THE EXPERT'S VOICE® IN .NET

Beginning F#

Discover the elegance and power of Microsoft's functional programming language

Robert Pickering

Forewords by Don Syme and Chance Coble

Apress®

Beginning F#

Robert Pickering

Forewords by Don Syme and Chance Coble

Apress®

Beginning F#

ISBN-13 (pbk): 978-1-4302-2389-4

ISBN-13 (electronic): 978-1-4302-2390-0

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

President and Publisher: Paul Manning

Lead Editor: Jonathan Hassell

Technical Reviewer: Michael de la Maza

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Debra Kelly

Copy Editors: Patrick Meader and Vanessa Porter

Compositor: Lynn L'Heureux

Indexer: John Collin

Artist: April Milne

Cover Designer: Anna Ishchenko

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 e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

Patrick wishes to dedicate this book to his father – "Everyday I attempt to approach his level of logic and perfection. Rest in peace."

Jim dedicates this book to his dad – "I've learned a lot from you and enjoyed the company."

Contents at a Glance

Contents

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.

Don Syme
Designer of F#, Microsoft Research
Original foreword to Foundations of F# (2007)

When Microsoft introduced F#, the .NET community gained a new paradigm—functional programming. That was a welcome event for coders who'd avoided .NET because the existing languages were geared toward rapid line-of-business development. But whether or not you've stayed clear of .NET, if you'd like to know what functional programming can bring to your work, Beginning F# is an excellent place to start the adventure.

When the first edition was published in 2007, functional languages were just starting to break into the mainstream. As it turned out, Robert Pickering displayed some impressive intuition about the importance they'd take on. In the short time since, they've become hot in the world of software architecture, so I'm especially pleased to see a second edition now.

Pickering is one of the most experienced F# programmers outside Microsoft, with a tremendous amount to offer people who are curious about the language. I'm excited to see that in this book he shares his perspective on everything from basic F# program design to large-scale software architecture. You'll find topics as deep as domain-specific languages and concurrency with a functional language. *Beginning F#* is truly geared toward professionals looking for real-world returns from this programming language.

I value that real-world approach—it helps me, and I believe it will help others use functional programming for their day-to-day work. I also prize the book as a powerful ally when I make the case that functional programming isn't just for academics anymore—it's a skill that software developers in the trenches should master. I even used the original edition of *Beginning F#* to convince my boss of functional programming's legitimacy. Thanks to the examples inside this book, I've witnessed more than one person make the transition from functional-programming novice to daily F# programmer, and I sincerely believe you'll have that experience.

Chance Coble
Chief Architect
Blacklight Solutions, LLC

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, Robert moved to London to catch the tail end of the dot-com boom, then went on to specialize in creating enterprise applications using the .NET Framework. He has worked as both a consultant and an engineer for a software house. After working on projects in Denmark, Holland, Belgium, and Switzerland, he finally settled in Paris, France, where he lives with his wife and three cats. He has been writing about F# almost since its beginning, and the F# wiki on his strangelights.com web site is among the most popular F# web sites.

About the Technical Reviewer

Michael de la Maza solves hard problems, often by applying computational techniques. He holds a PhD in computer science from MIT and is a Certified ScrumMaster, Certified Scrum Practitioner, and an IEEE Senior Member. Previously, he was VP of Corporate Strategy at Softricity (acquired by Microsoft in 2006) and a co-founder of Inquira.

Acknowledgments

If there is one person who must be acknowledged, it is Jim Huddleston, the editor of the first edition of this book. Jim was there from the beginning. He helped me get it commissioned, he worked with me to figure out the contents, he gave me much-needed encouragement and constructive criticism, and his skillful editing helped me convey the information effectively. Sadly, Jim died on Sunday, 25th February 2007, just as the orginal book was entering its final stages of production.

I feel very lucky to have worked on this project with my technical reviewer, Michael de la Maza, and lucky as well to have worked with the technical reviewer of the first edition, Don Syme, who went above and beyond the cause by contributing many ideas to the original book; his influence can still be seen in this edition.

Don, of course, is the creator and developer of F#, and I'd like to thank him and all the other members of the small but dedicated F# team. Specifically, I'd like to thank them for their hard work on the compiler, and to let them know that their quick responses to bugs and queries were very much appreciated.

I'm also indebted to the entire F# community, in particular, to Stephan Tolksdorf, who was a great help with the FParsec examples; André van Meulebrouck, who sent me many corrections; and Chance Coble for his encouragement and excellent foreword. And I'm grateful to Chris Barwick (a.k.a. optionsScalper) for his continued work on hubFS (`http://cs.hubfs.net`).

