Foundations of F#

Robert Pickering

Apress®

Foundations of F#

ISBN-13: 978-1-59059-757-6

ISBN-10: 1-59059-757-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: James Huddleston, Ewan Buckingham
Technical Reviewer: Don Syme
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, Chris Mills, Matthew Moodie, Jeffrey Pepper, Dominic Shakeshaft, Matt Wade
Project Manager: Elizabeth Seymour
Copy Edit Manager: Nicole Flores
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Lynn L'Heureux
Proofreader: Elizabeth Berry
Indexer: Broccoli Information Management
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code/Download section.

For Susan and for Jim

Contents at a Glance

Foreword xiii
About the Author xv
About the Technical Reviewer xvii
Acknowledgments xix
Preface: The Story of the Book xxi

CHAPTER 1 Introduction 1
CHAPTER 2 How to Obtain, Install, and Use F# 7
CHAPTER 3 Functional Programming 15
CHAPTER 4 Imperative Programming 55
CHAPTER 5 Object-Oriented Programming 81
CHAPTER 6 Organizing, Annotating, and Quoting Code 111
CHAPTER 7 The F# Libraries 129
CHAPTER 8 User Interfaces 167
CHAPTER 9 Data Access 209
CHAPTER 10 Distributed Applications 239
CHAPTER 11 Language-Oriented Programming 271
CHAPTER 12 The F# Tool Suite and .NET Programming Tools 299
CHAPTER 13 Compatibility and Advanced Interoperation 323

INDEX 345

Contents

Foreword xiii
About the Author xv
About the Technical Reviewer xvii
Acknowledgments xix
Preface: The Story of the Book xxi

CHAPTER 1 Introduction 1

What Is Functional Programming? 1
Why Is Functional Programming Important? 2
What Is F#? 2
Who Is Using F#? 3
Who Is This Book For? 4
What's Next? 5

CHAPTER 2 How to Obtain, Install, and Use F# 7

Obtaining F# 7
Installing F# on Windows 7
Installing F# on Linux 8
Using F# in Different Ways 9
Installing the Software Used in This Book 13
Summary 14

CHAPTER 3 Functional Programming 15

Identifiers 15
Keywords 16
Literals 17
Values and Functions 19
Scope 22
Recursion 25
Anonymous Functions 26
Operators 26

Lists 28
List Comprehensions 31
Control Flow 33
Types and Type Inference 34
Pattern Matching 37
Defining Types 42
Exceptions and Exception Handling 49
Lazy Evaluation 50
Summary 53

CHAPTER 4 Imperative Programming 55

The unit Type 55
The mutable Keyword 57
Defining Mutable Record Types 59
The ref Type 60
Arrays 62
Array Comprehensions 66
Control Flow 66
Loops over Comprehensions 69
Calling Static Methods and Properties from .NET Libraries 69
Using Objects and Instance Members from .NET Libraries 71
Using Indexers from .NET Libraries 73
Working with Events from .NET Libraries 74
Pattern Matching over .NET Types 76
The |> Operator 78
Summary 80

CHAPTER 5 Object-Oriented Programming 81

Casting 81
Type Tests 83
Type Annotations for Subtyping 83
Records As Objects 85
F# Types with Members 88
Object Expressions 90
Defining Interfaces 93
Implementing Interfaces 93
Classes, Fields, and Explicit Constructors 95
Implicit Class Construction 98
Classes and Inheritance 99
Classes and Methods 101

Accessing the Base Class . . . 103
Properties and Indexers . . . 104
Classes and Static Methods . . . 106
Overriding Methods from Non-F# Libraries . . . 107
Defining Delegates . . . 108
Structs . . . 108
Enums . . . 109
Summary . . . 110

CHAPTER 6 Organizing, Annotating, and Quoting Code . . . 111

Modules . . . 111
Namespaces . . . 112
Opening Namespaces and Modules . . . 114
Giving Namespaces and Modules Aliases . . . 115
Signature Files . . . 116
Module Scope . . . 116
Module Execution . . . 117
Optional Compilation . . . 119
Comments . . . 120
Doc Comments . . . 120
Custom Attributes . . . 122
Quoted Code . . . 124
Summary . . . 127

CHAPTER 7 The F# Libraries . . . 129

Libraries Overview . . . 129
The Native F# Library FSLib.dll . . . 130
The ML Compatibility Library MLLib.dll . . . 131
The Native F# Library FSLib.dll . . . 132
The Microsoft.FSharp.Core.Operators Module . . . 132
The Microsoft.FSharp.Reflection Module . . . 137
The Microsoft.FSharp.Collections.Seq Module . . . 139
The Microsoft.FSharp.Core.Enum Module . . . 149
The Microsoft.FSharp.Text.Printf Module . . . 151
The Microsoft.FSharp.Control.IEvent Module . . . 154
The Microsoft.FSharp.Math Namespace . . . 156
The ML Compatibility Library MLLib.dll . . . 160
The Microsoft.FSharp.Compatibility.OCaml.Pervasives Module . . . 160
The Microsoft.FSharp.Compatibility.OCaml.Arg Module . . . 164
Summary . . . 166

CHAPTER 8 User Interfaces 167

Introducing WinForms 167
Drawing WinForms 168
Working with Controls in WinForms 175
Using the Visual Studio Form Designer's Forms in F# 179
Working with WinForms Events and the IEvent Module 182
Creating New Forms Classes 186
Introducing ASP.NET 2.0 188
Creating an IHttpHandler 189
Working with ASP.NET Web Forms 192
Introducing Windows Presentation Foundation 195
Introducing Windows Presentation Foundation 3D 197
Summary 207

