CHAPTER 1

Introduction

F# is a typed functional programming language for the .NET Framework. It combines the succinctness, expressivity, and compositionality of typed functional programming with the runtime support, libraries, interoperability, tools, and object model of .NET. Our aim in this book is to help you become an expert in using F# and the .NET Framework.

Functional programming has long inspired researchers, students, and programmers alike with its simplicity and expressive power. Applied functional programming is booming: a new generation of typed functional languages is reaching maturity; some functional language constructs have been integrated into languages such as C#, Python, and Visual Basic; and there is now a substantial pool of expertise in the pragmatic application of functional programming techniques. There is also strong evidence that functional programming offers significant productivity gains in important application areas such as data access, financial modeling, statistical analysis, machine learning, software verification, and bio-informatics. More recently, functional programming is part of the rise of declarative programming models, especially in the data query, concurrent, reactive, and parallel programming domains.

F# differs from many functional languages in that it embraces imperative and object-oriented (OO) programming. It also provides a missing link between compiled and dynamic languages, combining the idioms and programming styles typical of dynamic languages with the performance and robustness of a compiled language. The F# designers have adopted a design philosophy that allows you to take the best and most productive aspects of these paradigms and combine them while still placing primary emphasis on functional programming techniques. This book will help you understand the power that F# offers through this combination.

F# and .NET offer an approach to computing that will continue to surprise and delight, and mastering functional programming techniques will help you become a better programmer regardless of the language you use. There has been no better time to learn functional programming, and F# offers the best route to learn and apply functional programming on the .NET platform.

The lead designer of the F# language, Don Syme, is one of the authors of this book. This book benefits from his authority on F# and .NET and from all the authors' years of experience with F# and other programming languages.

The Genesis of F#

F# began in 2002 when Don Syme and others at Microsoft Research decided to ensure that the "ML" approach to pragmatic but theoretically-based language design found a high-quality

expression for the .NET platform. The project was closely associated with the design and implementation of Generics for the .NET Common Language Runtime. The first major pre-release of F# was in 2005.

F# shares a core language with the programming language OCaml, and in some ways it can be considered an "OCaml for .NET." F# would not exist without OCaml, which in turn comes from the ML family of programming languages, which dates back to 1974. F# also draws from Haskell, particularly with regard to two advanced language features called *sequence expressions* and *workflows*. There are still strong connections between the designers of these languages and overlap in their user communities. The rationale for the design decisions taken during the development of F# is documented on the F# project website.

Despite the similarities to OCaml and Haskell, programming with F# is really quite different. In particular, the F# approach to type inference, OO programming, and dynamic language techniques is substantially different from all other mainstream functional languages. Programming in F# tends to be more object-oriented than in other functional languages. Programming also tends to be more flexible. F# embraces .NET techniques such as dynamic loading, dynamic typing, and reflection, and it adds techniques such as expression quotation and active patterns. We cover these topics in this book and use them in many application areas.

F# also owes a lot to the designers of .NET, whose vision of language interoperability between C++, Visual Basic, and "the language that eventually became C#" is still rocking the computer industry today. Today F# draws much from the broader community around the Common Language Infrastructure (CLI). This standard is implemented by the Microsoft .NET Framework, Mono, and Microsoft's client-side execution environment Silverlight. F# is able to leverage libraries and techniques developed by Microsoft, the broader .NET community, and the highly active open source community centered around Mono. These include hundreds of important libraries and major implementation stacks such as language-integrated queries using Microsoft's LINQ.

About This Book

This book is structured in two halves: Chapters 2 to 10 deal with the F# language and basic techniques and libraries associated with the .NET Framework. Chapters 11 to 19 deal with applied techniques ranging from building applications through to software engineering and design issues.

Throughout this book we address both *programming constructs* and *programming techniques*. Our approach is driven by examples: we show code, and then we explain it. Frequently we give reference material describing the constructs used in the code and related constructs you might use in similar programming tasks. We've found that an example-driven approach helps bring out the essence of a language and how the language constructs work together. You can find a complete syntax guide in the appendix, and we encourage you to reference this while reading the book.

Chapter 2, *Getting Started with F# and .NET*, begins by introducing F# Interactive, a tool you can use to interactively evaluate F# expressions and declarations and that we encourage you to use while reading this book. In this chapter you will use F# Interactive to explore some basic F# and .NET constructs, and we introduce many concepts that are described in more detail in later chapters.

Chapter 3, *Introducing Functional Programming*, focuses on the basic constructs of typed functional programming, including arithmetic and string primitives, type inference, tuples, lists, options, function values, aggregate operators, recursive functions, function pipelines, function compositions, pattern matching, sequences, and some simple examples of type definitions.

Chapter 4, *Introducing Imperative Programming*, introduces the basic constructs used for imperative programming in F#. Although the use of imperative programming is often minimized with F#, it is used heavily in some programming tasks such as scripting. You will learn about loops, arrays, mutability mutable records, locals and reference cells, the imperative .NET collections, exceptions, and the basics of .NET I/O.

Chapter 5, *Mastering Types and Generics*, covers types in more depth, especially the more advanced topics of generic type variables and subtyping. You will learn techniques you can use to make your code generic and how to understand and clarify type error messages reported by the F# compiler.