Finally, I'd like to thank everyone at Apress who took part in creating this book.

A number of people had to put up with me while I wrote this book, and they deserve special thanks. This includes my family: Mum, Dad, and sister, who got used to me sneaking off to write whenever I went to visit them; my work colleagues when writing the original book: 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; the people who helped keep me distracted in Geneva: Amy, Angela, Armand, Carmen, Emma, Erika, Francisco, Giovanna, Jordi, Laurent, Mattias, Peter, and Sameera; and the people I'm working with on my current project: Charels, Francois, Kyrylo, and Stefan. Last but by no means least, heartfelt thanks to my wife, Susan, for all the help and support she has given. Without her understanding, this book could never have happened.

Preface

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 was an API called Abstract IL, AbsIL for short. 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 than with C#. But a funny thing happened while writing the wrapper: even though in those days writing F# was a little hard going as the compiler was far from polished, I found I actually enjoyed programming in F#, so much so that when I finished the wrapper, I didn't want to go back to C#. In short, I was hooked.

During this period, I was working as a consultant, so I needed to regularly check out new technologies and APIs, and I got to do all my experimenting with F#. At the same time, a new way to communicate on the Web was emerging, and a new word was about to enter the English language: *blog*. I decided I should have a blog because anyone who was any one in technology seemed to have one, so I created `strangelight.com`, where my blog can still be found today. I later created a wiki about F#, also at `strangelight.com`, which continues to be very popular.

My job meant I had to do a lot of traveling, so I spent quite a lot of time in hotel rooms or on trains and planes, and I came to view these occasions as time to try out stuff in F#. I ended up exchanging quite a lot e-mails with Don Syme, and eventually we met up. 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 wonder what I should do with my new-found knowledge of F# and functional programming. About this time, a guy named Jim Huddleston posted to the F# mailing list to ask if anyone would like to write a book about F#. Well, I just couldn't help myself—it sounded like the job for me and in May, 2007, "Foundations of F#" was published.

About half a year later, it was announced that F# would be productized and made available as part of Visual Studio 2010. This seemed too good an opportunity to miss so I signed up to write a new version of the book, with the ambition of documenting the language as it is in Visual Studio 2010. The result is the book you are holding in your hands.

It has been great fun watching F# evolve and turn from a rudimentary language into the fully fledged and highly usable tool you see today. I hope reading this book changes your life as much as writing it 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 continual development of powerful FP languages—SML, Objective Caml (OCaml), APL, and Clean, among others—and FP-like languages—Erlang, Lisp, and Haskell 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) that 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 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 is distributed with Visual Studio 2010 and is available as a plug-in for Visual Studio 2008. It supports IntelliSense expression completion and automatic expression checking. It also gives tool tips to show what types have been inferred for expressions. Programmers often comment that this really helps bring the language to life.

F# was first implemented by Dr. Don Syme at Microsoft Research (MSR) in Cambridge. The project has now been embraced by Microsoft Corporate in Redmond, and the implementation of the compiler and Visual Studio integration is now developed by a team located in both Cambridge and Redmond.

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 the following:

> *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.

Chris Barwick, JJB Research (private e-mail)

Finally, I talked to Chance Coble, a software architect, about what F# bought to his work.

F# has made its case to me over and over again. The first project I decided to try F# on was a machine vision endeavor, which would identify and extract fingerprints from submitted fingerprint cards and load them into a biometrics system. The project plan was to perform the fingerprint extraction manually, which was growing cumbersome, and the automation turned out to be a huge win (with very little code). Later we decided to include that F# work in a larger application that had been written in C#, and accomplished the integration with ease. Since then I have used F# in projects for machine learning, domain specific language design, 3D visualizations, symbolic analysis, and anywhere performance intensive data processing has been required. The ability to easily integrate functional modules into existing production scale applications makes F# not only fun to work with, but an important addition for project leads. Unifying functional programming with a mature and rich platform like .NET has opened up a great deal of opportunity.

Chance Coble, Chief Architect, Blacklight Solutions, LLC (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://msdn.microsoft.com/fsharp`) 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, Chapter 10 covers concurrency and parallelism, and Chapter 11 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 12 looks at support for creating little languages or domain-specific languages (DSLs), a powerful and very common programming pattern in F#. Chapter 13 covers parsing text, with an emphasis on using this as a front end for DSLs. Finally, Chapter 14 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 was used to test the examples in this book.

