        if a<>0.0 then
          if d<0.0 then yield (a,b,c,None,None)
          else yield
               (a,b,c,
                Some((-b-Math.Sqrt(d))/2./a),
                Some((-b+Math.Sqrt(d))/2./a)) ]
```

2.6. Хвостовая рекурсия

Вернемся снова к рассмотрению простейшей функции вычисления длины списка:

```
let rec len = function
   [] -> 0
 | h::t -> 1+len t
```

Посмотрим на то, как эта функция вычисляется, например для списка [1;2;3]. Вначале от списка отделяется хвост, и рекурсивно вызывается `len [2;3]` – происходит рекурсивное погружение. На следующем уровне рекурсии вызывается `len [3]` и наконец `len []`, которая возвращает 0, – после чего происходит «всплывание» из рекурсии, для вычисления `len [3]` к 0 прибавляется 1, затем еще 1, и наконец вызванная функция завершается, возвращая результат – 3. Схематически процесс рекурсивного вызова `len [1;2;3]` изображен на рис. 2.1.

На каждом уровне рекурсии для рекурсивного вызова необходимо запомнить в стеке адрес возврата, параметры функции и возвращаемый результат – то есть такое определение len требует для своей работы $O(n)$ ячеек памяти. С другой стороны, очевидно, что для вычисления длины списка при императивном программировании не требуются дополнительные расходы памяти. Если бы на функциональном языке было невозможно совершать такие простые итеративные операции без расхода памяти, это было бы крайне негативной стороной, сводящей на нет многие преимущества.

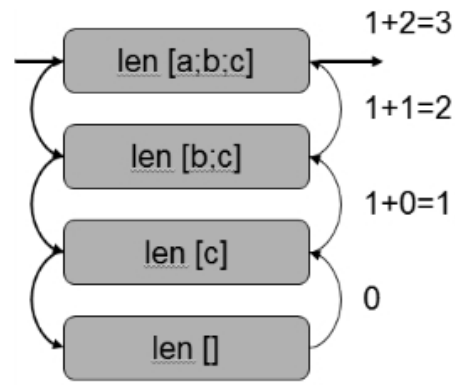

Рис. 2.1. Рекурсивное вычисление длины списка

К счастью, алгоритмы, реализуемые в императивных языках при помощи итерации, могут быть эффективно вычислены в функциональном под-

ходе при помощи так называемой *хвостовой рекурсии* (tail recursion). Суть хвостовой рекурсии сводится к тому, что в процессе рекурсии не выделяется дополнительная память, а рекурсивный вызов является последней операцией в процессе вычисления функции. В этом случае возможно сразу после вычисления функции перейти к следующему рекурсивному вызову без запоминания адреса возврата, то есть компилятор может распознать такую рекурсию и преобразовать ее к обычному циклу – сохранив при этом все преимущества рекурсивного определения.

Чтобы вычисление длины списка производилось без дополнительных расходов памяти, преобразуем функцию таким образом, чтобы прибавление единицы к длине происходило до рекурсивного вызова. Для этого введем счетчик – текущую длину списка – и каждый рекурсивный вызов будет сначала увеличивать счетчик и потом вызывать функцию с увеличенным счетчиком в качестве параметра:

```
let rec len a = function
   [] -> a
 | _::t -> len (a+1) t
```

При этом рекурсивный вызов располагается в конце вызова функции – поэтому адрес возврата не запоминается, а совершаются по сути циклические вычисления, как показано на рис. 2.2.

Чтобы вычислить длину списка, надо вызывать функцию с нулевым значением счетчика:

```
len 0 [1;2;3]
```

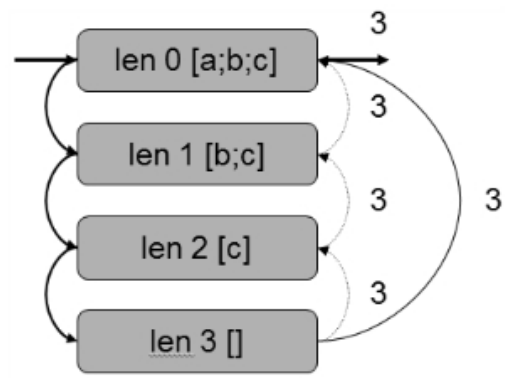

Рис. 2.2. Вычисление длины списка с использование хвостовой рекурсии

Понятно, что программисту, использующему функцию `len`, нет нужды знать про внутреннее устройство функции, поэтому правильно будет спрятать особенности реализации внутрь вложенной функции:

```
let len l =
  let rec len_tail a = function
     [] -> a
   | _::t -> len_tail (a+1) t
  len_tail 0 l
```

Другим примером, когда хвостовая рекурсия позволяет сильно оптимизировать решение, является реверсирование списка. Исходя из декларативных соображений, простейший вариант реверсирования может быть записан следующим образом:

```
let rec rev = function
  [] -> []
| h::t -> (rev t)@[h]
```

Если задуматься над сложностью такого алгоритма, то окажется, что она равна $O(n^2)$, поскольку в реверсировании используется операция `append`, имеющая линейную сложность. С другой стороны, алгоритм реверсирования может быть легко сформулирован итерационно: необходимо отделять по одному элементу с начала исходного списка и добавлять в начало результирующего – в этом случае последний элемент исходного списка как раз окажется первым в результате.

Для реализации такого алгоритма мы определим функцию `rev_tail`, которая в качестве первого аргумента будет принимать результирующий список, а вторым аргументом будет исходный список. На каждом шаге мы будет отделять голову исходного списка и добавлять его в начало первого аргумента при рекурсивном вызове. Когда исходный список исчерпается и станет пустым – мы вернем первый аргумент в качестве результата. При ближайшем рассмотрении также можно увидеть, что функция `rev_tail` использует хвостовую рекурсию.

```
let rev L =
   let rec rev_tail s = function
     [] -> s
   | h::t -> rev_tail (h::s) t in
   rev_tail [] L
```

Таким образом, мы видим, что декларативные реализации функций часто оказываются очень наглядными, но с точки зрения эффективности не самыми лучшими. Поэтому разработчику, пишущему код в функциональном стиле, стоит взять в привычку задумываться над особенностями выполнения кода, в частности постараться, где возможно, использовать хвостовую рекурсию. Основным сигналом к тому, что хвостовая рекурсия возможна, является итерационный алгоритм обработки, который при императивной реализации не требовал бы дополнительной памяти. При некотором навыке преобразование таких рекурсивных алгоритмов к хвостовой рекурсии станет механическим упражнением.

2.7. Сложностные особенности работы со списками

Как вы могли заметить, стиль работы со списками весьма отличается от работы с массивами. Может показаться, что из-за отсутствия прямого доступа к элементам списка работа с ним оказывается менее эффективной, чем работа с массивом. Это верно лишь отчасти – очень немногие алгоритмы, требующие прямого доступа (наиболее яркий представитель – двоичный поиск), плохо реализуются на списках. В подавляющем большинстве своем алгоритмы сводятся к итерации по списку, что в функциональных языках реализуется рекурсией или использованием встроенных библиотечных функций. Вообще говоря, многообразие встроенных функций работы со списками практически сводит на нет моменты, когда рекурсивную обработку списка необходимо реализовывать вручную – предоставляя

своеобразный аналог реляционной алгебры для обработки списков, с помощью которой можно выразить широкий спектр операций.

Тем не менее имеет смысл понимать сложностные ограничения при работе со списками. Сложность основных операций в сравнении с массивами с произвольным доступом отражена в следующей таблице.

Операция	Список	Массив со случайным доступом
Случайный доступ	O(n)	O(1)
Поиск	O(n)	O(n)
Вставка в начало / удаление первого элемента	O(1)	O(n)
Вставка / удаление элемента в конец	O(n)	O(n)
Вставка / удаление элемента в середину	O(n)	O(n)
Реверсирование	O(n)	O(n)

Из таблицы видно, что сложность доступа для списков и массивов более или менее паритетна. Невозможность реализовать на списках эффективный поиск приводит к тому, что вместо списков для хранения данных, требующих возможностей поиска, используются деревья, которые по своим сложностным характеристикам аналогичны двоичному поиску. В целом для решения специфических задач в функциональном программировании разработаны свои структуры данных, обладающие хорошими сложностными характеристиками доступа. Подробнее такие структуры данных изложены в диссертации и книге Криса Окасаки «Purely Functional Data Structures». Мы же здесь рассмотрим лишь один пример – реализацию очереди.

Очередь – это структура данных, в которую можно добавлять и из которой забирать элементы, при этом первый добавленный элемент будет извлекаться в первую очередь, по принципу FIFO (First In First Out). Для очереди мы определим операцию добавления элемента в очередь `put`, извлечения первого элемента `head` и удаления первого элемента `tail`.

Наивная реализация будет использовать список для представления очереди. При этом операции `head` и `tail` будут соответствовать операциям для списков, а добавление элемента будет добавлять элемент в конец очереди:

```
type 'a queue = 'a list
let tail = List.tail
let head = List.head
let rec put x L = L @ [x]
```

При такой реализации сложность доступа и удаления элемента из очереди равны O(1), а добавления – O(n). Если, наоборот, удалять элементы из конца очереди и добавлять в начало, то сложность добавления будет O(1), а удаления – O(n).

Наша же цель – разработать структуру данных, которая будет иметь сложность O(1) для обеих операций, хотя бы «почти всегда», то есть за исключением некоторых редких операций. Для представления очереди в этом случае мы будем

использовать два списка – из первого списка мы будем забирать элементы (это будет голова очереди), а во второй – добавлять (это будет хвост очереди, но расположенный в реверсированном порядке). Например, очередь [1;2;3;4] (где 1 – это голова, 4 – хвост) может быть представлена парой списков ([1;2],[4;3]). Естественно, такое представление не единственное; та же очередь может быть представлена как ([1],[4;3;2]) и другими способами. В любом случае, мы договоримся, что для непустой очереди первый список всегда будет непустым, пустой первый список будет означать пустую очередь (при этом второй список тоже будет пустым).

Взятие элемента из очереди всегда происходит из первого списка. Если при этом первый список вдруг становится пустым – нам необходимо осуществить перестройку очереди, поставив в первый список реверсированный второй, а второй сделав пустым. Такая операция будет иметь неконстантную сложность, однако следующие операции уже будут брать элементы из первого списка со сложностью O(1). Добавление элементов происходит в голову второго списка, всегда со сложностью O(1).

Реализация очереди будет выглядеть следующим образом:

```
type 'a queue =
      'a list * 'a list

let tail (L,R) =
  match L with
   [x] -> (rev R, [])   // осуществляем перестройку очереди
  | h::t -> (t,R);;     // удаление элемента без перестройки

let head (h::_,_) = h;; // первый элемент очереди - это всегда
     // первый элемент первого списка

let put x (L,R) =
  match L with
   [] -> ([x],R)       // добавляем в пустую очередь
  | _ -> (L,x::R)       // добавляем в непустую очередь - ко второму списку
```

Другая задача, которая часто встает перед разработчиками, – это реализация различных словарей, в которые добавляются элементы. Здесь важными операциями являются добавление ключа и поиск по ключу. Одним из подходов, принятых в императивных языках, является использование массива и двоичного поиска в массиве – при этом сложность вставки элемента (с раздвиганием массива) равна $O(n)$, а поиск осуществляется за $O(\log_2 n)$. В функциональном программировании более принято использовать деревья поиска, обычные или сбалансированные, которые дают нехудшие сложностные характеристики. Деревья будут рассмотрены ниже в этой главе. Также стоит отметить, что библиотека .NET предоставляет множество структур данных, таких как Hashtable, Dictionary и др., которые могут эффективно использоваться в F#. Примеры подобного использования мы увидим ниже.

2.8. Массивы

В библиотеке .NET часто используются массивы, в том числе в качестве аргументов функций в различных методах библиотеки. Поэтому совершенно логично, что F# должен поддерживать массивы как базовый тип данных, поскольку списки F# являются отдельным типом данных, напрямую не совместимым с классическими структурами .NET[1]. Однако поддержка массивов требует некоторого отступления от чистоты функционального программирования, так как массивы позволяют модифицировать свои элементы в процессе работы.

По сути, массивы очень напоминают списки, а над ними определены все те же функции обработки, что были рассмотрены для списков, – `map`, `filter` и др. Только соответствующие функции обработки находятся в пространстве имен `Array`. Поэтому если обработка данных сводится только к использованию библиотечных функций, то разработчику не очень принципиально, является ли структура данных массивом или списком.

Для массивов также определены конструкторы с таким же синтаксисом, как и для списков, – только ограничиваются они символами `[|` и `|]`. Например, список из чисел от 1 до 5 может быть задан как `[|1;2;3;4;5|]` или `[| 1..5 |]`.

На базовом уровне, однако, обработка массивов существенно отличается от списков и больше всего напоминает императивное программирование. Основная операция, применимая к массиву, – это взятие элемента по индексу, обозначаемая как A.[i]. В качестве примера рассмотрим функцию суммирования элементов целочисленного массива (мы описываем функцию в учебных целях, понятно, что на практике было бы проще воспользоваться `Array.sum`):

```
let sum (a : int []) =
   let rec sumrec i s =
      if i<a.Length then sumrec (i+1) (s+a.[i])
      else s
   sumrec 0 0
```

Здесь вложенная рекурсивная функция `sumrec` играет роль цикла со счетчиком и аккумулятором одновременно: первый аргумент `i` является счетчиком, изменяясь от 0 до длины массива, второй – накапливает искомую сумму. При этом используется хвостовая рекурсия.

Элементам массива можно также присваивать значения с помощью операции `<-`, например `A.[i] <- n`. Для примера рассмотрим функцию, которая создает целочисленный массив заданной длины *n*, заполненный числами от 1 до *n*:

[1] Естественно, для списков предусмотрены функции преобразования в массивы и обратно `List.toArray` и `List.ofArray`, а также в тип последовательности, совместимой с `IEnumerable`, и обратно `List.toSeq`/`List.ofSeq`.

```
let intarray n =
   let a = Array.create n 0
   Array.iteri (fun i _ -> a.[i] <- (i+1)) a
   a
```

Для заполнения массива используется библиотечная функция `iteri`, позволяющая удобно пройтись по всем элементам массива, а сам массив создается вначале при помощи вызова `Array.create`. Отметим, что аналогичного результата можно добиться с использованием функции библиотеки `Array.init`, или же конструктора массива:

```
let intarray n = Array.init n (fun i -> i+1)
let intarray n = [|1..n|]
```

Операция доступа .[] для массивов также позволяет делать срезы, то есть извлекать из массивов целые диапазоны элементов. Например, `A.[5..10]` извлечет из массива подмассив с 5-го по 10-й элемент (нумерация начинается с 0), а `A.[5..]` – элемент с 5-го по последний. Точно так же возможно присваивание подмассивов, например `A.[5..10] <- [|5..10|]`.

Другим типом, который стоит здесь упомянуть, является тип `List<_>` библиотеки .NET. Во избежание конфликта имен в библиотеке F# этот тип переименован в `ResizeArray`. Его удобно использовать в тех случаях, когда размер структуры данных заранее неизвестен, а по ходу дела надо добавлять новые значения. Для такого сценария достаточно удобно использовать списки, но `ResizeArray` может быть хорошей альтернативой, особенно в тех случаях, когда полезно иметь интерфейс с другими функциями библиотеки .NET или с модулями на других .NET-языках.

В качестве примера использования `ResizeArray` рассмотрим функцию, которая считывает с клавиатуры строки до тех пор, пока не будет введена точка, и возвращает список считанных строк:

```
let ReadLines() =
  let inp = new ResizeArray<string>()
  let rec recread() =
    let s = Console.ReadLine()
    if s<>"." then
      inp.Add(s)
      recread()
  recread()
  List.ofSeq inp
```

В этом примере присутствует побочный эффект, не только в том, что происходит считывание с консоли, но и в том, что в теле функции `recread` мы модифицируем внешний объект `inp`. Поэтому нам приходится в явном виде описывать `recread` как функцию, используя `()`, и то же самое делать для самой функции `ReadLines`.

2.9. Многомерные массивы и матрицы

Отдельного внимания заслуживает представление многомерных массивов в функциональных языках. Традиционно в классических функциональных языках, где основным способом представления последовательностей данных является список, многомерные массивы представляются списками списков (либо более хитрым способом, как в случае разреженных матриц). Однако поскольку F# является мультипарадигмальным языком и позволяет легко использовать библиотеку .NET и внешние сторонние библиотеки, то в ней возникает целый спектр возможностей, среди которых следует выбирать, исходя из конкретной задачи.

2.9.1. Списки списков, или непрямоугольные массивы (Jagged Arrays)

Рассмотрим для начала традиционный способ представления двумерных массивов списком списков (или массивом массивов). Соответствующий тип будет иметь вид `T list list`, или `T[][]`. В этом случае операции над такими массивами реализуются весьма непросто, поскольку операции со строками осуществляются легко (строка представляется целым последовательным списком), в то время как для выделения элементов столбца придется затратить значительные усилия, как вычислительные, так и при написании соответствующих функций). В качестве упражнения попробуйте реализовать операцию транспонирования матрицы, представленной в виде списка списков.

Однако в таком представлении есть и плюсы – в частности, возможность представлять массивы непрямоугольной формы, поскольку внутри списка могут, в свою очередь, быть списки разной длины. Именно поэтому такой способ представления называется *jagged arrays* – непрямоугольные массивы. Типичный пример подобной структуры – треугольник Паскаля, изображенный на рис. 2.3. В нем первый список имеет длину 2, 2-й – 3 и т. д. Число, находящееся в произвольном ряду, является суммой чисел, стоящих на позиции левее и правее данной в предыдущем ряду. Реализуем функцию для вычисления первых *n* строк треугольника Паскаля:

```
let pascal n =
  let rec pas L n =
    let A::t = L in
    if n = 0 then L
    else
      pas ((
      (1::[for i in 1..(List.length A-1) -> A.[i-1]+A.[i]])@[1])::L)
  (n-1)
  pas [[1;1]] n
```

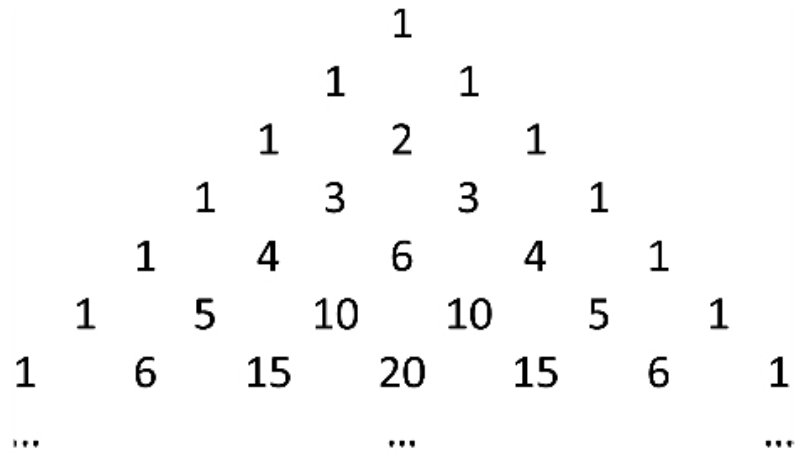

Рис. 2.3. Треугольник Паскаля

Треугольник Паскаля представляется списком списков переменной длины. Мы используем тот факт, что операция индексации .[] работает и для списков, – это позволяет нам сгенерировать очередную строку треугольника с помощью произвольного доступа к элементам предыдущей строки. Это не лучшее решение с точки зрения оптимальности (поскольку производится много лишних проходов по списку для нахождения произвольного элемента), но зато явно самое наглядное.

2.9.2. Многомерные массивы .NET

Поскольку F# – язык на платформе .NET, то в нем можно использовать многомерные массивы .NET. Тип таких массивов будет `T[,]`, и располагаться они будут в последовательной области памяти, что позволяет эффективно реализовывать операции срезов для таких массивов. Для использования двух-, трех- и четырехмерных массивов предназначены, соответственно, классы `Array2D`, `Array3D` и `Array4D`, в которых определены некоторые функции работы с массивами, такие как `map` и `iter`. К сожалению, из-за многообразия возможных вариантов обработки многих функций для многомерных массивов не предусмотрено, поэтому их придется реализовывать самостоятельно. Из плюсов – для многомерных массивов предусмотрены очень гибкие операции срезов, например конструкция `A.[0..,1..2]`, примененная к двумерной матрице, позволяет вырезать из нее необходимый диапазон столбцов (в данном случае – 1 и 2), оставив строки без изменения. Естественно, возможно одновременное вырезание строк и столбцов.

В качестве примера реализуем функцию свертки двумерного массива по столбцам. Эта функция будет принимать функцию свертки, начальное значение состояния и двумерный массив размера $m \times n$ и возвращать одномерный массив длины n, содержащий результат свертки по столбцам:

```
let fold_cols f i (A : 't[,]) =
   let n = Array2D.length2 A
   let res = Array.create n i
   Array2D.iteri (fun i j x -> res.[j] <- f res.[j] x) A
   res
```

Мы вначале создаем результирующий массив нужной размерности, заполненный начальным значением, а затем используем итерацию по всему исходному массиву, применяя функцию свертки к очередному элементу и соответствующему элементу массива-результата. Исходный тип массива описан достаточно полиморфным, что позволяет применять эту функцию к массивам различных типов (в примере ниже – `int` и `float`):

```
fold_cols (+) 0 (Array2D.init 3 4 (fun i j -> i*3+j))
fold_cols (+) 0.0 (Array2D.init 3 4 (fun i j -> float(i)*3.0+float(j)))
```

2.9.3. Специализированные типы для матриц и векторов

Поскольку программистам на F# очень часто приходится сталкиваться с математическими задачами, специально были выделены классы для работы с матрицами и векторами. Эти классы являются частью F# PowerPack и предоставляют намного больше операций для работы с матрицами, нежели определено для стандартных двумерных массивов.

Основные типы данных для линейно-алгебраических операций расположены в пространстве имен Microsoft.FSharp.Math. Это типы `Matrix<_>`, `Vector<_>` и `RowVector<_>`. Для легкого создания матриц и векторов из списков для типа float предусмотрены специальные функции-конструкторы:

```
let v = vector [1.;2.;3.]
let rv = rowvec [1.;2.;3.]
let m : Matrix<float> = matrix [ [ 1.;2.;3.];[4.;5.;6.];[7.;8.;9.]]
rv*v,v*rv,m*v,rv*m
```

В качестве примера реализуем функцию диагонализации матрицы методом Гаусса. Алгоритм состоит в том, что мы для каждой строки i проделываем следующее:

- убеждаемся, что m_{ii} не равно 0, если это так – переставляем i-ю строку с другой, в которой i-й столбец ненулевой. Это делает вложенная функция `swapnz`, а за саму перестановку строк отвечает функция `swaprow`. Обратите внимание, как `swaprow` оперирует срезами матриц, чтобы осуществлять присваивание строк одной операцией, без использования цикла;
- нормализуем i-ю строку таким образом, чтобы m_{ii} было равно 1, путем деления всех элементов на m_{ii};
- из всех строк j с номером, большим i, вычитаем i-ю строку, домноженную на m_{ji}, – таким образом обнуляются все элементы в строках, идущие перед i-м столбцом.

Соответствующая функция диагонализации приведена ниже. Обратите внимание, что она, вопреки традициям функционального программирования, изме-

няет исходную матрицу, а не порождает ее копию. Для работы с большими матрицами иногда такой подход может оказаться предпочтительным:

```
let diagonalize (m:Matrix<float>) =
   let nrows = m.NumRows-1
   let ncols = m.NumCols-1
   let norm j =
       (m.Row j) |> Seq.iteri (fun i x -> m.[j,i] <- x / m.[j,j])
   let swaprow i j =
       let r = m.[i..i,0..ncols]
       m.[i..i,0..ncols] <- m.[j..j,0..ncols]
       m.[j..j,0..ncols] <- r
   let rec swapnz i j =
       if j<=nrows then
         if m.[j,i]<>0. then swaprow i j
         else swapnz i (j+1)
   for i = 0 to nrows do
       if m.[i,i]=0. then swapnz i (i+1)
       if m.[i,i]<>0. then
           norm i
           for j = i+1 to nrows do
              let c = m.[j,i]
              for k=i to ncols do m.[j,k] <- m.[j,k]-m.[i,k]*c
   m
```

2.9.4. Разреженные матрицы

В научных расчетах матрицы часто возникают при решении систем линейных алгебраических уравнений (СЛАУ). При решении дифференциальных уравнений в частных производных, описывающих определенные свойства среды (например, распространение тепла или течение жидкости), часто возникают матрицы большой размерности, подавляющее большинство элементов которых равны 0. Такие матрицы называются *разреженными*. Использование разреженных матриц не только позволяет экономить на памяти при хранении данных, но и обеспечивает ускорение многих операций с матрицами, поскольку достаточно обрабатывать и учитывать лишь ненулевые элементы.

Наиболее естественное представление разреженных матриц – последовательностью элементов, где каждый элемент представляется своими координатами и значением. Альтернативно можно представлять матрицу порождающей функцией – функцией, которая по координатам возвращает соответствующее значение.

Тип `Matrix<_>` в библиотеке F# предоставляет поддержку разреженных матриц. На самом деле любая матрица в этой библиотеке может быть представлена в обычном или разреженном виде, при этом все арифметические операции прозрачным образом поддерживаются между разреженными и неразреженными матрицами. Для создания разреженной матрицы проще всего пользоваться инициализатором `Matrix.initSparse`, которому передается размерность и последовательность элементов с координатами:

```
let sparse = Matrix.initSparse 100 100 [for i in 0..99 -> (i,i,1.0)]
```

2.9.5. Использование сторонних математических пакетов

Очень часто стандартных возможностей, предоставляемых библиотекой F#, может не хватать для решения сложных прикладных задач. В этом случае имеет смысл воспользоваться сторонними библиотеками, которые предоставляют возможности по работе с матрицами. Подробнее использование сторонних библиотек для вычислений рассматривается в главе 7.

2.10. Деревья общего вида

Другим важным рекурсивным типом данных, который часто встречается в функциональных программах, являются деревья. Определение дерева очень похоже на определение списка, однако, поскольку деревья встречаются в разных задачах и их структура несколько варьируется, реализации деревьев нет в стандартной библиотеке F#. Мы в этом разделе рассмотрим несколько типовых применений деревьев, равно как и способы их описания.

В дискретной математике деревом называется ациклический связанный граф. В информатике обычно дают другое рекуррентное определение дерева общего вида типа T – это элемент типа T с присоединенными к нему 0 и более поддеревьями типа T. Если к элементу присоединено 0 поддеревьев, он называется терминальным, или листом, в противном случае узлом. В соответствии с этим дерево может быть представлено следующим образом:

```
type 'T tree =
  Leaf of 'T
| Node of 'T*('T tree list)
```

Соответственно, дерево, представленное на рис. 2.4, может быть описано следующим образом:

```
let tr = Node(1,[Node(2,[Leaf(5)]);Node(3,[Leaf(6);Leaf(7)]);Leaf(4)])
```

Основная процедура обработки дерева – это обход, когда каждый элемент дерева посещается (то есть обрабатывается) ровно один раз. Обход может быть с порождением другого дерева (map), или с аккумулятором (fold), но при этом базовый алгоритм обхода остается неизменным:

```
let rec iter f = function
  Leaf(T) -> f T
| Node(T,L) -> (f T; for t in L do iter f t done)
```

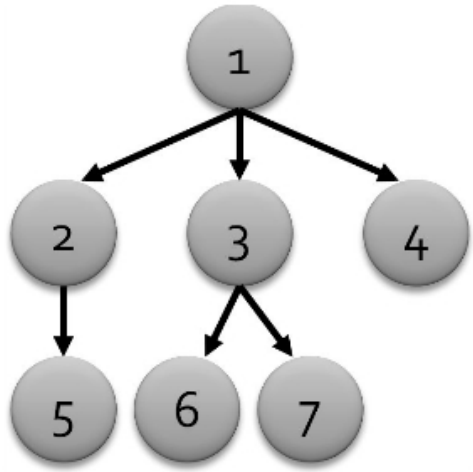

Рис. 2.4. Пример дерева общего вида

Иногда бывает полезным включать в обход также глубину соответствующего элемента, то есть количество узлов, отделяющее его от вершины:

```
let iterh f =
  let rec itr n = function
    Leaf(T) -> f n T
  | Node(T,L) -> (f n T; for t in L do itr (n+1) t done) in
  itr 0
```

Например, для красивой распечатки дерева с отступами можно использовать эту функцию (вспомогательная функция spaces генерирует строку из *n* пробелов):

```
let spaces n = List.fold (fun s _ -> s+" ") "" [0..n]
let print_tree T = iterh (fun h x -> printf "%s%A\n" (spaces (h*3)) x) T
```

В качестве примера функции обхода, генерирующей дерево, рассмотрим реализацию map для деревьев:

```
let rec map f = function
   Leaf(T) -> Leaf(f T)
 | Node(T,L) -> Node(f T,List.map (fun t -> map f t) L)
```

Эта функция для каждого листа возвращает преобразованный лист, а для узла общего вида применяет функцию f к элементу, и затем отображение map для списка поддеревьев которое к каждому поддереву применяет рекурсивным образом такое же древовидное отображение.

Важным примером древовидной структуры является структура папок (каталогов) операционной системы. В этом случае мы можем говорить о неявном порождении дерева путем вызова функций ввода-вывода для обхода дерева каталогов, как в этом примере, печатающем дерево папок, начиная с указанной:

```
let dir_tree path =
    let rec tree path ind =
       Directory.GetDirectories path |>
       Array.iter(fun dir ->
         printfn "%s%s" (spaces (ind*3)) dir;
         tree dir (ind+1))
    tree path 0
```

Сделать такую функцию более универсальной, то есть выделить процедуру обработки директории за пределы функции, можно несколькими способами:

- передавать процедуре dir_tree функцию-обработчик с двумя аргументами – именем директории и уровнем вложенности;
- возвращать дерево директорий в виде описанной ранее древовидной структуры и затем еще раз совершать обход дерева для его печати с помощью iterh;
- возвращать некоторое "упрощенное" представление дерева в виде последовательности пар из имени директории и уровня вложенности, которую затем можно будет просто обработать средствами работы со списками. Последовательности seq будут рассмотрены нами в следующей главе, однако как иллюстрацию упрощенного представления дерева мы приведем соответствующую процедуру здесь:

```
let dir_tree path =
    let rec tree path ind =
       seq {
           for dir in Directory.GetDirectories path do
             yield (dir,ind)
             yield! (tree dir (ind+1))
           }
    tree path 0

dir_tree @"c:\windows"
 |> Seq.iter (fun (s,h) -> printf "%s%s\n" (spaces (h*3)) s)
```

В качестве завершающего примера по работе с деревьями общего вида, которая также демонстрирует использование функций работы со списками, рассмотрим функцию, печатающую размер всех файлов в директории и ее поддиректориях, аналог UNIX-утилиты du:

```
let rec du path =
   Directory.GetDirectories path |>
   Array.iter(fun dir ->
     let sz = Directory.GetFiles dir |>
                Array.sumBy (fun f -> (new FileInfo(f)).Length)
     printfn "%10d %s" sz dir;
     du dir)
```

Эта функция во многом аналогична рассмотренной ранее функции dir_tree, но для каждой директории она вычисляет ее размер следующим образом: полу-

чает список файлов с помощью вызова `GetFiles` и затем вызывает функцию `sumBy`, которая по каждому имени файла порождает структуру `FileInfo` для вычисления длины файла.

2.11. Двоичные деревья

2.11.1. Определение

Другой важной разновидностью деревьев являются **двоичные деревья** – такие деревья, у каждого узла которых есть два (возможно, пустых) поддерева – левое и правое. Двоичные деревья не являются частным случаем деревьев общего вида, поэтому их стоит рассмотреть отдельно. Интересной особенностью двоичных деревьев также является тот факт, что любое дерево общего вида может быть представлено в виде двоичного, однако более подробное рассмотрение этого факта выходит за рамки данной книги.

В соответствии с определением двоичных деревьев для их описания удобно использовать следующий тип:

```
type 't btree =
   Node of 't * 't btree * 't btree
 | Nil
```

2.11.2. Обход двоичных деревьев

Обход двоичных деревьев, в отличие от деревьев общего вида, различается порядком обработки левого и правого поддеревьев и самого элемента в процессе обхода. Различают три основных порядка обхода, показанных в следующей таблице, и три симметричных им обхода, при которых правое поддерево обходится раньше левого:

Порядок обхода	Название		Пример (для дерева выражения)
Корень – левое поддерево – правое поддерево	Прямой	Префиксный	+ * 1 2 3
Левое поддерево – корень – правое поддерево	Обратный	Инфиксный	1 * 2 + 3
Левое поддерево – правое поддерево – корень	Концевой	Постфиксный	1 2 * 3 +

Опишем функцию обхода дерева, которая будет реализовывать три порядка обхода. При этом применим следующий хитрый прием: вместо того чтобы делать переключатели в коде, ограничивая возможные обходы тремя вариантами, будем описывать порядок обхода в виде функции, которая принимает три функции-аргумента для обработки корня, левого и правого поддеревьев и выполняет их в нужном порядке:

```
let prefix root left right = (root(); left(); right())
let infix root left right = (left(); root(); right())
let postfix root left right = (left(); right(); root())
```

В данном случае аргументы root, left и right имеют тип unit→unit, то есть выполнимой функции, которая не возвращает результата. Круглые скобки обозначают выполнение этой функции.

Описав таким образом три порядка обхода (и, возможно, три сопряженных порядка), нам останется в самой процедуре обхода лишь описать три соответствующие функции для обработки корня, левого и правого поддеревьев и передать их как аргументы в переданный в виде аргумента порядок обхода trav:

```
let iterh trav f t =
  let rec tr t h =
    match t with
     Node (x,L,R) -> trav
                     (fun () -> (f x h))  // обход корня
                     (fun () -> tr L (h+1))  // обход левого поддерева
                     (fun () -> tr R (h+1)); // обход прав. поддерева
     | Nil -> ()
  tr t 0
```

Пример инфиксного обхода дерева с использованием этой процедуры:

```
let print_tree T = iterh infix (fun x h -> printf "%s%A\n" (spaces h)x) T
```

Посмотрим также, как реализуется процедура свертки для двоичного дерева. Для простоты будем рассматривать инфиксную свертку:

```
let fold_infix f init t =
   let rec tr t x =
     match t with
       Node (z,L,R) -> tr L (f z (tr R x))
     | Nil -> x
   tr t init
```

С помощью такой процедуры свертки можно, например, преобразовать дерево в список:

```
let tree_to_list T = fold_infix (fun x t -> x::t) [] T
```

2.11.3. Деревья поиска

На практике часто возникают задачи, в которых необходимо создавать различного рода структуры данных, поддерживающие добавление элементов и поиск по некоторому ключевому значению. В таких случаях обычно используются дво-

ичный поиск, хеш-таблицы или деревья поиска. В библиотеке .NET уже присутствуют необходимые классы для работы со словарями и хеш-таблицами, которые мы опишем ниже. Сейчас же кратко рассмотрим, как устроены деревья поиска.

Дерево поиска – это двоичное дерево из элементов порядкового типа, в котором для каждого узла все элементы в левом поддереве меньше данного узла, а все элементы правого поддерева – больше. В таком дереве достаточно легко организовать поиск элемента – начиная с корня мы смотрим, меньше или больше искомый элемент корневого значения, и спускаемся в соответствующем направлении по дереву, пока либо не находим элемент, либо не доходим до листа – что означает, что элемента в дереве нет.

Добавление в дерево поиска реализуется аналогичным образом – когда мы доходим до листа и понимаем, что элемента в дереве нет, мы присоединяем его к листу в соответствующем месте (слева или справа) в зависимости от значения ключа:

```
let rec insert x t =
  match t with
    Nil -> Node(x,Nil,Nil)
  | Node(z,L,R) -> if z=x then t
                   else if x<z then Node(z,insert x L,R)
                   else Node(z,L,insert x R)
```

Для добавления целого списка элементов в дерево можем воспользоваться сверткой, где дерево выступает в роли аккумулятора:

```
let list_to_tree L = List.fold (fun t x -> insert x t) Nil L
```

В частности, мы можем описать сортировку списка, преобразуя список в дерево поиска и затем дерево – в список, используя ранее реализованную нами процедуру `tree_to_list`:

```
let tree_sort L = (list_to_tree >> tree_to_list) L
```

У читателя мог возникнуть вопрос, почему в этом примере (и ранее в нескольких местах) мы используем явный аргумент L, а не запишем определение в более простом виде:

```
let tree_sort = list_to_tree >> tree_to_list
```

Здесь вступает в силу ограничение системы вывода F#, называемое *value restriction*, – для неявных аргументов компилятор не производит обобщения типов и не может правильно определить обобщенный тип аргумента. Поэтому необходимо либо указывать аргумент в явном виде, либо описывать менее обобщенный тип функции, например:

```
let tree_sort : int list -> int list = list_to_tree >> tree_to_list
```

Рассмотренный простейший вариант дерева поиска обеспечивает сложность добавления элемента порядка $O(\log_2 n)$, а сложность поиска находится где-то посередине между $O(\log_2 n)$ для сбалансированного дерева до $O(n)$ в случае, если дерево в процессе построения представляет собой линейную цепочку элементов. На практике для повышения эффективности поиска обычно используют так называемые сбалансированные деревья, в которых при добавлении элемента учитывается балансировка дерева, то есть обеспечивается близость значений высот левого и правого поддеревьев. Рассмотрение сбалансированных деревьев выходит за рамки данной книги, поэтому мы отсылаем заинтересовавшегося читателя к любому классическому курсу информатики.

2.11.4. Деревья выражений и абстрактные синтаксические деревья (AST)

Другим часто используемым применением деревьев является грамматический разбор текста. В этом случае в виде дерева – так называемого абстрактного синтаксического дерева (AST, Abstract Syntax Tree) – удобно представлять структуру разобранного текста и затем уже проводить требуемые действия над текстом путем обработки этого дерева. Сам процесс синтаксического разбора может быть частично автоматизирован использованием специализированных утилит построения анализаторов `fslex` и `fsyacc`.

Частным случаем AST являются деревья арифметических выражений. Для представления арифметического выражения мы могли бы использовать дерево, в узлах которого могут находиться арифметические операции или значения, описав его на базе определенного нами выше типа `btree`:

```
type Operation = Add | Sub | Mul | Div
type ExprNode = Op of Operation | Value of int
type ExprTree = ExprNode btree
```

Тогда для представления простого выражения 1 * 2 + 3 нам пришлось бы описывать следующую структуру:

```
let ex = Node(Op(Add),Node(Op(Mul), Node(Value(1),Nil,Nil),
Node(Value(2),Nil,Nil)), Node(Value(3),Nil,Nil))
```

Однако при описании синтаксических деревьев всегда проще описывать тип данных, ориентированный на конкретное дерево, с учетом возможной структуры узлов. В нашем случае более простое описание будет иметь вид:

```
type Expr =
  Add of Expr * Expr
  | Sub of Expr * Expr
  | Mul of Expr * Expr
  | Div of Expr * Expr
```

```
  | Value of int

let ex = Add(Mul(Value(1),Value(2)),Value(3))
```

Мы видим, что в этом случае синтаксическое дерево приобретает простой и понятный вид, а также упрощаются и приобретают естественную семантику операции его обработки. Например, функция для вычисления выражения, заданного таким деревом, выглядит следующим образом:

```
let rec compute = function
   Value(n) -> n
 | Add(e1,e2) -> compute e1 + compute e2
 | Sub(e1,e2) -> compute e1 - compute e2
 | Mul(e1,e2) -> compute e1 * compute e2
 | Div(e1,e2) -> compute e1 / compute e2
```

2.12. Другие структуры данных

2.12.1. Множества (Set)

Для представления множества различных элементов какого-то типа служит тип `Set<_>`, который, с одной стороны, похож на список (в том, что для множества определены стандартные операции `map`, `filter` и др.), а с другой – реализован как двоичное дерево поиска, рассмотренное выше. Помимо классических «списковых» операций, над множествами определены теоретико-множественные операции – объединение (`+`), пересечение `Set.intersect`, разность (`-`) и др. Например:

```
let s1 = set [1;2;5;6]
let s2 = set [4;5;7;9]

s1+s2,s1-s2,Set.intersect s1 s2
```

Рассмотрим простейший пример, в котором нам надо построить множество букв, встречающихся во входной строке. Для этого идеально использовать множество как состояние в операции свертки по списку букв:

```
let letters (s:string) =
  s.ToCharArray()
  |> Array.fold (fun s c -> s+set[c]) Set.empty
```

2.12.2. Отображения (Map)

Чуть усложним задачу: пусть нам надо строить не множество букв, а частотный словарь их вхождения в строку, то есть для каждой буквы надо запоминать число вхождений. В этом случае следует использовать тип данных словаря, который сопоставляет букве-ключу целое число.

Соответствующий тип данных называется Map<char, int>. Основные операции, которые он поддерживает, – это добавление новой пары ключ – значение Map.add (старое соответствие в этом случае удаляется), поиск значения по ключу Map.find, проверка на наличие ключа в таблице Map.containsKey и удаление ключа Map.remove. Помимо этого, поддерживаются в том или ином виде все списочные операции map, filter и т. д., так же как и преобразование к списку пар (ключ, значение) Map.toList.

С использованием отображения функция подсчета частотного словаря запишется следующим образом:

```
//Строим частотный словарь букв в строке
let letters (s:string) =
   s.ToCharArray()
   |> Array.fold (fun mp c ->
                  if Map.containsKey c mp
                  then Map.add c (mp.[c]+1) mp
                  else Map.add c 1 mp)
       Map.empty
```

2.12.3. Хеш-таблицы

В предыдущем примере отображение Map использовалось в функциональном стиле, в том смысле что каждая вставка или изменение значения в таблице порождало новое отображение. Такой подход является естественным для программиста, привыкшего рассуждать в функциональном стиле, но может показаться непривычным для программиста на C#. Поэтому F# содержит изменяемые структуры данных, с которыми можно оперировать привычными методами – организовать цикл по списку и добавлять в одну и ту же структуру данных каждое новое вхождение символа. Такой изменяемой структурой данных является хеш-таблица HashMultiMap, с использованием которой подсчет частотного словаря можно реализовать следующим образом:

```
let letters (s:string) =
   let ht = new HashMultiMap<char,int>(HashIdentity.Structural)
   s.ToCharArray()
   |> Array.iter (fun c ->
                  if ht.ContainsKey c
                  then ht.[c] <- ht.[c]+1
                  else ht.[c] <- 1)
   ht
```

Напомним, что F# может работать со всеми структурами данных библиотеки .NET, которые также являются изменяемыми. Например, чтобы использовать словарь Dictionary, в код придется внести очень незначительные изменения:

```
open System.Collections.Generic
let letters (s:string) =
   let ht = new Dictionary<char,int>()
   s.ToCharArray()
   |> Array.iter (fun c ->
                  if ht.ContainsKey c then ht.[c] <- ht.[c]+1
                  else ht.[c] <- 1)
   ht
```

Если мы пишем код, который предполагается использовать из других .NET-языков, или внешнее API, то использование стандартных структур данных .NET является предпочтительным. Из минусов такого подхода следует отметить, что функциональные структуры данных, как правило, богаче и, в частности, содержат в себе реализацию списковых операций `map`, `filter` и др. Однако также не следует забывать, что в библиотеке F# содержится модуль Seq, определяющий операции типа `map`, `filter`, `fold` и др. для произвольного типа, поддерживающего интерфейс `IEnumerable`, что позволяет оперировать с любыми перечислимыми последовательностями в богатом функциональном стиле.

3. Типовые приемы функционального программирования

Мы надеемся, что, изучив предыдущие главы, читатель уже начал привыкать к функциональному стилю программирования и смог оценить для себя его преимущества. В этой главе мы рассмотрим ряд приемов, типичных для функционального программирования, которые позволят еще более эффективно использовать возможности этой парадигмы программирования.

3.1. Замыкания

В функциональных языках функции являются полноправными значениями, которые могут передаваться в качестве аргументов, связываться с именами, использоваться внутри структур данных и возвращаться в качестве результата работы других функций. Рассмотрим простой пример:

```
let filt = List.filter (fun x -> x%3=0) in
  [1..100] |> filt
```

Здесь мы определяем функцию `filt`, выбирающую из списка только числа, кратные трем, и затем применяем эту функцию для фильтрации списка чисел. Имя `filt` в данном случае связывается с функциональным значением типа `int list -> int list`.

Рассмотрим тот же пример, но записанный с использованием промежуточной переменной `n`:

```
let n = 3 in
  let filt = List.filter (fun x -> x%n=0) in
    [1..100] |> filt
```

Что в данном случае представляет из себя значение `filt`? В функциональном выражении `fun x -> x%n=0` используется ссылка на имя `n`, описанное вне данного определения во внешнем лексическом контексте. Поэтому определение анонимной функции `fun x -> x%n=0`, равно как и всей функции `filt`, должно содержать в себе ссылку на текущее значение имени `n`. В более общем случае, при определении функции, включающей в себя имена из внешнего лексического контекста, значения всех этих имен *на момент определения функции* должны быть зафиксированы и использованы в дальнейшем при вычислении функции. Такое функциональное значение, содержащее в себе, помимо собственно функции, слепок

переменных из внешнего контекста, на момент определения функции получило название *лексического замыкания* (lexical closure), или просто замыкания.

Таким образом, замыкания возникают всегда, когда мы определяем функцию, содержащую в себе некоторое имя из внешнего контекста. В более сложном случае замыкание может возвращаться как результат функции – как в примере ниже:

```
let divisible n = List.filter (fun x -> x%n=0)
let filt = divisible 3 in [1..100] |> filt
```

В этом примере функция `divisible` принимает на вход аргумент типа `int` и возвращает фильтрующую функцию, которая инкапсулирует в себе переданное на момент вызова `divisible` значение аргумента `n`. Таким образом, с помощью замыканий функциональные объекты могут содержать внутри себя некоторое внутреннее состояние.

Чтобы показать, что замыкание содержит в себе значение на момент создания замыкания, рассмотрим следующий диалог:

```
> let x = 4;;
> let adder y = x+y;;
> adder 1;;
val it : int = 5
> let x = 3;;
> adder 1;;
val it : int = 5
> adder;;
val it : (int -> int) = <fun:clo@17-5>
```

Мы видим, что при переопределении имени `x` замыкание по-прежнему ссылается на исходное имя, расположенное во внешней области видимости. Также видно, что значение `adder` представляет собой замыкание, – об этом говорят скобки вокруг типа `(int -> int)` и текст clo (от *closure* – замыкание) в значении мнемонической функциональной ссылки.

3.2. Динамическое связывание и mutable-переменные

Использование замыканий предполагает *статическое связывание имен*, то есть имена внешних переменных связываются со своими значениями на момент определения замыкания. В некоторых функциональных языках (например, в ЛИСПе) по умолчанию используется *динамическое связывание*, где переменные из внешней области видимости связываются на момент вызова замыкания. В таких языках для создания замыкания обычно используется специальная конструкция.

Чтобы смоделировать динамическое связывание на F#, необходимо познакомиться с понятием *изменяемых* (или мутирующих, mutable) переменных. Рассмотрим слегка видоизмененный пример из предыдущего раздела:

```
> let mutable x = 4;;
> let adder y = x+y;;
> adder 1;;
val it : int = 5
> x <- 3;;
> adder 1;;
val it : int = 4
```

В отличие от примера выше, здесь мы описали x как изменяемую переменную и использовали специальную операцию <- для изменения значения переменной. Значение изменяемой переменной не фиксируется в замыкании – вместо этого используется динамическое связывание, в результате чего после присваивания функция adder стала прибавлять уже новое значение.

Важно заметить, что mutable-переменные не являются, строго говоря, функциональным приемом программирования. Наоборот, введение mutable-переменных – это своего рода отход от чисто функционального языка, позволяющий, в частности, программировать даже в императивном стиле. Однако приведенное здесь применение mutable-переменных в замыканиях может использоваться для эмулирования имеющегося в ЛИСПе и других языках динамического связывания, позволяя менять в процессе исполнения внутреннее состояние замыкания. Как мы увидим в следующем разделе, это позволяет описывать весьма интересные функциональные объекты – генераторы.

3.3. Генераторы и ссылочные переменные ref

В предыдущей главе мы видели два способа представления последовательностей: списками и в виде функции от номера. Существует еще один способ задания последовательности – при помощи функции-генератора, которая будет при каждом вызове возвращать очередной член последовательности. В императивных .NET-языках наиболее близким эквивалентом генератора будет класс, реализующий интерфейс IEnumerable. В чисто функциональных языках реализация генератора затруднена из-за необходимости явно использовать окружение для хранения состояния, в то время как в F# и в других языках с динамическим связыванием (например, в ЛИСПе) мы можем использовать для реализации генераторов с состоянием замыкание с изменяемым полем.

Попробуем реализовать простейший генератор целых чисел, начиная с n. В F# нельзя непосредственно определять mutable-переменные внутри замыкания, поэтому мы используем специальный immutable-объект cell с mutable-полем:

```
type cell = { mutable content : int }
```

Функция new_counter будет создавать новый генератор, начинающийся с заданного числа n:

```
let new_counter n =
  let x = { content = n } in
  fun () ->
    (x.content <- x.content+1; x.content)
```

При создании генератора сначала формируется имя x, хранящее ячейку с начальным значением n, а затем возвращается анонимная функция, которая при вызове увеличивает содержимое счетчика на 1 и возвращает текущее значение.

Поскольку ситуация, когда желательно использовать изменяемые переменные там, где допустимы только immutable-значения, является достаточно типичной, F# предусматривает специальный синтаксис для изменяемых ячеек (здесь t – значение типа T, а R – изменяемая ячейка типа T ref):

Выражение	Тип	Действие
ref t	T ref	Создание ячейки
!R	T	Извлечение значения из ячейки
R := t	unit	Присвоение значения ячейке

С использованием такого синтаксиса приведенный выше пример с описанием генератора может быть записан так:

```
let new_counter n =
  let x = ref n in
  fun () ->
    (x := !x+1; !x)
```

Такой генератор определяет потенциально бесконечную последовательность, в том смысле что каждое обращение к функции генерирует очередной член. На практике в конечном итоге нас всегда интересует конечное количество элементов последовательности – например, чтобы их напечатать или агрегировать. Для этого нам будет полезна функция преобразования первых *n* элементов последовательности, задаваемой генератором, в список:

```
let rec take n gen =
   if n=0 then []
   else gen()::take (n-1) gen
```

Попробуем описать более общий вид генератора, который задается некоторой порождающей функцией fgen и начальным значением init:

```
let new_generator fgen init =
  let x = ref init in
  fun () ->
    (x:=fgen !x; !x)
```

Приведенный ранее пример счетчика легко определяется через такой более общий генератор следующим образом:

```
let new_counter n = new_generator (fun x-> x+1) n
```

Следует отметить, что состояние в таком генераторе может иметь весьма сложный вид. Например, чтобы определить генератор последовательности чисел Фибоначчи, будем использовать в качестве состояния пару чисел:

```
let fibgen = new_generator (fun (u,v) -> (u+v,u)) (1,1)
```

На каждом шаге пара чисел суммируется и подменяется парой из суммы и одного оставшегося числа. Такой генератор возвращает не просто последовательности чисел Фибоначчи, а последовательность пар. Чтобы получить из него последовательность самих чисел Фибоначчи, необходимо воспользоваться функцией, аналогичной функции `List.map`, примерно следующим образом:

```
let fib = map (fun (u,v) -> u) fibgen
```

Как же может быть описана функция map для генераторов? Если генератор представляет собой функцию, то применение map к генератору также должно возвращать генератор, который при каждом вызове получает очередное значение из исходного генератора (хранимого внутри замыкания), применяет к нему функцию отображения и возвращает результат:

```
let map f gen =
  fun () -> f (gen())
```

Аналогичным образом мы можем описать функцию `filter` – в этом случае нам придется воспользоваться вспомогательной функцией `repeat`, которая пропускает некоторое количество элементов последовательности, пока они удовлетворяют некоторому условию:

```
let rec repeat cond gen =
  let x = gen() in
  if cond x then x
  else repeat cond gen

let filter cond gen =
  fun () -> repeat cond gen
```

С использованием этих функций обработка последовательностей, задаваемых генераторами, будем происходить аналогично работе со списками. Например, если мы хотим получить первые 10 чисел Фибоначчи, делящихся на 3, то мы можем использовать такую конструкцию:

```
take 10 (filter (fun x -> x%3=0) fib)

> val it : int list =
    [3; 21; 144; 987; 6765; 46368; 317811; 2178309; 14930352; 102334155]
```

Обратите внимание, что в данном случае мы оперируем с достаточно большими числами, и попытка определить то же самое действие со списками привело бы к значительному расходу памяти под промежуточный список чисел Фибоначчи, из которого затем, после фильтрации, мы оставили бы список из 10 чисел. В данном случае, несмотря на идентичность записанного алгоритма, за счет использования генераторов промежуточный список не создается, требуемый алгоритм задается цепочкой применения функций к исходному генератору, а результат формируется в виде итогового списка «на лету».

Поскольку при использовании генераторов результирующие элементы последовательности формируются «по требованию» (в нашем примере такое формирование осуществляла функция `take`, преобразующая задаваемую генератором последовательность в список), то такие последовательности называются ленивыми последовательностями. Поскольку ленивые последовательности являются крайне важным элементом функционального стиля программирования, в F# существуют специальные синтаксические возможности для их описания, которые мы рассмотрим в следующем разделе.

3.4. Ленивые последовательности (seq)

Предыдущий пример с созданием генератора для последовательности чисел Фибоначчи может быть написан на F# с использованием последовательностей следующим образом:

```
let fibs = Seq.unfold
           (fun (u,v) -> Some(u,(u+v,u)))
           (1,1)
```

Функция `unfold` создает необходимый для задания последовательности генератор, который управляется некоторым внутренним состоянием. В нашем случае состояние – это пара идущих подряд чисел Фибоначчи, а функция смены состояния – это переход от пары (u,v) к паре $(u+v,u)$. При этом в качестве элементов сгенерированной последовательности надо возвращать первые элементы этой пары (u). Таким образом, функции `unfold` передаются два аргумента – функция смены состояния, которая по текущему состоянию возвращает пару из очередного элемента последовательности и нового состояния (в нашем случае это пара $(u,(u+v,u))$, и начальное значение состояния (1,1).

При этом функция смены состояния возвращает опциональное значение – когда она возвращает `None`, последовательность заканчивается. В нашем случае создается потенциально бесконечная последовательность.

Далее, чтобы решить задачу нахождения первых 10 чисел Фибоначчи, делящихся на три, мы можем использовать стандартные функции пакета `Seq`, аналогичные одноименным операциям со списками:

```
Seq.take 10 (Seq.filter (fun x -> x%3=0) fibs)
```

В отличие от списковых аналогов, функции над последовательностями не возвращают целиком полученные наборы элементов, а лишь оперируют функциями-генераторами, которые в случае необходимости могут получить необходимые значения из первоначального генератора последовательности. Для явной конвертации последовательности в список необходимо использовать функцию `Seq.toList` – при этом в явном виде срабатывают все цепочки функций-генераторов и формируется набор значений в памяти в виде списка:

```
Seq.take 10 (Seq.filter (fun x -> x%3=0) fibs) |> Seq.toList
```

Существуют также другие способы генерации последовательностей: например, с помощью функции, определяющей каждый элемент последовательности по его номеру:

```
let squares = Seq.init_infinite(fun n -> n*n)
let squares10 = Seq.init_finite 10 (fun n-> n*n)
```

Также бывает удобно задавать последовательности с помощью специальной конструкции `seq {…}` – внутри окруженного такой конструкцией фрагмента кода используются операции `yield` для возврата очередного элемента. Вот как, например, можно определить функцию, возвращающую ленивую последовательность строк текстового файла:

```
open System.IO
let ReadLines fn =
  seq { use inp = File.OpenText fn in
        while not(inp.EndOfStream) do
            yield (inp.ReadLine())
      }
```

Используя такое определение, мы можем обрабатывать файл, по сути дела не считывая его целиком в память. Пусть, например, у нас имеется большой файл с данными в формате CSV (данные, разделенные запятой) и нам нужно подсчитать сумму третьего по счету столбца:

```
let table = ReadLines "csvsample.txt"
|> Seq.map (fun s -> s.Split([|','|]))
let sum_age = table |> Seq.fold(fun x l -> x+Int32.Parse(l.[2])) 0
```

Для построения простых последовательностей, задаваемых перечислением или диапазоном значений, могут использоваться более простые конструкции, называемые range expressions и sequence comprehension:

```
seq { 1I..100000000000000I } // генерация seq of BigInteger
seq { for i in 1..10 -> (i,i*i) }
  // то же, что Seq.map (fun i->(i,i*i)) [1..10]
```

Последняя конструкция, по сути, эквивалентна операции `map`, но выглядит более наглядно – поэтому на ее основе можно строить достаточно сложную обработку последовательностей. Например, рассмотренное ранее нахождение первых 10 чисел Фибоначчи, делящихся на 3, может быть записано так:

```
seq { for x in fibs do if x%3=0 then yield x } |> Seq.take 10
```

В этом случае конструкция `seq {...}` сочетает в себе возможности операций `map` и `filter`.

3.4.1. Построение частотного словаря текстового файла

Рассмотрим чуть более содержательный пример, когда нам нужно построить частотный словарь некоторого (потенциально очень большого) файла, то есть посчитать, сколько в нем раз встречаются те или иные слова, и вывести 10 наиболее часто встречающихся слов. Для начала построим последовательность из слов файла – для этого возьмем последовательность строк, сгенерированную `ReadLines`, разобьем ее на слова при помощи `String.Split` и соберем все слова вместе в одну последовательность с помощью стандартной функции `Seq.collect`:

```
ReadLines @"c:\books\prince.txt" |>
Seq.collect (fun s -> s.Split([|',';' ';':';'!';'.'|]))
```

Для построения частотного словаря последовательности слов необходимо вначале осознать, что, по сути дела, мы строим по списку некоторый агрегатный результат – пусть не такой простой, как количество или суммарная длина слов, которые могут быть выражены числом. Отсюда вытекает, что необходимо использовать стандартную операцию свертки (`Seq.fold`) с частотным словарем в качестве состояния. Для представления частотного словаря можно использовать хеш-таблицу `Map` с ключами строкового типа, содержащую для каждого слова соответствующее ему количество вхождений. При построении свертки мы смотрим на каждое очередное слово и, если оно есть в таблице, увеличиваем связанный с ним счетчик на единицу, в противном случае добавляем его в таблицу с счетчиком 1:

```
let FreqDict S =
   Seq.fold (
     fun (ht:Map<_,int>) v ->
      if Map.containsKey v ht then Map.add v ((Map.find v ht)+1) ht
      else Map.add v 1 ht)
      (Map.empty) S
```

Далее для решения задачи остается только преобразовать хеш-таблицу в список для удобства обработки, исключить из него редко встречающиеся или слишком короткие слова, отсортировать в нужном порядке, после чего взять 10 первых элементов:

```
ReadLines @"c:\books\prince.txt" |>
Seq.collect (fun s -> s.Split([|',';' ';':';'!';'.'|])) |>
FreqDict |>
Map.toList |>
List.sortWith(fun (k1,v1) (k2,v2) -> -compare v1 v2) |>
List.filter(fun (k,v) -> k.Length>3) |>
Seq.take 10
```

Еще раз хотелось бы обратить внимание, что размер обрабатываемого файла в данном случае практически не ограничен, поскольку в памяти находится только частотный словарь, представляемый хеш-таблицей, а строки файла подкачиваются с диска по мере обработки.

3.4.2. Вычисление числа π методом Монте-Карло

Отметим, что ленивые последовательности могут использоваться не только для обработки больших файлов – они также удобны для представления потенциально бесконечных последовательностей в математических задачах. Рассмотрим задачу вычисления площади фигуры вероятностным методом Монте-Карло и применим этот метод для нахождения числа π.

Метод Монте-Карло состоит в следующем. Рассмотрим квадрат со стороной R и некоторую фигуру, определяемую функцией принадлежности h(x,y). Будем случайным образом «бросать» внутри квадрата N точек (x_i,y_i) и посчитаем количество точек, попавших внутрь фигуры: $H = \#\{ (x_i,y_i) \mid h(x_i,y_i), 1 \le i \le N \}$. Тогда очевидно, что отношение числа попаданий к общему числу бросков будет примерно равно отношению площади фигуры S к площади квадрата: $H/N \approx S/R^2$. Отсюда мы можем найти площадь фигуры $S = R^2H/N$.

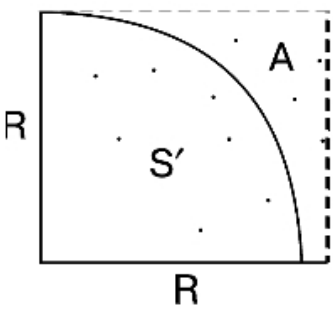

Рис. 3.1. Вычисление числа пи методом Монте-Карло

Для вычисления числа пи рассмотрим четверть круга, вписанного в квадрат, как показано на рис. 3.1. В этом случае функция принадлежности $h(x,y)=x^2+y^2\le R^2$, и $S = \pi R^2/4=R^2H/N$. Получаем, что $\pi=4H/N$.

Для начала определим функцию, создающую бесконечную последовательность псевдослучайных чисел. Используем содержащуюся в F# Power Pack функцию `Seq.generate`, которая принимает три аргумента: функцию-конструктор, возвращающую некоторый объект-состояние, функцию-итератор, которая по состоянию генерирует очередной член последовательности, и функцию-деструктор. Для генерации псевдослучайных величин мы исполь-

зуем встроенный механизм .NET Framework, передавая начальное значение псевдослучайного генератора в виде параметра:

```
let rand max n =
   Seq.generate
    (fun () -> new System.Random(n))
    (fun r -> Some(r.NextDouble()*max))
    (fun _ -> ())
```

Параметр max задает диапазон генерируемых значений – от 0 до max.

Альтернативно мы можем задать бесконечную последовательность непосредственно с помощью конструкции seq следующим образом:

```
let rand max n =
   seq {
     let r = new System.Random(n)
     while true do yield r.NextDouble()*max
   }
```

С точки зрения современного стиля программирования на F#, второй вариант считается более предпочтительным, хотя первый выглядит более «функционально».

Далее определим функцию MonteCarlo, которая вычисляет отношение H/N для произвольной функции принадлежности и заданного радиуса R и количества итераций N. Ее можно определить следующим образом:

```
let MonteCarlo hit R N =
  let hits = (float)(
      Seq.zip (rand R 134) (rand R 313) |> // составляем послед.пар
      Seq.take N |>                        // берем первые N элементов
      Seq.filter hit |>   // оставляем только попавшие в фигуру точки
      Seq.length) in      // считаем их количество
  hits/((float)N)
```

С помощью этой функции уже очень легко вычислить число пи:

```
let pi' = 4.0*MonteCarlo (fun (x,y) -> x*x+y*y<=1.0) 1.0 10000
```

Обратите внимание, что, несмотря на описываемую в тексте программы работу с длинными последовательностями, на самом деле эти последовательности не создаются и не занимают место в памяти, – описанная последовательность действий транслируется в цепочку функциональных вызовов, и на этапе подсчета количества попаданий (когда вызывается Seq.length) происходит циклическая последовательная обработка каждой из точек, сгенерированных псевдослучайным генератором Random.

3.5. Ленивые и энергичные вычисления

Ленивые последовательности на самом деле являются лишь одним из частных случаев более широкого понятия *ленивых вычислений*. Для функционального программирования понятие энергичной и ленивой стратегии вычислений является фундаментальным – существуют даже целые языки, например Haskell, использующие ленивую стратегию вычислений. Попробуем разобраться в этих понятиях.

В функциональном программировании основной операцией выступает применение функции к аргументу – аппликация. При этом аргумент, в свою очередь, может быть некоторым функциональным выражением, включающим дополнительные функциональные вызовы. Например:

```
let NoLongLines f = length (filter IsLineLong (ReadLines f))
```

Наиболее очевидная стратегия вызова функции $f\ x$ – это сначала вычислить аргумент x и затем передать в функцию уже вычисленное значение. Примерно так работает энергичная стратегия вычислений – в нашем примере сначала будет прочитан файл, выполнится `ReadLines f`, после чего отработает фильтрация, а затем вычислится длина получившейся последовательности строк.

В случае с ленивой стратегией вычисление выражения откладывается до самого последнего момента, когда нам нужно его значение. В нашем примере функция `length` в качестве аргумента получит целиком содержащееся в скобках выражение, после чего, когда для вычисления длины нам понадобится первый элемент списка, она начнет вызывать функцию `filter`. Эта функция, в свою очередь, будет совершать итерацию по списку, вызывая в случае необходимости функцию `ReadLines`.

Таким образом, при ленивой стратегии вычислений достаточно сложно предсказать, в каком порядке будут совершаться вычисления. Рассмотрим гипотетическую программу для вычисления корней квадратного уравнения:

```
let Solve =
   let a = Read "Введите a"
   let b = Read "Введите b"
   let c = Read "Введите c"
   let d = b*b-4.*a*c
   Print ((-b+sqrt(d))/2./a,(-b-sqrt(d))/2./a)
```

В случае с энергичной стратегией все просто: выполнение операции `let a = Read «Введите a»` приводит к вызову функции Read и диалогу с пользователем, в результате чего мы получаем стандартное приглашение для ввода трех коэффициентов, после чего полученные корни печатаются на экране.

Для ленивой последовательности вычислений все выглядит по-другому[1]. Соответствие имен a, b и c и соответствующих действий запоминалось бы, но сами действия выполнялись бы в последнюю очередь, когда возникает потребность оперировать значениями. Вычисления бы инициировались изнутри функции Print, что привело бы к вызову вычисления дискриминанта, который, в свою очередь, запросил бы ввод значения *b*, а затем уже *a* и *c*.

В языке F# используется энергичная стратегия вычислений. Это оправдано, поскольку требуется прозрачное взаимодействие с .NET Framework и компонентами, написанными на других языках платформы .NET, преимущественно императивными. Также энергичный порядок вычислений более привычен для программистов, первоначально знакомящихся с императивным подходом к программированию, что позволит большинству программистов проще использовать F# в тех задачах, где он действительно эффективен.

Однако в F# есть средства для поддержки ленивых вычислений. Один из механизмов – это рассмотренные выше ленивые последовательности, реализованные через генераторы. Другой механизм – это использование явных конструкций откладывания вычислений lazy/force.

Рассмотрим приведенный выше пример решения квадратного уравнения и добавим конструкции lazy для явного «откладывания» вычислений, где необходимо, и вызов метода Force() в том месте, где нужно получить вычисленное значение:

```
let Solve =
   let a = lazy(Read "Введите a")
   let b = lazy(Read "Введите b")
   let c = lazy(Read "Введите c")
   let d = lazy(b.Force()*b.Force()-4.*a.Force()*c.Force())
   Print ((-b.Force()+sqrt(d.Force()))/2./a.Force(),
(-b.Force()-sqrt(d.Force()))/2./a.Force())
```

Конструкция lazy(…), будучи применена к выражению типа T, возвращает значение типа Lazy<T>, содержащее в себе исходное выражение в невычисленной форме. При первом вызове метода Force() происходит вычисление выражения, и полученное значение сохраняется. Последующие вызовы Force() не приводят к повторному вычислению выражения, а сразу возвращается вычисленное ранее значение – именно поэтому значение b не запрашивается в программе выше несколько раз. Такое запоминание значений называется *мемоизацией* – это понятие мы более подробно рассмотрим в следующем разделе.

---

[1] Здесь мы рассматриваем весьма упрощенный пример. В реальной жизни в чисто-функциональных ленивых языках типа Haskell принципиально невозможны функции с побочным эффектом, которые мы используем в этом примере. В реальности функции ввода-вывода были бы снабжены аргументами, фиксирующими состояние внешней среды, и последовательность ввода-вывода соответствовала бы порядку записи таких операций в программе.

Ленивые вычисления могут использоваться вместо генераторов для определения ленивых последовательностей. Рассмотрим определение потока Stream<'a>, аналогичного по свойствам последовательности Seq<'a>:

```
type 'a SeqCell = Nil | Cons of 'a * 'a Stream
and  'a Stream = Lazy<'a SeqCell>
```

Последовательность представляется ленивой ячейкой, которая может содержать либо признак конца последовательности Nil, либо конструктор Cons, объединяющий голову и хвост. Для задания таких ленивых последовательностей в коде приходится в явном виде описывать каждую ячейку последовательности (либо определять функцию, конвертирующую List<'a> или Seq<'a> в Stream<'a> – сделать это мы предлагаем читателю в качестве самостоятельного упражнения):

```
let sample1 = lazy(Cons(1,lazy(Cons(2,lazy(Nil)))))
let sample2 = lazy(Cons(3,lazy(Cons(4,lazy(Nil)))))
```

Работа с такими ленивыми потоками сводится к тому, что на каждом шаге мы форсируем (Force()) вычисление очередной ячейки, то есть очередного члена последовательности. Рассмотрим в качестве примера реализацию функции конкатенации ленивых потоков:

```
let rec concat (s:'a Stream) (t: 'a Stream) =
    match s.Force() with
      Nil -> t
    | Cons(x,y) -> lazy(Cons(x,(concat y t)))
```

Такая функция по двум потокам возвращает поток, то есть ленивое значение, которое может быть при необходимости вычислено. Сама функция производит форсирование (вычисление) только первой ячейки, чтобы убедиться, является ли первый поток пустым. Если нет – рекурсивно вызывается функция concat, а результат снова «заворачивается» в ленивую ячейку SeqCell. Для пустого потока возвращается второй поток.

Ленивые вычисления могут применяться в тех случаях, когда вызов функции связан с большими накладными расходами, которые было бы полезно отложить «на потом». Ведь при использовании ленивых вычислений ленивое значение может никогда не понадобиться! Примером ленивого поведения является условный оператор if…then…else – в случае истинности условного выражения второе подвыражение никогда не вычисляется. Ленивые вычисления позволяют распространить такое поведение «вычислений по необходимости» на весь программный код.

Несмотря на все преимущества ленивых вычислений, их постоянное использование по умолчанию ведет к снижению производительности языка и затрудняет понимание программы разработчиком, привыкшим к традиционным алгоритми-

ческим языкам, использующим энергичные схемы. Поэтому в языке F# (и OCaml) по умолчанию используется именно энергичная стратегия.

3.6. Мемоизация

Как упоминалось ранее, при использовании ленивых вычислений важно уметь запоминать промежуточные результаты вычисления выражений, чтобы не производить каждый раз вычисления повторно. Чтобы проиллюстрировать эту идею, рассмотрим функцию возведения числа в квадрат:

```
let sqr x = x*x
```

Предположим, нам необходимо вычислить квадрат 10-го числа Фибоначчи (для вычисления которого будем использовать рассмотренную нами функцию `fib`), то есть получить значение `sqr(fib 10)`. При использовании энергичной стратегии сначала будет вычислено `fib 10`, а затем вызвана функция `sqr`, то есть возникнет следующая последовательность вычислений: `sqr(fib 10)` → `sqr 89` → `89*89` → `7921`. В случае ленивых вычислений последовательность будет другой (не вдаваясь в подробности вычисления самой функции `fib` в ленивом режиме): `sqr(fib 10)` → `(fib 10) * (fib 10)` → `89 * (fib 10)` → `89 * 89` → `7921`. Как видно, без использования мемоизации возникает проблема повторного вычисления функции `fib 10`, называемая *проблемой разделения*[1].

Реализация приведенной выше функции на F# выглядит следующим образом:

```
let lsqr (x: Lazy<int>) = lazy(x.Force()*x.Force())
Print (lsqr(lazy (Read "Enter number:")).Force())
```

Запустив этот код, мы увидим, что считывание входного аргумента происходит только один раз – благодаря использованию мемоизации.

Мемоизация может применяться не только с ленивыми вычислениями. Например, рассмотрим упомянутую выше функцию вычисления чисел Фибоначчи «наивным» способом:

```
let rec fib n =
  if n<2 then 1 else fib (n-1) + fib(n-2)
```

Вычисление, например, `fib 5` происходит следующим образом (опустив для простоты применение условного оператора):

[1] Такое название обусловлено тем, что при вызове функции sqr ее аргумент может несколько раз входить в определение функции, что приводит к тому, что он как бы разделяется на несколько копий. Реализация ленивых языков с мемоизацией требует, чтобы такие копии были связаны и вычислялись синхронно, то есть не более одного раза.

fib 5 → fib 4 + fib 3 → (fib 3 + fib 2) + (fib 2 + fib 1) → ((fib 2 + fib 1) + + (fib 1 + fib 0)) + ((fib 1 + fib 0) + fib 1) → ((fib 0 + fib 1)+1)+(1+1)+(1+1)+1) → (1+1+1+1+1+1+1+1) → 8

Из этого примера мы видим, что вычисление `fib 3` и `fib 2` производилось несколько раз повторно, благодаря чему в общем случае сложность такого алгоритма получается равной $O(2^n)$. Использование мемоизации в данном случае позволяет получить алгоритм линейной сложности – однако ценою расходования памяти для запоминания промежуточных результатов.

Мемоизацию здесь можно в явном виде запрограммировать использованием хеш-таблицы для запоминания результатов вычислений:

```
open System.Collections.Generic
let mfib =
   let d = new Dictionary<int,int>()
   let rec fib n =
     if d.ContainsKey(n) then d.[n]
     else
       let res = if n<2 then 1 else fib (n-1) + fib(n-2)
       d.Add(n,res)
       res
   fun n -> fib n
```

Здесь мы определяем основную функцию `mfib`, внутри которой создаются хеш-таблица для запоминания вычисленных значений и вложенная рекурсивная функция `fib`, внутри которой сначала производится проверка наличия уже вычисленного для данного аргумента значения в хеш-таблице, и если такое значение есть – оно сразу возвращается, в противном случае производятся вычисления и результат запоминается в хеш-таблице.

Понятно, что такая мемоизация «вручную» не очень удобна и требует написания существенного объема дополнительного кода. Однако функциональный подход предоставляет все возможности для отделения механизма мемоизации от самого вычисления функции, благодаря чему мы можем определить абстрактную функцию мемоизации `memoize`, которую можно будет использовать для мемоизации произвольных функций:

```
let memoize (f: 'a -> 'b) =
   let t = new System.Collections.Generic.Dictionary<'a,'b>()
   fun n ->
     if t.ContainsKey(n) then t.[n]
     else let res = f n
          t.Add(n,res)
          res
```

Для описания мемоизованной функции вычисления последовательности Фибоначчи в этом случае потребуется написать:

```
let rec fibFast =
       memoize (
         fun n -> if n < 2 then 1 else fibFast(n-1) + fibFast(n-2))
```

На этом примере можно лишний раз убедиться в возможностях функциональной декомпозиции, которая позволяет эффективно логически разделить процесс вычисления функции и запоминания результатов вычислений, предоставляя программисту простой синтаксический способ применения явной мемоизации.

3.7. Продолжения

Мы уже знаем, что в функциональном программировании функции являются полноправными значениями, поэтому некоторые приемы программирования основаны на манипулировании функциями. Один из таких подходов – *продолжения*. Его суть состоит в том, что мы передаем в качестве аргумента функции другую функцию, которую следует вызвать по окончании вычислений, таким образом продолжив процесс вычислений.

В качестве примера рассмотрим задачу реверсирования списка с использованием продолжений. В простейшем случае реверсирование реализуется следующим образом с помощью нехвостовой рекурсии:

```
let rec rev L =
  match L with
    [] -> []
  | h::t -> rev t @ [h]
```

Посмотрим, как использование продолжений позволит сделать рекурсию хвостовой. В этом случае в функцию реверсирования `rv` мы также будем передавать функцию-продолжение `f`. На каждом шаге от списка будет «откусываться» головной элемент `h`, а функция-продолжение будет дополняться присоединением значения `h` к голове списка-аргумента. Таким образом, функция-продолжение на каждом шаге будет представлять собой функцию, применив которую к пустому списку мы дополним его необходимыми элементами:

```
let rec rv l f =
    match l with
    [] -> (f [])
    | h::t -> rv t (f>>(fun x -> h::x))
```

Исходная функция `rev` в этом случае запишется так:

```
let rev l = rv l (fun x -> x)
```

По сути дела, в ходе выполнения rv исходный список разворачивается в цепочку функциональных вызовов, которая передается между рекурсивными вызовами функции, а по окончании погружения в рекурсию применяется к пустому списку, чтобы сконструировать требуемый результат.

Мы можем записать то же самое более «правильно», реализуя rv как вложенную функцию:

```
let rev L =
  let rec rv l f =
     match l with
       [] -> (f [])
     | h::t -> rv t (f>>(fun x -> h::x))
  rv L (fun x -> x)
```

Приведенный пример не является оптимальным способом реализации реверсирования – как мы видели ранее, это можно сделать проще, сразу конструируя результирующий список по ходу погружения в хвостовую рекурсию. Однако подход с использованием продолжений позволяет свести к хвостовой даже нелинейную рекурсию! Посмотрим, как это делается на примере функции, вычисляющей количество элементов в двоичном дереве. В простейшем случае эта функция выглядит так:

```
type 't tree = Nil | Node of 't*('t tree)*('t tree)

let rec size = function
   Nil -> 0
 | Node(_,L,R) -> 1+(size L)+(size R)
```

При использовании продолжений мы будем формировать и передавать через рекурсивные вызовы функцию-продолжение с целочисленным аргументом: эта функция будет прибавлять к аргументу количество аргументов в уже обработанной части дерева. Когда мы доходим до конца (листа) дерева по какому-то из направлений, функция-продолжение применяется к 0, возвращая размер соответствующей ветки:

```
let size t =
  let rec size' t cont =
    match t with
      Nil -> cont 0
    | Node(_,L,R) ->
       size' L (fun x1 ->
         size' R (fun x2 ->
           cont(x1+x2+1))) in
  size' t (fun x->x)
```

Функция-продолжение в данном случае формируется хитрым способом. Вначале мы порождаем функцию с аргументом `x1`, которая рекурсивно вызывает `size'` для левого поддерева, передавая в качестве аргумента еще одну функцию-продолжение с аргументом `x2`, формирующую продолжение для правого поддерева. В результате в памяти формируется древовидная структура функциональных вызовов, вычисление которой и дает искомый результат.

Может показаться, что продолжения используются только для сведения рекурсии к хвостовой. Однако это не совсем так: далее в книге мы встретим использование продолжений, например для описания параллельных и асинхронных вычислений.

4. Императивные и объектно-ориентированные возможности F#

4.1. Мультипарадигмальность языка F#

Мы уже говорили про то, что традиционные языки программирования являются императивными – в том смысле, что программы на таких языках состоят из конструкций-операторов, которые изменяют состояние памяти. Однако на практике современные языки программирования поддерживают не только одну парадигму программирования. Например, в языке программирования C# есть много функциональных элементов: лямбда-выражения, анонимные типы и т. д., – более того, внутри C# существует специальный синтаксис LINQ (Language-Integrated Query), который создает внутри императивного языка некоторое функциональное ядро.

Точно так же F# не является чисто функциональным языком. Поскольку он глубоко интегрирован с платформой .NET, ему приходится поддерживать изменяемые структуры данных, побочные эффекты, объектную ориентированность и другие свойства языков этой платформы. Мы уже сталкивались со многими императивными возможностями языка F# – с поддержкой массивов, с изменяемыми переменными и изменяемыми структурами данных. В этом разделе мы опишем некоторые чисто императивные конструкции F#, а также объектно-ориентированный синтаксис языка, который будет необходим для эффективного взаимодействия с другими языками на платформе .NET.

Из-за мультипарадигмальности языка может возникнуть вопрос, какой стиль программирования использовать. Иногда программисты, привыкшие к императивным языкам программирования, начинают программировать на F# в императивном стиле, из-за чего получается не очень красивый и иногда более громоздкий, нежели на C#, код. Мы сознательно отложили рассмотрение императивных возможностей F# до этой главы, понемногу рассказывая про различные императивные конструкции по ходу изложения, чтобы предварительно познакомить читателей с функциональным стилем программирования с минимальными императивными включениями. Именно такой стиль программирования будет наиболее эффективным при использовании F#, поскольку позволит во многом сохранить преимущества чистого функционального подхода, локализовать побочные эффек-

ты и добиться краткого кода, который содержит минимум ошибок и может эффективно распараллеливаться.

4.2. Элементы императивного программирования на F#

Мы не будем подробно рассматривать все императивные элементы F#, ограничившись несколькими простыми примерами, которые проиллюстрируют имеющиеся в языке возможности.

4.2.1. Использование изменяемых переменных и ссылок

Одно из основных отличий в стиле программирования, которое бросается в глаза, – это отсутствие в функциональных языках программирования циклов и необходимость использования рекурсии. Мы уже видели и многократно использовали цикл `for` – однако во многих случаях это было лишь безобидной заменой функции `for_loop`, описанной нами в первой главе. Основное отличие такого использования от императивных языков состоит в том, что внутри тела цикла нет возможности модифицировать какие-либо элементы и менять состояние, то есть все итерации цикла выполняются независимо друг от друга, без связи по данным.

Для того чтобы использовать императивный стиль программирования, достаточно добавить к такому циклу `for` изменяемые переменные. Мы уже встречались в главе 3 с изменяемыми mutable-переменными. С их использованием, например, можно закодировать суммирование элементов списка обычным для императивного программиста способом:

```
let sum_list l =
   let mutable acc = 0
   for x in l do
     acc <- acc+x
   acc
```

Здесь мы использовали конструкцию `<-` для изменения значения переменной, в остальном работа с такой переменной ничем не отличается от обычной.

Альтернативой изменяемым переменным служат ссылочные ref-переменные, которые представляют собой неизменяемую переменную типа запись с изменяемым полем внутри. С их использованием суммирование запишется следующим образом:

```
let sum_list l =
   let acc = ref 0
   for x in l do
      acc := !acc+x
   !acc
```

Мы видим, что ! используется для доступа к значению переменной, а для присвоения значения используется оператор присваивания :=. Несмотря на то что с использованием mutable-переменных алгоритм выглядит проще и нагляднее, использование ref-переменных хорошо тем, что в явном виде приходится описывать операции доступа, что лишний раз напоминает программисту о возможных побочных эффектах.

Лишний раз хотелось бы подчеркнуть, что приведенные здесь примеры лишь демонстрируют синтаксис языка, и такую реализацию ни в коем случае не следует предпочитать рекурсивному функциональному стилю программирования с использованием хвостовой рекурсии, который мы видели ранее в книге. Вдумчивый читатель мог уже оценить недостатки такого подхода, в котором каждая итерация цикла зависит от предыдущей и не может быть выполнена независимо и параллельно. Использование свертки fold имеет несомненные преимущества, поскольку оно сохраняет семантику независимости отдельных проходов свертки, то есть при желании можно будет легко перейти к параллельному коду простой заменой функции fold на параллельную версию. В приведенном императивном фрагменте кода такое полуавтоматическое распараллеливание невозможно в принципе.

4.2.2. Цикл с предусловием

Другой важной циклической конструкцией является цикл с пред- и постусловием. В функциональных языках такая конструкция не может быть использована напрямую, поскольку она требует изменения состояния, для того чтобы условие выхода из цикла могло выполниться. Ближайшим аналогом цикла с предусловием может быть выполнение операции над элементами последовательности до тех пор, пока не выполнится условие.

В языке F# есть конструкция цикла `while`, которую можно проиллюстрировать на следующем примере. Ранее в главе 2 мы показывали, как использовать `ResizeArray` для ввода строк с клавиатуры до тех пор, пока не будет введена точка. В соответствующем примере мы использовали рекурсию, здесь же посмотрим, как можно использовать для этого цикл с предусловием:

```
let ReadLines() =
   let a = new ResizeArray<string>()
   let mutable s = ""
   while s<>"." do
     s <- Console.ReadLine()
     a.Add(s)
   List.ofSeq a
```

С точки зрения порождаемого кода, как явный цикл с предусловием, так и хвостовая рекурсия практически эквивалентны. Мы рекомендуем по возможности воздерживаться от использования циклов с предусловием, чтобы научиться получать эстетическое удовольствие от функционального стиля программирования.

4.2.3. Условный оператор

В императивном стиле программирования возможно использование условного оператора `if` не для выбора значения, а для указания потока выполнения программы, например:

```
let print_sign x =
   if x>0 then printfn "Positive"
   elif x=0 then printfn "Zero"
   else printfn "Negative"
```

На самом деле такой оператор, по сути, является частным случаем обычного условного оператора, но возвращающего тип `unit`. Единственным отличием служит то, что для такого условного оператора допускается опускать ветку `else`:

```
let warn_if_negative x =
  if x<0 then printfn "Negative!!!"
```

4.2.4. Null-значения

В C#, равно как и в других языках на платформе .NET, принято использовать значение `null`, которое обычно указывает на неинициализированную переменную или на специальное выделенное значение. Поскольку в функциональном программировании нет переменных, а любые имена связываются со значением, необходимость в использовании `null` отпадает, а в ситуациях, когда функция должна вернуть некоторое выделенное значение, используется опциональный тип.

Однако при взаимодействии с другими библиотеками .NET функции могут вернуть значение `null`. Об этом не стоит забывать, поскольку в противном случае могут возникнуть непредвиденные исключительные ситуации в функциональном коде. Лучшим решением будет в явном виде преобразовывать `null`-значения в функциональный тип, например:

```
let getenv s = match System.Environment.GetEnvironmentVariable s with
                 null -> None
               | x -> Some(x)
```

4.2.5. Обработка исключительных ситуаций

Рассмотрим функцию чтения текстового файла в строку:

```
let ReadFile f =
   let fi = File.OpenText(f)
   let s = fi.ReadToEnd()
   fi.Close()
   s
```

В этой функции возможно возникновение целого ряда исключительных ситуаций: ей может быть передан неверный путь, содержащий недопустимые символы, или же файл может не существовать. Три возможные исключительные ситуации проиллюстрированы ниже:

```
ReadFile "dd\\:/"              // NotSupportedException
ReadFile "c:\nonexistant.txt"  // ArgumentException
ReadFile @"c:\nonexistant.txt" // FileNotFoundException
```

Мы можем обработать эти исключительные ситуации внутри нашей функции с помощью конструкции обработки исключений следующим образом:

```
let ReadFile f =
   try
     let fi = File.OpenText(f)
     let s = fi.ReadToEnd()
     fi.Close()
     Some(s)
   with
     | :? FileNotFoundException -> eprintfn "File not found"; None
     | :? NotSupportedException
     | :? ArgumentException -> eprintfn "Illegal path"; None
     | _ -> eprintfn "Unknown error"; None
```

В этом случае функция будет возвращать значение типа `string option`, а возникшие ошибки будут отображаться в стандартном потоке сообщений об ошибках – для этого служит функция `eprintf`. Операция `:?` позволяет проверить соответствие типа исключения и использовать различные пути выполнения в этом случае.

Альтернативно мы можем предпочесть ситуацию, чтобы функция `ReadFile` генерировала исключения, – в этом случае необходимо убедиться, что открытый поток ввода-вывода будет закрыт даже в том случае, если возникло исключение. Для этого используется конструкция `try...finally`:

```
let ReadFile f =
   let fi = File.OpenText(f)
   try
     let s = fi.ReadToEnd()
     s
   finally
     fi.Close()
```

Однако если смысл конструкции `finally` только в том, чтобы закрывать открытые ресурсы, проще использовать конструкцию `use` (аналогичную `using` в C#), которая обеспечивает освобождение ресурсов, как только инициализированная с помощью нее переменная выходит из области видимости.

Мы также можем предпочесть определить свой тип исключения для простоты его дальнейшей обработки и генерировать его в случае возникновения исклю-

чительной ситуации в нашей функции. С учетом этого окончательный вариант функции ReadFile может быть записан так:

```
exception CannotReadFile of string
let ReadFile f =
   try
     use fi = File.OpenText(f)
     fi.ReadToEnd()
   with
     | :? FileNotFoundException | :? NotSupportedException
     | :? ArgumentException -> raise (CannotReadFile(f))
     | _ -> failwith "Unknown error"
```

Здесь exception описывает новый тип исключительной ситуации с одним аргументом типа string, raise – генерирует возникновение этой ситуации, а failwith вызывает общую исключительную ситуацию System.Exception. Использование use гарантирует правильное закрытие ресурсов при выходе из функции.

Для обработки данной исключительной ситуации можно использовать следующую конструкцию:

```
try
 ReadFile @"c:\nonexistant.txt"
with
 CannotReadFile(f) -> eprintfn "Cannot read file: %s" f; ""
```

4.3. Объектно-ориентированное программирование на F#

Существуют различные подходы к реализации объектной ориентированности на функциональных языках. Одним из традиционных подходов считается использование замыканий для хранения инкапсулированного внутреннего состояния объектов с предоставлением доступа к этому состоянию через соответствующие функции.

F# основан на платформе .NET, поэтому он поддерживает соответствующую объектную модель, принятую в CLR. Однако F# вносит в нее некоторые изменения и расширения на уровне языка, делая более удобной для функционального стиля программирования.

4.3.1. Записи

Удобным средством представления набора значений и первым шагом в построении объектной модели являются записи. В некотором смысле упорядоченные кортежи хорошо справляются с задачей упаковки нескольких значений в единый объект – поэтому до сих пор мы успешно писали многие примеры без использова-

ния записей. По сравнению с кортежами, записи позволяют использовать именованные поля, например:

```
type Point = { x : float; y : float }
let p1 = { x=10.0; y=10.0 }
let p2 = { new Point with x=10.0 and y=0.0 }
```

Как мы видим, для создания значений типа запись существуют два разных синтаксиса. Первый, упрощенный синтаксис имеет смысл использовать тогда, когда компилятор может автоматически распознать тип значения. Однако если у нас есть другой тип, использующий те же имена полей, например:

```
type Circle = { x : float; y : float; r : float }
```

то необходимо будет в явном виде указывать тип создаваемой записи.

Для доступа к полям записи мы используем обычную точечную нотацию:

```
let distance a b =
   let sqr x = x*x
   Math.Sqrt(sqr(a.x-b.x)+sqr(a.y-b.y))
```

Запись может также использоваться при сопоставлении с образцом, например:

```
let quadrant p =
  match p with
     { x=0.0; y=0.0 } -> 0
   |  { x=u; y=v } when u>=0.0 && v>=0.0 -> 1
   |  { x=u; y=v } when u>=0.0 && v<0.0 -> 2
   |  { x=u; y=v } when u<0.0 && v<0.0 -> 3
   |  { x=u; y=v } when u<0.0 && v>=0.0 -> 4
```

4.3.2. Моделирование объектной ориентированности через записи и замыкания

В принципе, значительная часть объектной ориентированности может быть смоделирована через замыкания, как это делается в классическом функциональном программировании. Предположим, мы хотим описать различные классы геометрических фигур, для которых будет некоторый общий набор методов: рисование, вычисление площади и т. д. Мы можем описать набор методов в виде записи, которая будет содержать в себе функциональные переменные с типами, соответствующими необходимым операциям:

```
type Shape = { Draw : unit -> unit; Area : unit -> float }
```

Далее для описания самих объектов опишем функции-конструкторы, которые будут сохранять параметры фигуры (координаты центра, радиус или длину стороны) внутри замыкания и возвращать соответствующую запись типа Shape с заполненными процедурами работы с объектом:

```
let circle c r =
    let cent,rad = c,r
    { Draw = fun () -> printfn "Circle @(%f,%f), r=%f" cent.x cent.y rad;
      Area = fun () -> Math.PI*rad*rad/2.0 }

let square c x =
    let cent,len = c,x
    { Draw = fun () -> printfn "Square @(%f,%f),sz=%f" cent.x cent.y len;
      Area = fun () -> len*len }
```

В случае если нам необходимо иметь возможность изменять параметры объекта (например, двигать фигуры), мы можем сохранять внутри замыкания ref-ссылку.

Теперь мы можем определить коллекцию геометрических фигур и обращаться к ним через единый интерфейс Shape, наблюдая полиморфизм:

```
let shapes = [ circle {x=1.0; y=2.0} 10.0; square {x=10.0; y=3.0} 2.0 ]

shapes |> List.iter (fun shape -> shape.Draw())
shapes |> List.map (fun shape -> shape.Area())
```

4.3.3. Методы

F# позволяет приписывать объявляемым типам данных – записям и размеченным объединениям – методы, которые можно будет вызывать с использованием обычной точечной нотации. Вернемся к описанию типа Point – мы добавим к нему функции рисования и вычисления расстояния между точками:

```
type Point = { x : float; y : float }
   with
     member P.Draw() = printfn "Point @(%f,%f)" P.x P.y
     static member Zero = { x=0.0; y=0.0 }
     static member Distance (P1,P2) =
       let sqr x = x*x
       Math.Sqrt(sqr(P1.x-P2.x)+sqr(P1.y-P2.y))
     member P1.Distance(P2) = Point.Distance(P1,P2)
     static member (+) (P1 : Point, P2 : Point) =
       { x=P1.x+P2.x ; y = P1.y+P2.y }
     override P.ToString() = sprintf "Point @(%f,%f)" P.x P.y
   end
```

Обратите внимание на следующие особенности этого описания:

- методы определяются с помощью ключевого слова member, при этом перед именем метода указывается имя, которое будет играть роль самого объекта (this в терминологии C#) в описании метода;
- мы использовали ключевое слово override вместо member для перегрузки существующего метода. Перегрузка ToString() позволяет нам изменить вид отображения объекта;
- функцию Distance мы описали как метод класса и как статическую функцию. Использование статической функции с двумя аргументами больше соответствует духу функционального программирования, в то время как использование метода характерно для объектно-ориентированного подхода. При написании кода, используемого из других языков на платформе .NET, рекомендуется применять второй подход, в то время как в чисто функциональном коде статический каррированный метод может оказаться предпочтительнее. Обычные методы, как и статические, могут принимать аргументы в каррированной форме или в виде кортежей;
- в самом классе мы также описали характерный представитель класса Zero в виде статического поля;
- возможно перегружать операторы, описывая их как статические функции с соответствующим именем.

4.3.4. Интерфейсы

Наш пример из раздела выше, в котором мы описывали полиморфное поведение геометрических фигур, может быть также реализован с использованием понятия интерфейса. Интерфейс, по сути, представляет собой шаблон с описанием функциональности объекта, которую затем можно воплощать в конкретных объектах. В данном случае мы опишем интерфейс Shape, содержащий методы Draw и Area:

```
type Shape =
   abstract Draw : unit -> unit
   abstract Area : float
```

F# понимает, что Shape является интерфейсом, поскольку в нем определены только абстрактные методы, нет внутренних атрибутов или конструкторов. Также можно в явном виде указать, что мы хотим описать именно интерфейс, используя более подробный синтаксис:

```
type Shape = interface
   abstract Draw : unit -> unit
   abstract Area : float
end
```

Теперь мы можем описать функции-конструкторы circle и square, который будут создавать экземпляры объектов, реализующих интерфейс Shape, при этом

переопределяя методы в соответствии с требуемым поведением. Это делается при помощи конструкции, называемой **объектным выражением** (object expression):

```
let circle cent rad =
  { new Shape with
      member x.Draw() = printfn "Circle @(%f,%f), r=%f" cent.x cent.y rad
      member x.Area = Math.PI*rad*rad/2.0 }

let square cent sz =
 { new Shape with
      member x.Draw() = printfn "Square @(%f,%f),sz=%f" cent.x cent.y sz
      member x.Area = size*size }
```

Объектное выражение позволяет нам создавать конкретные экземпляры классов (и интерфейсов!), не только указывая значения конкретных полей, но и переопределяя (или доопределяя) некоторые методы. Например, с его помощью мы можем создавать легковесные «экземпляры» интерфейсов с необходимой функциональностью, например:

```
let SoryByLen (x : ResizeArray<string>) =
     x.Sort({ new IComparer<string> with
                member this.Compare(s1,s2) =
                  s1.Length.CompareTo(s2.Length) })
```

4.3.5. Создание классов с помощью делегирования

Предположим, что нам необходимо создать наборы классов наших геометрических фигур для рисования на различных поверхностях: на консоли (из звездочек), на изображении типа `Bitmap` и на экранной форме. Традиционным подходом в данном случае было бы использовать общий базовый класс для каждой геометрической фигуры с абстрактным методом `Draw` и затем породить от этого класса три класса для рисования на различных поверностях, переопределив метод `Draw`.

Однако возможен и другой подход. Мы можем абстрагировать функцию рисования в отдельный класс, или, для простоты, в одну функцию рисования точки на нашей поверхности. Далее будем передавать эту функцию в конструкторы классов наших геометрических фигур и определим их метод `Draw` соответствующим образом через переданную функцию рисования:

```
type Drawer = float*float -> unit

let circle draw cent rad =
  { new Shape with
      member x.Draw() =
        for phi in 0.0..0.1..(2.0*Math.PI) do
```

```
        draw (cent.x+rad*Math.Cos(phi),cent.y+rad*Math.Sin(phi))
    member x.Area = Math.PI*rad*rad/2.0 }
```

Тогда для создания классов, рисующих окружности на различных поверхностях, нам потребуется лишь передать им соответствующие функции рисования:

```
let ConsoleCircle cent rad =
  circle (fun (x,y) -> ... Console.Write("*") ...) cent rad

let BitmapCircle cent rad =
  circle (fun (x,y) -> ... Bitmap.Setpixel(x,y) ...) cent rad
```

Такой подход называется делегированием, поскольку мы делегируем некоторую внешнюю функциональность функциям, передаваемым как аргументы, при этом концентрируя внутри класса его базовую функциональность. При этом у нас получается разделить функции рисования в ортогональную иерархию по отношению к свойствам геометрических фигур, в то время как традиционный объектно-ориентированный подход с наследованием поощряет использование одной иерархии объектов. В целом наследование приводит к построению иерархии все более усложняющихся классов, в то время как фукнциональный подход обычно поощряет создание небольших абстракций, которые могут гибко комбинироваться между собой. В этом смысле делегирование больше соответствует духу функционального программирования и широко используется в библиотеке F#.

4.3.6. Создание иерархии классов

Принятый в других языках платформы .NET подход к созданию иерархической структуры классов также может быть реализован в F#. Реализуем иерархию геометрических объектов, при этом добавив возможности модификации координат объектов.

Определяя интерфейс `Shape`, как и ранее, базовым классом иерархии сделаем класс `Point`:

```
type Point (cx,cy) =
   let mutable x = cx
   let mutable y = cy
   new() = new Point(0.0,0.0)
   abstract MoveTo : Point -> unit
   default p.MoveTo(dest) = p.Coords <- dest.Coords
   member p.Coords
     with get() = (x,y) and set(v) = let (x1,y1) = v in x <- x1; y <- y1
   interface Shape with
     override t.Draw() = printfn "Point %A" t.Coords
     override t.Area = 0.0
   static member Zero = new Point()
```

Рассмотрим синтаксис такого описания. Во-первых, в описании класса ему передается список параметров (в данном случае cx, cy) – это признак того, что используется сокращенный синтаксис неявного конструктора классов, то есть основной конструктор класса можно описывать сразу после заголовка. Объявляемые там локальные переменные (x и y) становятся внутренними атрибутами класса. Также с помощью ключевого слова new можно объявлять дополнительные конструкторы – в нашем случае конструктор без параметров.

Методы могут описываться при помощи ключевых слов member, default, override и abstract. Member описывает метод, который не может быть перегружен в дочерних классах. Для описания метода, который допускает перегрузку, всегда необходимо использовать ключевое слово abstract с указанием типа (сигнатуры) метода. Если при этом мы хотим предоставить какую-то реализацию метода в данном классе, то есть описать то, что называется виртуальным методом в ООП, то необходимо одновременно описать эту реализацию при помощи ключевого слова default или override (они могут использоваться как синонимы) – как мы делаем с методом MoveTo.

Далее мы описываем свойство (property) Coords – для этого в явном виде указываем функции для чтения (get) и для изменения (set) свойства. Далее мы указываем, что класс должен поддерживать интерфейс Shape, и описываем методы этого интерфейса. В заключение описываем статическое поле класса Point.Zero.

Обратите внимание, что, несмотря на то что класс реализует интерфейс Shape, напрямую вызывать методы этого интерфейса нельзя. Чтобы вызвать метод Area для объекта типа Point, необходимо сначала осуществить приведение объектного типа с помощью специального оператора :> следующим образом: (p :> Shape).Area. Другой возможностью является описание функции, которая принимает на вход объект любого из типов, реализующих указанный интерфейс или наследующих от указанного типа. Например:

```
let draw (x: #Shape) = x.Draw()
```

В этом случае функции draw можно будет передавать как объекты типа Point, так и любые другие объекты, реализующие интерфейс Shape.

Теперь опишем другие классы иерархии, которые будут унаследованы от Point:

```
type Circle (cx,cy,cr) =
 class
   inherit Point(cx,cy)
   let mutable r = cr
   new () = new Circle(0.0,0.0,0.0)
   member p.Radius with get()=r and set(v)=r<-v
   interface Shape with
     override t.Draw() = printfn "Circle %A, r=%f" base.Coords r
     override t.Area = Math.PI*r*r/2.0
 end

type Square (cx,cy,sz) =
```

```
  inherit Point(cx,cy)
  let mutable size = sz
  new() = new Square(0.0,0.0,1.0)
  member p.Size with get()=size and set(v)=size<-v
  interface Shape with
    override t.Draw() = printfn "Square %A, sz=%f" base.Coords size
    override t.Area = size*size
```

Здесь при помощи ключевого слова `inherit` указываются базовый класс и вызов соответствующего конструктора. Помимо этого, мы предоставляем свои реализации для методов интерфейса `Shape` и дополнительные свойства для доступа к вновь появившимся атрибутам класса. Обратите также внимание, что для `Circle` мы явно используем ключевое слово `class` – это можно делать, если вы хотите подчернуть в явном виде, что тип является классом.

Для создания последовательности геометрических фигур нам приходится в явном виде приводить их к типу `Point`, а для вызова методов интерфейса `Shape` – к типу `Shape`:

```
let plist =
  [new Point(); new Square():>Point; new Circle(1.0,1.0,5.0):>Point]
plist |> List.iter (fun p -> (p:>Shape).Draw())
plist |> List.map (fun x -> (x.Coords,(x:>Shape).Area))
plist |> List.iter (fun p -> p.MoveTo(Point.Zero))
```

Последние два примера демонстрируют, что свойство `Coords` и метод `MoveTo` унаследованы всеми дочерними классами.

Отметим, что оператор `:>` обеспечивает приведение типа к более «абстрактному» или родительскому классу в иерархии (то есть обеспечивает upcasting). Для обратного приведения (downcasting) используется конструкция `:?>`, которая, однако, весьма опасна, поскольку проверить соответствие типов на этапе компиляции не представляется возможным, и исключение генерируется уже на этапе выполнения. Поэтому вместо нее рекомендуется использовать сопоставление с образцом, как в следующем примере:

```
let area (p:Object) =
   match p with
    | :? Shape as s -> s.Area
    | _ -> failwith "Not a shape"
```

Заметим, что оператор `:?` также позволяет проверять соответствие типов, возвращая результат типа `bool`, поэтому этот пример можно записать следующим образом:

```
let area (p:Object) =
  if (p :? Shape) then (p:?>Shape).Area
  else failwith "Not a shape"
```

4.3.7. Расширение функциональности имеющихся классов

F# также позволяет нам доопределять или переопределять методы существующих классов, создавая то, что в терминологии C# называется *extension methods*. Например, мы можем добавить методы для проверки четности и нечетности целых чисел:

```
type System.Int32 with
   member x.isOdd = x%2=1
   member x.isEven = x%2=0

(12).isEven
```

Если поместить соответствующие описания в модуль (см. следующий раздел), то открытие такого модуля делает доступными описанные расширения.

4.3.8. Модули

Функциональное программирование, помимо объектно-ориентированной декомпозиции предметной области, часто использует функциональную декомпозицию, благодаря чему объектно-ориентированный подход к разбиению задачи на подзадачи может оказать не слишком хорошим. Однако при функциональной декомпозиции необходимо иметь определенный способ разделения программного кода на независимые части. Традиционно в таком случае удобно использовать модульный подход, когда близкие по смыслу функции группируются в модули. F# предоставляет для этого соответствующие языковые средства.

С точки зрения объектной модели .NET, модуль представляет собой класс с набором типов, объектов и статических методов. Например, в рассмотренном в главе 2 примере реализации очереди мы могли бы оформить все соответствующие процедуры в отдельный модуль следующим образом:

```
module Queue =
    type 'a queue = 'a list * 'a list
    let empty = [],[]   // пустая очередь
    let tail (L,R) =    // удалить элемент из очереди
      match L with
       [x] -> (List.rev R, [])
      | h::t -> (t,R)
    let head (h::_,_) = h   // взять голову очереди
    let put x (L,R) =   // добавить элемент в очередь
      match L with
       [] -> ([x],R)
      | _ -> (L,x::R)
```

После такого описания в программе достаточно открыть соответствующий модуль, и можно пользоваться описанными в нем типами и процедурами:

```
open Queue

let q = Queue.empty
let q1 = Queue.put 5 (Queue.put 10 q)
Queue.head q1
```

5. Метапрограммирование

Благодаря наличию гибкой системы встроенных в язык типов, в том числе вариантного типа, F# становится удобной платформой для определения более специализированных языков высокого уровня, или так называемого *domain specific languages*, DSL. Дополнительные средства языка типа квотирования (*quotations*) дают доступ к исходному дереву функциональной программы, позволяя манипулировать исходными представлениями. В этом разделе мы рассмотрим такие средства и примеры использования языка, которые можно объединить под общим названием *метапрограммирования*.

Под метапрограммированием обычно понимают создание программ, которые, в свою очередь, манипулируют программами как данными. К метапрограммированию можно отнести, например, преобразование функциональных выражений или их трансляцию в другой язык, или же расширения языка для создания более специализированного языка высокого уровня.

Современные языки, как правило, содержат различные средства метапрограммирования. В C# к таковым можно отнести LINQ-синтаксис и деревья выражений (*expression trees*). Как мы увидим в следующих разделах, F# также не станет исключением.

5.1. Языково-ориентированное программирование

Вначале мы рассмотрим приемы использования F# для так называемого *языково-ориентированного программирования* (language-oriented programming), которые не требуют специальных, не рассмотренных ранее возможностей языка. Обычно, когда говорят про языково-ориентированное программирование, речь идет о создании *доменно-ориентированных языков* (DSL, Domain Specific Languages). DSL – это язык еще более высокого уровня, нежели язык общего назначения типа F# или C#, который содержит в себе специфические конструкции некоторой (достаточно узкой) предметной области и предназначен для решения соответствующих специализированных задач. Такие языки могут быть как графическими, так и текстовыми.

Рассмотрим простой пример реализации текстового DSL на F#. Предположим, нам необходимо описывать родословное дерево: набор людей с указанием некоторого набора данных плюс информацию о семьях. Пример описания родословного дерева мог бы выглядеть следующим образом:

```
person "Aaron" (born "21.03.1974")
person "Mary" unknown_birth
person "John" (born "30.12.1940")
person "Vickie" (born "14.05.2004")
person "Julia" unknown_birth
person "Justin" unknown_birth

family
  (father "Aaron")
  (mother "Julia")
  [child "Vickie"]

family
  (father "John")
  (mother "Mary")
  [child "Aaron";child "Justin"]
```

Может показаться странным, но это – текст на F#. Посмотрим, как можно добиться того, чтобы компилятор воспринимал такой текст и строил по нему модель родословного дерева.

Для начала опишем объект для хранения данных о человеке:

```
type Person (n : string) =
        let mutable name = n
        let mutable father : Person option = None
        let mutable mother : Person option = None
        let mutable birthdate = DateTime.MinValue
        member x.Name with get()=name and set(v) = name<-v
        member x.Father with get()=father and set(v) = father<-v
        member x.Mother with get()=mother and set(v) = mother<-v
        member x.Birthdate with get()=birthdate and set(v) = birthdate<-v
```

Будем хранить всех людей в глобальном словаре:

```
let People = new Dictionary<string,Person>()
```

Тогда ключевое слово нашего DSL для описания человека может выглядеть так:

```
let person name bdate =
    let P = new Person(name)
    P.Birthdate <- bdate
    People.Add(name,P)
    P
```

Здесь `bdate` – это дата рождения, для удобного описания которой мы вводим конструкции разбора даты:

```
let born str_date =
     DateTime.Parse(str_date)
let unknown_birth=DateTime.MinValue
```

Конструкция family для описания семьи воспринимает на вход двух родителей и список детей, и смысл этой конструкции – пройтись по списку детей и установить в соответствующих объектах правильные ссылки на родителей:

```
let rec family F M = function
    [] -> ()
  | (h:Person)::t ->
      h.Father <- Some(F)
      h.Mother <- Some(M)
      family F M t
```

Для того чтобы обеспечить красивый синтаксис DSL, введем также слова father, mother и child как синонимы для поиска ссылки на человека в словаре:

```
let father s = People.[s]
let mother s = People.[s]
let child s = People.[s]
```

Мы получили возможность использования простейшего DSL для описания семейных отношений. Правда, такой подход имеет множество недостатков – например, ключевые слова father и mother не несут соответствующей семантики, а тот факт, является ли кто-то отцом или матерью, определяется порядком следования выражений в конструкции family. Подобного недостатка можно избежать введением дополнительной «типизации» на уровне объектов предметной области:

```
type tperson = Father of Person | Mother of Person | Child of Person

let father s = Father(People.[s])
let mother s = Mother(People.[s])
let child s = Child(People.[s])

let family P1 P2 L =
    let rec rfamily F M = function
            [] -> ()
          | Child(h)::t ->
              h.Father <- Some(F)
              h.Mother <- Some(M)
              rfamily F M t
    match P1,P2 with
       Father(F),Mother(M) -> rfamily F M L
     | Mother(M),Father(F) -> rfamily F M L
     | _ -> failwith "Wrong # of parents"
```

Пример другого интересного DSL для описания конфигурации солнечной системы с целью последующей визуализации содержится в [4]. Само описание выглядит следующим образом:

```
let solarSystem =
   sun
   -- (rotate 80.00f 4.1f mercury)
   -- (rotate 150.0f 1.6f venus)
   -- (rotate 215.0f 1.0f
        (earth -- (rotate 20.0f 12.0f moon)))
```

В этом языке `rotate` является функцией, применяемой к объекту, описывающему планету, а оператор `--` выполняет роль конструктора, аналогичного конструктору списков `::`.

5.2. Активные шаблоны

Еще одним часто используемым средством для реализации DSL на F# являются так называемые **активные шаблоны** (*active patterns*). Обычно шаблоны могут использоваться в операторе сопоставления с образцом `match` и позволяют сопоставлять один объект сложной структуры с другим, попутно производя необходимые сопоставления имен. Активные шаблоны дают возможность программисту определять свои шаблоны, для проверки которых будет вызываться определенная функция.

Начнем с простого примера: пусть нам надо описать функцию для определения того, является ли число четным или нет:

```
let test x = if x%2=0 then printfn "%d is even" x
                      else printfn "%d is odd" x
```

То же самое можно сделать с помощью операции сопоставления с образцом так:

```
let test x =
    match x with
      | x when x%2=0 -> printfn "%d is even" x
      | _ -> printfn "%d is odd" x
```

А теперь представьте себе, как было бы удобно вместо этого писать более понятную конструкцию:

```
let test x =
    match x with
      | Even -> printfn "%d is even" x
      | Odd -> printfn "%d is odd" x
```

Такая конструкция и является активным шаблоном! Для описания подобного активного шаблона используется следующий синтаксис:

```
let (|Even|Odd|) x = if x%2=0 then Even else Odd
```

Видя, что в match используется активный шаблон, компилятор вызывает соответствующую функцию и затем по результатам ее работы производит сопоставление.

В нашем случае шаблон получился не слишком «активным», в том смысле что используемый код был очень простым. Это не всегда так. Например, через активный шаблон можно реализовать сопоставление строки с регулярным выражением следующим образом:

```
let (|Match|_|) (pat : string) (inp : string) =
    let m = Regex.Match(inp, pat)
    if m.Success then Some (List.tail [ for g in m.Groups -> g.Value ])
                 else None
```

Данный шаблон является неполным, то есть если сопоставление при работе шаблона не произойдет, будет продолжено сопоставление с другого доступного активного или обычного шаблона. На это указывают использование _ в числе вариантов шаблона и тот факт, что шаблон возвращает опциональный тип. Шаблону передаются регулярное выражение pat и входной аргумент (то есть то выражение, которое необходимо сопоставлять) inp, а на выходе получается список найденных совпадений, который сопоставляется со вторым аргументом активного шаблона. Использовать активный шаблон можно следующим образом:

```
match "My daughter is 16 years old" with
  | Match "(\d+)" x -> printfn "found %A" x
  | _ -> printfn "Not found"
```

Применение активных шаблонов может сильно повысить читаемость кода и выразительность языка. Именно поэтому создатели F# считают активные шаблоны одним из важнейших доступных в языке средств метапрограммирования наряду с рассмотренными ниже монадическими выражениями.

5.3. Квотирование

Важным средством метапрограммирования в F# является *квотирование* (quotations, дословно – цитирование). Средства квотирования позволяют окружить фрагмент программы на F# специальными кавычками <@ ... @> или <@@ ... @@>, при этом соответствующий фрагмент не компилируется, а остается в форме дерева функциональной программы.

Скобки <@@ ... @@> называются *нетипизированным квотированием* – несмотря на то что внутри производится проверка типов в соответствии с синтаксисом

F#, в результате возвращается значение типа Expr. Кавычки <@ ... @> соответственно обозначают *типизированное квотирование*, и если выражение под кавычками имеет тип T, то возвращается Expr<T>.

Чтобы понять, что же представляет собой дерево выражений, посмотрим на результат <@ 1+2*3 @>:

```
printf "%A" (<@ (1+2)*3 @>)

Call (None, Int32 op_Multiply[Int32,Int32,Int32](Int32, Int32),
      [Call (None, Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
             [Value (1), Value (2)]), Value (3)])val it : unit = ()
```

Мы видим, что в данном случае вызов функции обозначается функтором Call, аргументами которого являются имя функции и список аргументов. Константы обозначаются функтором Value. Чуть более сложным образом представляется каррированная функция, например:

```
printf "%A" (<@ (+)1 @>)

