Предисловие Дона Сайма 6

Предисловие автора 8

0. Введение 10
- 0.1. Зачем изучать функциональное программирование 10
- 0.2. О чем и для кого эта книга 12
- 0.3. Как установить и начать использовать F# 13

1. Основы функционального программирования 17
- 1.1. Применение функций vs. Присваивание 17
- 1.2. Упорядоченные кортежи, списки и вывод типов 19
- 1.3. Функциональные типы и описание функций 20
- 1.4. Каррирование 22
- 1.5. Условный оператор и опциональный тип 23
- 1.6. Типы данных, размеченное объединение и сопоставление с образцом 25
- 1.7. Рекурсия, функции-параметры и цикл for 27
- 1.8. Конструкции >>, |> 28
- 1.9. Пример – построение множества Мандельброта 29
- 1.10. Интероперабельность с .NET 31

2. Рекурсивные структуры данных 34
- 2.1. Списки и конструкторы списков 34
- 2.2. Сопоставление с образцом 35
- 2.3. Простейшие функции обработки списков 36
- 2.4. Функции высших порядков 37
  - 2.4.1. Отображение 37
  - 2.4.2. Фильтрация 39
  - 2.4.3. Свертка 41
  - 2.4.4. Другие функции высших порядков 43
- 2.5. Генераторы списков 44
- 2.6. Хвостовая рекурсия 45
- 2.7. Сложностные особенности работы со списками 47
- 2.8. Массивы 50
- 2.9. Многомерные массивы и матрицы 52
  - 2.9.1. Списки списков, или непрямоугольные массивы (Jagged Arrays) 52
  - 2.9.2. Многомерные массивы .NET 53
  - 2.9.3. Специализированные типы для матриц и векторов 54
  - 2.9.4. Разреженные матрицы 55

2.9.5. Использование сторонних математических пакетов 56
2.10. Деревья общего вида 56
2.11. Двоичные деревья 59
2.11.1. Определение 59
2.11.2. Обход двоичных деревьев 59
2.11.3. Деревья поиска 60
2.11.4. Деревья выражений и абстрактные синтаксические деревья (AST) ... 62
2.12. Другие структуры данных 63
2.12.1. Множества (Set) 63
2.12.2. Отображения (Map) 63
2.12.3. Хеш-таблицы 64

3. Типовые приемы функционального программирования 66
3.1. Замыкания 66
3.2. Динамическое связывание и mutable-переменные 67
3.3. Генераторы и ссылочные переменные ref 68
3.4. Ленивые последовательности (seq) 71
3.4.1. Построение частотного словаря текстового файла 73
3.4.2. Вычисление числа π методом Монте-Карло 74
3.5. Ленивые и энергичные вычисления 76
3.6. Мемоизация 79
3.7. Продолжения 81

4. Императивные и объектно-ориентированные возможности F# 84
4.1. Мультипарадигмальность языка F# 84
4.2. Элементы императивного программирования на F# 85
4.2.1. Использование изменяемых переменных и ссылок 85
4.2.2. Цикл с предусловием 86
4.2.3. Условный оператор 87
4.2.4. Null-значения 87
4.2.5. Обработка исключительных ситуаций 87
4.3. Объектно-ориентированное программирование на F# 89
4.3.1. Записи 89
4.3.2. Моделирование объектной ориентированности через записи и замыкания 90
4.3.3. Методы 91
4.3.4. Интерфейсы 92
4.3.5. Создание классов с помощью делегирования 93
4.3.6. Создание иерархии классов 94
4.3.7. Расширение функциональности имеющихся классов 97
4.3.8. Модули 97

5. Метапрограммирование 99
5.1. Языково-ориентированное программирование 99
5.2. Активные шаблоны 102
5.3. Квотирование 103

5.4. Конструирование выражений, частичное применение функции и суперкомпиляция 106
5.5. Монады 107
5.5.1. Монада ввода-вывода 108
5.5.2. Монадические свойства 110
5.5.3. Монада недетерминированных вычислений 111
5.6. Монадические выражения 112

6. Параллельное и асинхронное программирование 115
6.1. Асинхронные выражения и параллельное программирование 115
6.2. Асинхронное программирование 116
6.3. Асинхронно-параллельная обработка файлов 118
6.4. Агентный паттерн проектирования 120
6.5. Использование MPI 122

7. Решение типовых задач 127
7.1. Вычислительные задачи 127
7.1.1. Вычисления с высокой точностью 127
7.1.2. Комплексный тип 128
7.1.3. Единицы измерения 128
7.1.4. Использование сторонних математических пакетов 129
7.2. Доступ к данным 131
7.2.1. Доступ к реляционным базам данных (SQL Server) 131
7.2.2. Доступ к слабоструктурированным данным XML 136
7.2.3. Работа с данными в Microsoft Excel 139
7.3. Веб-программирование 143
7.3.1. Доступ к веб-сервисам, XML-данным, RSS-потокам 144
7.3.2. Доступ к текстовому содержимому веб-страниц 144
7.3.3. Использование веб-ориентированных программных интерфейсов на примере Bing Search API 147
7.3.4. Реализация веб-приложений на F# для ASP.NET Web Forms 148
7.3.5. Реализация веб-приложений на F# для ASP.NET MVC 150
7.3.6. Реализация веб-приложений на F# при помощи системы WebSharper 152
7.3.7. Облачное программирование на F# для Windows Azure 156
7.4. Визуализация и работа с графикой 158
7.4.1. Двухмерная графика на основе Windows Forms API 159
7.4.2. Использование элемента Chart 160
7.4.3. 3D-визуализация с помощью DirectX и/или XNA 164
7.5. Анализ текстов и построение компиляторов 171
7.4.1. Реализация синтаксического разбора методом рекурсивного спуска 171
7.4.2. Использование fslex и fsyacc 174
7.5. Создание F#-приложений для Silverlight и Windows Phone 7 179