Obtaining F#

F# is now included in Visual Studio 2010 by default, so if you have this installed on your machine you may already have F# installed. If you have Visual Studio 2010 installed but you can't see F#, then you need to ensure that you installed the package. This can be done through the Add/Remove Programs or Programs section of the control panel (see Figure 2-1).

If you're not a Visual Studio user or would prefer to use Visual Studio 2008 rather than 2010, then you'll have to download the F# distribution separately. The best place to look for all F#-related information is the MSDN F# resource center at `http://msdn.microsoft.com /fsharp/`. A link to the compiler distribution is included in the top, left-hand corner of the F# resource center page. There are two versions—an MSI version, which will automatically install F# Visual Studio integration if Visual Studio is installed, and a ZIP version of the distribution, which is primarily targeted at non-Windows users. The package includes the compiler `fsc.exe`, as well as `fsi.exe` (the F# interactive console), some F#-based parsing tools, the F# base class libraries, the F# documentation, and some F# samples.

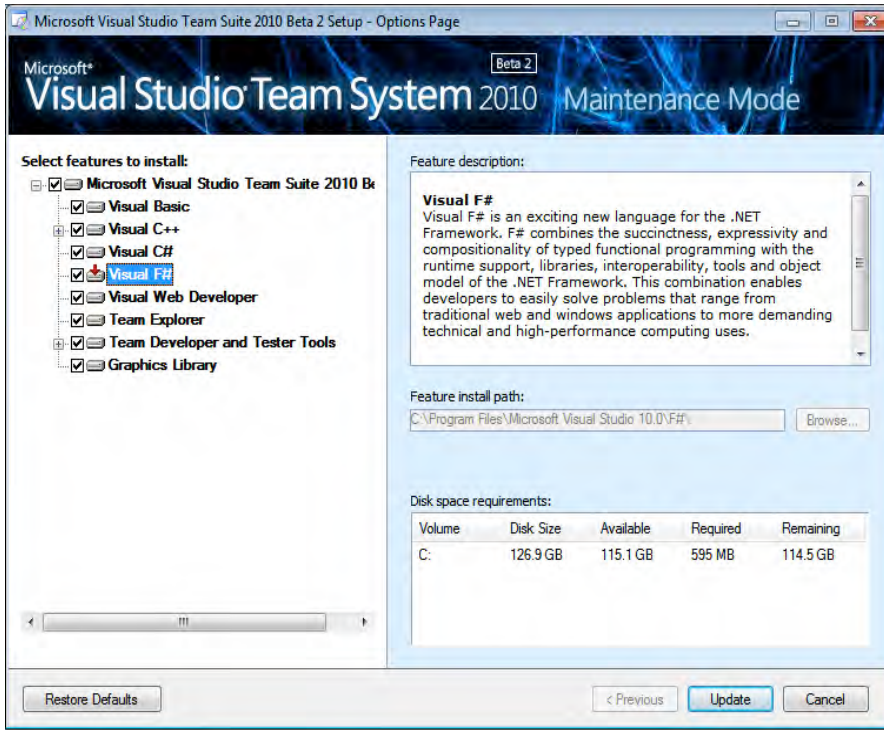

Figure 2-1. *Enabling F# in Visual Studio 2010*

Installing F# on Windows with Visual Studio 2008

This section is for people using an older version of Visual Studio. (As noted, Visual Studio 2010 users will already have F# installed.) Installing F# on Windows with Visual Studio 2008 is straightforward. You need to be running an account with system administrator privileges, then download the MSI version as described in the previous section, and execute it.

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. However, you can install F#'s plug-in on top of the free Visual Studio 2008 Shell. See the MSDN Visual Studio Extensibility center for more details on Visual Studio 2008 Shell at `http://msdn.microsoft.com/vsx2008/`.

Installing F# on Linux

If you are unfamiliar with Linux and would like to try Mono, the simplest way is to download the SUSE Linux virtual machine (VM) image available on the Mono web site at `http://www.go-mono.com/mono-downloads`.

■ **Note** Mono is a free, open-source, multi-platform implementation of the Common Language Runtime (CLR), which is compliant with the ECMA specifications of the Common Language Infrastructure (CLI) and compatible with Microsoft .NET. It is implemented by Novell with the aid of a number of community volunteers.

This VM image comes with the latest Mono runtime and development tools preinstalled, so there's no need to worry about setting up any of these. But the image does not currently include the F# compiler, so you will need to install it using the following instructions. I performed all of these steps as the `root` account.