CHAPTER 9 Data Access 209

The System.Configuration Namespace 209
The System.IO Namespace 212
The System.Xml Namespace 214
ADO.NET 216
The EntLib Data Access Block 221
Data Binding 223
Data Binding and the DataGridView 225
ADO.NET Extensions 228
Introducing LINQ 232
Using LINQ to XML 234
Using LINQ to SQL 236
Summary 238

CHAPTER 10 Distributed Applications 239

Networking Overview 239
Using TCP/IP Sockets 240
Using HTTP 249
Calling Web Services 250
Creating Web Services 255
Windows Communication Foundation 261
Hosting WCF Services 265
Summary 270

CHAPTER 11 Language-Oriented Programming 271

What Is Language-Oriented Programming? 271
Data Structures As Little Languages 271
A Data Structure–Based Language Implementation 272
Metaprogramming with Quotations 278
An Arithmetic-Language Implementation 280
The Abstract Syntax Tree 281
Tokenizing the Text: Fslex 281
Generating a Parser: Fsyacc 284
Using the Parser 286
Interpreting the AST 288
Compiling the AST 289
Compilation vs. Interpretation 293
Summary 297

CHAPTER 12 The F# Tool Suite and .NET Programming Tools 299

Using Useful fsc.exe Command-Line Switches 299
Basic Compiler Switches 300
Compiler Optimization Switches 300
Compiler Warning Switches 302
Compiler Target Switches 303
Signing and Versioning Switches 303
Printing the Interface Switches 304
Adding Resources Switches 304
Generating HTML Switches 305
CLI Version Switches 306
Compilation Details Switches 307
Statically Linking Switches 307
Using fsi.exe Effectively 307
fsi.exe Commands 307
Controlling the fsi.exe Environment 308
fsi.exe Command-Line Switches 310
Using the Source Directory Macro 311
Writing NUnit Tests 311
Using Assembly Browsers 313
Using Debugging Tools 315

Using Profiling Tools 316
Ntimer 317
Perfmon and Performance Counters 317
NProf 320
CLR Profiler 321
Summary 322

CHAPTER 13 Compatibility and Advanced Interoperation 323

Calling F# Libraries from C# 323
Returning Tuples 324
Exposing Functions That Take Functions As Parameters 324
Using Union Types 326
Using F# Lists 329
Defining Types in a Namespace 329
Defining Classes and Interfaces 330
Using F# with the .NET Framework Versions 1 and 1.1 332
Calling Using COM Objects 335
Using P/Invoke 336
Using Inline IL 338
Using F# from Native Code via COM 340
Summary 343

INDEX 345

Foreword

A new language needs a simple and clear introductory book that makes it accessible to a broad range of programmers. In *Foundations of F#*, Robert Pickering has captured the essential elements that the professional programmer needs to master in order to get started with F# and .NET. As the designer of F#, I am thrilled to see Robert take up the challenge of presenting F# in a way that is accessible to a wide audience.

F# combines the simplicity and elegance of typed functional programming with the strengths of the .NET platform. Although typed functional programming is relatively new to many programmers and thus requires some learning, in many ways it makes programming simpler. This is mainly because F# programs tend to be built from compositional, correct foundational elements, and type inference makes programs shorter and clearer. Robert first introduces the three foundational paradigms of F#: functional programming, imperative programming, and object-oriented programming, and he shows how F# lets you use them in concert. He then shows how this multiparadigm approach can be used in conjunction with the .NET libraries to perform practical programming tasks such as GUI implementation, data access, and distributed programming. He then introduces some of the particular strengths of F# in the area of "language-oriented" programming.

F# is a practical language, and Robert has ensured that the reader is well equipped with information needed to use the current generation of F# tools well. Many computer professionals first encounter functional programming through a short section of the undergraduate curriculum and often leave these courses uncertain about the real-world applicability of the techniques they have been taught. Similarly, some people encounter functional programming only in its purest forms and are uncertain whether it is possible to combine the elements of the paradigm with other approaches to programming and software engineering. Robert has helped remove this uncertainty: typed functional programming is practical, easy to learn, and a powerful addition to the .NET programming landscape.

F# is also a research language, used in part to deliver recent advances in language design, particularly those that work well with .NET. It combines a stable and dependable base language with more recent extensions. Robert's book describes F# 2.0, the latest release of the language at the time of writing. The rest of the F# team and I are very grateful to Robert's many suggestions, and the language has been greatly improved through this. I hope you enjoy reading this book as much as I enjoyed being its technical reviewer.

Don Syme
Cambridge, UK

About the Author

ROBERT PICKERING was born in Sheffield, in the north of England, but a fascination with computers and the "madchester" indie music scene led him to cross the Pennines and study computer science at the University of Manchester.

After finishing his degree, he moved to London to catch the tail end of the dot-com boom working at marchFirst; then he moved to Avanade to do some more serious work. At Avanade, he specialized in creating enterprise applications using the .NET Framework, and he got the chance to work on projects in Denmark, Holland, and Belgium; he finally settled in Paris, France, where he lives now with his wife and their two cats. He has been writing about F# almost since it began, and the F# wiki on his `http://www.strangelights.com` web site is among the most popular F# web sites.

He currently works for LexiFi, which is an innovative ISV that specializes in software for analyzing and processing complex financial derivatives—products such as swaps and options. LexiFi has pioneered the use of functional programming in finance in order to develop a rigorous formalism for representing financial instruments.

About the Technical Reviewer

DON SYME is a researcher at Microsoft Research, Cambridge. Born in Toowoomba, Australia, his love for programming was sparked by family and teachers at age 10. He studied at the Australian National University and the University of Cambridge, becoming an expert in the application of automated proof to real-world problems, participating in the team that formally verified the correctness of aspects of the Intel Pentium IV floating-point circuits. Joining Microsoft in 1998, he saw the opportunity to enhance and transform the design of the .NET Framework by including elements of functional programming, beginning with the addition of generics to C# 2.0 and the .NET common language runtime, a project he saw through to completion in Visual Studio 2005. In 2003 he began the design and implementation of F#, which has now become the premier functional programming language for the .NET Framework. He continues to be a driving force in the design, implementation, and enhancement of the language.