Вместо заключения 185

Рекомендуемая литература 190

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, Microsoft Research 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

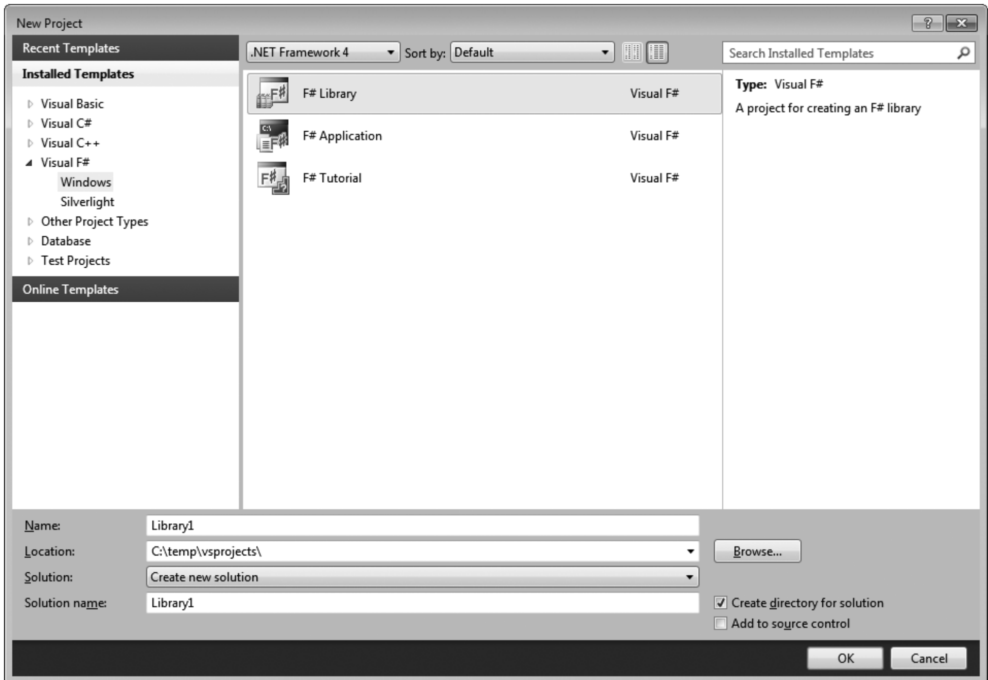

Рис. 0.1. Создание F#-проекта в Visual Studio 2010

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

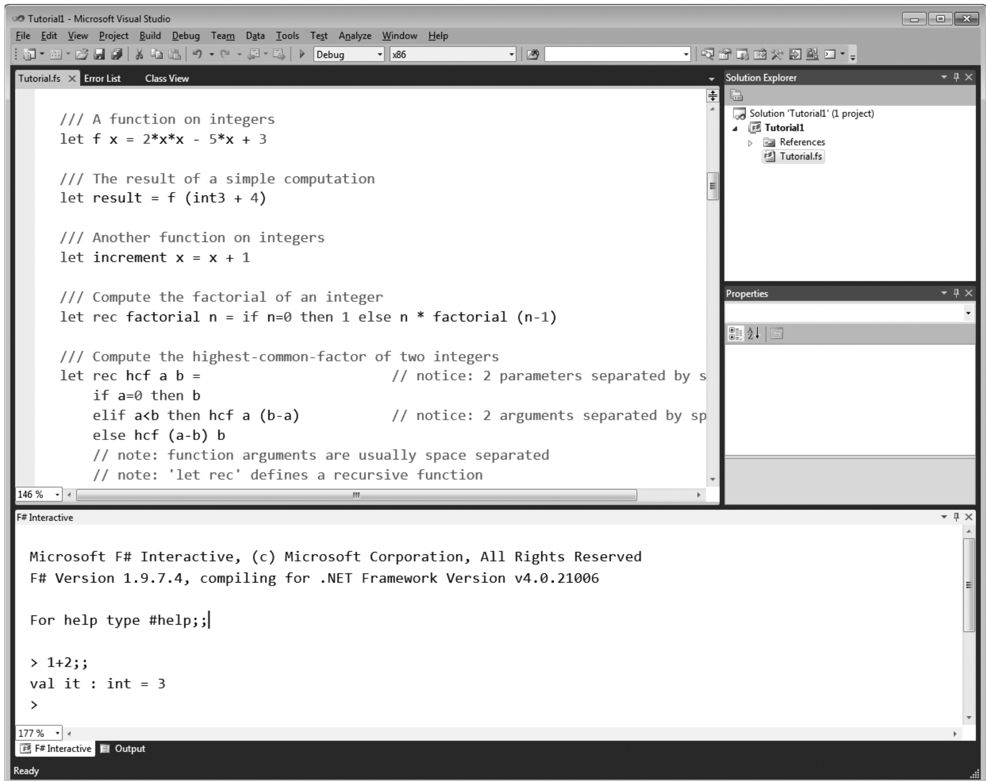

Рис. 0.2. Окно псевдо-интерпретатора F# Interactive

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

Конец ознакомительного фрагмента.
Приобрести книгу можно
в интернет-магазине
«Электронный универс»
e-Univers.ru