Let (x, Value (1),
  Lambda (y,
    Call(None,Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
      [x, y])))
```

Использование квотирования позволяет программисту самому реализовать обработчик фрагментов функциональной программы, представленной в виде дерева. Например, мы можем преобразовывать такую программу для выполнения на другом процессоре (например, на графическом процессоре GPU) или же в другой язык (система запросов к реляционным данным LINQ в F# реализуется именно через квотирование).

Мы рассмотрим сравнительно простой пример преобразования арифметических выражений в язык стекового калькулятора. Известно, что любое арифметическое выражение можно преобразовать в постфиксную форму, которая затем вычисляется при помощи операций на стеке. Если рассматривать арифметические выражения с основными операциями, то для вычисления можно использовать абстрактную стековую машину со следующими командами:

- Push(c) – поместить значение константы c на верхушку стека;
- Add, Sub, Mul, Div – проделать соответственно операцию сложения, вычитания, умножения или деления с двумя верхними элементами стека, поместив результат на стек. Исходные значения при этом удаляются из стека;
- Print – напечатать верхушку стека.

Такую абстрактную машину на F# можно описать следующим образом:

```
type Command = Push of int
             | Add
```

```
            | Sub
            | Mult
            | Div
            | Print

type Prog = Command list
```

Программа для такой стековой машины будет представляться списком, который для удобства будет записываться в обратном порядке. Например, для вычисления выражения 1+2*3 необходима будет следующая последовательность команд: `[Add; Mult; Push 3; Push 2; Push 1]`.

Для реализации трансляции дерева квотирования в последовательность команд реализуем предикат `comp: Expr->Prog`. Он будет сводиться к предикату `compile: Prog -> Expr -> Prog`, который будет принимать на вход некоторый список команд и дополнять его командами вычисления заданного выражения. Тогда comp можно будет определить так[1]:

```
let comp : (Expr->Command list) = compile []
```

Для реализации `compile` нам потребуются вспомогательные взаимно рекурсивные функции: `compileop` будет рассматривать все возможные арифметические операции и добавлять в список соответствующую команду (`Add`, `Mult` и т. д.), а `compilel` будет рекурсивно обрабатывать список выражений (вызывая `compile` для каждого из подвыражений), добавляя в список команды для их вычисления. Для описания взаимно рекурсивных функций используется один let-блок, в котором каждое описание разделяется ключевым словом `and`:

```
let rec compile l E =
 match E with
   Call(_,Op,Args) -> compileop (compilel l Args) Op
 | Value(x,_) -> Push(int(x.ToString()))::l
and compileop l Op =
  if Op.Name="op_Addition" then Add::l
  else if Op.Name="op_Multiply" then Mult::l
  else if Op.Name="op_Subtraction" then Sub::l
  else if Op.Name="op_Division" then Div::l
  else l
and compilel l = function
    [] -> l
  | h::t -> compilel (compile l h) t
```

Например, при компиляции выражения 1+2 дерево квотирования содержит операцию `Call(Null,op_Addition, [Value(1),Value(2)])`, и для получения резуль-

[1] Здесь нам потребовалось явно задать тип функции comp, поскольку без этого механизм вывода типов F# выводил слишком общий тип. Избежать необходимости задания типа можно было, введя дополнительную переменную-аргумент следующим образом: `let comp x = compile [] x`. Однако мы посчитали такое определение менее «функциональным».

тата сначала вызывается `complilel [Value(1), Value(2)]`, выдающее в результате применения compile список команд `[Push(2),Push(1)]`, а потом `compileop`, добавляющий операцию сложения `Add`. В результате генерируется искомый набор команд `[Add,Push(2),Push(1)]`.

5.4. Конструирование выражений, частичное применение функции и суперкомпиляция

Рассмотрим еще одно интересное применение квотирования для реализации эффективного механизма частичного применения функции. Предположим, у нас определена функция возведения в степень следующим образом:

```
let rec power n x =
  if n=0 then 1
  else x*power (n-1) x
```

и нам необходимо определить на основе этой функции другую функцию возведения в конкретную степень 5:

```
let pow5 = power 5
```

Такое определение будет неплохо работать – однако заметим, что определение функции в виде явного произведения было бы более эффективным:

```
let pow5 x = x*x*x*x*x
```

С помощью квотирования мы можем определить функцию `metapower`, которая для заданного `n` будет порождать выражения n-кратного умножения переменной x саму на себя, которые затем могут быть JIT-скомпилированы во время выполнения программы и эффективно выполнены.

Для этого мы воспользуемся возможностью включения переменных в квотированные выражения. Например,

```
let test x = <@ %x+1 @>
```

описывает функцию из `Expr<int>` в `Expr<int>`, выполняя, по сути дела, подстановку одного выражения в другое. С учетом этого опишем функцию `metapower : int -> Expr<int> -> Expr<int>`, которая по заданному натуральному числу n будет формировать выражение для возведения произвольного выражения в указанную степень:

```
let rec metapower n x =
  if n=0 then <@ 1 @>
  else (fun z -> <@ %x * %z @>) (metapower(n-1) x)
```

Если мы с учетом такого описания определим функцию возведения в какую-то конкретную степень, например

```
let pow5 x = metapower 5 <@ x @>
```

то вычисление степени будет производиться путем подстановки аргумента `x` в заранее сформированное выражение `x*x*x*x*x`, которое затем может быть эффективно вычислено.

Подобное частичное применение функции, при котором создается специализированная программа, которая может выполняться более эффективно за счет учета особенностей выполнения на конкретных данных, лежит в основе идеи суперкомпиляции. Любая программа может быть описана как функция из входных данных в выходные: P : I → O. Если при этом некоторая часть входных данных является статической, то мы можем описать программу как P: $I_{st} \to I_{dyn} \to O$. В этом случае частичное применение (P I_{st}) будет представлять собой функцию $I_{dyn} \to O$, и задача суперкомпилятора – построить оптимизированную программу для (P I_{st}).

Интересным частным случаем является так называемая проекция Футамуры. В ней в качестве программы рассматривается интерпретатор некоторого языка программирования Intr, который принимает на вход программу Source и входные данные I и выдает результат O: Intr: Source → I → O. В этом случае, применяя суперкомпиляцию, мы можем получить из интерпретатора компилятор, поскольку оптимизированное частичное применение (Intr Source) и будет представлять собой результат компиляции программы.

5.5. Монады

Помимо квотирования, в F# существует другой важный механизм метапрограммирования – это *монадические выражения*[1] (*computational expressions*). На базе этого механизма в F# реализуются параллельные и асинхронные вычисления, конструкция `seq {...}` и другие элементы. Прежде чем перейти к описанию синтаксиса монадических выражений в F#, поговорим про *монады* – важный элемент функционального подхода к программированию.

В чистом функциональном подходе возникают проблемы с реализацией алгоритмов, требующих явной последовательности вычислений. Для некоторых задач, например при последовательном выводе на экран, нам важно, чтобы операции выполнялись в заданном порядке. В то же время для ленивых языков порядок вычисления выражений зависит от внутреннего устройства функций, а не от порядка их записи в выражении, что может приводить к неожиданным эффектам. Кроме того, в чистом функциональном подходе отсутствуют побочные эффекты, и ввод-вывод в привычном виде (например, в виде операции `WriteLine`, выводящей что-то

[1] Дословно computational expression переводится как «вычислительное выражение». Однако, по мнению автора, на русском языке эта фраза звучит весьма тавтологично, термин «монадическое выражение» намного лучше передает суть этой конструкции.

на экран) вообще невозможен. Для реализации «императивного» подхода внутри чисто функционального языка могут использоваться монады.

5.5.1. Монада ввода-вывода

Рассмотрим для начала реализацию ввода-вывода в чисто функциональном языке. Поскольку ввод или вывод приводят к модификации некоторых объектов «вне» программы (то есть консоли), для явного задания такой модификации вводят понятие *состояния*. При этом все операции ввода-вывода должны в явном виде принимать на вход состояние до операции ввода-вывода, а возвращать модифицированное состояние. Например, операция печати строки `print_str` будет иметь тип `string -> State -> State`.

Рассмотрим для начала пример организации вывода. В этом случае состояние будет моделироваться одним списком, представляющим собой последовательность выводимых на экран сообщений (записанную в обратном порядке, чтобы облегчить вывод следующего сообщения путем присоединения к началу списка):

```
type State = string list
```

Для печати значений различных типов определим операции `print`:

```
let print_int (x:int) = fun l -> x.ToString() :: l
let print_str (x:string) = fun l -> x::l
```

Обратите внимание, что функция print является преобразователем состояния – по данному значению строки она преобразует исходное состояние в состояние после печати.

Если мы хотим напечатать последовательно несколько строк, то нам придется использовать весьма неудобную конструкцию, например:

```
print_str "!" (print_int (3*5) (print_str "The result is" []))
```

Мы можем весьма упростить эту запись, если введем специальную операцию >>= следующим образом:

```
let (>>=) A B = B A
```

Тогда написанный выше код можно будет записать в более «последовательном» виде:

```
[] >>= print_str "The result is" >>= print_int 15 >>= print_str "!"
```

Операция >>= представляет собой операцию комбинирования преобразователей состояния, которая позволяет объединить преобразователи в цепочку, применяя их во вполне определенной последовательности. В некотором смысле такое объединение напоминает рассмотренное выше использование продолжений.

Теперь перейдем к более сложному примеру ввода-вывода. В этом случае состояние будем моделировать при помощи двух списков: списка ввода (содержащего все вводимые программой строки) и списка вывода:

```
type State = string list * string list
```

Операция ввода в этом случае должна будет инкапсулировать в себе введенное значение, поэтому, помимо преобразования состояния, мы должны уметь «протаскивать» это значение через цепочку вычислений. Введем для этого специальный тип IO<'t>, который представляет собой совокупность из преобразователя состояния и значения типа 't (такой тип называется *монадическим типом*):

```
type IO <'t> = 't*State
```

Для описания «чистого» преобразователя состояния мы будем использовать IO<unit>. Например, операция печати print будет возвращать такой тип:

```
let print (t:string) = fun (I,0) -> ((),(I,t::0))
```

Можно представлять себе функцию print следующим образом: она берет строку и возвращает «программу» (называемую *монадой*), применяя которую к состоянию, мы получаем результирующее состояние с напечатанной строкой.

Аналогично определяется тип чтения строки, который имеет тип IO<string>:

```
let read = fun (I,0) -> (List.head I,(List.tail I,0))
```

Операция комбинирования >>= в этом случае применяется к монаде a типа IO<'a> и функции, которая по значению типа 'a возвращает новую монаду типа IO<'b>:

```
let (>>=) a b =
  fun s0 ->
    let (ra,s1) = a s0 in
b ra s1
```

Результатом композиции a>>=b является преобразователь состояния, который начиная с состояния s0 сначала применяет a, а затем применяет функцию b к результату и результирующему состоянию s1 – получая на выходе новую монаду-преобразователь состояния.

Рассмотрим простейший пример, когда мы хотим напечатать на экране введенную строку. При этом композиция read >>= print дает нам искомую программу, применив которую к начальному состоянию, мы получим конечное состояние, в котором строка будет напечатана на экране (то есть перемещена из списка ввода в список вывода):

```
(read >>= print) (["hello"],[])
```

В более сложном примере мы хотим ввести имя пользователя и напечатать «Привет,» и введенное имя. Для этого мы должны в качестве второй «программы» передать в операцию композиции функцию, которая конкатенирует строки и печатает результат:

```
(read >>= fun t -> print ("I like "+t)) (["Vasya"],[])
```

Того же эффекта можно достичь и по-другому: сначала применив программу read, затем – программу, модифицирующую нужное нам значение, и потом программу печати результата print:

```
(read >>= fun t -> ret ("I like "+t) >>= print) (["Vasya"],[])
```

Здесь операция ret (return) – это функция, которая по значению типа string возвращает IO<string> – программу, подменяющую передаваемое в цепочке вычислений значение на заданное, не меняя состояния:

```
let ret (x:string) = fun S -> (x,S)
```

Если у нас возникнет потребность напечатать подряд несколько строк, то мы не сможем использовать конструкцию print «...» >>= print «...», поскольку операция >>= вторым аргументом ожидает функцию t->IO<'t>. Для комбинирования двух монадических операций используется конструкция >> – в этом случае значение, возвращенное первой операцией, отбрасывается:

```
let (>>) f g =
  fun s0 ->
    let (_,s1) = f s0 in
    g s1
```

С использованием этой операции можно записать более содержательный пример ввода-вывода:

```
(print "Your name:" >>
 read >>=
 (fun t -> print ("I like "+t)) >>
 print "Goodbye")
   (["Mike"],[])
```

5.5.2. Монадические свойства

Чтобы более строго определить понятие монады, посмотрим, какие основные конструкции нам пришлось определить. Для типа 't определялся некоторый монадический тип M<'t>, для которого были определены следующие операции:

- return : 't → M<'t> (который мы называли ret, поскольку слово return является ключевым в F#);

- >>= : M<'a> → ('a→M<'b>) → M<'b> (эта операция обычно называется *связыванием* и читается *bind*).

Также мы определяли операцию композиции >> : M<'a>→M<'b>→M<'b>, которая, однако, является производной от связывания и может быть определена как

```
let (>>) f g = f >>= (fun _ -> g)
```

Для того чтобы объект мог считаться монадой, необходимо также выполнение трех так называемых монадических свойств:

- return x >>= f должно быть эквивалентно f x (если мы оборачиваем в монадический тип значение x и подаем это на вход f, то это эквивалентно тому, что мы сразу применим функцию f к x);
- m >>= return должно быть эквивалентно m (если мы берем некоторую операцию, возвращающую значение x, и потом в явном виде выполняем return x – то это эквивалентно исходной операции);
- (m>>=f)>>=g должно быть эквивалентно m>>=(fun x -> f x >>= g) (по сути дела, это свойство ассоциативности, то есть если мы связываем действия m и f и потом добавляем действие g, то можно сначала сгруппировать f и g и потом связать это с m.

Объекты, удовлетворяющие перечисленным свойствам, и называются монадами.

5.5.3. Монада недетерминированных вычислений

Монады могут использоваться не только для ввода-вывода. В качестве другого примера рассмотрим *монаду недетерминированных вычислений*. Речь идет о вычислениях, которые могут возващать не одно, а несколько возможных значений. Например, если у нас есть два значения, одно из которых равно 2, а другое – 1 или 2, то их сумма, в свою очередь, будет равна 3 или 4.

Монаду будем представлять в виде списка возможных значений, то есть

```
type Nondet<'a> = 'a list
```

Помимо стандартной операции `ret`, завертывающей значение типа `t` в монадический тип, определим также операцию `ret'`, которая будет возвращать монаду – список возможных значений. Также определим операцию `fail`, которая будет соответствовать отсутствию результатов вычислений (неудаче):

```
let ret x = [x]
let ret' x = x
let fail = []
```

Операция связывания в этом случае определяется просто:

```
let (>>=) mA (b:'b->Nondet<'b>) = List.collect b mA
```

Мы берем на вход исходный список вариантов mA и применяем к каждому элементу функцию b, которая возвращает список возможных вариантов. Далее все эти списки нужно объединить вместе – это как раз эквивалентно действию встроенной функции collect.

Вот пример использования недетерминированных вычислений: мы берем число, которое равно либо 1, либо 2, либо 3, и умножаем его либо на 4, либо на 5 – в результате получаем список возможных вариантов:

```
ret' [1;2;3] >>=
  fun x -> [x*4;x*5]
```

5.6. Монадические выражения

Приведенная выше запись монад хотя и относительно наглядна, но требует большой внимательности и отслеживания, как передаются аргументы от одной монады к другой при их связывании. Для упрощения записи F# позволяет определять так называемые *монадические выражения* (*computational expressions*). С их помощью, например, приведенный выше пример недетерминированных вычислений может быть записан так:

```
nondet { let! x = [1;2;3] in return! [x*2;x*3] }
```

nondet в данном случае называется конструктором монадического выражения, может быть определен произвольным образом с помощью задания базовых операций >>=, return и, возможно, некоторых других, тем самым расширяя язык новыми конструкциями с задаваемой разработчиком семантикой. В нашем случае для создания конструктора nondet необходимо написать:

```
type NondetBuilder() =
  member m.Return(x) = ret x
  member m.Bind(mA,b) = mA >>= b
  member m.Zero() = fail
  member m.Combine(a,b) = a@b
  member m.Delay(x) = x()
  member m.ReturnFrom(x) = x

let nondet = new NondetBuilder()
```

Здесь мы описываем некоторый класс NondetBuilder, в котором определяем функции связывания bind, функцию оборачивания значения в монадический тип return, нулевой элемент zero, функцию комбинирования монад >> и некоторые

другие. После этого, создается экземпляр этого класса `nondet`, имя которого и используется при создании монадических выражений.

Монада недетерминированных вычислений, по сути дела, расширяет F# конструкцией, позволяющей рассматривать несколько возможных вариантов решения задачи, наподобие того, как это делается в языке Пролог[1]. В качестве примера рассмотрим решение логической задачки про Алису, Льва и Единорога с помощью перебора:

Однажды Алиса повстречала Льва и Единорога, отдыхавших под деревом. Странные это были существа. Лев лгал по понедельникам, вторникам и средам и говорил правду во все остальные дни недели. Единорог же вел себя иначе: он лгал по четвергам, пятницам и субботам и говорил правду во все остальные дни недели. Они высказали следующие утверждения:

Лев: Вчера был один из дней, когда я лгу.

Единорог: Вчера был один из дней, когда я тоже лгу.

Из этих двух высказываний Алиса сумела вывести, какой день недели был вчера. Что это был за день?

Для решения этой задачки заведем три списка – список дней недели, а также списки, показывающие, в какие дни лев и единорог говорят правду либо лгут:

```
let lev = [false;false;false;true;true;true;true]
let edi = [true;true;true;false;false;false;true]
let days = ["mon";"tue";"wed";"thu";"fri";"sat";"sun"]
let data = List.zip3 lev edi days
```

Здесь список data представляет собой список троек из дней недели и значений лживости или правдивости льва и единорога. Дальше определим функцию prev, которая позволит определять предыдущий день недели по отношению к тому, который определяется функцией `hit`:

```
let rec prev last hit l =
  match l with
    [] -> last
  | h::t -> if hit h then last else prev h hit t
```

Этой функции в качестве первого аргумента надо передавать «предыдущий» день недели, в качестве которого можно использовать `(true,true,»sun»)`.

Другая вспомогательная функция `realday` позволит понять, что на самом деле говорит животное: ей передаются состояние (лжет или говорит правду) и само высказывание (говорил правду или ложь):

[1] Следует, однако, четко понимать, что варианты в Прологе рассматриваются в процессе возврата, в то время как монада недетерминированных вычислений оперирует списками всех возможных решений, моделируя обход дерева решений в ширину, что с практической точки зрения не всегда оптимально.

```
let realday state said =
  if state then said else not(said)
```

После этого само решение задачи получается с помощью следующего недетерминированного перебора:

```
let res =
 nondet {
   let! (l,e,d) = data in
    let (l1,e1,d1) = prev (true,true,"sun") (fun (_,_,x) -> x=d) data in
    if (realday l false) = l1 && (realday e false) = e1
  then return (l1,e1,d1)
  }
```

Мы рассматриваем «по очереди»[1] все дни недели из списка `data` и для каждого из них находим предыдущий день с помощью функции `prev`. Далее мы сравниваем высказывания животных с учетом функции `realday`, и если смысл совпал – возвращаем требуемый результат.

Таким образом, мы видим, что с помощью монадических выражений можно достаточно сильно изменить семантику интерпретации выражений, при этом сохраняя синтаксис F#. В следующей главе мы увидим, как монадические выражения используются для построения асинхронных и параллельных вычислений.

[1] На самом деле все рассмотрения происходят одновременно и сводятся к операциям со списками всех возможных решений. Однако для понимания решения задачи проще представлять себе перебор как последовательный процесс.

6. Параллельное и асинхронное программирование

Проблема написания программного кода, который может распараллеливаться на несколько ядер процессора или на несколько компьютеров, сейчас стоит как нельзя более остро. В то время как многоядерные процессоры становятся общедоступными, программирование приложений, способных эффективно использовать несколько потоков вычислений, остается чрезвычайно трудоемкой задачей.

Функциональное программирование позволяет существенно упростить параллельное программирование, поскольку в функциональной программе нет совместно используемых несколькими потоками областей памяти, а каждая функция оперирует только полученными на вход данными. Однако задача эффективного разбиения процесса вычислений на параллельные потоки все равно остается.

Вторая проблема, которая возникает перед разработчиками, – это асинхронные вычисления и асинхронный ввод-вывод. Программирование в асинхронном режиме ведет к такк называемой инверсии управления, порождая малочитаемый и сложный в отладке код. Мы увидим, как использование монадических выражений позволяет сильно упростить написание асинхронного кода.

6.1. Асинхронные выражения и параллельное программирование

Простейшей конструкцией для распараллеливания кода является монадическое выражение `async {…}`. Оно сопоставляет выражению типа `t` асинхронное вычисление `Async<t>`, которое затем может объединяться в более сложные конструкции и вычисляться параллельно и независимо.

Например, если мы хотим вычислить параллельно два арифметических выражения, это можно сделать следующим образом:

```
let t1 = async { return 1+2 }
let t2 = async { return 3+4 }
Async.RunSynchronously(Async.Parallel [t1;t2])
```

`Async.Parallel` позволяет объединить список из async-выражений в одно параллельное вычисление, а `Async.RunSynchronously` запускает вычисления параллельно и завершается только после завершения каждого из вычислений, возвращая список полученных значений.

В качестве более содержательного примера рассмотрим реализацию параллельной функции map, которая применяет указанную функцию ко всем элементам списка параллельно:

```
let map' func items =
     let tasks =
         seq {
             for i in items -> async {
                 return (func i)
             }
          }
     Async.RunSynchronously (Async.Parallel tasks)
```

Здесь для каждого из элементов входного списка формируется async-выражение, применяющее к элементу заданную функцию, и затем к списку таких async-выражений применяется операция синхронного вычисления.

Внимательный читатель может засомневаться в эффективности такого подхода, поскольку для каждого из элементов списка как бы запускается свое параллельное вычисление. Однако на практике вычисление async-выражения является весьма легковесной операцией и не ведет к созданию отдельного потока, а вычисляется в рамках некоторого заданного пула потоков, поэтому такая реализация является приемлемой. Конечно, распараллеливание приводит к появлению дополнительных вычислительных расходов, но эти расходы несопоставимо малы по сравнению с созданием нитей исполнения (threads) .NET. Приведенный ниже пример использования параллельной map позволяет на двухядерной системе получить выигрыш в скорости более чем в полтора раза[1]:

```
time (fun () -> List.map (fun x -> fib(x)) [30..35])
time (fun () -> map' (fun x -> fib(x)) [30..35])
```

6.2. Асинхронное программирование

Другой важной проблемой, которая очень эффективно решается в функциональной парадигме, является *асихронное программирование*. В современных компьютерах при работе с файлами на диске используется прямой доступ в память, поэтому при операциях чтения или записи центральный процессор простаивает, когда мог бы использоваться для решения других задач. При построении распределенных систем такая же ситуация имеет место при удаленном вызове веб-сервисов: ответ от удаленного сервиса может поступить не сразу, и во время

[1] Здесь функция fib определена обычным образом как дважды рекурсивная функция, поэтому для вычисления fib для больших значений аргумента требуется значительное время – время вычислений в этом примере существенно больше, чем накладные расходы на создание потоков вычислений.

ожидания наша программная система могла бы выполнять другие полезные вычисления.

Асинхронное программирование обычно реализуется с помощью продолжений или callback-функций. Вызывается операция асинхронного чтения, которой, в свою очередь, передается функция, которую следует вызвать по окончании процесса чтения и передать в качестве аргумента считанный файл. При этом сама функция асинхронного чтения быстро завершается, и после ее завершения можно выполнять другую полезную работу. Такое раздвоение потока управления вместе с необходимостью разносить процесс обработки по нескольким несвязанным функциям носит название *инверсии управления*, поскольку превращает исходно линейный алгоритм в множество на первый взгляд не связанных функций.

Функциональный подход обеспечивает сравнительно простое использование продолжений. Рассмотрим пример обработки изображения, который в синхронном виде записывается простой последовательностью шагов:

```
let image = Read "source.jpg"
let result = f image
Write "destination.jpg" image
printf "Done"
```

Асинхронный вариант этого процесса с использованием продолжений мог бы записываться так:

```
ReadAsync("source.jpg",fun image ->
  let result = f image
  WriteAsync("destination.jpg",result, fun () ->
    printf "Done"))
```

Надо понимать, что такая функция достаточно быстро завершается, и необходимо загрузить программу работой, которая будет происходить параллельно с процессом чтения и записи.

Здесь приходят на помощь асинхронные выражения. Оказывается, приведенный выше алгоритм может быть записан в виде async-выражения следующим образом:

```
async { let! image = ReadAsync "source.jpg"
        let result = f image
        do! WriteAsync image2 "destination.jpg"
        do printfn "done!"
      }
```

Как можем видеть, в такой записи алгоритм выглядит как обычная последовательная программа – лишь после асинхронных операций используется специальный синтаксис `let!` и `do!`. На самом деле приведенная выше запись примерно эквивалентна следующему:

```
async.Delay(fun () ->
    async.Bind(ReadAsync "source.jpg", (fun image ->
        let image2 = f image
        async.Bind(writeAsync "destination.jpg",(fun () ->
            printfn "done!"
            async.Return(image2))))))
```

Далее такие асинхронные операции могут объединяться вместе в параллельно выполняемые блоки, и в момент асинхронного выполнения ввода-вывода в одном потоке будут выполняться вычислительные операции в другом. Например, для асинхронно-параллельной обработки 100 изображений можно использовать следующий код:

```
let ProcessImageAsync () =
    async { let  inStream  = File.OpenRead(sprintf "Image%d.jpg" i)
            let! pixels    = inStream.ReadAsync(numPixels)
            let  pixels'   = TransformImage(pixels,i)
            let  outStream = File.OpenWrite(sprintf "Image%d_tn.jpg" i)
            do!  outStream.WriteAsync(pixels')
            do   Console.WriteLine "done!"  }

let ProcessImagesAsyncWorkflow() =
    Async.RunSynchronously (Async.Parallel
                [ for i in 1 .. 100 -> ProcessImageAsync i ])
```

Асинхронная обработка также эффективна для интернет-запросов. Примеры такого использования асинхронных выражений мы увидим в дальнейших разделах.

6.3. Асинхронно-параллельная обработка файлов

Рассмотрим более содержательный пример. Предположим, у нас есть директория с большим количеством текстовых файлов, и нам нужно для каждого файла построить и записать в результирующий файл частотный словарь. Вначале реализуем обычную последовательную обработку.

Чтение и запись файлов сделаем чуть менее тривиальным способом, чтобы иметь возможность преобразовать код в асинхронный. Мы используем считывание во временно создаваемый массив в памяти, а затем с помощью объекта encoding преобразуем его в строку; аналогичным образом устроена и запись в файл:

```
let ReadFile fn =
   use f = File.OpenRead fn
   let len = (int)f.Length
   let cont = Array.zeroCreate len
   let cont' = f.Read(cont,0,len)
   let converter = System.Text.Encoding.UTF8
```

```
   converter.GetString(cont)

let WriteFile (str:string) fn =
    use f = File.OpenWrite fn
    let conv = System.Text.Encoding.UTF8
    f.Write(conv.GetBytes(str),0,conv.GetByteCount(str))
```

Для обработки файла и построения частнотного словаря опишем функцию ProcessFile, основанную на описанной ранее функции построения частотного словаря FreqDict:

```
let ProcessFile f =
    let dict =
      ReadFile f |>
      ToWords |>
      FreqDict |>
      Map.toSeq |>
      Seq.filter (fun (k,v) -> v>10 && k.Length>3)
    WriteDict dict (f+".res")
```

Теперь преобразуем эту программу к асинхронно-параллельному варианту. Для этого функции чтения и записи файлов заменим на асихронные:

```
let ReadFileAsync fn =
  async {
   use f = File.OpenRead fn
   let len = (int)f.Length
   let cont = Array.zeroCreate len
   let! cont' = f.AsyncRead(len)
   let converter = System.Text.Encoding.UTF8
   return converter.GetString(cont)
  }

let WriteFileAsync (str:string) fn =
   async {
    use f = File.OpenWrite fn
    let conv = System.Text.Encoding.UTF8
    do! f.AsyncWrite(conv.GetBytes(str),0,conv.GetByteCount(str))
  }
```

Функция ReadFileAsync возвращает Async<string>, то есть отложенное вычисление, которое может вернуть значение типа string. Аналогично WriteFileAsync имеет тип string -> string -> Async<unit>. Как видим, с точки зрения синтаксиса мы взяли обычные, синхронные определения, окружили их async-блоком, заменили операции чтения и записи на асинхронные и поставили в соответствующих строках восклицательный знак после let и do. Таким образом, преобразование обычной программы в асинхронную может быть выполнено весьма механическим образом.

Аналогично для превращения функции обработки в асинхронную достаточно выполнить такую же процедуру: заменить функции ввода-вывода на асинхронные, не забыть про восклицательные знаки и окружить все async-блоком:

```
let ProcessFileAsync f =
   async {
    let! str = ReadFileAsync f
    let dict =
      ToWords str |>
      FreqDict |>
      Map.toSeq|>
      Seq.filter (fun (k,v) -> v>10 && k.Length>3)
    let st = Seq.fold (fun s (k,v)->s+(sprintf "%s: %d\r\n" k v)) "" dict
    do! WriteFileAsync st (f+".res")
   }
```

Для обработки всех файлов в заданной директории используем функцию:

```
let ProcessFilesAsync ()=
  Async.Parallel
      [ for f in GetTextFiles(@"c:\books") -> ProcessFileAsync f ] |>
  Async.RunSynchronously
```

Она собирает все порожденные `ProcessFileAsync` отложенные вычисления в список, после чего применяет к нему `Async.RunSynchronously`. Именно в этот момент начинаются вычисления, и библиотека F# сама занимается распределением процессорного времени между задачами.

Такая модификация программы позволяет добиться существенного увеличения скорости на простой двухъядерной машине. Это объясняется тем, что блокирующий ввод-вывод обычно занимает значительное время в работе программы.

Мы видим, что достаточно простые манипуляции с текстом программы позволили нам превратить ее в асинхронную. С использованием языков программирования типа C# нам пришлось бы существенным образом перерабатывать структуру программы, здесь же за счет конструкции асинхронных выражений и системы вывода типов текст программы изменился мало, но ее суть при этом поменялась достаточно сильно. Если исходная программа состояла из функций, занимающихся обработкой информации, то асинхронный вариант возвращает отложенные асинхронные вычисления и манипулирует ими до тех пор, пока вызов `Async.RunSynchronously` не запустит эти вычисления на выполнение.

6.4. Агентный паттерн проектирования

В данном случае распараллеливание программы не представляло большого труда, поскольку она изначально состояла из большого количества одинаковых независимых потоков выполнения. Что же касается более сложных систем, распа-

раллеливание может оказаться более сложной задачей. Для этого может оказаться необходимым пересмотреть архитектуру системы.

Одним из подходов к построению распараллеливаемой архитектуры является использование агентного паттерна проектирования. В этом случае программа строится из легковесных, но частично автономных модулей – агентов, которые могут выполнять определенные действия в обмен на получение сообщений. Такие агенты могут работать параллельно, распределяя процессорное время гибким образом между собой.

Рассмотрим построение агента, который строит частотный словарь файла. Библиотека F# обеспечивает механизм передачи сообщений через класс `MailboxProcessor`. При создании агента мы используем метод `MailboxProcessor.Start`, передавая ему функцию работы с очередью сообщений в соответствии с таким шаблоном:

```
let ProcessAgent =
    MailboxProcessor.Start(fun inbox ->
       let rec loop() = async {
        let! msg = inbox.Receive()
        printf "Processing %s\n" msg
        do! ProcessFileAsync msg
        printf "Done processing %s\n" msg
        return! loop()
       }
       loop()
     )
```

Для обработки файла надо передать такому агенту сообщение при помощи метода `Post`:

```
ProcessAgent.Post(@"c:\books\prince.txt")
```

Для обработки множества файлов можно передать сразу целую последовательность сообщений – их обработка будет производиться *последовательно*, последовательность будет обеспечиваться механизмом очереди MailboxProcessor'а.

```
for f in GetTextFiles(@"c:\books") do ProcessAgent.Post f
```

Казалось бы, переход к агентному паттерну лишил наше решение способности обрабатывать файлы параллельно. Однако очень просто с помощью того же агентного паттерна мы можем построить распараллеливающий агент:

```
let MailboxDispatcher n f =
    MailboxProcessor.Start(fun inbox ->
        let queue = Array.init n (fun i -> MailboxProcessor.Start(f))
        let rec loop i = async {
            let! msg = inbox.Receive()
            queue.[i].Post(msg)
```

```
            return! loop((i+1)%n)
        }
        loop 0
    )
```

При создании `MailboxDispatcher` мы передаем ему количество параллельных агентов и такую же функцию обработки очереди сообщений, что и `MailboxProcessor`'у, после чего порождается массив из соответствующего количества `MailboxProcessor`'ов, которые могут работать независимо. При приходе сообщения диспетчеру оно переправляется следующему из созданных агентов в соответствии с простейшим механизмом распределения нагрузки – в данном случае *roundrobin*.

Параллельный агент, выполняющий обработку файлов в 2 дочерних агентах, будет описываться так:

```
let ParallelProcessAgent =
       MailboxDispatcher 2 (fun inbox ->
       let rec loop() = async {
        let! msg = inbox.Receive()
        printf "Processing %s\n" msg
        do! ProcessFileAsync msg
        printf "Done processing %s\n" msg
        return! loop()
       }
       loop()
     )
```

Агентный паттерн построения распределенных систем приобретает все большую актуальность в связи с возрастающей сложностью программных продуктов и все большей ориентированностью на архитектуру интернет-сервисов. Похожий подход является базовым для таких появляющихся в последние годы языков, как Google Go и Axum от Microsoft Research. Эти языки позволяют строить взаимодействующие по сети между разными узлами сообщества агентов.

В то время как агенты в F# представляют собой легковесные элементы, взаимодействующие в рамках одного приложения, использование такого паттерна проектирования в будущем позволит масштабировать систему за границы приложения. Для будущего развития F# предполагается улучшать и расширять агентный подход к построению приложений, возможно, заимствовав какие-то идеи из проекта Axum.

6.5. Использование MPI

Еще одно направление, где приходится сталкиваться с параллельным программированием, – это высокопроизводительные научные расчеты, проводимые на компьютерных кластерах. Такие кластеры обычно содержат некоторое количество высокопроизводительных компьютеров, соединенных высокоскоростными

каналами связи. Для решения задачи на всех компьютерах кластера запускается по экземпляру вычислительной программы, которые совместно вырабатывают решения, обмениваясь сообщениями.

Традиционно программирование для кластера требует четкого понимания того, как будет происходить обмен сообщениями для получения результата, и явного описания алгоритма обмена сообщениями. На текущий момент стандартом де-факто для протокола обмена сообщениями является MPI (Message Passing Interface) – API достаточно низкого уровня, позволяющее включать в программы простые операции обмена сообщениями, коллективную посылку сообщений (broadcast), операции агрегирования сообщений от нескольких узлов и т. д. Для платформы .NET существует реализация MPI.NET от университета штата Индиана, которая упрощает разработку программ для кластера на платформе .NET.

Для того чтобы разрабатывать программы для кластера, совсем не обязательно иметь в распоряжении такой кластер – достаточно на рабочую станцию (обычный компьютер) с Windows-совместимой системой установить MS-MPI, который свободно скачивается с сайта Microsoft (на момент написания книги текущая версия называлась HPC Pack 2008 R2 MS-MPI Redistributable Package). Если же вы запускаете программы на кластере под управлением Windows HPC Server – на нем уже установлены соответствующие библиотеки. Поверх MS-MPI необходимо установить упомянутую выше MPI.NET с сайта http://www.osl.iu.edu/research/mpi.net.

Программа для кластера обычно создается как консольное приложение. Для запуска нескольких копий приложения на кластере служит утилита `mpiexec`, которая автоматически производит копирование программы на все узлы кластера с последующим запуском всех экземпляров. Например, для запуска 10 экземпляров программы `fsmpi.exe` на кластере служит команда

```
mpiexec -n 10 FSMPI.exe
```

Для одномашинной конфигурации в MS-MPI также существует утилита `mpiexec`, которая запускает несколько процессов на одной машине, обеспечивая между ними правильное взаимодействие по интерфейсу MPI. Более того, на многоядерной конфигурации тоже можно использовать MPI как механизм распараллеливания приложения с целью его ускорения (однако для этого также существуют более простые средства, рассмотренные выше).

Внутри приложения для доступа к MPI мы используем специальный объект `Communicator.world` типа `Intracommunicator`. Количество запущенных на кластере экземпляров приложения доступно через свойство `Size`, а каждому запущенному приложению присваивается уникальный номер от `0` до `Size-1`, доступный через свойство `Rank`.

Типичное MPI-приложение на F# выглядит следующим образом:

```
let args = Environment.GetCommandLineArgs()
let env = new Environment(ref args)
let W = Communicator.world
// Проводим вычисления
env.Dispose()
```

Здесь мы получаем аргументы командной строки приложения (в них мы можем передавать какие-то осмысленные аргументы нашему приложению, а также дополнительные ключи для MPI-окружения), инициализируем MPI-окружение, получаем экземпляр объекта `Intracommunicator` для проведения дальнейших коммуникаций, проводим непосредственно вычисления, после чего не забываем вызвать метод `Dispose()` для MPI-окружения (еще более правильно было бы создавать объект `env` внутри `using`-блока, чтобы метод `Dispose` вызывался автоматически при выходе из области видимости).

Основные операции, предоставляемые интерфейсом MPI, следующие:

- `BroadCast` – посылка некоторого значения всем узлам сети от родительского узла `root`. Например, в результате выполнения такого фрагмента кода переменные `x` во всех экземплярах программы станут равны значению переменной на нулевом узле (переменная `x` должна быть объявлена как `mutable`):