Acknowledgments

If there is one person I feel I should be acknowledging, it is Jim Huddleston, the book's editor. Jim was there from the book's beginning. He helped me get it commissioned, he aided me in working out the contents, he gave me much encouragement and constructive criticism, and he showed great skill as an editor. Sadly, Jim died on February 25, 2007, just as the book was entering its final stages of production. Even though I never met Jim in person, never even talked to him on the telephone, I feel a real sense of loss knowing he is gone. My thoughts are with his family at this very sad time, and I'm very disappointed that I never got to meet him in person and that he never saw the finished book.

Sadly, Jim's was not the only death that coincided with the closing stages of writing this book. On March 28, 2007, my uncle Gordon lost his battle with cancer. He was a great lover of life, with many interests. He was a maths teacher who was an avid *New Scientist* reader with a deep interest in maths and science and a passion for music; he was a talented musician who played many gigs across Orkney and the north of Scotland. He will be greatly missed by me and all my family.

I feel very lucky to have worked on the project with my technical reviewer, Don Syme, who went above and beyond the cause by contributing many ideas to the book, helping improve the implementations of many of the samples, and giving me much encouragement. I'd also like to thank all the Apress staff who took part in creating this book, especially Elizabeth Seymour, Kim Wimpsett, and Laura Cheu.

I'd also like to thank Don in another capacity, as the creator and developer of F#, along with James Margetson and all the others at Microsoft Research, Cambridge, who worked on F#. Specifically, I'd like to thank them for their hard work on the compiler, and I'd like to let them know that their quick response times to bugs and queries have been very much appreciated. I'd also like to thank all the F# community, in particular Christopher J. Barwick (a.k.a. optionsScalper), who did so much to boost the F# community when he created the hubFS (`http://cs.hubfs.net`).

I'd like to thank all the people who had to put up with me while I wrote this book. My family: Mum, Dad, and Sister had to put up with me sneaking off to write whenever I went to visit them. Also, my work colleagues often suffered grumpy mornings after late nights of F# hacking and writing: Arnaud, Aurélie, Baptiste, Buuloc, Daniel, Dennis, Emmanuel, Fabrice, François, Frederik, Guillaume, Ibrahima, Jean-Marc, Laurent, Lionel, Oussama, Patrice, Philippe, Regis, Sebastien J., Sebastien P., Stefaan, Stefany, and Stephane—I thank you all. Last but by no means least, I'd like to thank my wife, Susan, for all the help and support she has given; without her understanding, this book could never have happened.

Preface: The Story of the Book

In 2003 I was looking for a way to process IL, the intermediate language into which all .NET languages are compiled. At the time, .NET was fairly new, and there weren't a lot of options for doing this. I quickly realized that the best option at the time, and probably still today, was an API called Abstract IL (AbsIL). AbsIL was written in a language called F#, and I decided to use this language to write a small wrapper around AbsIL so I could extract the information I needed from a DLL in a form more usable from C#. But a funny thing happened while writing the wrapper: even though in those days writing F# was a little hard going because the compiler was far from polished, I found that I actually enjoyed programming in F# so much that when I had finished the wrapper, I didn't want to go back to C#. In short, I was hooked.

At the time I was working as a consultant, so I needed to regularly check out new technologies and their APIs; therefore, I got to do all my experimentation using F#. At the same time, people were talking about a new way to communicate on the Web, and a new word was about to enter the English language: *blog*. I decided I should have a blog because anyone who was anyone in technology seemed to have one, so I created `http://www.strangelights.com` (where my blog can still be found to this today). This was later followed by a wiki about F#, which can also be found at `http://www.strangelights.com` and which continues to be very popular.

My job meant I had to do a lot of traveling, so this meant quite a lot of time in hotel rooms or on trains and planes, and I came to view this time as time to try out stuff in F#. So, I ended up exchanging quite a lot emails with Don Syme, and then eventually we met. We went for a beer in the pub where Watson and Crick went after they first pieced together the structure of DNA. Will people talk about the pub were Syme and Pickering first met years from now? Errrm, perhaps not. Anyway, all this led me to the point where I was wondering what I should do with my newfound knowledge of F# and functional programming. About this time a guy called Jim Huddleston mailed the F# mailing list and asked whether anyone would like to write a book about F#. Well, I just couldn't help myself—it sounded like the job for me.

So, with much help and encouragement from Jim, I started writing the book. Some of it was written in Paris where I was living on the weekends, some of it was written in Brussels were I was working during the week, and much of it was written while I was traveling between the two on the Thalys (the high-speed train between France and Belgium). A little of it was written as far north as the Orkney Islands in Scotland while visiting my aunt and uncle, and a little of the reviewing was done while meeting my in-laws in New Zealand. Finally, thanks to the contacts I made while writing the book, I got a new job working for the prestigious ISV LexiFi.

It has been great fun watching the language evolve over time and turn from the beginnings of a language into the fully fledged and highly usable language you see today. I hope reading this book changes your life as much as writing it has changed mine.

CHAPTER 1

Introduction

This introductory chapter will address some of the major questions you may have about F# and functional programming.

What Is Functional Programming?

Functional programming (FP) is the oldest of the three major programming paradigms. The first FP language, IPL, was invented in 1955, about a year before Fortran. The second, Lisp, was invented in 1958, a year before Cobol. Both Fortran and Cobol are imperative (or procedural) languages, and their immediate success in scientific and business computing made imperative programming the dominant paradigm for more than 30 years. The rise of the object-oriented (OO) paradigm in the 1970s and the gradual maturing of OO languages ever since have made OO programming the most popular paradigm today.