- Still in the `/usr/lib/fsharp` directory, run the command `sh install-mono.sh`.
- Unpack the F# distribution and copy the resulting files to `/usr/lib/fsharp`.
- In the `/usr/lib/fsharp` directory, run `chmod +x install-mono.sh`.
- Run the `dos2unix` tool on the text file `install-mono.sh`.
- 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 the command line of any account 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`, the shell script for this is as follows:

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

Figure 2-2 shows F# interactive running under Mono and Linux.

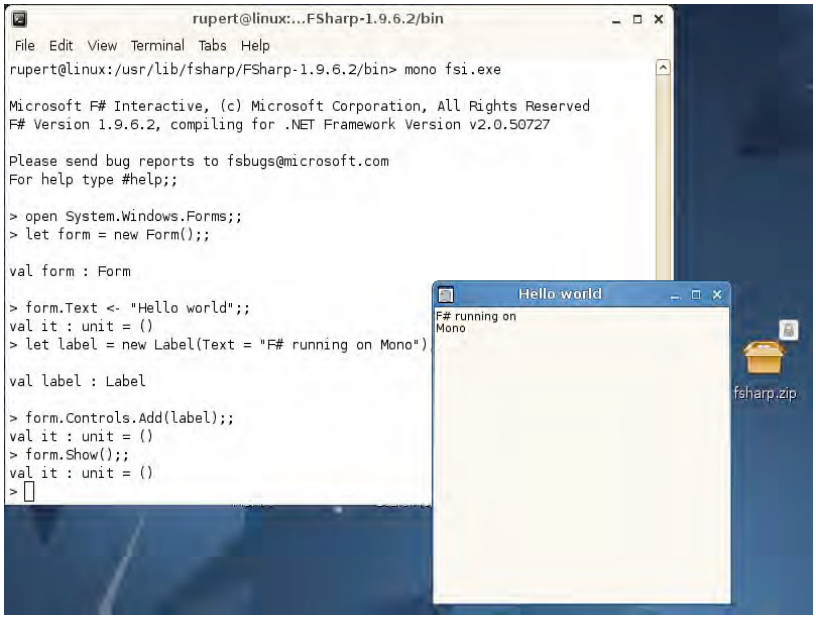

Figure 2-2. *F# interactive running under Mono and Linux*

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`:

```
printfn "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
```

Visual Studio

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 2-3). 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.