```
W.BroadCast(ref x,0)
```

- посылка/прием сообщений осуществляется методами `Send/Receive` – при этом мы можем указывать явно номера узлов отправителя/получателя или же принимать сообщения от любых узлов. Также можно снабжать сообщения целочисленным тегом и ожидать приема конкретных тегов. Сами сообщения – это значения переменных, которые автоматически сериализуются и десериализуются. Посылка и прием сообщений по умолчанию являются блокирующими, то есть выполнение программы приостанавливается до успешной посылки;
- неблокирующая посылка сообщения осуществляется методом `ImmediateSend/ImmediateReceive`. Например, `ImmediateSend` инициирует посылку сообщения и продолжает выполнение программы:

```
let status = W.ImmediateSend(ref x,source,dest)
// продолжаем вычисления
status.Wait()
```

- синхронизация всех процессов производится вызовом метода `Barrier` – все процессы, подойдя к такому барьеру, ожидают друг друга и продолжают выполняться совместно;
- сбор данных со всех работающих экземпляров проводится функцией `Gather`, которая формирует на одном из узлов массив значений, полученных от каждого из других узлов. Обратную задачу – распределение массива значений по разным узлам – выполняет функция `Scatter`;
- во многих случаях нас интересуют не сами значения, полученные с каждого узла, а их агрегатная функция – сумма, количество и т. д. В этом случае используется функция `Reduce`;
- существуют и другие, более сложные функции, предоставляемые интерфейсом MPI, но мы не будем на них останавливаться.

Рассмотрим пример вычисления числа π методом Монте-Карло на кластере. Сам алгоритм вычисления был рассмотрен ранее в главе 3. Мы также позаимству-

ем оттуда определение функции rand для генерации псевдослучайной последовательности. Основной код программы приведен ниже:

```
let args = Environment.GetCommandLineArgs()
let env = new Environment(ref args)
let W = Communicator.world
let N = 10000
Console.WriteLine("Running on node {0} out of {1}",W.Rank+1,W.Size)
let n = Seq.zip (rand 1.0 (W.Rank*7+1)) (rand 1.0 (W.Rank*3+2))
         |> Seq.take N
         |> Seq.filter (fun (x,y) -> x*x+y*y<1.0)
         |> Seq.length
if W.Rank=0 then
   let res = W.Reduce<int>(n,Operation<int>.Add,0)
   let pi = 4.0*float(res)/float(N)/float(W.Size)
   Console.WriteLine("Pi={0}",pi)
else
   W.Reduce<int>(n,Operation<int>.Add,0) |> ignore
env.Dispose()
```

Как вы помните, вычисление числа пи сводится к «бросанию» большого количества псевдослучайных точек в квадрат со стороной 1 и вычислению количества точек, попавших в четверть круга. В нашем случае каждый узел кластера будет бросать N точек и вычислять количество попаданий n – соответствующий алгоритм уже был рассмотрен в главе 3.

Далее мы разделяем программу на две части – узел 0 (Rank=0) отвечает за сбор суммарного количества попаданий и вычисление числа пи, а остальные узлы лишь посылают ему вычисленное ими количество попаданий. Вся механика обмена сообщениями осуществляется функцией Reduce – будучи вызванной на «главном» узле (его номер передается в качестве последнего агрумента), она собирает все значения, применяет к ним агрегирующую операцию и возвращает результат, в то время как на остальных узлах производится только отправка значения.

Для сравнения – аналогичный код, основанный на явной отправке и получении сообщений с вычислением суммы на нулевом узле, выглядит следующим образом:

```
let args = Environment.GetCommandLineArgs()
let env = new Environment(ref args)
let W = Communicator.world
let N = 10000
let size = W.Size
let n = Seq.zip (rand 1.0 (W.Rank*7+1)) (rand 1.0 (W.Rank*3+2))
         |> Seq.take N
         |> Seq.filter (fun (x,y) -> x*x+y*y<1.0)
         |> Seq.length
if W.Rank=0 then
   let res = [1..(size-1)]
```

```
                |> List.map(fun _ ->
 W.Receive<int>(Communicator.anySource,Communicator.anyTag))
                |> List.sum
    let pi = 4.0*float(n+res)/float(N)/float(W.Size)
    Console.WriteLine("Pi={0}",pi)
else
    W.Send<int>(n,0,0)
env.Dispose()
```

Из примера видно, что код для распараллеливания на кластере и на локальной машине сильно отличается, хотя используемые подходы – обмен сообщениями, синхронизация и т. д. – весьма похожи. Ожидается, что со временем мы будем наблюдать все большую конвергенцию подходов к параллельным вычислениям и в то же время упрощение этих подходов, чтобы подавляющее большинство разработчиков могло комфортно создавать параллельные приложения. Очевидно, что использование функционального подхода – это один из шагов в этом направлении.

Также следует отметить все большую распространенность облачных вычислений, которые также открывают путь к использованию огромной вычислительной мощности большого числа компьютеров, расположенных в дата-центрах облачных провайдеров. Ниже в главе 7 мы рассмотрим использование F# для создания облачных распределенных вычислительных сервисов.

7. Решение типовых задач

В этой главе мы остановимся на нескольких практических моментах использования F# для решения типовых задач. Хочется надеяться, что в ней вы найдете много готовых сценариев использования F#, которые затем сможете использовать для решения своих задач, объединяя простые фрагменты кода, приведенные в этой книге, как кирпичики, для достижения более сложной функциональности. Тут мы рассматриваем вопросы построения веб-приложений и приложений для клиентов, доступ к данным, вычислительные задачи и другие примеры, в которых использование F# и функционального подхода к программированию представляется эффективным.

7.1. Вычислительные задачи

7.1.1. Вычисления с высокой точностью

Часто при решении математических задач возникает необходимость вычислений с высокой точностью. Для этого платформа .NET предусматривает специальные типы данных. Например, тип данных `System.Numerics.BigInteger` позволяет оперировать с целыми числами произвольной длины.

В качестве примера рассмотрим вычисление факториала. Обычное определение

```
let rec fact = function
    1 -> 1
  | n -> n*fact (n-1)
```

не позволяет вычислять большие значения факториала – например, `fact 17` уже дает отрицательный результат, что говорит о переполнении целого типа.

Для борьбы с этой проблемой используем тип BigInteger в качестве результата функции. В сам текст функции придется внести минимальные изменения:

```
let rec fact = function
    1 -> 1I
  | n -> BigInteger(n)*fact (n-1)
```

Аналогичным образом для точного представления дробей используется тип `BigRational` (или BigNum, как он называется в библиотеке F#). С его помощью мы

можем, например, точно посчитать аппроксимацию функций путем «наивного» суммирования ряда Тейлора. Например, чтобы вычислить

$$e^x = \sum_{n=1}^{\infty} \frac{x^n}{n!},$$

мы можем использовать следующие определения:

```
let nth n x = BigNum.PowN(x,n) / BigNum.FromBigInt(fact n)
let exp x = 1N+Seq.fold(fun s n -> s+(nth n x)) 0N [1..50]
```

Функция nth вычисляет n-ый член ряда Тейлора, используя для этого определенную нами ранее функцию вычисления факториала с произвольной точностью, а далее мы используем свертку списка для вычисления суммы ряда.

7.1.2. Комплексный тип

Операции для работы с комплексным типом Complex содержатся в F# Power Pack. Пример с построением множества Мандельброта в главе 1 содержит содержательные фрагменты кода, использующие комплесный тип, для определения сходимости функциональной последовательности на комлексной плоскости.

7.1.3. Единицы измерения

Во многих физических задачах величины имеют определенную размерность, и контроль за соблюдением размерности является лишней возможностью проверить правильность вычислений. Поскольку F# предназначен для решения вычислительных задач, в него на уровне синтаксиса языка была введена возможность указания размерности используемых величин.

Рассмотрим простейшую физическую задачу моделирования броска тела под углом. В этой задаче нам потребуются основные физические размерности: метр и секунда, – которые мы можем описать следующим образом:

```
[<Measure>]
type m

[<Measure>]
type s
```

При описании значений мы можем теперь указывать их размерность, например константа g=9.8 м/с^2 мы опишем так:

```
let g = 9.8<m/s^2>
```

При этом тип такой константы будет float<m/s^2> и, строго говоря, не будет совместим с типом float. В частности, мы не сможем передать эту константу стандартным функциям, таким как sin или exp, – придется применять явное приведе-

ние типов: sin(float g). Если же мы хотим описать свою функцию, которая будет принимать значения указанной размерностью, то мы можем использовать либо явное указание размерности в типе, либо шаблон _:

```
let double x = x*2.0<_>
```

Описанная таким образом функция double сможет применяться ко всем значениям типа float с любой размерностью.

Для решения нашей задачи мы опишем фунцию go, которая будет принимать в качестве аргумента текущее положение тела и текущую скорость, а также текущее время. Затем, пока координата тела по оси Y не станет отрицательной (то есть пока тело не коснется земли), мы будем вычислять следующее положение тела и вызывать функцию рекурсивно:

```
let rec go (vx,vy) (x,y) (time:float<s>) =
  printf "Time: %A Pos: (%A,%A), Speed: (%A,%A)\n" time x y vx vy
  if y >= 0.0<m> then
    let h = 0.1<s>
    let dx = h*vx
    let dy = h*vy
    let dspy = -h*g
    go (vx,vy+dspy) (x+dx,y+dy) (time+h)

go (3.0<m/s>,3.0<m/s>) (0.0<m>,0.0<m>) 0.0<s>
```

Обратите внимание, что, помимо проверки типов, в данном случае осуществляются проверка и вывод размерностей! Например, размерности dx и dy автоматически выводятся как float<m>, а dspy – как float<m/s>. Попытка передать функции параметры с неправильной размерностью будут обнаружены на этапе компиляции.

Следует также отметить, что в библиотеке Microsoft.FSharp.Math.SI содержатся стандартные единицы измерения системы Си, а в Microsoft.FSharp.Math.PhysicalConstants – значения основных физических констант вместе с единицами измерения.

7.1.4. Использование сторонних математических пакетов

Если вы планируете программировать задачи, связанные с серьезными математическими или статистическими расчетами, вы сможете, как правило, найти неплохие существующие библиотеки на платформе .NET, которые реализуют множество математических алгоритмов: работу с векторами и матрицами (включая весьма нетривиальные алгоритмы типа вычисления собственных чисел/векторов, решение СЛАУ и т. д.), расширенную работу с комплексными числами, функции для построения статистических распределений, для интерполяции,

численного дифференцирования и интегрирования и т. д. Подробное рассмотрение предоставляемых такими библиотеками функций выходит за рамки этой книги – ограничимся лишь упоминанием нескольких библиотек, хорошо работающих с F#:

- Extreme Optimization Library является де-факто стандартом для математических расчетов на платформе .NET. Библиотека является платной, ее 60-дневная пробная версия доступна с сайта http://www.extremeoptimization.com. Пример использования Extreme Optimization Library с F# для нахождения собственных значений и векторов матрицы есть в книге [6];
- библиотеки F# for Numerics и F# for Visualization от Flying Frog Consultancy представляют собой набор хорошо интегрированных между собой библиотек, специально разработанных для языка F#. Библиотеки являются платными и могут поставляться со специальными книгами, описывающими их использование;
- Math.NET Numerics – это библиотека с открытым исходным кодом, доступная по адресу http://mathnetnumerics.codeplex.com. Она является частью большого проекта Math.NET (http://www.mathdotnet.com), который, помимо численной библиотеки, будет включать в себя модуль для символьных вычислений и обработки сигналов. В Math.NET Numerics вы найдете почти все из описанных выше возможностей для численных математических расчетов. Поскольку библиотека является свободно распространяемой, с возможностью посмотреть исходный код, мы приведем несколько примеров ее использования.

Для использования Math.NET Numerics необходима основная библиотека `MathNet.Numerics.DLL`. Рассмотрим простейший пример применения Math.NET Numerics для статистической обработки данных.

Для начала используем пространство имен `MathNet.Numerics.Distributions` для генерации псевдослучайной последовательности с заданным распределением. Помимо использованного в этом примере нормального распределения, в пакет входит множество других распределений, как дискретных, так и непрерывных:

```
open MathNet.Numerics.Distributions
let d = new Normal()
d.Variance <- 0.5
let seq = d.Samples()
```

В результате мы получаем бесконечную последовательность значений `seq`, распределенных нормально с указанными параметрами.

Теперь используем средства статистического анализа для построения гистограммы данной последовательности значений. Для этого создадим объект Histogram и передадим ему последовательность данных:

```
open MathNet.Numerics.Statistics
let H = new Histogram(Seq.take 1000 seq,10)
```

Нам в явном виде пришлось ограничить последовательность тысячью элементов, чтобы избежать зацикливания. Второй параметр показывает количество интервалов, на которые разбивается диапазон входной последовательности для построения гистограммы. Сами интервалы доступны в виде объектов `Bucket`. Например, построим графическое изображение гистограммы в виде звездочек:

```
for i in 0..H.BucketCount-1 do
  let bk = H.[i]
  for j in 0 .. int(bk.Count/10.0) do printf "*"
  printfn ""
```

Другой пример использования Math.NET Numerics мы увидим чуть ниже, когда будем рассматривать пример с интерполяцией зависимостей.

Надо учитывать, что Math.NET – достаточно молодой проект, поэтому многие его части находятся в активной разработке.

7.2. Доступ к данным

Как мы отмечали, язык F# идеально подходит для обработки данных. В этом разделе мы рассмотрим различные источники, из которых эти данные могут поступать. Ранее в книге мы уже видели примеры работы с текстовыми файлами, в том числе с файлами данных, разделенных запятой (CSV). Здесь мы рассмотрим работу с базами данных, с Microsoft Excel, а также со слабоструктурированными XML-данными.

7.2.1. Доступ к реляционным базам данных (SQL Server)

Как полноценный язык на платформе .NET, F# поддерживает весь спектр технологий для доступа к реляционным данным: ADO.NET, Entity Framework и др. Для реализации не слишком сложных приложений, по мнению автора, проще всего использовать технологию LINQ (Language Integrated Query), реализация которой для F# входит в состав F# Power Pack.

Технология LINQ позволяет использовать квотирование для формулирования запросов к базе данных на языке F#. При этом библиотека преобразует запрос в SQL-запрос, который отправляется СУБД, а получившийся на выходе результат преобразует в последовательность (`seq`) объектов, которая и возвращается в приложение.

Для того чтобы такая технология работала, необходимо сгенерировать набор классов, соответствующий по структуре конкретной базе данных, к которой осуществляется доступ. Это можно сделать при помощи утилиты командной строки SqlMetal либо из Visual Studio – в обоих случаях генерируются C#-классы, которые можно скомпилировать в виде отдельного проекта в DLL-файл и затем использовать из F# в диалоговом режиме или в составе проекта.

В качестве примера рассмотрим доступ к базе данных AdventureWorks, которая входит в стандартный набор примеров к SQL Server 2008. Скачать и установить базу данных можно по адресу http://msftdbprodsamples.codeplex.com.

Чтобы упростить работу со сгенерированными классами, проще всего создать решение в Visual Studio, куда включить один проект на F# и проект на C#, содержащий автоматически сгенерированные классы. Выберем F# Application как тип проекта для F# (см. рис. 7.1) и добавим к решению еще один проект типа C# Class Library (для этого при создании проекта надо будет выбрать опцию «добавить к существующему решению» (Add to solution) вместо используемой по умолчанию опции «создать новое решение» (Create Solution).

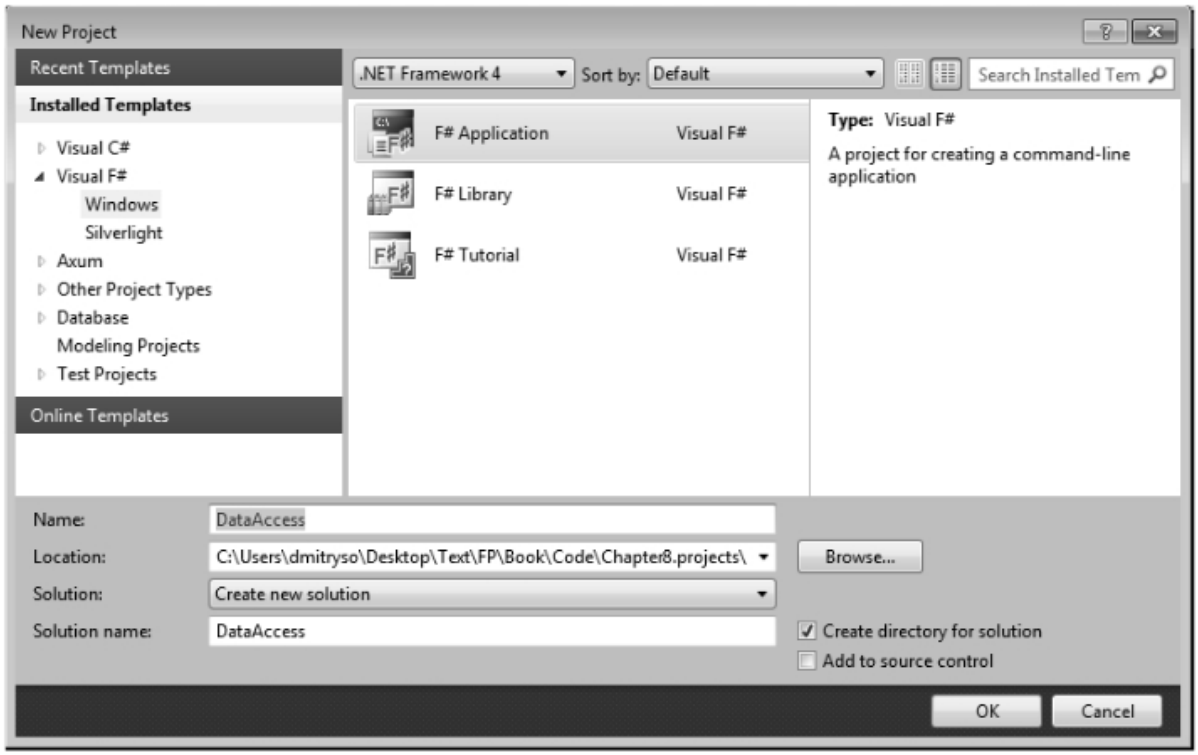

Рис. 7.1. Создание F#-проекта

В C#-проекте удалим файл `Class1.cs`, добавленный по умолчанию, и добавим к проекту новый элемент типа LINQ to SQL Classes. При этом создастся файл с расширением .dbml, при открытии которого Visual Studio будет показывать графический дизайнер структуры базы данных. Для создания классов, соответствующих таблицам базы данных, необходимо перетащить на этот дизайнер требуемые таблицы (см. рис. 7.2). Получившийся результат можно видеть на рис. 7.3.

Для доступа к полученной модели из проекта на F# необходимо добавить в него ссылки на стандартные модули System.Data и System.Data.Linq, а также на FSharp.Powerpack и FSharp.PowerPack.Linq.

Весь доступ к базе данных осуществляется с помощью так называемого контекста данных (`DataContext`). Если мы назвали файл модели данных `AdventureWorks`, то в нем был автоматически сгенерирован класс `AdventureWorksDataContext`:

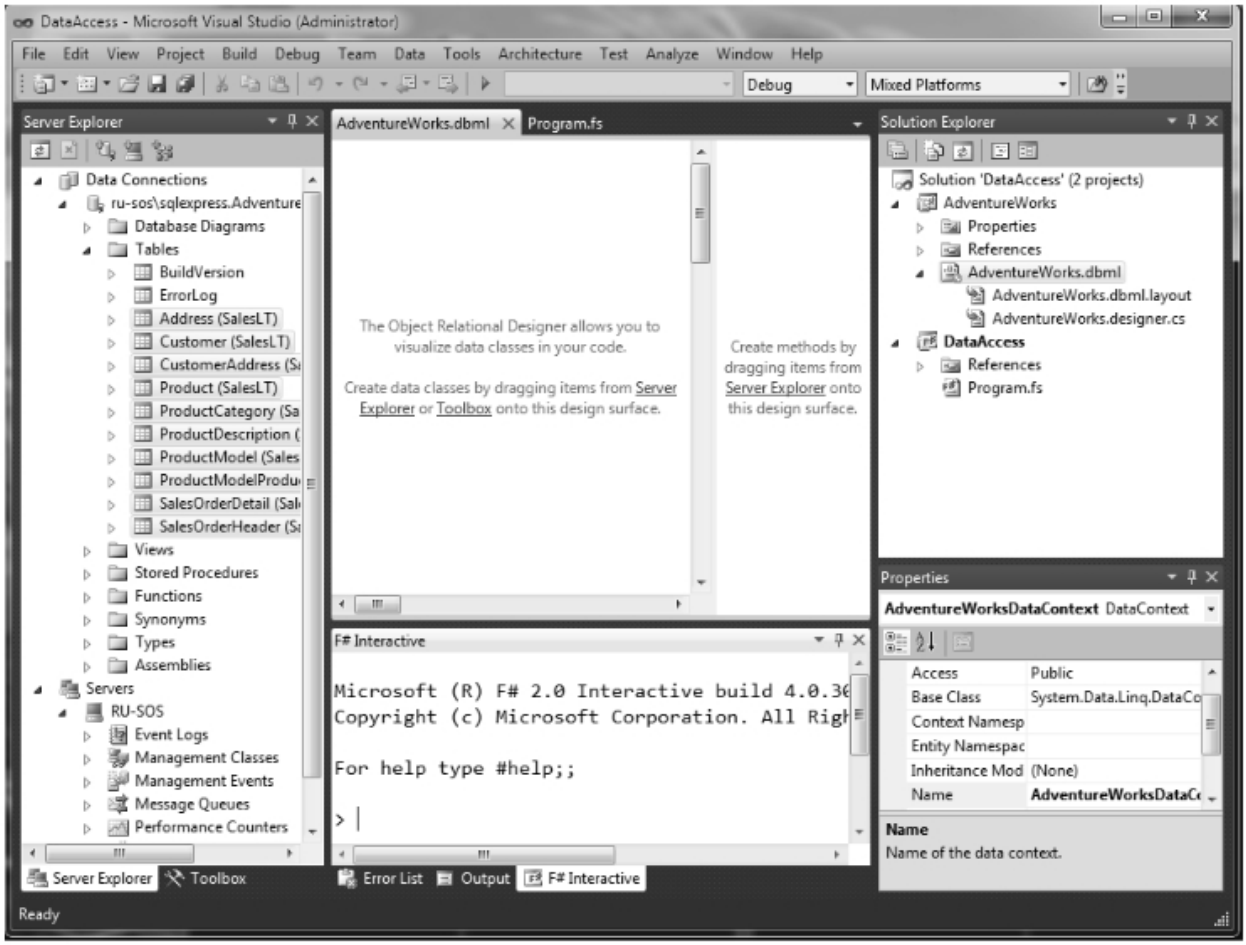

Рис. 7.2. Решение с проектом на F# и моделью данных на C#

```
open AdventureWorks
open Microsoft.FSharp.Linq.Query

let db = new AdventureWorksDataContext()
```

Для чтения данных из таблицы продуктов Products мы используем следующий синтаксис:

```
let pr = query <@ seq { for p in db.Products -> (p.Name,p.ListPrice) } @>
```

Здесь `query` – метод, определенный в Microsoft.FSharp.Linq.Query, который отвечает за преобразование квотированного выражения в SQL, запуск его на стороне СУБД и возвращение в F#-код ленивой последовательности-результата. Обратите внимание, что сама по себе приведенная выше строчка не ведет к обращению к базе данных, – обращение производится тогда, когда программа запрашивает данные из ленивой последовательности, например в момент показа данных на экране:

```
let display T = for x in T do printfn "%A" x
display pr
```

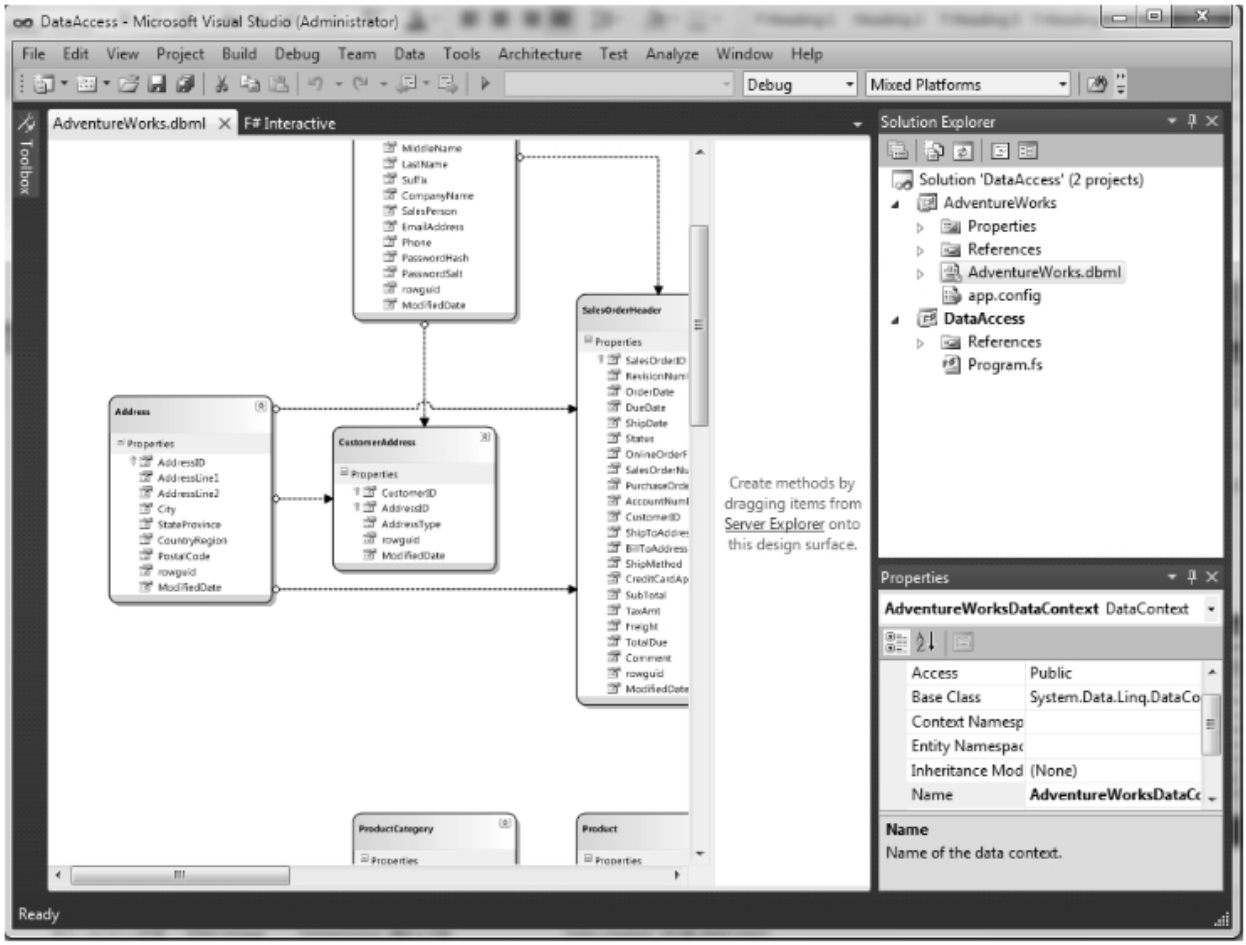

Рис. 7.3. Построенная модель данных LINQ to SQL в Visual Studio

Возможно использование в рамках запроса более сложных конструкций, например:

```
let prods =
  query <@ seq { for p in db.Products do
                 for c in db.ProductCategories do
                   if p.ProductCategoryID.Value = c.ProductCategoryID
                    && c.Name = "Tires and Tubes"
                    then yield p}
           |> Seq.sortBy (fun p -> p.Name)
           |> Seq.map (fun p -> (p.Name,p.ListPrice)) @>
```

Здесь мы производим объединение (join) двух таблиц: продуктов и категорий продуктов, с последующей выборкой только тех продуктов, которые находятся в категории «Tires and Tubes». После этого мы производим сортировку по названию продукта и выборку необходимых полей данных – названия и цены.

Обратите внимание, что поскольку сортировка и выборка расположены внутри квотирования, они будут также выполняться на стороне SQL-сервера. Еще при трансляции такого запроса генерируется неявное объединение двух таблиц

с помощью составного SELECT-запроса, которое, тем не менее, оптимизируется в объединение на уровне SQL-сервера.

В данном случае выражение представляло собой объединение двух таблиц. Однако не всегда такая запись объединения может быть правильно интерпретирована анализатором query и преобразована в эффективный SQL-код. Для надежности можно также использовать операцию `Query.join` в явном виде, которая генерирует эффективный SQL-код с оператором JOIN:

```
let prods =
  query <@ join db.Products db.ProductCategories
            (fun p -> p.ProductCategoryID.Value)
            (fun pc -> pc.ProductCategoryID)
            (fun p pc -> (p.Name,pc.Name)) @>
```

В этом вызове первая функция выбирает ключ для объединения из первой таблицы, вторая – из второй, а третья по обеим таблицам выбирает необходимый результат.

Обратите внимание, что для достижения такого же результата – списка продуктов и их категорий – в принципе, можно было бы использовать более простой запрос:

```
let prods3 =
  query <@ seq { for p in db.Products -> (p.Name,p.ProductCategory) } @>
```

Здесь мы используем тот факт, что порожденные с помощью визуального редактора или SqlMetal объекты достаточно «умны», и они могут сами осуществлять запрос к связанной таблице `ProductCategory` для нахождения названия категории продукта. Причем механизм LINQ достаточно умен для того, чтобы в этом случае не осуществлять заполнение этого поля для каждого объекта, а также, как и в предыдущем примере, использовать объединение таблиц и один SQL-запрос. Однако явное объединение в некотором смысле имеет то преимущество, что мы в меньшей степени зависим от «прозорливости» анализатора запросов LINQ.

С помощью порожденного кода мы можем также просто добавлять записи в базу данных. Для этого необходимо породить объект соответствующего типа, после чего вставить его в нужные таблицы и вызвать `SubmitChanges`:

```
let pc = new ProductCategory()
pc.Name <- "Child Clothing"
pc.ParentProductCategoryID <- new Nullable<int>(3)
pc.rowguid <- Guid.NewGuid() ; pc.ModifiedDate <- DateTime.Now
db.ProductCategories.InsertOnSubmit(pc)
db.SubmitChanges()
```

Использование неявного доступа к базам данных существенно упрощает процесс доступа по сравнению с более низкоуровневыми библиотеками доступа типа

ADO.NET, однако оставляет меньше контроля разработчику. Поэтому в тех случаях, когда важна производительность, рекомендуется контролировать SQL-код, выполняемый LINQ-запросами. Для этого можно перенаправить логпоток в произвольный объект типа `Stream`, например:

```
db.Log <- Console.Out
```

В логе будут показываться все выполняемые SQL-запросы к базе данных.

7.2.2. Доступ к слабоструктурированным данным XML

Другой важный источник данных, с которым наверняка придется столкнуться в серьезных проектах, – это слабоструктурированные данные в формате XML. В виде XML можно рассматривать веб-страницы в формате XHTML, поток блог-заметок в виде RSS, результат вызова веб-сервисов и многое другое. Рассмотрим, как можно работать с XML-данными на примере разбора RSS-ленты блога.

RSS-лента имеет приблизительно такой формат:

```
<?xml version="1.0" encoding="utf-8"?>
<rss xmlns:dc="http://purl.org/dc/elements/1.1/"
     version="2.0">
  <channel>
    <title>Название блога</title>
    <link>http://blogs.msdn.com/sos</link>
    <description>Про блог</description>
    <item xml:lang="ru-RU">
      <dc:creator>Дмитрий Сошников</dc:creator>
      <title>Заголовок заметки</title>
      <link>http://blogs.msdn.com/...</link>
      <pubDate>Wed, 27 Aug 2008 13:02:55 GMT</pubDate>
      <description>Краткое описание</description>
      <body xmlns="http://www.w3.org/1999/xhtml">Содержимое</body>
      <category>Категория 1;Категория 2</category>
    </item>
  </channel>
</rss>
```

Для работы с XML-данными в библиотеке .NET существуют три основных способа:

- чтение потоков XML-данных с использованием объектов `XmlReader/XmlWriter`. Такой подход хорошо подходит для длинных документов, которые можно обрабатывать последовательно без загрузки в память целиком. Тем, кто захочет писать высокопроизводительные приложения, работающие с большими объемами данных, стоит использовать такой подход;
- разбор XML-документа в памяти с использованием объекта `XmlDocument`;

- использование механизма LINQ для XML. Такой подход представляется наиболее простым и перспективным.

Для начала рассмотрим, как можно работать с документом с помощью XmlDocument. Мы загружаем документ в память следующим образом (при этом осуществляются его разбор и анализ, и на этапе загрузки генерируется исключение, если документ не является правильно построенным):

```
open System.Xml
let xfile = @"c:\samples\blog.xml"

let xdoc = new XmlDocument()
xdoc.Load(xfile)
```

Для извлечения заголовков всех записей из RSS-потока можно использовать следующую конструкцию:

```
let titles =
   seq{ for t in xdoc.SelectNodes("//item/title") -> t.InnerText }
```

Конструкция SelectNodes выбирает последовательность узлов в соответствии с XQuery-запросом; синтаксис //item означает выбрать все узлы <item> (вне зависимости от их местоположения в дереве документа), а title указывает на конкретное поле <title> внутри узла <item>.

Если мы хотим выбрать сразу несколько полей внутри <item> за один проход, мы можем выбирать узлы при помощи xdoc.SelectNodes(«//item») и затем внутри такого узла искать подузлы, например:

```
type BlogRecord = { title: string; desc: string; categories: string[] }
let records =
  let node (t:XmlNode) x = t.SelectSingleNode(x).InnerText
  seq{
    for t in xdoc.SelectNodes("//item") ->
       { title=node t "title"; desc = node t "description" ;
         categories = (node t "category").Split(';') }}
```

В этом примере мы одновременно описываем тип записи для хранения заметки в блоге, а также производим преобразование строки со списком категорий в более удобный для обработки массив строк. Далее с такой последовательностью удобно прозводить различные операции, например можно вычислить количество заметок по каждой из категорий следующим образом (функция FreqDict построения частотного словаря последовательности слов была нами описана ранее в главе 6):

```
records |> Seq.collect (fun x -> x.categories) | FreqDict
```

Теперь рассмотрим решение такой же задачи с использованием LINQ для XML. Для этого необходимо подключить к проекту System.Xml.Linq.dll и открыть соответствующие модули:

```
#r "System.Xml.Linq.dll"
open System.Xml
open System.Xml.Linq
```

Для работы с XML-данными используется тип XDocument:

```
let xdoc = XDocument.Load(xfile)
```

По объекту типа XDocument можно перемещаться вниз по дереву тегов с помощью методов Element/Attribute; также можно искать все узлы ниже текущего с помощью метода Descendants. Все эти методы принимают аргумент типа XName, представляющий собой полностью квалифицированное имя узла. Для получения объекта типа XName из текстового имени определим простую функцию:

```
let xn s = XName.op_Implicit s
```

С учетом этого описанные выше функции titles и records можно определить так:

```
let titles =
  seq{ for t in xdoc.Descendants(xn "item") ->
t.Element(xn "title").Value }

let records =
  let xv (t:#XElement) n = t.Element(xn n).Value
  seq{ for t in xdoc.Descendants(xn "item") ->
         {  title=xv t "title";
        desc=xv t "description";
          categories=(xv t "category").Split(';') }}
```

Хотя на первый взгляд может показаться, что большой разницы от использования XmlDocument или XDocument нет, это не так. При реализации более сложных алгоритмов работы с XML-документами вы почувствуете преимущества LINQ в попеременной работе с атрибутами и элементами. Кроме того, с помощью LINQ очень просто реализуется генерация XML-файлов, что демонстрируется следующим примером:

```
let xd = new XDocument(
          new XElement(xn "root",
           seq { for x in 1..3 ->
                  new XElement(xn "item", x.ToString(),
                  new XAttribute(xn "type","numeric")) }))
```

Этот фрагмент кода создает следующий XML-файл:

```
<root>
  <item type="numeric">1</item>
```

```
  <item type="numeric">2</item>
  <item type="numeric">3</item>
</root>
```

Благодаря тому что конструкторы `XElement/XAttribute` могут принимать в качестве аргументов последовательности других элементов, оказывается возможным использовать такой удобный синтаксис генерации XML-файла, перемежая имена элементов и атрибутов с циклическими конструкциями.

В качестве примера рассмотрим преобразование двух XML-файлов: RSS-поток новостей преобразуется следующей функцией в XHTML-код, отображающий список из всех заголовков блога:

```
let html = new XDocument(
            new XElement(xn "html",
             new XElement(xn "body",
              new XElement(xn "h1", "Blog Content"),
              new XElement(xn "ul",
               seq { for x in xdoc.Descendants(xn "item") ->
                       new XElement(xn "li",x.Element(xn "title").Value)
})))))
```

Следует также упомянуть, что для генерации XML-файлов могут использоваться различные методы:

- генерация текстового файла, имеющего форму правильного XML. Такой подход имеет много недостатков, поскольку есть риск сформировать неправильный XML-файл, не учтя какого-нибудь из тонких моментов XML-синтаксиса (в стандарте XML есть много тонкостей, связанных с кодировкой файла, с кодированием символов, входящих в состав XML-тегов, и т. д.);
- использование `XmlWriter` – этот подход предпочтителен для эффективной генерации очень больших документов;
- использование `XmlDocument` для конструирования XML-файла чрезвычайно трудоемко и неэффективно;
- `XDocument` / LINQ следует использовать в подавляющем большинстве случаев.

7.2.3. Работа с данными в Microsoft Excel

Во многих задачах, особенно в финансовых расчетах, очень часто исходные данные содержатся в таблицах Microsoft Excel. Один из вариантов получения доступа к таким файлам – это сохранить их в текстовом формате CSV (*Comma-Separated Values*), который затем обрабатывать стандартными функциями для работы со списками. Пример обработки такого файла мы уже видели в главе 3.

Другой альтернативой является использование средств офисного программирования Visual Studio Tools for Office, которые позволяют управлять работающим приложением Microsoft Excel из программы на F#. Для этого необходимо подключить к проекту несколько библиотек:

```
#r "FSharp.Powerpack"
#r "Microsoft.Office.Interop.Excel"
#r "Office"

open Microsoft.Office.Interop.Excel
```

Рассмотрим простую задачу – пусть в Excel содержатся координаты точек, для которым мы построим линейную аппроксимацию методом наименьших квадратов. Начальный вид Excel-таблицы приведен на рис. 7.4 – нам необходимо заполнить колонки Y_SQ и Diff.

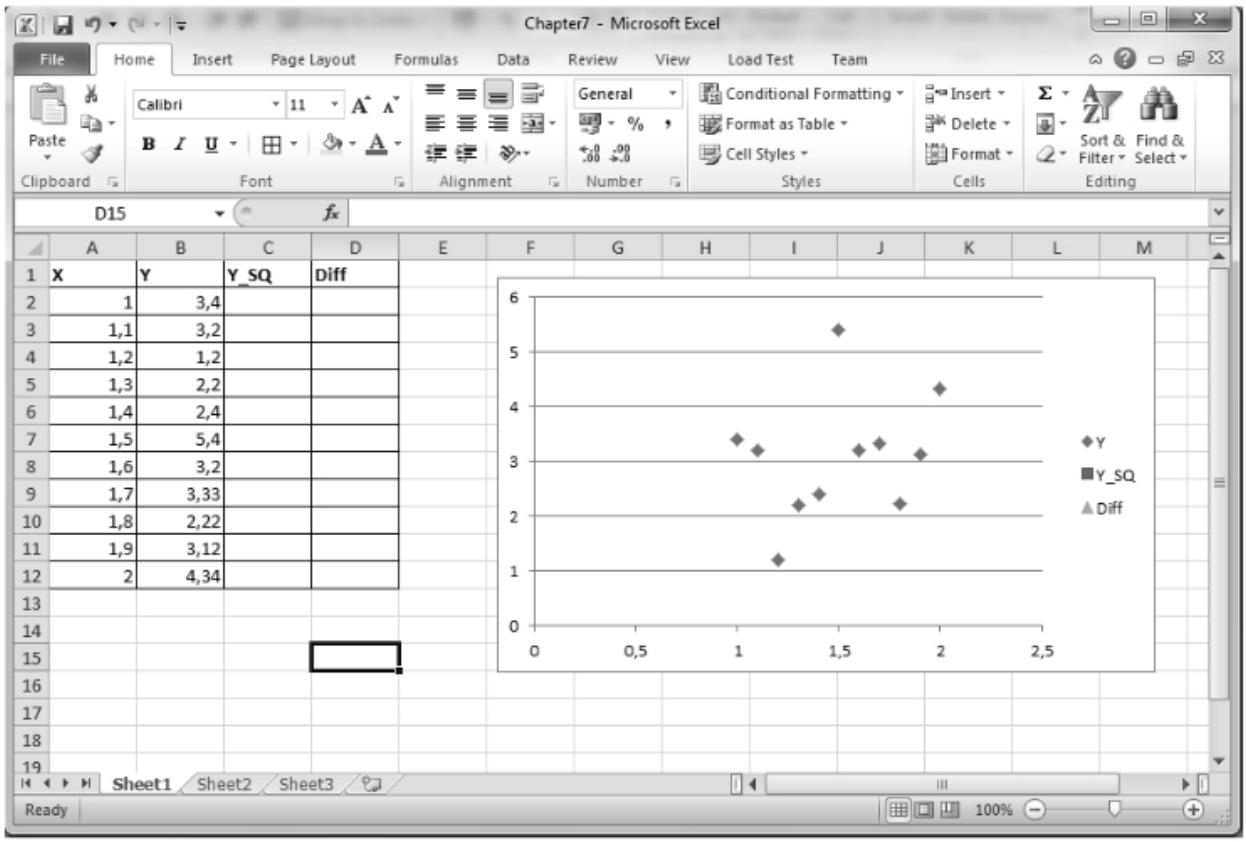

Рис. 7.4. Начальные данные в Excel-таблице

Для того чтобы открыть Excel и загрузить в него исходный файл, необходимо использовать следующий код:

```
let app = new ApplicationClass(Visible = true)
app.Workbooks.Open(@"c:\fsbook\code\Chapter7.xlsx")
```

Использование атрибута `Visible=true` позволяет отображать работающее приложение Excel и наблюдать, что происходит при управлении приложением из F#. Если в дальнейшем предполагается развертывать приложение на сервере, то лучше использовать невидимый режим – однако следует помнить, что для работоспо-

собности такого подхода необходимо установленное на компьютере приложение Microsoft Excel.

Определим базовые функции для доступа к ячейкам таблицы:

```
let cell x y =
  let range = sprintf "%c%d" (char (x + int 'A')) (y+1)
  (app.ActiveSheet :?> _Worksheet).Range(range)

let get x y = (cell x y).Value(System.Reflection.Missing.Value)
let set x y z = (cell x y).Value(System.Reflection.Missing.Value) <- z
```

Функция `cell` дает ссылку на соответствующую ячейку по целочисленным координатам – сначала из координат формируется именованное название ячейки в терминах Excel, а затем вызывается соответствующий метод, возвращающий ссылку на ячейку с таким номером в текущем рабочем листе, открытом в Excel. Функции `get` и `set` считывают значение из ячейки и записывают значения в ячейки соответственно.

Теперь, имея доступ к индивидуальным ячейкам, опишем функции, которые возвращают последовательность элементов в столбце:

```
let col x =
   Seq.unfold
    (fun i ->
       let v = get x i
       if v=null then None
       else Some(v,i+1)) 0
```

`Seq.unfold` – это конструктор последовательности с состоянием. В роли состояния здесь выступает целое число – номер текущей строки, который начинается с 0. Далее для каждого следующего члена извлекается значение из ячейки с текущим номером `i`, и если оно не равно `null` – возвращается пара из значения `v` и следующего состояния `i+1`. В противном случае возвращается `None`, что является признаком конца последовательности.

Данная функция возвращает последовательность типа `object`, которая может содержать объекты разных типов. Для наших нужд мы опишем функцию, которая будет отсекать заголовок и возвращать последовательность типа `float`:

```
let fcol n =
  (Seq.skip 1 (col n)) |> Seq.map (fun o -> o :?> float)
```

С помощью этой функции мы легко получим множество координат из Excel-файла в виде последовательности пар как `Seq.zip (fcol 0) (fcol 1)`.

Для записи результатов в файл опишем функцию `writecol`:

```
let writecol n seq =
  Seq.iteri (fun i x -> set n i x) seq
```

Для записи последовательности типа float с заголовком типа string используем функцию:

```
let fwritecol n (hd:string) s =
   writecol n (Seq.append [hd:>Object] (Seq.map (fun x -> x:>Object) s))
```

Пусть основной алгоритм описан в виде функции interpolate, которая возвращает линейную функцию, являющуюся результатом интерполяции. Тогда запись результатов интерполяции в файл можно описать так:

```
let f = interpolate (Seq.zip (fcol 0) (fcol 1))

fwritecol 2 "Y_SQ" (Seq.map (fun x -> f x) (fcol 0))
fwritecol 3 "Diff" (Seq.map2 (fun x y -> abs(x-y)) (fcol 1) (fcol 2))
```

Нам осталось лишь описать функцию interpolate, реализующую метод наименьших квадратов. Для последовательности точек $\{x_i,y_i\}_{i-1}^{n}$ нам необходимо подобрать коэффициенты кривой $y=ax+b$, которая минимизирует квадрат отклонения точек от прямой. Известно, что для двумерного случая эти коэффициенты определяются соотношениями:

$$b = (\Sigma y_i \Sigma x_i^2 - \Sigma x_i y_i \Sigma x_i)/(n\Sigma x_i^2 - (\Sigma x_i)^2);\ a = (\Sigma y_i - nb)/\Sigma x_i.$$

Здесь все суммы подразумеваются от 1 до n, где n – количество точек. Тогда функция интерполяции запишется следующим образом:

```
let interpolate (s: (float*float) seq) =
   let (sx,sy) = Seq.fold (fun (sx,sy) (x,y) -> (sx+x,sy+y)) (0.0,0.0) s
   let (sx2,sy2)=Seq.fold(fun (sx,sy) (x,y) -> (sx+x*x,sy+y*y)) (.0,.0) s
   let sxy = Seq.fold (fun s (x,y) -> s+x*y) 0.0 s
   let n = Seq.length s
   let b = (sy*sx2-sxy*sx)/(float(n)*sx2-sx*sx)
   let a = (sy-float(n)*b)/sx
   fun x -> a*x+b
```

Вначале мы при помощи свертки вычисляем сумму всех координат последовательности, их квадратов и взаимного произведения. Обратите внимание, что вычисление суммы обеих координат производится в рамках одной свертки для оптимизации алгоритма. Еще более оптимальным с точки зрения производительности было бы вычислять все 5 компонент (а точнее 4, поскольку сумма yi2 вычисляется исключительно из симметрии и дальше не используется) в одной свертке – мы этого не делали исключительно из соображений читаемости кода.

Далее вычисляется длина последовательности n, и в явном виде записывается приведенная выше формула, которая дает нам коэффициенты a и b. После этого нам остается всего лишь вернуть линейную функцию $ax+b$, которую мы порождаем при помощи функциональной константы.

Дополним решение нашей задачи и рассмотрим, как можно использовать Math.NET Numerics для интерполяции функции по точкам. Пусть у нас есть та-

блица аргумента и значения функции, как в предыдущем примере, но сами значения заданы не во всех точках, а лишь в некоторых. Наша задача – внести значения функции, полученные в результате интерполяции, в третий столбец таблицы.

Для получения последовательности пар (x_i, y_i) снова используем `Seq.unfold`:

```
let coords x = Seq.unfold
                      (fun i ->
                         let u = get x i
                         let v = get (x+1) i
                         if u=null then None
                         else Some((u:?>float,
                                if v=null then None
else Some(v:?>float)),i+1)) 20
```

В результате получается последовательность пар типа `float*float option`, поскольку значение второго аргумента определено не для всех точек. Число 20 показывает начальную строчку, начиная с которой в Excel-таблице расположена требуемая последовательность.

Далее реализуем функцию интерполяции при помощи Math.NET Numerics:

```
open MathNet.Numerics.Interpolation
let interpolate (s: (float*float) seq) =
   let xs = Seq.map fst s |> Seq.toArray
   let ys = Seq.map snd s |> Seq.toArray
   let i = MathNet.Numerics.Interpolation.Interpolate.Common (xs,ys)
   fun x -> i.Interpolate(x)
```

Как и в прошлом примере, функция возвращает интерполированную функцию типа `float->float`.

Для построения интерполированной функции нам нужно отобрать только те пары, возвращаемые функцией `coords`, в которых значение y_i определено, а также преобразовать пары к типу `float`:

```
let f = coords 0 |> Seq.filter (fun (u,v) -> Option.isSome(v))
         |> Seq.map (fun (u,Some(v))->(u,v)) |> interpolate
```

После этого для вставки значений в Excel используем обычную итерацию по последовательности:

```
coords 0 |> Seq.iteri (fun i (x,y) -> set 2 (20+i) (f x))
```

7.3. Веб-программирование

Сегодня сложно найти программную систему, которая бы не использовала в своей работе возможности сети Интернет. В этой связи веб-программирование приобретает очень важную роль. Здесь речь идет не только о создании динами-

ческих сайтов с использованием F#, но также и о системах, которые используют Интернет как средство коммуникации, предоставляя или используя веб-сервисы.

7.3.1. Доступ к веб-сервисам, XML-данным, RSS-потокам

F# хорошо подходит для обработки данных, а многие данные на сегодняшний день доступны через Интернет. Поэтому часто бывает удобным реализовывать на F# операции обработки веб-страниц, получаемых прямо из Сети.

Простейший способ получения и работы с веб-страницей, особенно в том случае, если она поддерживает стандарт XHTML, то есть является также правильным XML-файлом, является использование рассмотренных ранее инструментов работы с XML: `XmlDocument` или `XDocument`. Эти классы могут считывать документ прямо из Интернета, если в качестве имени файла указать URL, например для считывания RSS-ленты блога достаточно указать:

```
let xdoc = XDocument.Load(@"http://blogs.msdn.com/b/sos/rss.aspx")
```

После этого можно обрабатывать XML-дерево рассмотренными выше способами.

7.3.2. Доступ к текстовому содержимому веб-страниц

Во многих случаях бывает, что нам нужно работать с содержимым веб-страниц как с текстом, – это бывает необходимо, учитывая, что чаще всего страницы используют HTML, который не является правильно построенным XML. Тогда стоит использовать классы в `System.Net`, например:

```
open System.Net
open System.IO
let http (url:string) =
   let rq = WebRequest.Create(url)
   use res = rq.GetResponse()
   use rd = new StreamReader(res.GetResponseStream())
   rd.ReadToEnd()
```

Одного вызова такой функции `http` достаточно, чтобы считать содержимое веб-страницы в строку, например:

```
http "http://www.yandex.ru"
```

Попробуем построить на основе этой функции простейшего паука, который будет индексировать Интернет. Для начала используем механизм регулярных выражений для выделения из текста страницы всех ссылок:

```
open System.Text.RegularExpressions
let links txt =
  let mtch = Regex.Matches(txt,"href=\s*\"[^\"h]*(http://[^&\"]*)\"")
  [ for x in mtch -> x.Groups.[1].Value ]
```

Эта функция в ответ на содержимое страницы возвращает список содержащихся в ней URL. Для обхода Интернета воспользуемся поиском в ширину: заведем очередь URL для обхода, а также словарь `internet`, который будет хранить содержимое страниц по URL:

```
let internet = new Dictionary<string,string>()
let queue = new Queue<string>()
```

Сам алгоритм будет состоять в том, что, выбрав очередную страницу из очереди и посетив ее, мы будем добавлять все содержащиеся в ней ссылки в конец очереди:

```
let rec crawl n =
   if n>0 then
       let url = queue.Dequeue()
       if not (internet.ContainsKey(url)) then
          printf "%d. Processing %s..." n url
          let page = try http url with _ -> printfn "Error"; ""
          if page<>"" then
              internet.Add(url,page)
              let linx = page |> links
              linx |> Seq.iter(fun l -> queue.Enqueue(l))
              printf "%d bytes, %d links." page.Length (Seq.length linx)
              printfn "Done"
       crawl (n-1)
```

Функции `crawl` мы в явном виде передаем количество страниц, которые необходимо обработать. Для запуска обхода используем функцию `engine`:

```
let engine url n =
   queue.Clear()
   queue.Enqueue(url)
   crawl n
```

Мы можем очень существенно повысить эффективность работы паука, используя асинхронный доступ к веб-ресурсам. В этом случае можно использовать такую функцию для асинхронной обработки страницы и извлечения из нее ссылок:

```
let process' (url:string) =
   async {
     let! html =
```

```
    async {
       try
           let req = WebRequest.Create(url)
           use! resp = req.AsyncGetResponse()
           use reader = new StreamReader(resp.GetResponseStream())
           return reader.ReadToEnd()
       with _ -> return "" }
   return links html }
```

Для реализации самого паука используем агентный паттерн, при этом функции-обработчику будем передавать множество уже посещенных URL и счетчик, ограничивающий количество пройденных ссылок:

```
let crawler =
  MailboxProcessor.Start(fun inbox ->
    let rec loop n (inet:Set<string>) =
       async {
         if n>0 then
           let! url = inbox.Receive()
           if not (Set.contains url inet) then
             printfn "Processing %d -> %s" n url
             do Async.Start(
                        async {
                          let! links = process' url
                          for l in links do inbox.Post(l) })
           return! loop (n-1) (Set.add url inet)}
    loop 100 Set.empty)
```

В данном случае паук практически не совершает полезной работы (то есть содержимое страниц не сохраняется), и, кроме того, запросы не совершаются параллельно. Для перехода к параллельному выполнению запросов в несколько потоков можно использовать агент-диспетчер, как это было рассмотрено ранее в главе, посвященной параллельному и асинхронному программированию. Код такого решения мы в книге не приводим, отсылая заинтересованного читателя к примеру кода, опубликованному на сайте книги.

Еще один тонкий момент, который возникнет при создании реального паука, состоит в сборе данных из нескольких параллельно работающих потоков выполнения – например, если бы нам требовалось сохранять содержимое всех пройденных страниц или индексировать их. В случае агентной архитектуры для такой задачи можно выделить отдельного агента, посылая ему сообщения из разных потоков. Если же мы, вопреки традициям функционального программирования, захотим использовать глобальную изменяемую структуру данных для хранения результатов, то необходимо будет позаботиться о блокировке записи в эту структуру – либо явно при помощи семафоров или защищенных секций кода, либо используя структуру, предназначенную для параллельного доступа (thread-safe).

7.3.3. Использование веб-ориентированных программных интерфейсов на примере Bing Search API

Многие современные веб-службы, такие как Windows Live, Bing Search, twitter и др., предоставляют программные интерфейсы (API) на основе XML-протоколов. Для примера рассмотрим использование API-поисковой системы Bing.

Чтобы использовать этот API, вам необходимо зарегистрироваться и получить уникальный ключ приложения (Application ID). Используя этот ключ приложения, вы затем можете формулировать запросы в виде обыкновенных GET-запросов, получая в ответ XML-сообщения определенного вида.

Рассмотрим, как можно использовать Bing Search API для сравнения популярности нескольких терминов в Интернете. Предположим, у нас есть список ключевых слов, и мы хотим вернуть список из соответствующих количеств результатов поисковых запросов с этими терминами.

Для решения задачи в синхронном режиме мы могли бы использовать подход, описанный в разделе 7.3.1. Здесь же мы покажем, как эта задача может быть решена асинхронно.

Поиск количества результатов поискового запроса s с помощью Bing API с использованием асинхронных вычислений выглядит следующим образом:

```
let SearchCountAsync s =
    let AppID = "[Вставьте сюда Ваш AppID]"
    let url = sprintf
      "http://api.search.live.net/xml.aspx?Appid=%s&sources=web&query=%s"
       AppID s
    async {
        let req = WebRequest.Create url
        use! resp = Async.FromBeginEnd(req.BeginGetResponse,
     req.EndGetResponse)
        use stream = resp.GetResponseStream()
        let xdoc = XDocument.Load(stream)
        let webns =
          System.Xml.Linq.XNamespace.op_Implicit
              "http://schemas.microsoft.com/LiveSearch/2008/04/XML/web"
        let sx = xdoc.Descendants(webns.GetName("Total"))
        let cnt = Seq.head(sx).Value
        return Int32.Parse(cnt)
    }
```

Обратите внимание, что эта функция возвращает `Async<int>`, то есть отложенное вычисление, которое может быть вычислено асинхронно. В этой функции в начале в явном виде описывается ключ приложения AppID для Bing API – возможно, вы захотите вынести эту переменную в конфигурационный файл или отдельный раздел с константами. Далее с использованием AppID формируется по-

исковый запрос, полученный XML-поток разбирается с помощью LINQ to XML, и выбирается нужное поле – количество результатов поиска. Поскольку все окружено async-блоком, то реальный запрос происходит позднее, когда асинхронные вычисления объединяются в параллельный блок и запускаются.

Для получения искомого списка нам надо по исходному списку слов построить список async-вычислителей, объединить их в одно параллельное вычисление и запустить его:

```
let Compare L =
     L |> List.map SearchCountAsync
       |> Async.Parallel |> Async.RunSynchronously
```

Эта функция по входному списку слов возвращает искомый массив целых чисел – количеств вхождений этих слов «в Интернет».

7.3.4. Реализация веб-приложений на F# для ASP.NET Web Forms

Обратимся собственно к вопросу создания веб-приложений. В настоящее время Майкрософт предлагает две основные технологии для создания динамических приложений, работающих на стороне сервера: ASP.NET Web Forms и ASP.NET MVC.

В первом случае мы используем визуальный дизайнер Visual Studio для проектирования веб-интерфейса примерно таким же образом, как это делается при создании оконных Windows-приложений. Затем на «события», генерируемые элементами управления, навешивается программный код. Инфраструктура ASP.NET берет на себя все сложности, связанные с поддержкой событийной архитектуры приложения в веб-среде, не имеющей состояния. Такой подход сравнительно прост для начинающего программиста, однако не очень естествен для опытного разработчика и, как правило, менее эффективен.

Для визуального проектирования интерфейса ASP.NET Web Forms Visual Studio поддерживает C# и Visual Basic. Использовать для этой цели F# не рекомендуется из-за отсутствия инструментальной поддержки, хотя теоретически и возможно. Например, следующая страничка ASP.NET реализует серверные вычисления на F#:

```
<%@ Page Language="F#"  %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<script language="F#" runat="server">
      member this.prime100 =
       let rec primes = function
            [] -> []
          | h::t -> h::primes(List.filter (fun x->x%h>0) t)
       primes [2..100] |> List.fold (fun x y -> x+",¨+y.ToString()) ""
```

```
                    |> fun x-> x.Substring(1)
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
  <p>The first 100 prime numers are <%= this.prime100 %></p>
</body>
</html>
```

Для того чтобы язык F# стал доступен как скриптовый язык серверного программирования ASP.NET-страниц, необходимо добавить в файл `Web.Config` проекта следующую секцию:

```
<configuration>
…
  <system.codedom>
    <compilers>
      <compiler language="F#;f#;fs;fsharp"
      extension=".fs"
      type="Microsoft.FSharp.Compiler.CodeDom.FSharpAspNetCodeProvider,
            FSharp.Compiler.CodeDom,Version=1.9.9.9,Culture=neutral,
            PublicKeyToken=a19089b1c74d0809"/>
    </compilers>
  </system.codedom>
</configuration>
```

Однако на практике, если у вас возникает желание запрограммировать логику веб-приложения на F#, разумнее будет конструировать интерфейс на C# с использованием визуального конструктора, а весь F#-код спрятать в отдельную библиотеку, отвечающую за бизнес-логику. Если при этом вы будете проектировать и уровень доступа к данным на базе LINQ to SQL, то его вы, скорее всего, будете также реализовывать на C#, как мы рассматривали в предыдущей главе.

Рассмотрим пример веб-приложения, которое просит пользователя ввести URL сайта и возвращает список найденных в нем ссылок в виде списка (`ListBox`). В этом случае мы создадим проект ASP.NET и добавим еще один проект – F# Library. В библиотеке F# разместим следующий код, отвечающий за считывание содержимого сайта из Интернета и выделение ссылок:

```
module WebExplore
    open System.Net
    open System.IO
    let http (url:string) =
       let rq = WebRequest.Create(url)
       use res = rq.GetResponse()
       use rd = new StreamReader(res.GetResponseStream())
       rd.ReadToEnd()

    open System.Text.RegularExpressions
```

```
    let links txt =
      let mtch = Regex.Matches(txt,"href=\s*\"[^\"h]*(http://[^&\"]*)\"")
      [| for x in mtch -> x.Groups.[1].Value |]

    let explore url = http url |> links
```

Здесь мы возвращаем список ссылок в виде массива, чтобы к нему было удобнее обращаться из C#-кода.

На странице ASP.NET при помощи визуального редактора разместим поле ввода `TextBox1`, кнопку `Button1` и список `ListBox1`. На нажатие кнопки повесим следующее событие, которое очищает список, после чего вызывает функцию `Explore` F#-модуля, помещая затем результат в список:

```
protected void Button1_Click(object sender, EventArgs e)
{
      var links = WebExplore.explore(TextBox1.Text);
      ListBox1.Items.Clear();
      foreach (var s in links)
      {
          ListBox1.Items.Add(s);
      }
}
```

Чтобы модуль `WebExplore` был доступен из веб-приложения, необходимо добавить ссылку на него в разделе References проекта.

7.3.5. Реализация веб-приложений на F# для ASP.NET MVC

Другим современным подходом к созданию веб-приложений является архитектура Model-View-Controller, которая реализована в технологии ASP.NET MVC. В этом случае оказывается возможным явным образом разделить логику работы приложения (Controller), уровень модели (Model, зачастую совпадающий с моделью данных приложения) и представление (View). Мы не будем подробно вдаваться в устройство приложений ASP.NET MVC, отсылая читателя к книге [15]. В ASP.NET MVC компоненту View по-прежнему удобнее реализовывать на ASP.NET и C#, в то время как модель данных и бизнес-логика могут успешно программироваться на F#.

Для приложения ASP.NET MVC на базе F# уже разработан готовый шаблон «F# and C# Web App (ASP.NET, MVC2)», который находится в галерее онлайн-шаблонов Visual Studio 2010. Его также можно скачать по адресу http://visualstudiogallery.msdn.microsoft.com/en-us/c36619e5-0d4a-4067-8ced-decd18e834c9?SRC=VSIDE.

Для того чтобы реализовать вариант рассмотренного выше приложения на MVC, необходимо сделать следующее:

- добавить такой же, как и выше, модуль `WebExplore` в проект и установить ссылку на него из модуля `Controllers`;

- для того чтобы передавать внутрь приложения адрес Url, введенный пользователем, нам потребуется простейшая модель данных, которая оформляется в виде отдельного класса в проекте Models:

```
namespace FSharpMVC2.Web.Models
open System.ComponentModel

type InputModel() =
    let mutable url=""
    [<DisplayName("Enter Url:")>]
    member x.Url with get()=url and set(v) = url <- v
```

- наша главная страничка view Index.aspx будет основана на этой модели, что явно указывается в заголовке страницы. Также на этой странице динамически генерируется поле ввода для заполнения модели (то есть ввода пользователем Url) и выводятся результаты, содержащиеся в данных View (ViewData), если они есть. Страничка для удобства приводится в слегка сокращенном виде:

```
<%@ Page Language="C#"
    MasterPageFile="~/Views/Shared/Site.Master"
 Inherits="System.Web.Mvc.ViewPage<FSharpMVC2.Web.Models.InputModel>" %>

<asp:Content ID="TheContent"
             ContentPlaceHolderID="MainContent" runat="server">
    <% using (Html.BeginForm()){ %>
    <%: Html.TextBoxFor(model => model.Url) %>
    <% } %>
    <ul>
        <% if (ViewData["Urls"] != null) {
               foreach (var x in (string[])ViewData["Urls"]){ %>
           <li><%=x%></li>
        <% }} %>
    </ul>
</asp:Content>
```

Здесь мы предполагаем, что результат работы, то есть список встречающихся в страничке ссылок, содержится в поле Urls объекта ViewData в виде массива строк. Для их отображения нам даже приходится использовать явное приведение типов;

- основная логика работы программируется в контроллере HomeController, который может отвечать сразу за несколько представлений View:

```
namespace FSharpMVC2.Web.Controllers
open FSharpMVC2.Web.Models
open System.Web
open System.Web.Mvc

[<HandleError>]
type HomeController() =
```

```
    inherit Controller()
    member x.Index (m:InputModel) : ActionResult =
        x.ViewData.["Urls"] <- WebExplore.explore m.Url
        x.View() :> ActionResult
    member x.About () =
        x.View() :> ActionResult
```

За представление `Index.aspx` в данном случае отвечает метод `index`, и его задача состоит в том, чтобы, получив на вход модель данных от пользователя (в данном случае состоящую только из одной Url), выполнить необходимые действия, вернуть результат в виде набора данных `ViewData` и указать представление, которое будет выдано пользователю;

- за отображение URL на конкретные контроллеры и странички в MVC-приложении отвечает специальный файл `Global.fs`, в нашем случае также реализованный на F#.

Как читатель мог заметить, F# оказывается весьма удобным для реализации логики в приложениях ASP.NET MVC. Однако для реализации модели данных во многих случаях удобнее использовать C#, поскольку он поддерживает автоматическое создание LINQ-to-SQL-объектов по базе данных.

7.3.6. Реализация веб-приложений на F# при помощи системы WebSharper

Благодаря особенностям языка F#, таким как развитые средства метапрограммирования, на нем стало возможно воплотить в жизнь интересную среду веб-программирования WebSharper, в которой код веб-приложения – включая клиенские и серверные составляющие – реализуется на одном языке F#. При этом для выполнения на клиенте внутри веб-браузера код на F# прозрачным для пользователя образом транслируется в JavaScript. Помимо этого, WebSharper включает в себя своего рода DSL для программирования HTML-разметки страницы внутри того же кода на F#.

WebSharper – это свободно распространяемая среда от IntelliFactory, которую можно скачать с http://www.intellifactory.com или http://www.websharper.com. После установки станет доступен шаблон веб-приложения WebSharper для Visual Studio, содержащий достаточно исчерпывающий пример реализации клиент-серверного приложения ввода данных. Мы здесь приведем несколько более простых фрагментов кода, чтобы вам было проще разобраться в среде WebSharper и в дальнейшем использовать ее в своих проектах.

WebSharper проще всего использовать совместно с ASP.NET Web Forms. В нашем случае (так же как и в случае создания приложения из шаблона) приложение будет содержать проект ASP.NET, включающий в себя сами веб-страницы, и проект с кодом на F#.

ASP.NET-страница может содержать в себе дизайн, мастер-страницы и т. д., однако основная логика добавляется в виде элементов управления, реализованных внутри F#-кода. Вот несколько упрощенный пример страницы:

```
<%@ Page Language="C#" %>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head runat="server">
    <WebSharper:ScriptManager runat="server" />
  </head>
  <body>
    <ws:MainPage runat="server"/>
  </body>
</html>
```

Основная логика приложения обеспечивается элементом управления MainPage, внешний вид и поведение которого описываются внутри отдельного F#-проекта. В качестве примера рассмотрим все то же приложение, вычисляющее простые числа:

```
[<JavaScriptType>]
type MainPage() =
    inherit Web.Control()
    [<JavaScript>]
    override this.Body =
       let result = P [Text ""]
       Div [
          P [Text "Press to retrieve data"]
          Input [Type "Button"; Value "Get Data"]
          |>! OnClick (fun e a -> result.Text <- Application.GetPrimes())
            result ]
```

Здесь мы определяем элемент управления MainPage, в котором за отображение отвечает перегруженный метод Body. Для формирования HTML-представления элемента мы используем проблемно-ориентированный язык (DSL), в котором используются вложенные функции, одноименные HTML-тегам. В рамках этого же языка мы можем навешивать обработчики событий, которые будут транслироваться в JavaScript и работать на клиенте внутри браузера. Обратите внимание, что весь элемент описывается с атрибутом <JavaScript>, то есть будет отрабатываться на клиенте.

Чтобы при нажатии кнопки на странице появлялись простые числа, необходимо отдельно определить функцию GetPrimes:

```
module Application =
    [<JavaScript>]
    let rec primes = function
              [] -> []
            | h::t -> h::primes(List.filter (fun x -> x%h>0) t)
    [<JavaScript>]
    let GetPrimes() = primes [2..100]
                      |> List.fold (fun s i -> s+" "+i.ToString()) ""
```

Здесь все функции помечены атрибутом <JavaScript>, что означает, что они также будут работать на клиентском компьютере, и вызов функции из события OnClick будет представлять собой обычный Javascript-вызов. Для этого WebSharper реализует трансляцию подмножества F#-кода в Javascript.

Мы также можем реализовать вычисление простых чисел на стороне сервера, используя технологию AJAX. В этом случае нам всего лишь придется пометить соответствующие функции вычисления простых чисел другим тегом <Rpc> – все остальное среда WebSharper сделает за нас, включая генерацию заглушки функции на стороне клиента и JSON веб-сервиса на сервере, сериализацию и десериализацию аргументов функции и результата и т. д. Код с вычислением на сервере будет иметь вид:

```
module Application =
    let rec primes = function
              [] -> []
            | h::t -> h::primes(List.filter (fun x -> x%h>0) t)
    [<Rpc>]
    let GetPrimes() = primes [2..100]
                      |> List.fold (fun s i -> s+" "+i.ToString()) ""
```

Помимо простой инфраструктуры трансляции F#-кода в JavaScript и возможности прозрачного AJAX-вызова, WebSharper содержит библиотеку, упрощающую построение весьма сложных клиент-серверных веб-приложений. Основными компонентами приложения являются:

- пейджлеты (pagelets) – элементы управления, представляющие собой фрагменты кода, работающего на клиенте, как в предыдущем примере;
- формлеты (formlets) – это компактные пейджлеты, отвечающие за ввод определенного набора данных формы. Формлеты привязаны к компактным типам данных из модели данных предметной области, при этом они могут быть зависимыми друг от друга, могут расширяться дополнительными свойствами типа проверки вводимых данных (на стороне клиента) и т. д.;
- флоулеты (flowlets) – позволяют определять порядок показа последовательности формлетов пользователю.

Предположим, мы хотим запрашивать у пользователя правую границу простых чисел для вычисления. В этом случае за форму ввода будет отвечать формлет типа Formlet<int>, для формирования которого мы используем монадическое formlet-выражение. Запускается формлет при помощи функции Run, которая принимает обработчик данных формы (в данном случае функцию int->unit) и возвращает DOM-иерархию формы, которую можно вставить в тело нашего основного элемента управления MainPage:

```
[<JavaScript>]
override this.Body =
            let res = P [Text ""]
```

```
Div [
  formlet {
    let! max = Controls.Input "100"
                |> Validator.IsInt "Must be int"
    return max |> int }
  |> Enhance.WithTextLabel "Enter max number:"
  |> Enhance.WithValidationIcon
  |> Enhance.WithSubmitAndResetButtons
  |> Enhance.WithFormContainer
  |>Formlet.Run(fun n-> res.Text<-Application.GetPrimes(n))
  res ]
```

В данном примере декларативным образом задается множество функционала – валидатор целочисленного поля ввода, наличие кнопок Submit/Reset в форме, использование визуального контейнера формы и т. д.

Для формирования формлетов может использоваться и другой синтаксис, отличный от монадического. В качестве примера вернемся к рассмотренному нами ранее примеру приложения, возвращающего все найденные в указанной веб-странице ссылки. Пусть в качестве входных данных мы запрашиваем у пользователя URL и количество ссылок, которые необходимо вернуть. Вся логика приложения может быть описана следующим образом (предполагая, что метод `WebExplore.explore` реализован в виде `<Rpc>`-метода и возвращает массив строк):

```
[<JavaScript>]
override this.Body =
    let res = Div []
    Div [
        Formlet.Yield(fun s n -> s,n|>int)
          <*> (Controls.Input "http://www.yandex.ru"
              |> Validator.IsNotEmpty "Must be entered"
              |> Enhance.WithTextLabel "Enter URL:")
          <*> (Controls.Input "100"
              |> Validator.IsInt "Must be int"
              |> Enhance.WithTextLabel "Enter max number:")
          |> Enhance.WithValidationIcon
          |> Enhance.WithSubmitAndResetButtons
          |> Enhance.WithFormContainer
          |> Formlet.Run(fun (s,n) ->
                   res.Append(UL[
                     for i in (WebExplore.explore s n) -> LI[Text i]
                   ]))
        res ]
```

Этот пример, помимо нового синтаксиса определения формлетов и демонстрации использования сложных формлетов с несколькими элементами данных, также показывает способ динамического создания фрагментов DOM-дерева «на лету». Из списка строк, возвращенных функцией `explore`, мы в цикле формируем

список <UL>, который затем при помощи метода Append добавляем к существующему элементу DOM-модели – изначально пустому тегу .

7.3.7. Облачное программирование на F# для Windows Azure

При размещении современных сервисов и приложений в Интернете, особенно в том случае, если требуются высокая надежность и масштабируемость, имеет смысл задуматься об использовании облачных технологий. Выбирая платформу Майкрософт для разработки, вы получаете в свое распоряжение модель облачного программирования, которая позволяет вам использовать привычные подходы для разработки веб-приложений и сервисов (ASP.NET, ASP.NET MVC и др.). Разработанные таким образом сервисы работают в облаке под управлением «облачной операционной системы» Windows Azure.

В данной книге мы не ставим задачу охватить все аспекты облачного программирования – однако важно отметить, что такое программирование почти всегда связано с решением задач с высокой нагрузкой, в том числе задач обработки данных, поэтому использование F# для облачных вычислений более чем оправдано. Именно поэтому в стандартной поставке Windows Azure SDK предусмотрено использование F# для облачных проектов обработки данных.

Говоря слегка упрощенно, Windows Azure предоставляет программисту два вида ресурсов: **вычислительные ресурсы** (compute) и услуги по **хранению данных** (storage). Вычислительные ресурсы могут быть двух типов: веб-роли (web role) и роли-обработчики (worker role).

Веб-роли – это либо веб-приложения (в смысле ASP.NET Web Forms или MVC – то есть приложения, имеющие пользовательский интерфейс и отвечающие на HTTP-запросы от пользователей), либо веб-сервисы (с программным интерфейсом типа SOAP, REST или WCF). Поскольку веб-роли чаще всего используют визуальный режим программирования, принятый в ASP.NET, для них предусматривается использование C# как базового языка программирования.

Роли-обработчики представляют собой процессы, которые работают в облаке и не имеют явного программного интерфейса с пользователем или с другими системами. Такие роли чаще всего используются для массированной обработки данных, происходящей в фоновом режиме. Именно для таких ролей, помимо C#, предлагается использовать также и F# – соответствующий шаблон есть в Windows Azure SDK.

Чтобы лучше понять смысл ролей, представим себе поисковую систему, размещенную в облаке. Такая система, скорее всего, будет использовать роли-обработчики для реализации паука, обходящего сеть и индексирующего страницы, и веб-роли для пользовательского интерфейса. В зависимости от нагрузки мы сможем изменять количество экземпляров каждой из ролей, которые будут запущены в облачной инфраструктуре для обеспечения ответов на запросы, тем самым гибко распределяя ресурсы по требованию.

Веб-роли и роли-обработчики обычно общаются между собой с помощью хранилищ данных: очередей, таблиц или blob-хранилищ. Например, упомянутая выше поисковая система, скорее всего, будет строить индекс в масштабируемом хранилище в виде таблиц[1], а отложенные на обработку URL вновь обнаруженных ссылок помещать в очередь.

Мы рассмотрим упрощенный пример использования Windows Azure для параллельного интегрирования функции. Мы приводим здесь лишь фрагменты кода, целиком пример содержится в примерах кода в данной книге.

В нашем случае мы будем использовать одну веб-роль для организации интерфейса с пользователем и одну роль-обработчик для вычислений. Роль-обработчик будет обрабатывать сообщения следующих двух типов из входной очереди:

- I:<левая граница>:<правая граница> – необходимость распределить работу по вычислению интеграла функции в указанных границах путем распределения работы между n другими агентами;
- i:<левая граница>:<правая граница> – необходимость непосредственно вычислить интеграл и поместить результат в выходную очередь.

Таким образом, для обработки сообщений мы можем использовать функцию такого вида:

```
let ProcessMessage (s:string) =
    let a = s.Split(':')
    let l = Double.Parse(a.[1])
    let r = Double.Parse(a.[2])
    match a.[0] with
      "I" ->
         let h = (r-l)/float(num)
         for i in 1..num do
            let l1 = l+float(i-1)*h
            let r1 = l1+h
            let s = "i:"+l1.ToString()+":"+r1.ToString()
            queue_in.AddMessage(new CloudQueueMessage(s))
    | "i" ->
         log ("Integrating "+s) "Information"
         let res = integrate func l r
         queue_out.AddMessage(
            new CloudQueueMessage("r:"+res.ToString())
```

Роль-обработчик, по сути дела, содержит одну важную функцию – `Run`, которая запускается при выполнении роли и работает до выполнения всех заданий (в нашем случае, как и во многих других сценариях, до бесконечности, ожидая все новых сообщений во входной очереди для обработки):

[1] Которые, строго говоря, не являются реляционными таблицами, поскольку в таблицах Windows Azure не обеспечиваются соответствие данных схеме и поддержка контроля целостности. Однако мы не будем вдаваться в подробности.