Despite the vigorous and continuous development of powerful FP languages (SML, OCaml, Haskell, and Clean, among others) and FP-like languages (APL and Lisp being the most successful for real-world applications) since the 1950s, FP remained a primarily academic pursuit until recently. The early commercial success of imperative languages made it the dominant paradigm for decades. Object-oriented languages gained broad acceptance only when enterprises recognized the need for more sophisticated computing solutions. Today, the promise of FP is finally being realized to solve even more complex problems—as well as the simpler ones.

Pure functional programming views all programs as collections of functions that accept arguments and return values. Unlike imperative and object-oriented programming, it allows no side effects and uses recursion instead of loops for iteration. The functions in a functional program are very much like mathematical functions because they do not change the state of the program. In the simplest terms, once a value is assigned to an identifier, it never changes, functions do not alter parameter values, and the results that functions return are completely new values. In typical underlying implementations, once a value is assigned to an area in memory, it does not change. To create results, functions copy values and then change the copies, leaving the original values free to be used by other functions and eventually be thrown away when no longer needed. (This is where the idea of garbage collection originated.)

The mathematical basis for pure functional programming is elegant, and FP therefore provides beautiful, succinct solutions for many computing problems, but its stateless and recursive nature makes the other paradigms convenient for handling many common programming tasks. However, one of F#'s great strengths is that you can use multiple paradigms and mix them to solve problems in the way you find most convenient.

Why Is Functional Programming Important?

When people think of functional programming, they often view its statelessness as a fatal flaw, without considering its advantages. One could argue that since an imperative program is often 90 percent assignment and since a functional program has no assignment, a functional program could be 90 percent shorter. However, not many people are convinced by such arguments or attracted to the ascetic world of stateless recursive programming, as John Hughes pointed out in his classic paper "Why Functional Programming Matters":

> *The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.*
>
> John Hughes, Chalmers University of Technology
> (`http://www.math.chalmers.se/~rjmh/Papers/whyfp.html`)

To see the advantages of functional programming, you must look at what FP permits, rather than what it prohibits. For example, functional programming allows you to treat functions themselves as values and pass them to other functions. This might not seem all that important at first glance, but its implications are extraordinary. Eliminating the distinction between data and function means that many problems can be more naturally solved. Functional programs can be shorter and more modular than corresponding imperative and object-oriented programs.

In addition to treating functions as values, functional languages offer other features that borrow from mathematics and are not commonly found in imperative languages. For example, functional programming languages often offer *curried functions*, where arguments can be passed to a function one at a time and, if all arguments are not given, the result is a residual function waiting for the rest of its parameters. It's also common for functional languages to offer type systems with much better "power-to-weight ratios," providing more performance and correctness for less effort.

Further, a function might return multiple values, and the calling function is free to consume them as it likes. I'll discuss these ideas, along with many more, in detail and with plenty of examples, in Chapter 3.

What Is F#?

Functional programming is the best approach to solving many thorny computing problems, but pure FP isn't suitable for general-purpose programming. So, FP languages have gradually embraced aspects of the imperative and OO paradigms, remaining true to the FP paradigm but incorporating features needed to easily write any kind of program. F# is a natural successor on this path. It is also much more than just an FP language.

Some of the most popular functional languages, including OCaml, Haskell, Lisp, and Scheme, have traditionally been implemented using custom runtimes, which leads to problems such as lack of interoperability. F# is a general-purpose programming language for .NET, a general-purpose runtime. F# smoothly integrates all three major programming paradigms. With F#, you can choose whichever paradigm works best to solve problems in the most effective way. You can do pure FP, if you're a purist, but you can easily combine functional,

imperative, and object-oriented styles in the same program and exploit the strengths of each paradigm. Like other typed functional languages, F# is strongly typed but also uses inferred typing, so programmers don't need to spend time explicitly specifying types unless an ambiguity exists. Further, F# seamlessly integrates with the .NET Framework base class library (BCL). Using the BCL in F# is as simple as using it in C# or Visual Basic (and maybe even simpler).

F# was modeled on Objective Caml (OCaml), a successful object-oriented FP language, and then tweaked and extended to mesh well technically and philosophically with .NET. It fully embraces .NET and enables users to do everything that .NET allows. The F# compiler can compile for all implementations of the Common Language Infrastructure (CLI), it supports .NET generics without changing any code, and it even provides for inline Intermediate Language (IL) code. The F# compiler not only produces executables for any CLI but can also run on any environment that has a CLI, which means F# is not limited to Windows but can run on Linux, Apple Mac OS X, and OpenBSD. (Chapter 2 covers what it's like to run F# on Linux.)

The F# compiler can be integrated into Visual Studio, supporting IntelliSense expression completion and automatic expression checking. It also gives tooltips to show what types have been inferred for expressions. Programmers often comment that this really helps bring the language to life.

F# was invented by Dr. Don Syme and is now the product of a small but highly dedicated team he heads at Microsoft Research (MSR) in Cambridge, England. However, F# is not just a research or academic language. It is used for a wide variety of real-world applications, whose number is growing rapidly.

Although other FP languages run on .NET, F# has established itself as the de facto .NET functional programming language because of the quality of its implementation and its superb integration with .NET and Visual Studio.

No other .NET language is as easy to use and as flexible as F#!

Who Is Using F#?