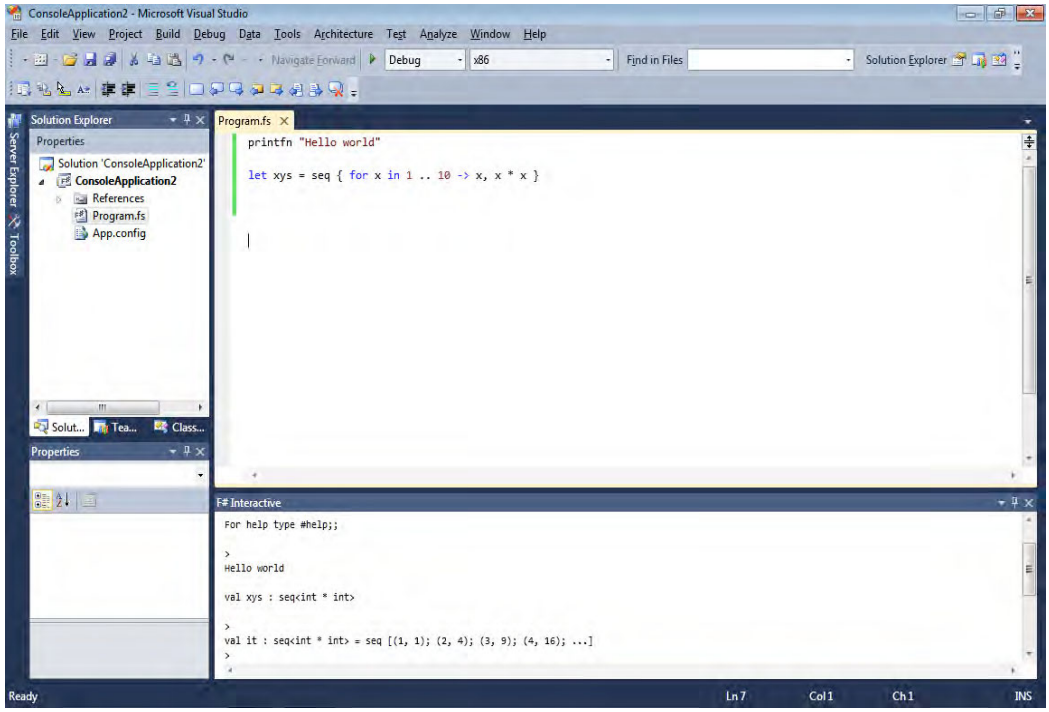

Figure 2-3. *Visual Studio 2010 hosting F# interactive*

SharpDevelop

SharpDevelop is an open-source IDE with F# bindings that can be used for .NET and Mono development. The F# bindings are packaged with SharpDevelop, so to use them all you need to do is ensure that F# is installed then install SharpDevelop. After that it's just a matter of creating a new F# project and off you go. The F# bindings for SharpDevelop do not offer as much functionality as Visual Studio does—only syntax highlighting and F# interactive are available. However, the development environment is still very useable, and the bindings are open source. So if you wish to extend them, you can help out with the project. Figure 2-4 shows SharpDevelop with an F# project open.

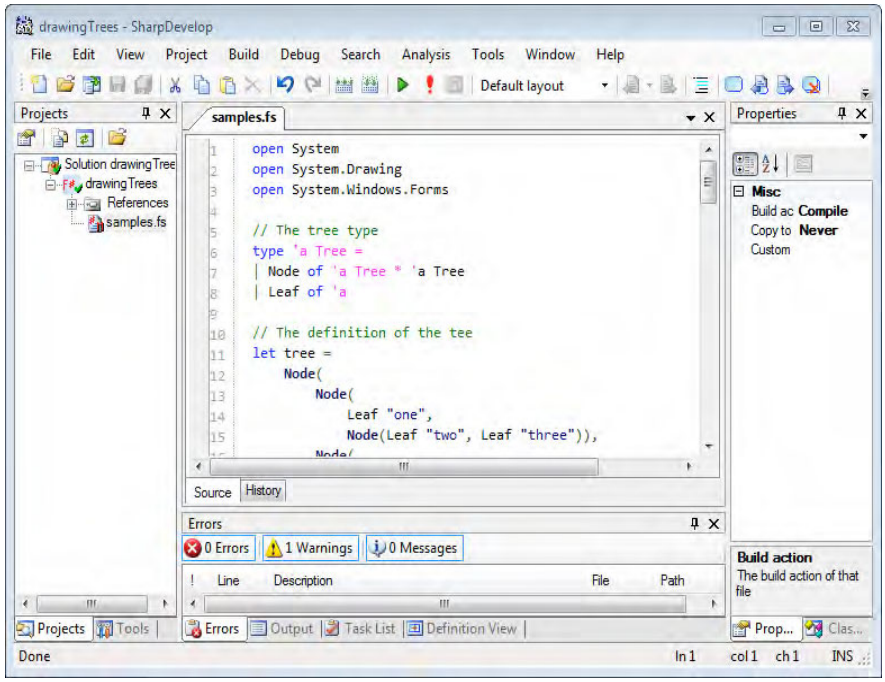

Figure 2-4. *SharpDevelop with an F# project open*

F# Interactive Command-Line

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 2-5.

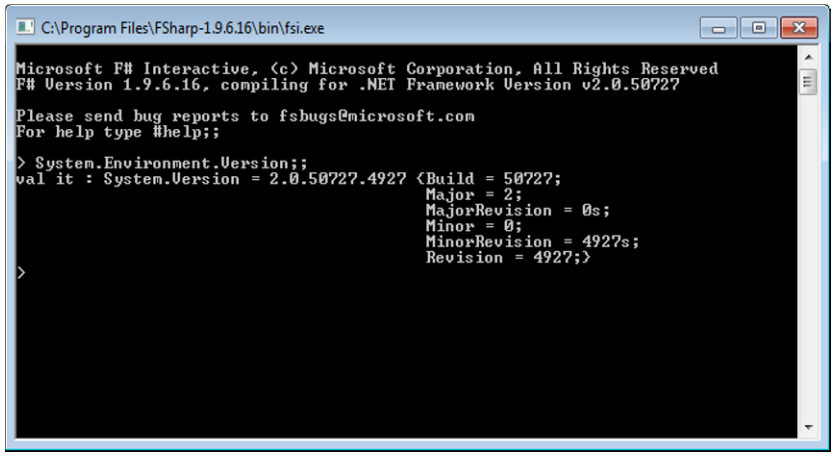

Figure 2-5. *The F# interactive console running in stand-alone mode*

When you use the interactive console, you type the code you want. 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 equal 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.

The Examples in This Book

The code in this book will focus on using `fsc.exe` rather than `fsi.exe`. 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.

The samples can be downloaded from `http://bfs.codeplex.com/`. All the samples in this book were tested using .NET 4.0 running on Windows 7. A subset has also been tested running under Mono 2.4.2.3 on Linux.