```
type Worker() =
    inherit RoleEntryPoint()

    let log message kind = Trace.WriteLine(message, kind)

    let mutable queue_in : CloudQueue = null
    let mutable queue_out : CloudQueue = null
    let mutable num = 10
    let mutable num_slices = 5000
    let mutable func = fun x -> sqrt(x)

    override wr.Run() =
        log "Starting computation agent..." "Information"
        while(true) do
            let msg = queue_in.GetMessage()
            if msg=null
                then Thread.Sleep(1000)
                else
                    ProcessMessage(msg.AsString)
                    queue_in.DeleteMessage(msg)
            log "Processing queue" "Information"
```

Другой метод отвечает за начальную инициализацию сервиса и получение ссылок на соответствующие структуры данных Azure Storage (которые в случае необходимости создаются):

```
    override wr.OnStart() =
    ...
        let storageAccount =
    CloudStorageAccount.FromConfigurationSetting("DataConnectionString")
        let cloudQueueClient = storageAccount.CreateCloudQueueClient()
        queue_in <- cloudQueueClient.GetQueueReference("queue-in")
        queue_out <- cloudQueueClient.GetQueueReference("queue-out")
        queue_in.CreateIfNotExist() |> ignore
        queue_out.CreateIfNotExist() |> ignore
        base.OnStart()
```

Веб-роль представляет собой почти обычное ASP.NET-приложение, которое в нашем случае реализовано на C#. Мы здесь не будем его подробно рассматривать.

7.4. Визуализация и работа с графикой

Часто в научных задачах возникает потребность визуализировать полученные результаты. В то время как F# оказывается удобным языком для обработки данных, построение интерфейсов на F# может оказаться менее удобным занятием.

Однако в этом разделе мы рассмотрим несколько подходов, которые могут использоваться для эффективной визуализации данных.

7.4.1. Двухмерная графика на основе Windows Forms API

На самом деле мы уже рассматривали соответствующий пример ранее, когда строили в окошке множество Мандельброта. По сути дела, платформа .NET дает нам в распоряжение набор классов для попиксельного манипулирования двухмерными графическими объектами с помощью класса `Bitmap`, что позволяет нам легко реализовывать на F# как интерактивные программы визуализации (выводящие графики или изображения в форму на дисплее), так и пакетные программы, порождающие множество результатов визуализации в виде набора графических файлов.

В качестве примера рассмотрим обработку изображений – это позволит нам посмотреть не только то, как записывать (генерировать) изображения, но и как получать доступ к пискелям исходного изображения. Для примера реализуем утилиту для преобразования произвольного JPEG-изображения к полутоновому (черно-белому).

Для начала опишем функцию, которая будет производить произвольную попиксельную обработку изображения. Напомним, что объект `Bitmap` может считывать изображения в различных форматах с диска и предоставляет доступ к цветовой информации отдельных пикселей с помощью методов `GetPixel/SetPixel`:

```
let pixprocess func fin fout =
   let bin = new Bitmap(fin:string)
   let bout = new Bitmap(width=bin.Width,height=bin.Height)
   for i in 0..(bin.Width-1) do
      for j in 0..(bin.Height-1) do
        bout.SetPixel(i,j,bin.GetPixel(i,j)|>func)
   bout.Save fout
```

Здесь мы рассматриваем простейшую попиксельную обработку. Для достижения лучшей функциональности мы могли бы обрабатывать изображение целыми фрагментами с использованием методов `LockBits/UnlockBits`, однако в нашем случае мы скорее преследуем наглядность, нежели производительность.

Далее опишем функцию, преобразующую цвет к градиенту серого:

```
let uncolor (c:Color) =
    let x = (int)(0.3*c.R + 0.59*c.G + 0.11*c.B)
    Color.FromArgb(x,x,x)
```

После этого сама утилита может быть записана следующим образом:

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

```
  Console.WriteLine("Image Uncolorifier")
  if args.Length<>2 then
    Console.WriteLine("Format: uncolor <input file> <output file>"); 1
  else pixprocess uncolor args.[0] args.[1]; 0
```

Стоит обратить внимание на то, как реализована работа с командной строкой. Помеченная атрибутом [<EntryPoint>] функция в качестве аргументов принимает на вход массив параметров, передаваемых приложению. Мы рассматриваем случай, когда передаются только два параметра: входное и выходное имена файла, над которыми надо осуществить преобразование. В этом случае мы вызываем функцию для осуществления преобразования и возвращаем код завершения 0 (что рассматривается ОС Windows как признак удачного завершения программы). Если число аргументов не равно 2, то мы печатаем сообщение и завершаемся с кодом ошибки 1.

Здесь мы не рассматриваем случай, когда пользователь указал несуществующее имя входного файла, или выходной файл не может быть открыт на запись, или же формат исходного изображения не поддерживается. Во всех этих случаях функция pixprocess сгенерирует исключение, которое приведет к исключению в программе. Более правильно было бы обработать исключение в нашей утилите следующим образом:

```
[<EntryPoint>]
let main(args: string array) =
  Console.WriteLine("Image Uncolorifier")
  if args.Length<>2 then
    Console.WriteLine("Format: uncolor <input file> <output file>"); 1
  else
    try
      pixprocess uncolor args.[0] args.[1]; 0
    with e -> Console.WriteLine("Error: "+e.Message); 2
```

Читатель, наверное, уже догадался, что на основе реализованной функции легко построить утилиту для пакетной обработки множества изображений в асинхронном режиме – на самом деле такой пример (без детализации алгоритма обработки изображений) уже рассматривался нами ранее в разделе 6.2.

7.4.2. Использование элемента Chart

Достаточно богатым средством отображения данных могут стать различные диаграммы. Много задач по анализу и отображению успешно решаются в Excel при помощи элемента Chart – аналогичный подход можно применить и для визуализации в F#. Конечно, одним из подходов мог бы быть экспорт данных в Excel с последующей визуализацией – однако здесь мы покажем, как использовать всю мощь Excel прямо из F#.

В версиях .NET 3.5 и более ранних для отображения диаграмм приходилось использовать различные сторонние элементы управления. Начиная с версии

.NET 4.0, в состав .NET Framework входит элемент Chart, обладающий мощными средствами двухмерной и псевдотрехмерной визуализации. Причем этот элемент может использоваться как в приложениях на основе Windows Forms, так и в веб-приложениях, на ходу порождая необходимые изображения для отображения на сайте.

Рассмотрим, как можно использовать элемент Chart для визуализации данных в форме[1]. Для начала необходимо подключить соответствующие библиотеки и DLL:

```
open System.Drawing
open System.Windows.Forms
#r "System.Windows.Forms.DataVisualization"
open System.Windows.Forms.DataVisualization.Charting
```

После этого опишем класс для создания окна визуализации и помещения внутрь элемента Chart:

```
type Visualiser(title, style, init_fun:Chart->unit) =
  inherit Form( Text=title )
  let chart = new Chart(Dock=DockStyle.Fill)
  let area = new ChartArea(Name=title)
  let series = new Series()
  do series.ChartType <- style
  do series.ChartArea <- title
  do chart.Series.Add(series)
  do chart.ChartAreas.Add(area)
  do init_fun chart
  do base.Controls.Add(chart)
```

Этот класс унаследован от `Form`, то есть представляет собой форму, которую можно в нужный момент показать на экране. В качестве конструктора мы передаем название окна, тип диаграммы (линии, точки и т. д. – в классе `SeriesChartType` определено множество различных типов), а также функцию для инициализации диаграммы, которая должна добавить необходимые для отображения точки во внутреннее поле элемента `Chart` типа `Series`.

Для отображения на экране простой последовательности точек определим функцию

```
let SeqVisualiser title style (xs: float seq) =
  new Visualiser(title,style,
     fun chart -> xs |> Seq.iter (chart.Series.[0].Points.Add >> ignore))
```

Эта функция берет последовательность чисел типа `float` и добавляет их на диаграмму – значения по оси X генерируются автоматически последовательно.

[1] Приведенный ниже код частично заимствован у Джеймса Хаггарда, см. http://stackoverflow.com/questions/3276357/how-do-i-plot-a-data-series-in-f.

Например, такую функцию можно использовать для отображения графика функции (рис. 7.5*а*) или построения множества каких-либо финансовых показателей (рис. 7.5*б*) следующим образом:

```
let V1 = SeqVisualiser "Data" (SeriesChartType.Line) (seq {-6.0..0.01..6.0}|> Seq.map
sin)
V1.Show()

let V2 = SeqVisualiser "Data" SeriesChartType.Bar
           (seq { let R = new System.Random()
                  for i in [1..10] -> R.NextDouble()*3.0 })
V2.Show()
```

а *б*

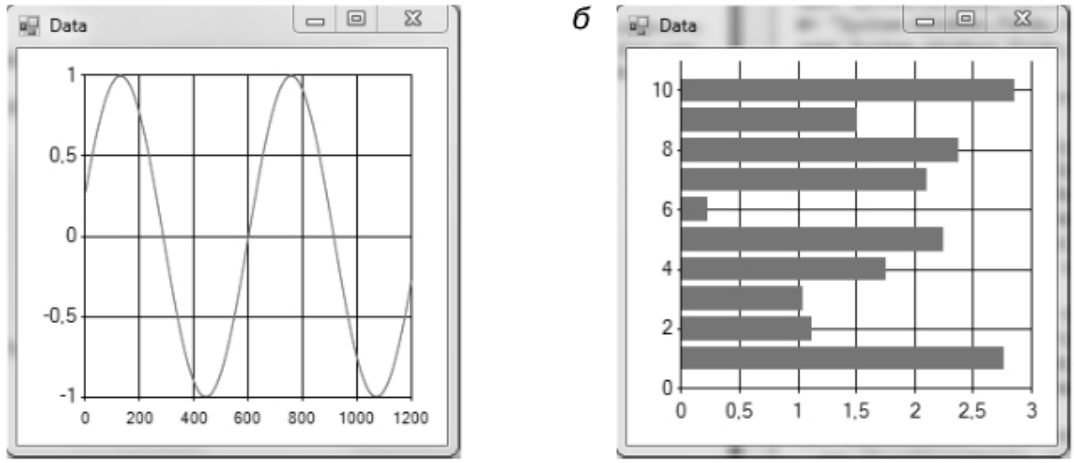

Рис. 7.5. Графическое отображение последовательностей в Chart Control

Иногда нам может понадобиться строить 2D-графики, в которых заданы координаты точек (x,y). Для такого случая мы можем описать функцию, которая принимает на вход последовательность пар точек:

```
let PointVisualiser title style xs =
  new Visualiser(title,style,
    fun chart ->
      xs |>
  Seq.iter (fun (x,y) -> chart.Series.[0].Points.AddXY(x,y)|>ignore))

(PointVisualiser "Data" SeriesChartType.Bubble
           (seq { let R = new System.Random()
                  for i in [1..10] ->
                   (R.NextDouble()*3.0,R.NextDouble()*3.0) })).Show()
```

Полученный в результате график приведен на рис. 7.6*а*. Также с помощью более изощренных типов диаграмм мы можем строить красивые псевдотрехмерные диаграммы. Например, попробуем реализовать приложение для отслеживания популярности операционных систем по количеству вхождений их названий на

сайт http://osys.ru. Для этого определим функцию count по подсчету количества вхождений на основе регулярных выражений и получим список из названий операционных систем и количества вхождений названия в текст страницы:

```
let page = http "http://osys.ru/"
let count s page = Regex.Matches(page,s).Count
let os = ["Windows";"UNIX";"Linux";"DOS"]
          |> Seq.map (fun s -> (s,float(count s page)))
```

Здесь мы используем описанную нами ранее функцию http для считывания страницы из Интернета. Для отображения определим функцию построения круговой диаграммы:

```
let LabelVisualiser3D title style xs =
  new Visualiser(title,style,
   fun chart ->
       chart.ChartAreas.[0].Area3DStyle.Enable3D <- true
       chart.ChartAreas.[0].Area3DStyle.Perspective <- 10
       xs |> Seq.iter (fun (y:string,x:float) ->
                         let pt = chart.Series.[0].Points.Add(x)
                         pt.Label <- y))

(LabelVisualiser3D "OS" SeriesChartType.Pie os).Show()
```

Результат выполнения программы вы можете видеть на рис. 7.6*б*.

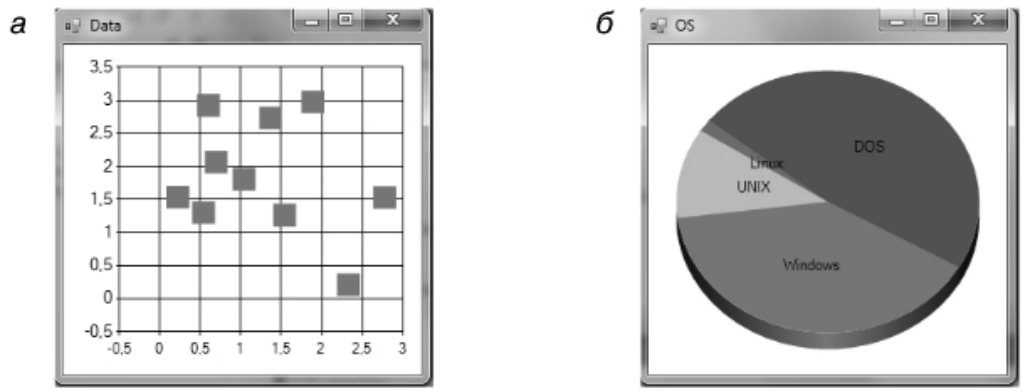

Рис. 7.6. Более сложные типы визуализации в Chart Control

Помимо возможностей визуализации, рассмотренный пример демонстрирует типичный для функционального программирования паттерн проектирования. Вместо того чтобы наследовать от исходного класса Visualiser различные типы диаграмм, приводя к появлению все более сложных специализированных классов, мы используем экземпляры классов, передавая им соответствующие функции инициализации. В зависимости от различных функций инициализации мы можем получать объекты (экземпляры класса) с очень различным поведением – причем весьма просто, без необходимости перегрузки методов и создания большого коли-

чества классов в иерархии. Похожим образом устроены многие функции и объекты стандартной библиотеки классов F#.

7.4.3. 3D-визуализация с помощью DirectX и/или XNA

В заключение рассмотрим задачу визуализации трехмерных объектов или данных, которая часто встречается в научных задачах. Сразу заметим, что в практических задачах часто эффктивнее оказывается использовать сторонние визуализаторы данных (например, gnuplot), а F# использовать для обработки. Однако за счет того, что F# является языком на платформе .NET, в которой есть необходимые средства для работы с 3D-графикой, задача написания своего визуализатора также решается весьма просто.

Для трехмерной графики на платформе .NET существуют два сравнительно схожих подхода – использование библиотеки Managed DirectX или XNA. Первая библиотека представляет собой простую управляемую обертку над графическими средствами DirectX, в то время как XNA является полноценной средой для построения игр со своей программной моделью в виде игрового цикла, основанной на DirectX (и во многом унаследованной от Managed DirectX).

Мы здесь рассмотрим использование XNA для трехмерной визуализации. Использование Managed DirectX несильно отличается от XNA – хороший пример такого использования для построения трехмерного графика функции и моделирования движения шариков по поверхности графика содержится в стандартных демонстрационных примерах F#, которые можно скачать с сайта MSDN по адресу http://archive.msdn.microsoft.com/fsharpsamples.

По умолчанию Visual Studio предлагает нам создать XNA-проект только на основе C#. Мы можем выбрать такой подход, реализовав на C# логику управления визуализацией (например, вращение камеры, увеличение и т. д.), а F# использовать для генерации исходной трехмерной модели. Мы же рассмотрим еще более радикальный подход и реализуем игру целиком на F#.

Для этого начнем создавать обычный Windows-проект на F# и подключим к нему дополнительно библиотеки `Microsoft.Xna.Framework`, `Microsoft.Xna.Framework.Graphics` и `Microsoft.Xna.Framework.Game`. Структура игры в XNA имеет следующий вид:

```
module XnaDemo

open System
open System.Drawing
open System.Windows.Forms
open Microsoft.Xna.Framework
open Microsoft.Xna.Framework.Graphics

type FGame() as self =
    inherit Game()

    let graphics = new GraphicsDeviceManager(self)
```

```
    override this.Initialize() =
        // код для инициализации игры
        base.Initialize()

    override this.Update(gameTime : GameTime) =
        // код для обновления состояния игры в каждом цикле игры
        base.Update gameTime

    override this.Draw(gameTime : GameTime) =
        // код для отрисовки очередного цикла игры

let fgame = new FGame()
fgame.Run()
```

Таким образом, XNA-приложение состоит из быстро повторяющегося «цикла игры», заключающегося в попеременном вызове методов `Update` и `Draw`.

В качестве примера рассмотрим приложение, строящее на экране вращающуюся пирамиду. Большинство трехмерных фигур в DirectX строятся из множества треугольников. В случае с пирамидой ее разбиение на треугольники очевидно (три грани и основание), позже мы увидим, как более сложные фигуры также могут быть триангулированы. Пусть пирамида состоит из трех вершин, расположенных в основании пирамиды на окружности радиусом R, и еще одной вершины, расположенной на расстоянии R от основания. Тогда координаты вершин пирамиды мы сможем вычислить следующим образом:

```
let n = 3
let CreateVertices() =
  let phi = MathHelper.TwoPi / float32(n)
  let R = 10.f
  new VertexPositionColor(new Vector3(0.f,0.f,R),Color.Blue) ::
   [ for i in 0..(n-1) ->
          new VertexPositionColor(
             new Vector3(R*sin(phi*float32(i)),
                         R*cos(phi*float32(i)), 0.0f), Color.Yellow)]
```

Каждая вершина здесь задается объектом `VertexPositionColor`, содержащим трехмерный вектор вершины и ее цвет. Вершина пирамиды сделана голубой, а точки в основании – желтыми. Заметим также, что в XNA в качестве базового типа для компонент вектора используется тип `float32`, что приводит к необходимости часто использовать операции преобразования типов.

Для построения пирамиды нам также понадобится набор индексов, указывающих, какие треугольники (из каких вершин, задаваемых по номерам) надо строить. Такое хранение данных (отдельно вершины и наборы индексов) сделано для того, чтобы минимизировать количество пересылаемых в видеокарту данных и исключить дублирование одинаковых вершин.

В нашем случае для создания массива индексов можно записать такую функцию:

```
let CreateIndices() =
  [ for i in 1..n do
     let nx = if i<n then i+1 else 1
     yield! [(int16)0,(int16)I,(int16)nx]]
  @ [ (int16)1;(int16)2;(int16)3 ]
```

Здесь сначала для каждой точки основания определяется треугольник, ведущий из вершины к этой и следующей точке (для последней точки «следующей» считается первая), а затем вручную добавляется треугольник основания. Каждый треугольник описывается последовательностью из трех целых чисел типа `int16` (отсюда снова необходимость преобразования типов!).

Построение фигур – это дело графического адаптера, который программируется с помощью так называемых эффектов. К счастью, XNA уже содержит множество стандартных эффектов, и нам для работы потребуется лишь создать экземпляр соответствующего класса, что мы сделаем вместе с некоторыми другими операциями в методе инициализации, предварительно описав несколько вспомогательных переменных:

```
    let graphics = new GraphicsDeviceManager(self)
    let cameraPosition = new Vector3(0.f, -20.f, 20.f)
    let mutable effect = null

    let mutable Vertices = CreateVertices()
    let mutable Indices = CreateIndices()
    let mutable rot = 0.f

override this.Initialize() =
        let rasterizerState = new RasterizerState()
        rasterizerState.CullMode <- CullMode.None
        self.GraphicsDevice.RasterizerState <- rasterizerState
        effect <- new BasicEffect(self.GraphicsDevice)
        effect.VertexColorEnabled <- true
        self.setUpMatrices()
        base.Initialize()
```

Здесь вектор `CameraPosition` определяет положение камеры, которая будет смотреть на пирамиду, переменная rot – текущий угол поворота пирамиды. Этот угол поворота мы будем медленно менять в методе `Update`, чтобы пирамида поворачивалась:

```
    override this.Update(gameTime : GameTime) =
        rot <- rot + 0.01f
        self.setUpMatrices()
        base.Update gameTime
```

Основной интерес представляет метод отрисовки `Draw`, который и делает всю работу по рисованию пирамиды:

```
override this.Draw(gameTime : GameTime) =
    self.GraphicsDevice.Clear(Color.Gray)
    for pass in effect.CurrentTechnique.Passes do
        pass.Apply()
        this.GraphicsDevice.DrawUserIndexedPrimitives(
          PrimitiveType.TriangleList,
          Array.ofList(Vertices),
          0, Vertices.Length,Array.ofList(Indices),
          0, Indices.Length / 3)
    base.Draw gameTime
```

Каждый эффект может состоять из нескольких «проходов»[1], поэтому нам нужно в явном виде перебрать эти проходы в цикле и для каждого прохода вызвать методы рисования. В нашем случае пирамида рисуется одним вызовом функции `DrawUserIndexedPrimitives`, которой мы передаем построенные ранее массивы вершин и индексов.

Обычно массивы вершин и индексов объединяют вместе в каркасы (Mesh), а наборы каркасов – в модели (Model). XNA определяет метод для более простого рисования модели, поэтому альтернативным подходом было бы формирование целого объекта типа `Model` с последующей отрисовкой его. Особенно полезным такой подход окажется в том случае, если F# используется лишь для формирования модели, а вся отрисовка делается на C# – тогда нам достаточно вернуть из F#-библиотеки экземпляр модели, который будет содержать в себе всю необходимую информацию для отображения.

Осталось разобраться лишь с методом `setUpMatrices`:

```
    member this.setUpMatrices() =
        effect.World <-
            Matrix.CreateFromAxisAngle(new Vector3(0.f,0.f,1.0f),rot)
        effect.View <-
            Matrix.CreateLookAt(cameraPosition, Vector3.Zero, Vector3.Up)
        effect.Projection <-
            Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
               self.GraphicsDevice.DisplayMode.AspectRatio, 1.f, 100.f)
```

Такое количество матриц нужно для того, чтобы преобразовать трехмерную модель для отображения на двухмерной поверхности экрана. Матрица `View` определяет направление взгляда камеры и вычисляется на основе вектора позиции камеры и того, куда камера направлена. Матрица `Projection` определяет используемую проекцию – в нашем случае мы используем перспективную проекцию с углом обзора 90°. Третья матрица `World` определяет дополнительное преобразование, которое применяется к модели перед отображением – в качестве такого преобразования мы задаем поворот на угол `rot` вокруг вертикального вектора, чтобы обеспечить вращение пирамиды.

[1] Здесь также хитрым образом вычисляется цвет вершины – мы предоставляем читателю приятную возможность самостоятельно разобраться, как это делается.

Получившийся результат можно посмотреть на рис. 7.7.

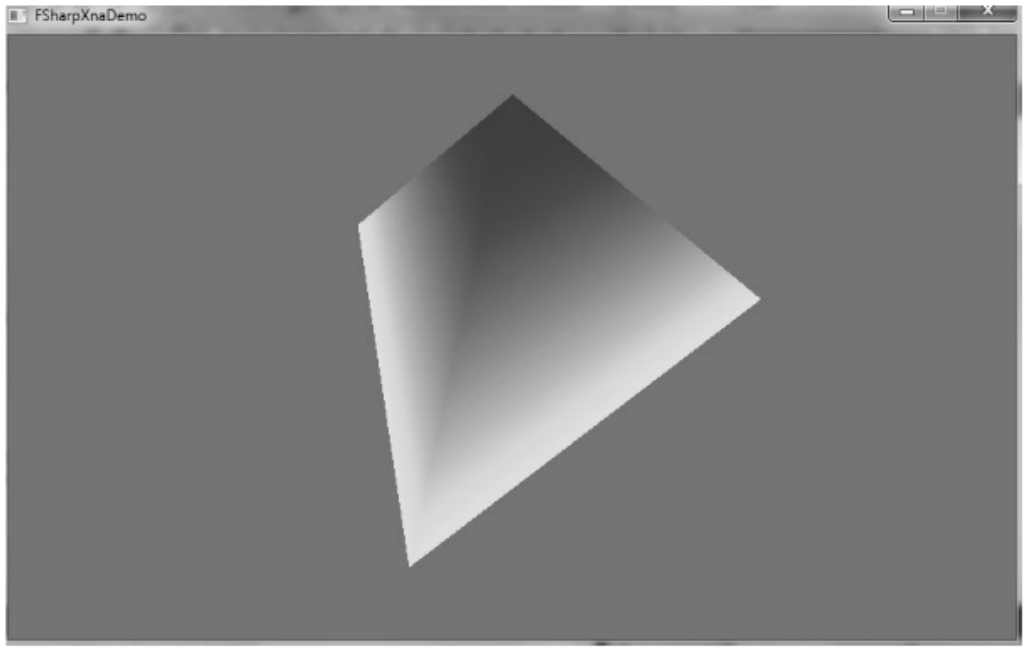

Рис. 7.7. Построение вращающейся пирамиды на XNA

В наборе примеров к этой книге содержится также проект для построения на XNA графика трехмерной функции. На самом деле от рассмотренного только что примера он отличается функциями триангуляции. Для начала зададим саму функцию `f` и ее вариант `func` с учетом преобразования типов к `float32` и масштабирования:

```
let mutable f = fun x y t ->
        let r = sqrt(x*x+y*y)
        exp(r)*sin(10.*Math.PI*r+t)

let func x y t =
        let x,y,t = float(x/100.f), float(y/100.f), float(t/3000.f)
        float32(f x y t)* 7.f
```

Для построения графика разобьем область определения (квадрат на плоскости XY) на некоторое количество квадратов – это количество по каждой оси в каждом из квадрантов обозначим как `scale`. Таким образом, получим $(2*\texttt{scale})^2$ квадратов и на этих узловых точках и будем строить значения функции. Тогда вершины треугольников можно будет вычислить следующим образом:

```
member this.updateFunctionVertices(gameTime : GameTime) =
        let t = gameTime.TotalGameTime.TotalMilliseconds
        functionVertices <-
          [ for i in -scale..scale do
```

```
            for j in -scale..scale do
              let y = func ((float32)i) ((float32)j) ((float32)t)
              let color =
                 new Microsoft.Xna.Framework.Color(
                    (-Math.Abs(y+5.f)*0.15f+1.f),
                    (-Math.Abs(y)*0.15f+1.f),
                    (-Math.Abs(y-5.f)*0.15f+1.f))
              yield new VertexPositionColor(
                      new Vector3((float32)i, y, (float32)j), color)]
```

Для самой триангуляции нам потребуется каждую квадратную область разбить на два треугольника:

```
member this.createFunctionIndices() =
        functionIndices <-
          [ for i in 0..2 * scale - 1 do
              for j in 0..2 * scale - 1 do
                let ulIndex = (int16)(i * (2 * scale + 1) + j)
                let urIndex = ulIndex + 1s
                let dlIndex = (int16)((i + 1) * (2 * scale + 1) + j)
                let drIndex = dlIndex + 1s
                yield! [dlIndex;drIndex;urIndex;dlIndex;urIndex;ulIndex]]
```

Переменные `ulIndex`, `urIndex`, `dlIndex`, `drIndex` обозначают индексы четырех углов квадрата (то есть номера соответствующих вершин из массива вершин), из которых затем в явном виде составляются два треугольника, представленных шестью последовательными номерами в списке.

Для построения координатной сетки в плоскости XY мы заполняем массив вершин следующим образом:

```
member this.createGridVertices() =
  let c = Color.Black
  gridVertices <-
    [ for i in -scale..scale do
        let fi,fs = (float32)i, (float32)scale
        yield new VertexPositionColor(new Vector3(fi, 0.f, fs), c)
        yield new VertexPositionColor(new Vector3(fi, 0.f, -fs), c)
        yield new VertexPositionColor(new Vector3(fs, 0.f, fi), c)
        yield new VertexPositionColor(new Vector3(-fs, 0.f, fi), c)]
```

Для отрисовки сетки в метод `Draw` перед построением самой функции `DrawUserIndexedPrimitives` вызовом мы добавляем следующий вызов:

```
            self.GraphicsDevice.DrawUserPrimitives(PrimitiveType.LineList,
              Array.ofList(gridVertices), 0, gridVertices.Length / 2)
```

Для добавления возможности вращения и масштабирования графика при помощи мыши мы добавляем в метод `Update` отслеживание состояния мыши на пре-

дыдущем и на текущем шагах и, исходя из приращения координат, формируем факторы увеличения и углы поворота:

```
override this.Update(gameTime : GameTime) =
  if (Keyboard.GetState().IsKeyDown(Keys.Escape))
    then self.Exit()
  let xRotation = (float32)(oldMouseState.X - Mouse.GetState().X)/100.f
  let yRotation = (float32)(oldMouseState.Y - Mouse.GetState().Y)/10000.f
  let yRotationAxis = Vector3.Cross(cameraPosition, Vector3.Up)
  let zoom = (float32)(oldMouseState.ScrollWheelValue -
                    Mouse.GetState().ScrollWheelValue)/1000.f
  let xRotationMatrix = Matrix.CreateRotationY(xRotation)
  let yRotationMatrix = Matrix.CreateFromAxisAngle(
                               yRotationAxis, -yRotation)
  let zoomMatrix = Matrix.CreateScale(1.f + zoom)

  cameraPosition <- Vector3.Transform(cameraPosition,
                 xRotationMatrix * yRotationMatrix * zoomMatrix)
  self.setUpMatrices(cameraPosition)

  oldMouseState <- Mouse.GetState()
  self.updateFunctionVertices(gameTime)
  base.Update gameTime
```

Результирующий график, который строит данная программа, приведен на рис. 7.8. Мы настоятельно рекомендуем посмотреть полный текст данного примера в электронном виде, чтобы разобраться в деталях.

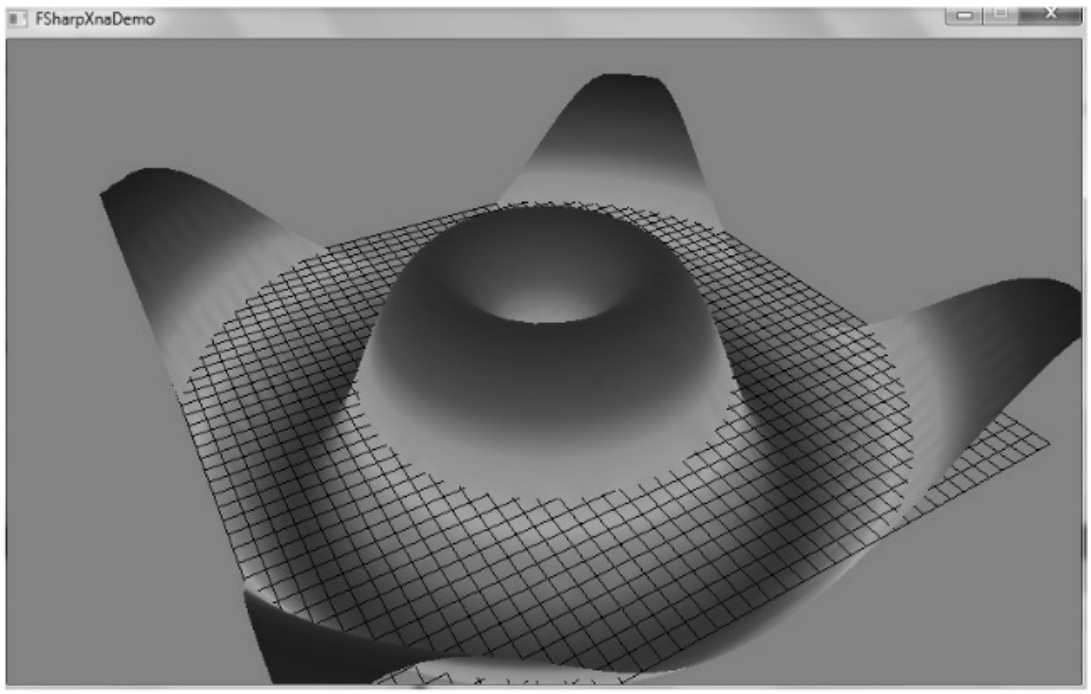

Рис. 7.8. Построение графика трехмерной функции на XNA

Мы показали, как XNA может использоваться совместно с F# для визуализации. Однако основное назначение XNA – это все-таки создание компьютерных игр. Здесь также F# может играть большую роль – например, для разработки искусственного интеллекта игровых персонажей. Примеры такого использования F# для разработки коммерческих игр, находящихся сейчас в рынке приложений XBox Live Arcade, уже есть.

7.5. Анализ текстов и построение компиляторов

Поскольку F# отлично подходит для решения задач обработки данных, вполне естественно, что он применяется и для построения систем анализа текстов, в том числе компиляторов с различных искусственных языков, языков программирования и т. д. Например, как в случае большинства продвинутых языков программирования, сам компилятор языка F# также написан на F#. Любопытный читатель уже, наверное, заинтересовался вопросом – как же компилировали первую версию такого компилятора?

При построении компилятора первым шагом идет *лексический анализ* текста, который преобразует входной поток символов в последовательность *лексем*, каждая из которых задает некоторый смысловой блок языка: ключевое слово, идентификатор, число и т. п. Далее следует *синтаксический анализ*, при котором последовательность лексем преобразуется в *абстрактное синтаксическое дерево*. Иногда эти процессы могут быть реализованы одновременно в рамках одного анализатора.

Далее в процессе компиляции синтаксическое дерево преобразуется в машинный или промежуточный код, проводятся различные оптимизации. Интерпретаторы могут непосредственно использовать дерево для выполнения программы.

В этом разделе мы рассмотрим реализацию лексических и синтаксических анализаторов на F#, что позволит создавать не только языки программирования, но и независимые доменно-ориентированные языки, которые не требуют среды программирования на F# для своего запуска.

7.4.1. Реализация синтаксического разбора методом рекурсивного спуска

Существует множество подходов к реализации задачи синтаксического анализа, которые подробно рассмотрены в специализированной литературе. Хочется подчеркнуть, что вне зависимости от выбранного подхода F#, скорее всего, окажется очень удобным языком для реализации синтаксического анализатора. В качестве примера рассмотрим алгоритм разбора по методу рекурсивного спуска.

Чтобы не усложнять пример, рассмотрим арифметические выражения в инфиксной форме. Например, выражение (1+2)*3 будет в инфиксной форме записываться как * + 1 2 3. Мы реализуем лексический и синтаксический анализаторы для преобразования такого выражения в дерево выражений следующего вида:

```
type Expr =
    | Add of Expr*Expr
    | Sub of Expr*Expr
    | Mul of Expr*Expr
    | Div of Expr*Expr
    | Value of int
```

Лексический анализатор принимает на вход строку, например «* + 1 2 3», и возвращает последовательность лексем – в данном примере список [Op('*'),Op('+'),Int(1),Int(2),Int(3)]. Для описания возможных лексем используем некоторый заданный пользователем тип (в нашем случае очень простой):

```
type Lexem = Int of int | Op of char
```

Такого рода лексический анализатор всегда реализуется в виде конечного автомата, который на каждом шаге «откусывает» от входной строки и обрабатывает по одному символу. При этом также важно, в каком состоянии находится сейчас автомат: например, если во входной строке нужно распознать число 12, то, прочитав цифру 1, автомат переходит в состояние чтения целочисленного числа, и следующий символ будет восприниматься не как новое число, а как продолжение предыдущего. В нашем случае состояние будет двух видов: либо нормальное чтение строки (WhiteSpace), либо чтение целого числа – в этом случае состояние будет содержать значение прочитанного на данный момент начала числа:

```
type State = WhiteSpace | IntSpace of int
```

Функция лексического анализатора будет принимать на вход строку и текущее состояние и возвращать список лексем. По идее, лексический анализатор должен рассматривать все состояния конечного автомата и все возможные типы входных символов (в нашем случае – пробел, знак операции или цифра) и для каждой комбинации что-то добавлять к списку выходных лексем и переходить в новое состояние. Для простоты мы рассматриваем здесь вариант без хвостовой рекурсии и надеемся, что читателю не составит труда в качестве упражнения привести рекурсию к хвостовой:

```
let rec lex (s:string) state =
    if s="" then
        match state with
                    WhiteSpace -> []
                  | IntSpace(n) -> [Int(n)]
    else
        match s.[0] with
            ' ' ->
              match state with
                 WhiteSpace -> lex (s.Substring(1)) WhiteSpace
               | IntSpace(n) -> Int(n)::(lex (s.Substring(1)) WhiteSpace)
```

```
          | x when x>='0'&&x<='9' ->
               match state with
                 WhiteSpace -> lex (s.Substring(1)) (IntSpace(ToInt(x)))
                |IntSpace(n) -> lex
(s.Substring(1)) (IntSpace(n*10+ToInt(x)))
          | '+' | '-' | '*' | '/' ->
                 match state with
                    WhiteSpace ->
                     Op(s.[0])::(lex (s.Substring(1)) WhiteSpace)
                   |IntSpace(n) ->
                     Op(s.[0])::Int(n)::(lex (s.Substring(1)) WhiteSpace)
```

Следующим этапом нам надо описать синтаксический анализатор, преобразующий последовательность лексем в дерево. По методу рекурсивного спуска такой анализатор строится как множество взаимно-рекурсивных функций, каждая из которых отвечает за свое правило грамматики разбора и возвращает построенный фрагмент дерева и остаток входной поседовательности лексем. Например, если мы применяем функцию разбора к последовательности `[Op('+'),Int(1),Int(2),Int(3)]`, то на входе должны получить дерево `Add(Value(1),Value(2)` и остаток последовательности `[Int(3)]`. В нашем случае грамматика имеет простой вид, поэтому мы обойдемся одной рекурсивной функцией `parse`:

```
let parse x =
    let rec parse' = function
          [] -> failwith "Error"
        | Op(c)::l ->
            let (t1,l1) = parse' l
            let (t2,l2) = parse' l1
            match c with
              '+' -> (Add(t1,t2),l2)
            | '-' -> (Sub(t1,t2),l2)
            | '*' -> (Mul(t1,t2),l2)
            | '/' -> (Div(t1,t2),l2)
        | Int(n)::l -> (Value(n),l)
    let (t,l) = parse' x
    if List.isEmpty l then t
    else failwith "Error"
```

Здесь мы рассматриваем два варианта возможных лексем, операцию или число. В случае числа сразу возвращается дерево из одного узла (и остаток последовательности лексем), а в случае операции дважды рекурсивно вызывается фукнция `parse'`, при этом остаточная последовательность лексем передается по цепочке от одного вызова к другому и из полученных поддеревьев собирается результирующее дерево. В случае если в какой-то момент последовательность лексем становится пустой, а функции `parse'` требуются лексемы для разбора, генерируется исключение, указывающее на синтаксически неверную последовательность. Также для удобства вызова функция `parse'` обернута в функцию `parse`, которая, кроме того,

проверяет, чтобы после разбора последовательность лексем была пустой, иначе это также соответствует случаю синтаксически ошибочной последовательности.

7.4.2. Использование fslex и fsyacc

Вы могли заметить, что даже для сравнительно простой задачи лексический и синтаксический анализаторы получаются весьма объемными. В то же время принципы построения такого кода весьма просты. Это позволяет автоматизировать построение лексических и синтаксических анализаторов – для этого служат специализированые утилиты `fslex` и `fsyacc`, берущие свое начало из аналогичных инструментов системы программирования на Си для ОС UNIX.

Рассмотрим пример построения интерпретатора для простого функционального языка программирования, пример программы на котором приведен ниже:

```
letrec fact =
   fun x ->
     if <.x.1
        then 1
        else *.x.(fact.(-.x.1))
in fact.5
```

Мы здесь для простоты в явном виде обозначаем точкой аппликацию, то есть применение функции к аргументу, – в остальном же язык программирования весьма похож по синтаксису на F# или OCaml.

Нашей первой задачей будет преобразование такой программы к синтаксическому дереву типа `expr`:

```
module Ast
type id = string
type expr =
   Var of id
 | Lam of id*expr
 | App of expr*expr
 | Int of int
 | Cond of expr*expr*expr
 | Let of id*expr*expr
 | LetRec of id*expr*expr
 | PFunc of id
 | Op of id*int*expr list
 | Closure of expr*env
 | RClosure of expr*env*id
and
 env = Map<id,expr>

type prog = expr
```

Здесь под окружением `Env` понимается текущее отображение имен на выражения, которое возникает в процессе выполнения (вычисления) программы. На

самом деле конструкции Closure, RClosure и env нужны для дальнейшей интерпретации программы и в процессе синтаксического разбора возникать не будут.

Приведенный выше пример с факториалом преобразуется в следующее абстрактное синтаксическое дерево:

```
(LetRec ("fact", Lam ("x", Cond (App (App (PFunc "<", Var "x"),
Int "1"), Int "1", App (App (PFunc "*", Var "x"),
App (Var "fact", App (App (PFunc "-", Var "x"), Int "1"))))),
App (Var "fact", Int "5")), [])
```

Сначала займемся лексическим анализатором, который по программе должен вернуть последовательность лексем. Для этого надо описать правила генерации лексем в некотором файле (например, lex.fsl), по которому затем утилита lex автоматически построит лексический анализатор. Файл состоит из нескольких секций: вначале в фигурных скобках описываются произвольные F#-предложения, которые должны быть включены в выходной файл. В нашем случае мы описываем имя модуля, открываем модуль синтаксического анализатора Pars и стандартный модуль Lexing из F# Power Pack:

```
{
 module Lex
 open System
 open Pars
 open Microsoft.FSharp.Text.Lexing
}
```

Далее мы можем описать некоторые регулярные выражения для использования в дальнейшем:

```
let digit = ['0'-'9']
let whitespace = [' ' '\t' ]
let newline = ('\n' | '\r' '\n')
```

Основную часть парсера составляют правила rule, которые для каждого регулярного выражения определяют лексему, которую необходимо вернуть. Правила могут быть взаимно рекурсивными (в этом случае они разделяются ключевыми словами and вместо rule), но в нашем случае мы обойдемся одним правилом (для сокращения текста множество очевидных правил пропущено и заменено на многоточие):

```
rule token = parse
| whitespace { token lexbuf }
| newline    { token lexbuf }
| "let"     { LET }
| "then"     { THEN }
| "else"     { ELSE }
| "("        { LPAREN }
```

```
| ">"       { MORE }
...
| ";"       { SEMI }
| ['a'-'z']+
            { ID(LexBuffer<_>.LexemeString lexbuf) }
| ['-']?digit+
            { INT (Int32.Parse(LexBuffer<_>.LexemeString lexbuf)) }
| eof   { EOF }
```

Как видно, многие правила просто разбирают ключевые слова из входного потока, возвращая соответствующую лексему. Более сложные правила могут возвращать лексему с параметром – например, лексема `ID` содержит в себе встретившийся в потоке идентификатор, а `INT` – встретившееся целое число (типа `int`). Для выделения текущей лексемы из потока используется конструкция `LexBuffer<_>.LexemeString lexbuf`, к которой может применяться произвольный код на F# для получения требуемого результата. В ней переменная `lexbuf` – это описанная в основной программе переменная типа `LexBuffer`, содержащая поток входных символов. Чтобы построить по описанию лексического анализатора исходный код, мы используем такую команду (ключ `unicode` необходим для построения анализатора с входным потоком типа `char`, а не `byte`, ключ `-o` указывает имя результирующего файла):

```
fslex -o lex.fs --unicode lex.fsl
```

Для написания синтаксического анализатора (парсера) используется утилита `fsyacc`[1]. Ей на вход подается файл с расширением fsy, а в результате получается F#-модуль (.fs-код и .fsi-интерфейс), строящий дерево. Файл начинается с прелюдии, которая переносится в F#-код напрямую:

```
%{
  open Ast
%}
```

Предполагается, что в модуле `Ast` находится описание абстрактного синтаксического дерева программы – типа `expr`, приведенного выше.

Далее в прелюдии описываются стартовое правило, с которого начинается разбор, возможные лексемы (токены) с указанием типа их параметров и указывается, какой будет базовый тип синтаксического дерева (в нашем случае `Ast.prog`) и из какого стартового правила он получается. После этого идут два знака процента `%%`, обозначающие начало секции правил:

```
%start start

%token <string> ID
```

[1] YACC – это сокращение от Yet Another Compilers Compiler, еще один компилятор компиляторов.

```
%token <System.Int32> INT
%token LET LETREC FUN IN LPAREN RPAREN END BEGIN
%token IF THEN ELSE SEMI EOF PLUS MINUS TIMES DIV EQ ARROW DOT LESS MORE

%type < Ast.prog > start

%%
```

Далее следуют правила грамматики для синтаксического разбора. Каждое правило содержит набор терминальных лексем и нетерминальных правил грамматики. Например, простое выражение языка (не содержащее аппликации) может описываться так:

```
SimpleExpr:
    | ID                        { Var($1) }
    | INT                       { Int($1)  }
    | PLUS { PFunc("+")  } | MINUS { PFunc("-")  } ...
    | FUN ID ARROW Expr { Lam($2,$4) }
    | LPAREN Expr RPAREN { $2 }
    | IF Expr THEN Expr ELSE Expr { Cond($2,$4,$6) }
    | LET ID EQ Expr IN Expr { Let($2,$4,$6) }
    | LETREC ID EQ Expr IN Expr { LetRec($2,$4,$6) }
```

Рассмотрим, например, правило для конструкции `fun`. Оно говорит о том, что конструкция должна начинаться с лексемы `FUN`, затем должен идти идентификатор `ID`, после чего – стрелка и некоторое выражение `Expr` – это нетерминальный символ, описывающийся не лексемой, а другим правилом грамматики (правило для Expr будет описано ниже). В результате должна быть получена конструкция `Lam` с двумя аргументами – идентификатором ($2 в описании означает, что нужно взять параметр лексемы, идущей вторым номером в последовательности, то есть лексемы `ID`) и деревом выражения `Expr` (оно идет в списке четвертым, поэтому обозначается как $4). Аналогично определяются другие правила разбора.

Выражение в общем случае может быть последовательностью аппликаций простых выражений, возможно, состоящей из одного простого выражения. Для этого мы используем следующие рекурсивные правила разбора:

```
Expr: AppList { $1 }

AppList:
    | SimpleExpr { $1 }
    | AppList DOT SimpleExpr { App($1,$3) }
```

Нам остается определить правило для разбора всей программы (вся программа – это одно большое выражение!) и стартовое правило `start`:

```
start: Prog { $1 }

Prog: Expr { $1 }
```

Для преобразования полученного описания парсера в программный модуль на F# используем команду:

```
fsyacc -o pars.fs --module Pars pars.fsy
```

В ключе `module` мы указываем имя модуля, который необходимо сгенерировать.

Для реализации интерпретатора используем технику Eval/Apply-интерпретатора, описанную подробно в [11]. Интерпретатор будет состоять из двух взаимно рекурсивных функций – `eval` для вычисления выражения `exp` в некотором окружении `env` и `apply` для реализации аппликации. Аппликация устроена таким образом, что она применяется либо к встроенной функции Op (в этом случае, пока не будет последовательно в цепочке аппликаций накоплено нужное количество аргументов функции, они хранятся в списке, потом функция применяется), либо к замыканиям (обычному или рекурсивному), которые содержат в себе необходимые окружения. Определение `apply` содержит в себе основные правила упрощения выражений, включая отдельное рассмотрение условного оператора и абстракции, которая ведет к построению замыкания.

Ниже приводится слегка сокращенный (там, где использовано многоточие) текст интерпретатора, заинтересованного читателя мы отсылаем к книге [11].

```
module LambdaInterpreter
open Ast

let arity = function _ -> 2

let funof = function
  "+" -> (function [Int(a);Int(b)] -> Int(a+b))
…
| "<=" -> (function [Int(a);Int(b)] -> if a<=b then Int(1) else Int(0))

let rec eval exp env =
  match exp with
    App(e1,e2) -> apply (eval e1 env) (eval e2 env)
  | Int(n) -> Int(n)
  | Var(x) -> Map.find x env
  | PFunc(f) -> Op(f,arity f,[])
  | Op(id,n,el) -> Op(id,n,el)
  | Cond(e0,e1,e2) ->
     if Int(1)=eval e0 env then eval e1 env else eval e2 env
  | Let(id,e1,e2) ->
    let r = eval e1 env in
      eval e2 (Map.add id r env)
  | LetRec(id,e1,e2) ->
      eval e2 (Map.add id (RClosure(e1,env,id)) env)
  | Lam(id,ex) -> Closure(exp,env)
  | Closure(exp,env) -> exp
and apply e1 e2 =
  match e1 with
     Closure(Lam(v,e),env) -> eval e (Map.add v e2 env)
```

```
   | RClosure(Lam(v,e),env,id) ->
eval e (Map.add v e2 (Map.add id e1 env))
   | Op(id,n,args) ->
      if n=1 then (funof id)(args@[e2])
      else Op(id,n-1,args@[e2])

let E exp = eval exp Map.empty;;
```

Наконец, основная программа, которая принимает имя файла с программой на нашем упрощенном функциональном языке и интерпретирует ее, выглядит так:

```
module Interp
open Ast
open LambdaInterpreter
open Microsoft.FSharp.Text.Lexing

[<EntryPoint>]
let main(argv) =
    if argv.Length <> 1 then begin
        printf "usage: interp.exe <file>\n"; exit 1;
    end;

    let stream = new StreamReader(argv.[0])
    let myProg =
        let lexbuf =
     Microsoft.FSharp.Text.Lexing.LexBuffer<_>.FromTextReader stream
        try
            Pars.start Lex.token lexbuf
        with e ->
            let pos = lexbuf.EndPos
            printf "error near line %d, character %d\n%s\n"
 pos.Line pos.Column (e.ToString);
            exit 1

    printf "Execution Begins...\n%A\n" (E myProg); 0
```

В программе мы получаем имя файла, открываем текстовый поток, создаем объект типа `LexBuffer` – после чего вызываем парсер, который пользуется лексическим анализатором и выдает на выходе абстрактное синтаксическое дерево программы `myProg`. После этого нам остается лишь вызвать интерпретатор и вывести на печать результат выполнения программы.

7.5. Создание F#-приложений для Silverlight и Windows Phone 7

При создании современных пользовательских или интернет-приложений с богатым пользовательским интерфейсом (так называемых RIA-приложений, Rich Internet Applications) на платформе Майкрософт все чаще используется тех-

нология Silverlight. Она основана на принципе разделения программного кода и декларативной разметки интерфейса на языке XAML, при этом богатство графического языка позволяет описывать не только современные графические интерфейсы (использующие возможности графических процессоров), но и многие анимационные эффекты, связывание с данными и т. д., вынося в код приложения только необходимую бизнес-логику.

Помимо внутрибраузерных приложений, Silverlight эффективно используется для создания внебраузерных приложений для Windows, а также является базовым средством программирования недавно выпущенной ОС для смартфонов Windows Phone 7. Все это делает Silverlight чрезвычайно привлекательной программной моделью для создания широкого спектра приложений: как деловых, так и мультимедийно-развлекательных.

Silverlight содержит независимую облегченную среду выполнения .NET, что позволяет запускать его в различных браузерах и даже на различных платформах (во многом с помощью проекта moonlight, основанного на Mono-реализации .NET). Несмотря на это, в Silverlight-приложениях также можно использовать различные языки программирования, включая F#. Более того, использование F# часто оказывается оправданным, поскольку RIA-приложения на Silverlight часто обращаются за данными к серверу (так как реализовать прямой доступ к СУБД в Silverlight-приложении невозможно), а обмен данными с сервером уместно делать в асинхронном режиме.

Стандартная поставка Visual Studio 2010 позволяет создавать Silverlight-библиотеки на F# – для этого предусмотрен специальный тип проекта Silverlight Library. Создание Silverlight-приложения исключительно на F# не поддерживается – примерно по тем же соображениям, что и программирование визуального интерфейса ASP.NET-приложений. Однако это ограничение при желании[1] можно обойти, создав соответствующие проекты вручную.

В качестве примера рассмотрим создание Silverlight-приложения для Windows Phone 7 с помощью F#. Для упрощения создания такого приложения можно воспользоваться шаблоном «F# and C# Windows Phone Application (Silverlight)», доступным в онлайн-каталоге шаблонов Visual Studio 2010. При создании такого приложения создаются два проекта – `App`, содержащий основной код приложения на языке F# (весь код по умолчанию помещается в один файл `AppLogic.fs`), и AppHost – проект на C#, содержащий в себе основную визуальную разметку приложения на языке XAML и соответствующие codebehind-файлы на C#, выполняющие роль заглушек.

При создании приложения файл `AppLogic.fs` содержит в себе базовую инфраструктуру приложения, а также описание крайне полезного оператора ?, позволяющего осуществлять динамическое разрешение имен элементов в XAML-разметке.

---

[1] Хотя обычно такого желания возникать не должно, поскольку взаимодействие между фрагментами программы на C# и F# осуществляется прозрачно, что позволяет эффективно реализовывать визуальную часть приложения на C#, а логику и асинхронный ввод-вывод и обращение к веб-сервисам – на F#.

В примерах, доступных для читателей этой книги на сайте http://www.soshnikov.com/fsharp, содержится пример панорамного приложения для Windows Phone 7, которое осуществляет сравнение популярности двух терминов по количеству найденных вхождений этих терминов в Интернете. Для определения количества вхождений мы используем Bing Search API в асинхронном режиме на основе примера из раздела 7.3.3. В нашем случае для поиска мы используем класс Searcher, имеющий следующий вид:

```
type BingSearcher (s:string) =
  let AppID = "[Your AppID Here]"
  let url = sprint
    "http://api.search.live.net/xml.aspx?Appid=%s&sources=web&query=%s"
    AppID s
  let resultEvent = new Event<SearchResultEventArgs>()
  let AsyncResult () =
    async {
      let req = WebRequest.Create url
      use! resp =
 Async.FromBeginEnd(req.BeginGetResponse,req.EndGetResponse)
      use stream = resp.GetResponseStream()
      let xdoc = XDocument.Load(stream)
      let webns =
         System.Xml.Linq.XNamespace.op_Implicit
"http://schemas.microsoft.com/LiveSearch/2008/04/XML/web"
      let sx = xdoc.Descendants(webns.GetName("Total"))
      let cnt = Seq.head(sx).Value
      return Int32.Parse(cnt)
    }
  member x.Pull() =
     let res = AsyncResult()
     let wrk = new AsyncWorker<_>(res)
     wrk.JobCompleted.Add(fun args ->
        resultEvent.Trigger(new SearchResultEventArgs(args.Result)))
     wrk.Start()
  [<CLIEvent>]
  member x.ResultAvailable = resultEvent.Publish
```

В этом классе мы определяем событие `ResultAvailable`, которое срабатывает, как только приходит результат от Bing Search API. Для запуска процесса необходимо вызвать метод `Pull`, использующий для своей работы специальный класс `AsyncWorker`. Этот весьма хитрый класс позволяет нам запустить на выполнение задачу, передаваемую как аргумент, в асинхронном режиме, а после ее завершения вызвать соответствующее событие в потоке выполнения пользовательского интерфейса (GUI thread).

```
    type AsyncWorker<'T>(job: Async<'T>) =
       let error         = new Event<System.Exception>()
           let canceled      = new Event<OperationCanceledException>()
```

```
        let jobCompleted  = new Event<JobCompletedEventArgs<'T>>()
        let cancellationCapability = new CancellationTokenSource()

        /// Start an instance of the work
        member x.Start()    =
            let syncContext = SynchronizationContext.CaptureCurrent()
            let raiseEventOnGuiThread(evt, args) =
                    syncContext.RaiseEvent evt args
            let work = async { let! result = job
                               syncContext.RaiseEvent
                                  jobCompleted
                                    (new JobCompletedEventArgs<'T>(result))
                               return result }
            Async.StartWithContinuations
                ( work,
                  (fun res -> raiseEventOnGuiThread(allCompleted, res)),
                  (fun exn -> raiseEventOnGuiThread(error, exn)),
                  (fun exn -> raiseEventOnGuiThread(canceled, exn)),
                    cancellationCapability.Token)

        [<CLIEvent>]
        member x.JobCompleted = jobCompleted.Publish
        [<CLIEvent>]
        member x.Canceled = canceled.Publish
        [<CLIEvent>]
        member x.Error = error.Publish
```

В нашем приложении на основной странице будут находиться два текстовых поля для ввода сравниваемых терминов (`term1box` и `term2box`) и кнопка (`mainBtn`), а вторая страница панорамы будет содержать два текстовых поля (`textItem1`/`textItem2`) и два прямоугольника (`barItem1`/`barItem2`) для графического отображения результатов. Это будет описываться примерно таким кодом на языке XAML:

```
<controls:Panorama Title="Search Master">
<controls:Panorama.Background>
  <ImageBrush ImageSource="Images\PanoramaBackground.png"/>
</controls:Panorama.Background>
<controls:PanoramaItem Header="search terms">
  <StackPanel>
    <TextBlock Height="30" Text="Item #1" VerticalAlignment="Top" />
    <TextBox x:Name="Term1" Height="72" Text="Microsoft" />
    <TextBlock Height="30" Text="Item #2" VerticalAlignment="Top" />
    <TextBox x:Name="Term2" Height="72" Text="UNIX" />
    <Button x:Name="MainBtn" Content="Compare" Height="72" Width="160" />
  </StackPanel>
</controls:PanoramaItem>
<controls:PanoramaItem Header="second item">
  <StackPanel>
    <TextBlock x:Name="TextItem1" Text="Item #1"></TextBlock>
    <Rectangle x:Name="BarItem1" Fill="Yellow" Stroke="Yellow" />
    <TextBlock x:Name="TextItem2" Text="Item #2"></TextBlock>
```

```
    <Rectangle x:Name="BarItem2" Fill="Red" Stroke="Red"/>
  </StackPanel>
   </controls:PanoramaItem>
</controls:Panorama>
```

Основной код нашего приложения будет достаточно простым: необходимо описать событие, возникающее при нажатии кнопки:

```
   do mainBtn.Click.Add(fun e ->
        let res1 = new Shwarsico.BingSearcher(term1box.Text)
        let res2 = new Shwarsico.BingSearcher(term2box.Text)
        res1.ResultAvailable.Add(fun res -> redraw 1 res.Count)
        res2.ResultAvailable.Add(fun res -> redraw 2 res.Count)
        textitem1.Text <- "Querying..." ; res1.Pull()
        textitem2.Text <- "Querying..." ; res2.Pull()
        )
```

Это событие создает два поисковых объекта и определяет события, срабатывающие при приходе ответа от сервера; после чего запускается асинхронный поиск (одновременно посылаются оба запроса серверу). За приход результатов отвечает метод redraw, который обновляет экран с результатами (при этом умеет это правильно делать и в том случае, когда пришел только первый результат):

```
let mutable p1,p2,max = 100,50,100

let redraw n x =
   if x>max then max <- x
   match n with
     | 1 -> p1 <- x; textitem1.Text <- x.ToString()
     | 2 -> p2 <- x; textitem2.Text <- x.ToString()
   baritem1.Width <- 400.*float(p1)/float(max)
   baritem2.Width <- 400.*float(p2)/float(max)
```

Все эти функции описываются внутри объекта, отвечающего за отображение главной страницы приложения:

```
type MainPage() as this =
    inherit PhoneApplicationPage()
    do Application.LoadComponent(this,
       new System.Uri("/WindowsPhonePanoramaApp;component/MainPage.xaml",
      System.UriKind.Relative))
    let root = new PhoneApplicationFrame()
    let term1box : TextBox = this?Term1
...
    let mainBtn : Button = this?MainBtn
```

Здесь же вначале с помощью оператора ? разрешаются ссылки на все внутренние элементы управления XAML-страницы.

Полный код приложения можно найти в примерах кода к этой книге, а вид первой страницы приложения – на рис. 7.9. Для более детального понимания процесса разработки под Windows Phone 7 мы отсылаем читателя к этому исходному коду, а также рекомендуем поэкспериментировать самостоятельно. Более общие концепции разработки под Windows Phone 7 как на Silverlight, так и на XNA содержатся в книге Чарльза Петцольда [16], доступной бесплатно в электронном виде.

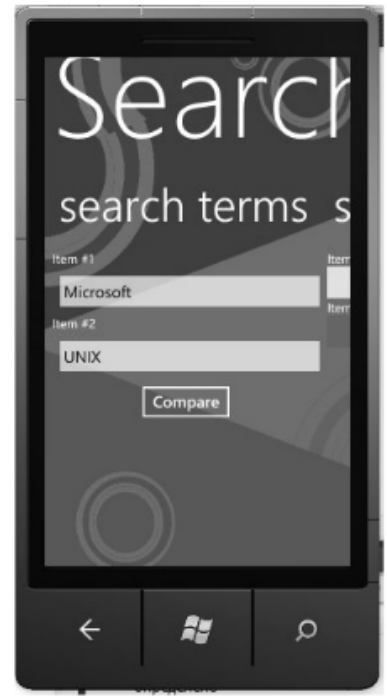

Рис. 7.9. Панорамное приложение Windows Phone 7

Вместо заключения

Обычно в заключении принято подводить некоторый итог книги, заканчивать на оптимистической ноте и обещать счастья каждому, кто сумел дочитать книгу до конца. Мне бы хотелось поступить немного нетрадиционно и рассмотреть здесь еще один красивый игрушечный пример использования F# для получения эстетического удовольствия.

В самом начале нашего изучения F# мы научились строить фрактальное изображение – множество Мандельброта. Сейчас же мы снова будем строить фракталы, на этот раз так называемые L-системы (L-Systems). Подробнее о L-системах вы сможете прочитать в [17], а также в замечательной интернет-статье по адресу http://habrahabr.ru/blogs/biotech/69989, которой во многом и был вдохновлен этот пример.

Говоря очень упрощенно, L-системы – это почти что грамматики, которые в некотором роде моделируют деление клеток в биологическом организме. L-система содержит некоторое количество правил переписывания, начальный символ – и далее применением правил переписывания к текущей строке, начиная с начального символа, мы получаем все новые и новые строки, генерируемые L-системой. Отличие от грамматики состоит в том, что мы одновременно заменяем все вхождения символов в строке на правые части правил, а также в отсутствие явного условия окончания (терминальных правил), поскольку в идеале мы можем получать бесконечные L-структуры.

Например, рассмотрим грамматику с правилом F → F-F++F-F и с начальной строкой F++F++F++. При первом применении мы заменим все F в начальной строке на правую часть правила, получив строку F-F++F-F++F-F++F-F++F-F++F-F++F-F++. Следующее применение правила даст строку F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++F-F++F-F-F-F++F-F++. И так далее, потенциально до бесконечности...

Возникает вопрос: а причем тут фракталы? Так вот, если интерпретировать получаемые строки как команды некоторой черепашки-рисователя, где F означает движение вперед, + и - – повороты направо и налево соответственно на определенный угол, то полученные нами строки будут определять следующие фигуры (называемые, кстати, снежинками Коха), показанные на рис. 1.

Таким образом, этот пример удивительным образом сочетает обработку символьных данных – преобразования строк на основе грамматик – и рассмотренные нами возможности платформы .NET по визуализации. Именно поэтому мне бы и хотелось рассмотреть этот пример.

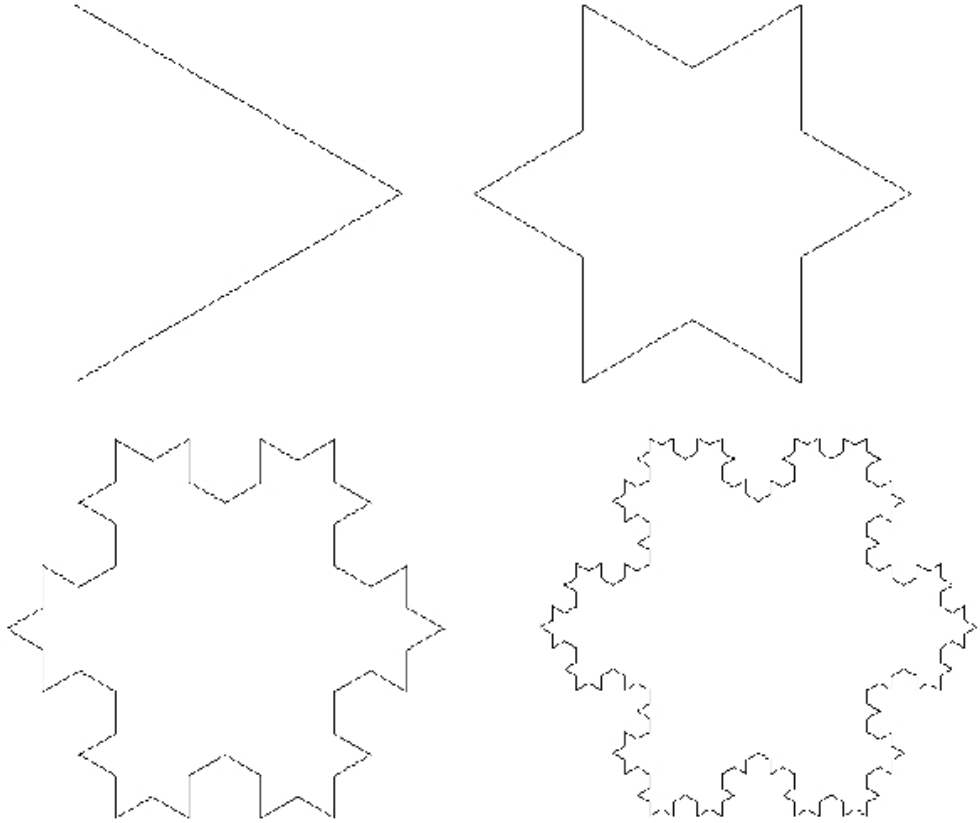

Рис. 1. Снежинка Коха

Начнем с грамматик. Будем задавать строки как списки символов, каждое правило грамматики – как пару из символа левой части и списка подстановки, а всю грамматику – как список таких правил:

```
type Rule = char * char list
type Grammar = Rule list
```

Для замены одного символа на список определим функцию

```
let FindSubst c (gr:Grammar) =
   match List.tryFind (fun (x,S) -> x=c) gr with
     | Some(x,S) -> S
     | None -> [c]
```

Если соответствующий символ c не находится в грамматике, то он заменяется сам на себя, то есть возвращается список из одного этого символа, в противном случае возвращается список, стоящий в правой части грамматики.

Однократное применение грамматики ко всем символам текущей строки запишется весьма компактно с помощью функции `collect`:

```
let Apply gr = List.collect (fun c -> FindSubst c gr)
```

Для многократного применения функции нам достаточно использовать рассмотренный нами выше (при построении множества Мандельброта) комбинатор n-кратного применения функции rpt:

```
let NApply n gr = rpt n (Apply gr)
```

Теперь нам надо научиться интерпретировать строки как язык черепашьей графики. Для этого опишем следующую функцию:

```
let TurtleBitmapVisualizer n delta cmd =
    let W,H = 1600,1600
    let b = new Bitmap(W,H)
    let g = Graphics.FromImage(b)
    let pen = new Pen(Color.Black)
    let NewCoord (x:float) (y:float) phi =
       let nx = x+n*cos(phi)
       let ny = y+n*sin(phi)
       (nx,ny,phi)
    let ProcessCommand x y phi = function
     | 'f' -> NewCoord x y phi
     | '+' -> (x,y,phi+delta)
     | '-' -> (x,y,phi-delta)
     | 'F' ->
         let (nx,ny,phi) = NewCoord x y phi
         g.DrawLine(pen,(float32)x,(float32)y,(float32)nx,(float32)ny)
         (nx,ny,phi)
     | _ -> (x,y,phi)
    let rec draw x y phi = function
     | [] -> ()
     | h::t ->
         let (nx,ny,nphi) = ProcessCommand x y phi h
         draw nx ny nphi t
    draw (float(W)/2.0) (float(H)/2.0) 0. cmd
    b
```

Вначале мы создаем чистый Bitmap для рисования и объект Graphics на его основе, который позволит рисовать на изображении графические примитивы – в нашем случае линии. Основную работу по рисованию берет на себя функция draw, которая для каждой встретившейся в списке команды выполняет ее при помощи ProcessCommand и рекурсивно обрабатывает конец списка. Текущие координаты пера «черепашки» и текущий угол поворота передаются аргументами draw, а функция ProcessCommand по текущим координатам и команде возвращает новую тройку состояния, по ходу дела рисуя линии на изображении.

Приведенные выше на рис. З.1 изображения были построены с помощью описанной выше функции, например:

```
let str (s:string) = s.ToCharArray() |> List.ofArray
let gr = [('F',str "F-F++F-F")]
```

```
let lsys = NApply 2 gr (str "F++F++F++")
let B = TurtleBitmapVisualizer 40.0 (Math.PI/180.0*60.0) lsys
B.Save(@"c:\pictures\bitmap.jpg")
```

Таким же образом можно получать другие интересные изображения, приведенные на рис. 2.

Рис. 2. Более сложные L-системы

Слегка усовершенствовав нашу рисовальную машину, мы сможем получать изображения, похожие на настоящие растения! Для этого нам потребуется реализовать возможность ветвления, то есть запоминания текущего положения черепашки с возможностью последующего возврата на это положение. Для запоминания положения в стеке будем использовать команду [, а для возврата –]. Запоминать надо текущие координаты и угол поворота:

```
type TurtleState = float * float * float
```

В начале функции `TurtleBitmapVisualizer` опишем стек для положений черепашки:

```
let stk = new Stack<TurtleState>()
```

а во вложенную функцию `ProcessCommand` добавим обработку квадратных скобок:

```
let ProcessCommand x y phi = function
 | 'f' -> …
 …
 | '[' -> stk.Push((x,y,phi)); (x,y,phi)
 | ']' -> stk.Pop()
 | _ -> (x,y,phi)
```

Такие команды позволят нам рисовать изображения L-систем, приведенных на рис. 3.

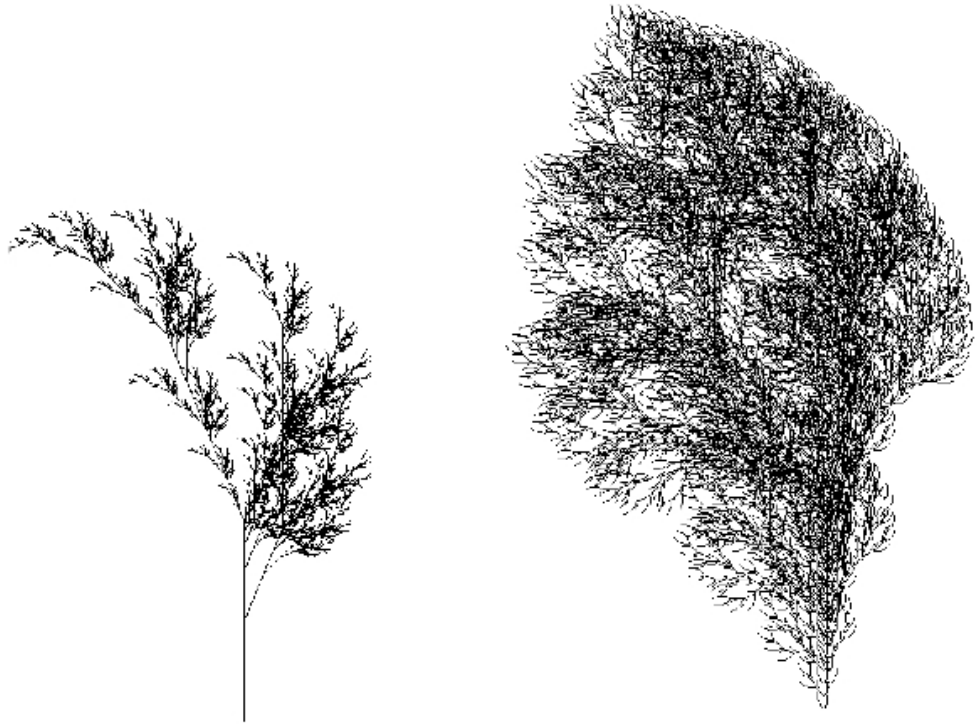

Рис. 3. Искусственная жизнь на F#

Этот пример наглядно показывает, что F# может использоваться для создания искусственной жизни! А если серьезно – то мне бы хотелось, чтобы после прочтения этой книги F# запомнился бы вам именно таким, как в этой задаче: языком, позволяющим быстро прототипировать весьма нетривиальные задачи обработки и визуализации данных, при этом получая удовольствие, используя минимум программного кода и не испытывая проблем при отладке.

Желаю вам, дорогие читатели, удачного дальнейшего изучения F# и других функциональных языков и продуктивного использования их на практике. Буду рад услышать ваши отзывы по электронной почте dmitri@soshnikov.com или в твиттере http://twitter.com/shwars. Доброго кода!

Рекомендуемая литература

[1] D.Syme, A.Granicz, A.Cisternio. Expert F# 2.0. Apress, 2010.

[2] C. Smith, Programming F#: A comprehensive guide for writing simple code to solve complex problems. O'Reilly, 2010.

[3] T.Neward, A.Erickson, T.Crowell, R.Minerich. Professional F# 2.0. Wiley Publishing, 2011.

[4] T.Petricek, J.Skeet. Real World Functional Programming: With Examples in F# and C#. Manning Publications, 2010.

[5] R.Pickering, Beginning F#, Apress, 2009.

[6] J.Harrop, F# for Scientists, Wiley Publishing, 2008.

[7] Сошников Д.В. Видео-курс «Функциональное программирование». Интернет-университет информационных технологий ИНТУИТ.РУ. http://bit.ly/funcprovideo

[8] R.Pickering, Foundations of F#, Apress, 2008.

[9] D.Syme, A.Granicz, A.Cisternio. Expert F#. Apress, 2008.

[10] E. Chailloux, P. Manoury, B.Pagano. Développement d'applications avec Objective Caml. O'Reilly, 2000. Русский перевод: http://shamil.free.fr/comp/ocaml/

[11] Филд А., Харрисон П. Функциональное программирование. – М.: Мир, 1993.

[12] Harrison, J. Introduction to Functional Programming. Lecture Notes, Cambridge University, 1997.

[13] Хювёнен Э., Сеппенен И. Мир Lisp'а. В 2-х томах. М.: Мир, 1990.

[14] Thompson S. Haskell: The Craft of Functional Programming. 2-nd edition, Addison-Wesley, 1999.

[15] Г. Магдануров, В. Юнев. ASP.NET MVC Framework. Серия: Профессиональное программирование. – СПб.: БХВ-Петербург, 2010.

[16] C.Petzold. Programming Windows Phone 7. Microsoft Press, 2010. Бесплатная электронная версия: http://bit.ly/petzwp7

[17] P.Prusinkiewicz, A.Lindenmayer. The Algorithmic Beauty of Plants. Springer-Verlag, 1990. Электронный вариант доступен по адресу http://algorithmicbotany.org/papers/#abop

Сошников Дмитрий Валерьевич

Функциональное программирование на F#

практическое электронное издание

Главный редактор *Мовчан Д. А.*

Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Для создания электронного издания использовано:
Microsoft Word 2013, приложение pdf2swf из ПО Swftools,
ПО IPRbooks Reader,
разработанное на основе Adobe Air

Подписано к использованию 04.04.2017 г.
Объем данных 2 Мб.