F# has a strong presence inside Microsoft, both in MSR and throughout the company as a whole. Ralf Herbrich, coleader of MSR's Applied Games Group, which specializes in machine learning techniques, is typical of F#'s growing number of fans:

> *The first application was parsing 110GB of log data spread over 11,000 text files in over 300 directories and importing it into a SQL database. The whole application is 90 lines long (including comments!) and finished the task of parsing the source files and importing the data in under 18 hours; that works out to a staggering 10,000 log lines processed per second! Note that I have not optimized the code at all but written the application in the most obvious way. I was truly astonished as I had planned at least a week of work for both coding and running the application.*
>
> *The second application was an analysis of millions of feedbacks. We had developed the model equations and I literally just typed them in as an F# program; together with the reading-data-from-SQL-database and writing-results-to-MATLAB-data-file the F# source code is 100 lines long (including comments). Again, I was astonished by the running time; the whole processing of the millions of data items takes 10 minutes on a*

> *standard desktop machine. My C# reference application (from some earlier tasks) is almost 1,000 lines long and is no faster. The whole job from developing the model equations to having first real world data results took 2 days.*
>
> Ralf Herbrich, Microsoft Research
> (`http://blogs.msdn.com/dsyme/archive/2006/04/01/566301.aspx`)

F# usage outside Microsoft is also rapidly growing. I asked Chris Barwick, who runs hubFS (`http://cs.hubFS.net`), a popular web site dedicated to F#, about why F# was now his language of choice, and he said this:

> *I've been in scientific and mathematics computing for more than 14 years. During that time, I have waited and hoped for a platform that would be robust in every manner. That platform has to provide effective tools that allow for the easy construction and usage of collateral and that makes a scientific computing environment effective. .NET represents a platform where IL gives rise to consistency across products. F# is* the *language that provides for competent scientific and mathematical computing on that platform. With these tools and other server products, I have a wide range of options with which to build complex systems at a very low cost of development and with very low ongoing costs to operate and to improve. F# is the cornerstone needed for advanced scientific computing.*
>
> Christopher J. Barwick, JJB Research (private email)

Finally, I talked to Jude O'Kelly, a software architect at Derivatives One, a company that sells financial modeling software, about why Derivatives One used F# in its products:

> *We tested our financial models in both C# and F#; the performance was about the same, but we liked the F# versions because of the succinct mathematical syntax. One of our problems with F# was the lack of information; we think this book improves this situation enormously.*
>
> Jude O'Kelly, Derivatives One (private email)

Who Is This Book For?

This book is aimed primarily at IT professionals who want to get up to speed quickly on F#. A working knowledge of the .NET Framework and some knowledge of either C# or Visual Basic would be nice, but it's not necessary. All you really need is some experience programming in any language to be comfortable learning F#.

Even complete beginners, who've never programmed before and are learning F# as their first computer language, should find this book very readable. Though it doesn't attempt to teach introductory programming per se, it does carefully present all the important details of F#.

What's Next?

This book teaches F#, by example, as a compiled language rather than a scripting language. By this I mean most examples are designed to be compiled with the `fsc.exe` compiler, either in Visual Studio or on a command line, rather than executed interactively with `fsi.exe`, the F# interactive environment. In reality, most examples will run fine either way.

Chapter 2 gives you just enough knowledge about setting up an F# development environment to get you going.

Chapters 3, 4, 5, and 6 cover the core F# syntax. I deliberately keep the code simple, because this will give you a better introduction to how the syntax works.

Chapter 7 looks at the core libraries distributed with F# to introduce you to their flavor and power, rather than to describe each function in detail. The F# online documentation (`http://research.microsoft.com/fsharp/manual/namespaces.html`) is the place to get the details.

Then you'll dive into how to use F# for the bread-and-butter problems of the working programmer. Chapter 8 covers user interface programming, Chapter 9 covers data access, and Chapter 10 covers how applications can take advantage of a network.

The final chapters take you through the topics you really need to know to master F#. Chapter 11 looks at support for creating little languages or domain-specific languages (DSLs), a powerful and very common programming pattern in F#. Chapter 12 covers the tools you can use to debug and optimize F# programs. Finally, Chapter 13 explores advanced interoperation issues.

CHAPTER 2

How to Obtain, Install, and Use F#

This chapter is designed to get you up and running with F# as quickly as possible. You'll look at how to obtain F#, how to install it on both Windows and Linux, and how to use the compiler in various ways. I'll also discuss what version of software the examples in this book were tested with and what extra software you might need to install.

Obtaining F#

You can download F# from the Microsoft Research F# Download page at `http://research.microsoft.com/fsharp/release.aspx`. The package includes various versions of the compiler, which are compatible with different versions of the CLR, `fsi.exe` (the F# interactive console), some F#-based parsing tools, the F# base class libraries, the F# documentation, and some F# samples.

Installing F# on Windows

Installing F# on Windows is straightforward. You need to be running as an account with system administrator privileges to install F#. Simply unzip the contents of the F# distribution to a temporary location, and then run the `InstallFSharp.msi` package, which is in the root of the distribution. The `.msi` should work whether or not Visual Studio 2003 or Visual Studio 2005 is installed.

If you'd prefer not to use an `.msi`, you can compile from the command line simply by unzipping to your preferred location and running `alternative-install.bat`, which will install the F# runtime libraries into the global assembly cache (GAC). You can also use this batch file to install F# against the Shared Source Common Language Infrastructure (SSCLI), more commonly known as Rotor, by using the `-sscli` command-line switch.

■ **Note** The SSCLI is a compressed archive of the source code for a working implementation of the ECMA CLI and the ECMA C# language specifications. This implementation builds and runs on Windows XP, and you can also compile and run it on other operating systems such as Linux or Mac OS X. This implementation is ideal if you really want to get under the covers and see what's going on; however, you may find it more difficult to use than .NET, so you're probably best off sticking with .NET while reading this book.

If you use the `alternative-install.bat` batch file, Visual Studio integration will not be installed. For installing Visual Studio integration, two further batch files are available, `alternative-install-vs2003.bat` and `alternative-install-vs2005.bat`. Please note that at the time of this writing the free Express Editions of Visual Studio do not support plug-ins, so you cannot use F# integration with them.

Installing F# on Linux

It's difficult to write a simple guide to installing F# on Linux, because there are so many different distributions of Linux and so many ways you can configure them. The following are the steps that worked well for me running SUSE Linux on a single computer. I performed all these steps as the `root` account.

1. Install Mono using the packages provided with the SUSE Linux distribution; you can find these by searching for *mono* and then *sharp* in the Install Software dialog box available from the Computer menu.
2. Unpack the F# distribution, and copy the resulting files to `/usr/lib/fsharp`.
3. In the `/usr/lib/fsharp` directory, run `chmod +x install-mono.sh`.
4. Run the `dos2unix` tool on the text file `install-mono.sh`.
5. Still in the `/usr/lib/fsharp` directory, run the command `sh install-mono.sh`.

After performing those steps, I was able to use F# from any account from the command line by running `mono /usr/lib/fsharp/bin/fsc.exe`, followed by the command-line options. Obviously, this was inconvenient to run every time, so I created a shell script file in `/usr/bin` and as `fsc`:

```
#!/bin/sh
exec /usr/bin/mono $MONO_OPTIONS /usr/lib/fsharp/bin/fsc.exe "$@"
```

I then ran `chmod +x fsc` to give users permission to execute it. After this, running the F# compiler was as simple as typing `fsc` at the command line. The F# interactive compiler, `fsi.exe`, will also run under Linux, but on the installation I used at the time of this writing, I needed to use the `--no-gui` switch. The shell script for this is as follows:

```
#!/bin/sh
exec /usr/bin/mono $MONO_OPTIONS /usr/lib/fsharp/bin/fsi.exe --no-gui "$@"
```

■ **Note** I used SUSE Linux, available from `http://www.novell.com/linux/`, because I found it installed smoothly on real and virtual machines, requiring very little manual setup.

Using F# in Different Ways

F# programs are just text files, so you can use any text editor to create them. Just save your program with the extension `.fs`, and use `fsc.exe` to compile them. For example, if you had the following program in the text file `helloworld.fs`:

```
#light
print_endline "Hello World"
```

you could just run `fsc.exe helloworld.fs` to compile your program into `helloworld.exe`, which would output the following to the console:

```
Hello World
```

In my opinion, the easiest and quickest way to develop F# programs is in Visual Studio in conjunction with the F# interactive compiler (see Figure 2-1). You can type F# programs into the text editor, taking advantage of syntax highlighting and IntelliSense code completion; compile them into executables; and debug them interactively by setting breakpoints and pressing F5. Also, you can execute parts of your code interactively using F# interactive. Just highlight the code you want to execute, and press Alt+Enter; F# interactive will execute the code and show the results. This is great for testing snippets individually.

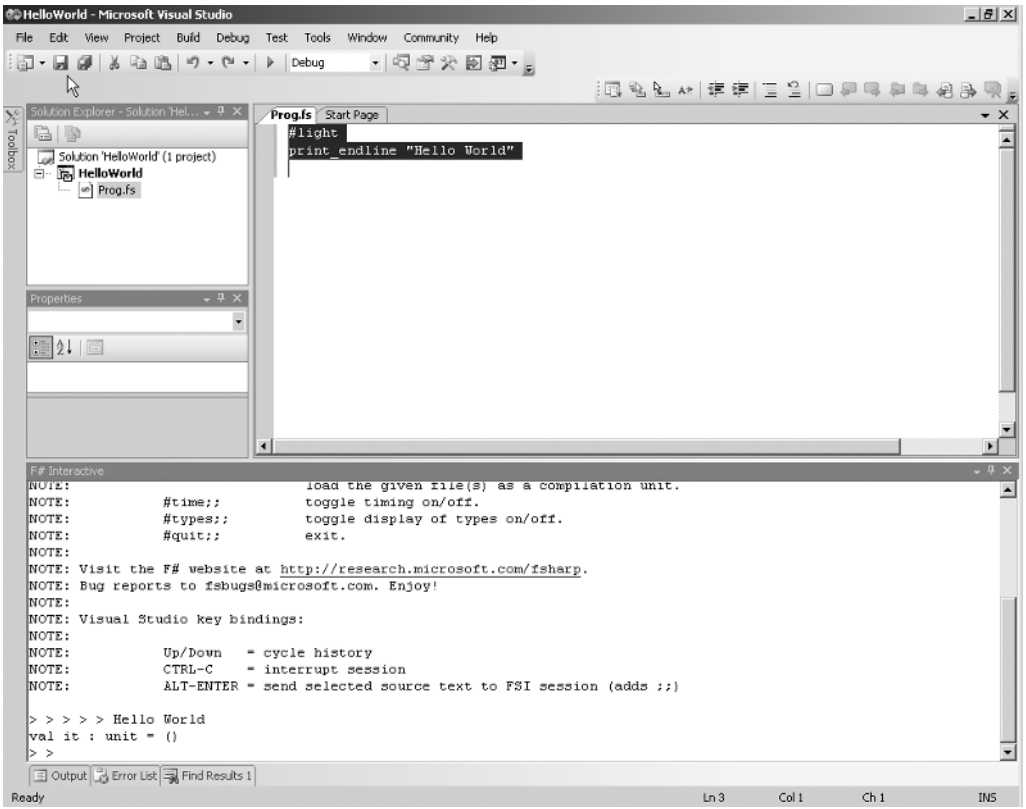

Figure 2-1. *Visual Studio hosting F# interactive*

■ **Note** If you are not convinced you want to invest in a copy of Visual Studio, trial versions of this software are available at `https://www.tryvs2005.com`.

If you prefer, you can type your programs into the F# interactive console directly when it's running in stand-alone mode, as shown in Figure 2-2.

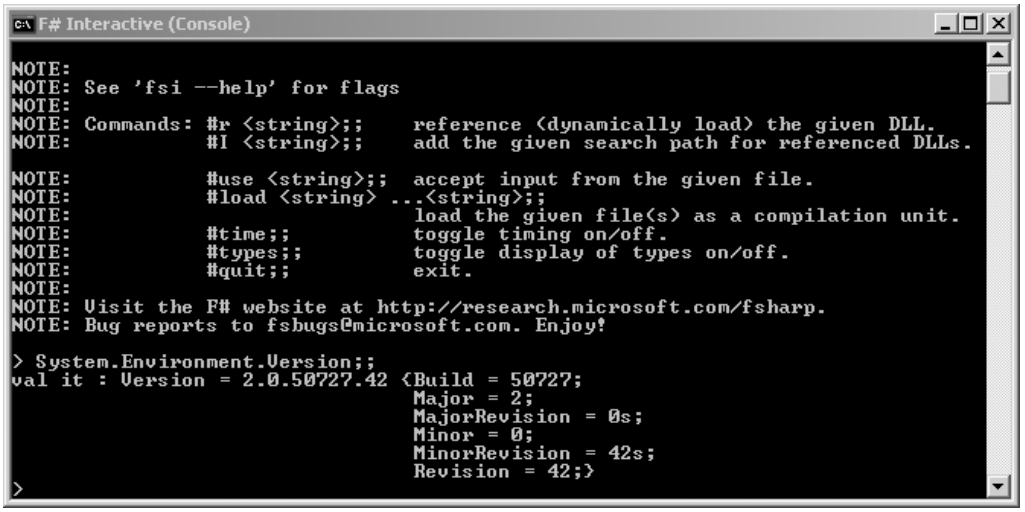

Figure 2-2. *The F# interactive console running in stand-alone mode*

When you use the interactive console, you type the code you want; then when you've completed a section, you use two semicolons (`;;`) to indicate that the compiler should compile and run it.

F# interactive responds to commands in two ways. If you bind a value to an identifier, it prints the name of the identifier and its type. So, typing the following into F# interactive:

```
> let i = 1 + 2;;
```

gives the following:

```
val i : int
```

However, if you just type a value into F# interactive, it will respond slightly differently. Typing the following into F# interactive:

```
> 1 + 2;;
```

gives the following:

```
val it : int = 3
```

This means the value has been bound to a special identifier, called `it`, that is available to other code within the F# interactive session. When any expression is evaluated at the top level, its value is also printed, after the equals sign; note the `3` in the previous example. As you get to know `fsi.exe` and F# in general, using F# interactive will become more and more useful for debugging programs and finding out how they work. (I discuss values, identifiers, and types in more detail in Chapter 3.)

You can get code completions by pressing Tab. I find this mode of working useful in testing short programs by copying and pasting them into the console or for checking properties on existing libraries. For example, in Figure 2-2 I checked the `System.Environment.Version` property. However, I find this mode inconvenient for creating longer programs since it's difficult to store the programs once they're coded; they have to be copied and pasted from the console. Using Visual Studio, even if you don't intend to just run them interactively, you can still easily execute snippets with Alt+Enter.

If you save your program with the `.fsx` extension instead of the `.fs` extension, you can run your programs interactively by right-clicking them and selecting the Run with F# Interactive menu option, as shown in Figure 2-3. This scripting style of programming is great for creating small programs to automate repetitive tasks. This way your program's configuration, such as the file paths it uses, can be stored inside regular strings in the program and can be quickly edited by the programmer using any text editor as needed.

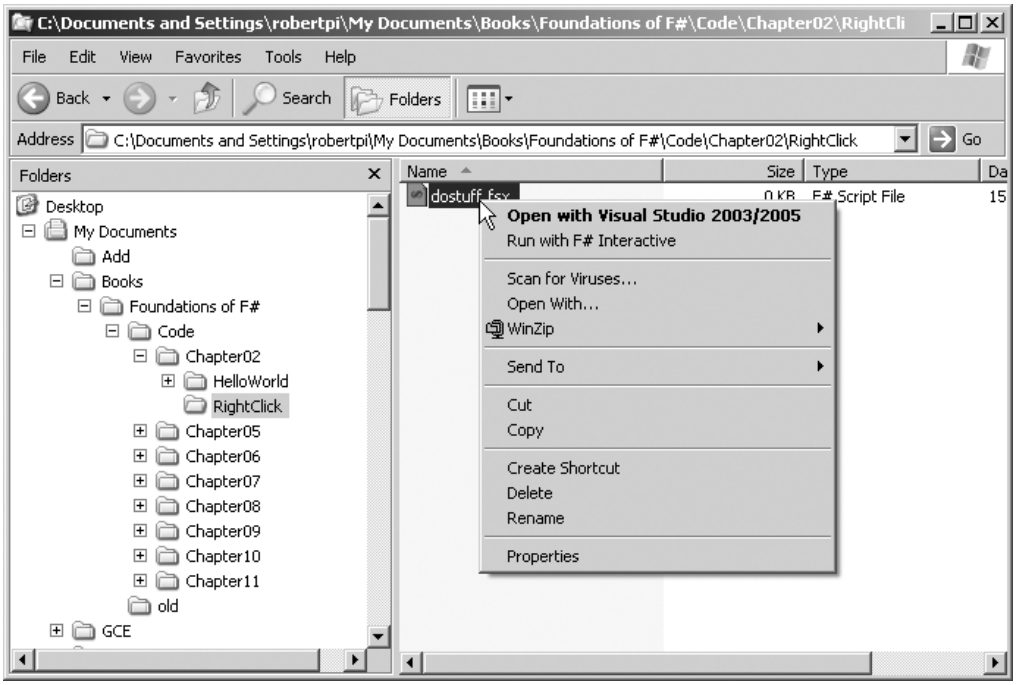

Figure 2-3. *Running an F# script by right-clicking it*

You can find more information about the F# programming tools and general programming tools for .NET in Chapter 11.

Installing the Software Used in This Book

The code in this book will focus on using `fsc.exe`, rather than `fsi.exe`. This is because although `fsi.exe` is great for testing code, running simple scripts, and running experiments, I believe `fsc.exe` is more useful for producing finished software. Since there's little difference between the syntax and the commands, most examples will work with little or no adaptation in `fsi.exe`, and I'll warn you when any changes are necessary.

All the samples in this book were created using .NET 2.0 running on Windows XP Professional. If you're using .NET 1.0 or 1.1, you'll experience problems with many of the samples because quite a few of them use classes and methods from the .NET 2.0 base class library (BCL) that aren't available in version 1.0 or 1.1.

The most common problem you will face when working with .NET 1.0 and 1.1 is that I use `System.Collections.Generic.List`, referred to as `ResizeArray` in F#, and `System.Collections.Generic.Dictionary`. You can always work around this by replacing these two classes with `System.Collections.ArrayList` and `System.Collections.Hashtable`, respectively. There may be other places where I use methods or classes not available in .NET 1.0 and 1.1, but generally you will be able to work around this with a little extra coding.

At the time of this writing, Mono shipped with its version of Framework 2.0, which the F# compiler targets by default; however, this was still in beta, with a production-quality version due to ship in mid-2007. A small subset of this book's examples has been tested on Mono 2.0, and the examples ran without problems.

A small number of examples use several other software libraries and packages. It's not necessary to immediately download and install all these software packages, but for specific examples, as listed in Table 2-1, you'll need to do this at some point.

Table 2-1. *Additional Software Used Within This Book*

Software	Used In	URL
.NET Framework 3.0	Chapter 8, Chapter 10	`http://www.microsoft.com/downloads/details.aspx?FamilyId=10CC340B-F857-4A14-83F5-25634C3BF043&displaylang=en`
SDK for .NET Framework 3.0	Chapter 8, Chapter 10	`http://www.microsoft.com/downloads/details.aspx?familyid=C2B1E300-F358-4523-B479-F53D234CDCCF&displaylang=en`
SQL Server 2005 Express Edition	Chapter 9	`http://msdn.microsoft.com/vstudio/express/sql/register/default.aspx`
SQL Server 2005 Samples	Chapter 9	`http://www.microsoft.com/downloads/details.aspx?familyid=E719ECF7-9F46-4312-AF89-6AD8702E4E6E&displaylang=en`
Microsoft .NET LINQ Preview (May 2006)	Chapter 9	`http://www.microsoft.com/downloads/details.aspx?familyid=1e902c21-340c-4d13-9f04-70eb5e3dceea&displaylang=en`
Windows Server 2003 Resource Kit Tools	Chapter 12	`http://www.microsoft.com/downloads/details.aspx?FamilyID=9d467a69-57ff-4ae7-96ee-b18c4790cffd&DisplayLang=en`
NUnit	Chapter 12	`http://www.nunit.org/index.php?p=download`

continued

Table 2-1. *Continued*

Software	Used In	URL
NProf	Chapter 12	`http://www.mertner.com/confluence/display/NProf/Home`
CLR Profiler for .NET 2.0	Chapter 12	`http://www.microsoft.com/downloads/details.aspx?familyid=a362781c-3870-43be-8926-862b40aa0cd0&displaylang=en`
Reflector	Chapter 12	`http://www.aisto.com/roeder/dotnet/`

Obviously, some of these links are a little long to type, so I've summarized them all at `http://strangelights.com/FSharp/Foundations/default.aspx/FSharpFoundations.Downloads` where I'll keep them updated.

Summary

This chapter described how to install and run F# and the different ways you can work with it. The following chapters will explain how to program with F#, starting in Chapter 3 with functional programming in F#.

CHAPTER 3

Functional Programming

You saw in Chapter 1 that pure functional programming treats functions as values, relies on recursion for looping, and does not allow changes to state. In this chapter, you'll survey the major language constructs of F# that support the functional programming paradigm.

Identifiers

Identifiers are the way you give names to values in F# so you can refer to them later in a program. You define an identifier using the keyword `let` followed by the name of the identifier, an equals sign, and an expression that specifies the value to which the identifier refers. An *expression* is any piece of code that represents a computation that will return a value. The following expression shows a value being assigned to an identifier:

```
let x = 42
```

To most people coming from an imperative programming background, this will look like a variable assignment. There are a lot of similarities, but there are key differences. In pure functional programming, once a value is assigned to an identifier, it never changes. This is why I will refer to them throughout this book as *identifiers* and not *variables*. You will see in the "Scope" section later in this chapter that, under some circumstances, you can redefine identifiers and that in imperative programming in F#, in some circumstances, the value of an identifier can change.

An identifier can refer to either a value or a function, and since F# functions are really values in their own right, this is hardly surprising. (I discuss this relationship in detail in the "Functions and Values" section later in this chapter.) This means F# has no real concept of a function name or parameter name; they are all just identifiers. You write a function definition the same way as a value identifier, except a function has two or more identifiers between the `let` keyword and the equals sign, as follows:

```
let raisePowerTwo x = x ** 2.0
```

The first identifier is the name of the function, `raisePowerTwo`, and the identifier that follows it is the name of the function's parameter, `x`.

Keywords

Most, if not all, programming languages have the concept of keywords. A *keyword* is a language token that the compiler reserves for special use. In F# you cannot use a keyword as an identifier name or a *type* name (I discuss types later in this chapter in "Defining Types"). The following are the F# keywords:

```
abstract        lsl
and             lsr
as              lxor
assert          match member
asr             mod
begin           module
class           mutable namespace
default         new
delegate        null
do              of
done            open
downcast        or
downto          override
else            rec
end             sig
exception       static
false           struct
finally         then
for             to
fun             true
function        try
if              type
in              val
inherit         when
inline          upcast
interface       while
land            with
lor
```