Chapter 6, *Working with Objects and Modules*, introduces object-oriented programming in F#. You will learn how to define concrete object types to implement data structures, how to use object-oriented notational devices such as method overloading with your F# types, and how to create objects with mutable state. You will then learn how to define object interface types and a range of techniques to implement objects, including object expressions, constructor functions, delegation, and implementation inheritance.

Chapter 7, *Encapsulating and Packaging Your Code*, shows the techniques you can use to hide implementation details and package code fragments together into .NET assemblies. You will also learn how to use the F# command-line compiler tools and how to build libraries that can be shared across multiple projects. Finally, we cover some of the techniques you can use to build installers and deploy F# applications.

Chapter 8, *Mastering F#: Common Techniques*, looks at a number of important coding patterns in F#, including how to customize the hashing and comparison semantics of new type definitions, how to precompute and cache intermediary results, and how to create lazy values. You'll also learn how to clean up resources using the .NET idioms for disposing of objects, how to avoid stack overflows through the use of tail calls, and how to subscribe to .NET events and publish new .NET-compatible events from F# code.

Chapter 9, *Introducing Language-Oriented Programming*, looks at what is effectively a fourth programming paradigm supported by F#: the manipulation of structured data and language fragments using a variety of concrete and abstract representations. In this chapter you'll learn how to use XML as a concrete language format, how to convert XML to typed abstract syntax representations, how to design and work with abstract syntax representations, and how to use F# active patterns to hide representations. You will also learn three advanced features of F# programming: F# computation expressions (also called *workflows*), F# reflection, and F# quotations. These are used in later chapters, particularly Chapters 13 and 15.

Chapter 10, *Using the F# and .NET Libraries*, gives an overview of the libraries most frequently used with F#, including the .NET Framework and the extra libraries added by F#.

Chapters 11 to 19 deal with applied topics in F# programming. Chapter 11, *Working with Windows Forms and Controls*, shows how to design and build graphical user interface applications using F# and the .NET Windows Forms library. We also show how to design new controls using standard object-oriented design patterns and how to script applications using the controls offered by the .NET libraries directly.

Chapter 12, *Working with Symbolic Representations*, applies some of the techniques from Chapter 9 and Chapter 11 in two case studies. The first is symbolic expression differentiation and rendering, an extended version of a commonly used case study in symbolic programming. The second is verifying circuits with propositional logic, where you will learn how to use symbolic techniques to represent digital circuits, specify properties of these circuits, and verify these properties using binary decision diagrams (BDDs).

Chapter 13, *Reactive, Asynchronous, and Concurrent Programming*, shows how you can use F# for programs that have multiple logical threads of execution and that react to inputs and messages. You will first learn how to construct basic background tasks that support progress reporting and cancellation. You will then learn how to use F# asynchronous workflows to build scalable, massively concurrent reactive programs that make good use of the .NET thread pool and other .NET concurrency-related resources. This chapter concentrates on message-passing techniques that avoid or minimize the use of shared memory. However, you will also learn the fundamentals of concurrent programming with shared memory using .NET.

Chapter 14, *Building Web Applications*, shows how to use F# with ASP.NET to write server-side scripts that respond to web requests. You will learn how to serve web page content using ASP.NET controls. We also describe how open source projects such as the F# Web Toolkit let you write both parts of Ajax-style client/server applications in F#.

Chapter 15, *Working with Data*, looks at several dimensions of querying and accessing data from F#. You'll first learn how functional programming relates to querying in-memory data structures, especially via the LINQ paradigm supported by .NET and F#. You'll then look at how to use F# in conjunction with relational databases, particularly through the use of the ADO.NET and LINQ-to-SQL technologies that are part of the .NET Framework.

Chapter 16, *Lexing and Parsing*, shows how to deal with additional concrete language formats beyond those already discussed in Chapter 9. In particular, you will learn how to use the F# tools for generating lexers and parsers from declarative specifications and how to use combinator techniques to build declarative specifications of binary format readers.

Chapter 17, *Interoperating with C and COM*, shows how to use F# and .NET to interoperate with software that exports a native API. You will learn more about the .NET Common Language Runtime itself, how memory management works, and how to use the .NET Platform Invoke mechanisms from F#.

Chapter 18, *Debugging and Testing F# Programs*, shows the primary tools and techniques you can use to eliminate bugs from your F# programs. You will learn how to use the .NET and Visual Studio debugging tools with F#, how to use F# Interactive for exploratory development and testing, and how to use the NUnit testing framework with F# code.

Chapter 19, *Designing F# Libraries*, gives our advice on methodology and design issues for writing libraries in F#. You will learn how to write "vanilla" .NET libraries that make relatively little use of F# constructs at their boundaries in order to appear as natural as possible to other .NET programmers. We will then cover functional programming design methodology and how to combine it with the object-oriented design techniques specified by the standard .NET Framework design guidelines.

The appendix, *F# Brief Language Guide*, gives a compact guide to all key F# language constructs and the key operators used in F# programming.

Because of space limitations, we have only partially addressed some important aspects of programming with F#. It is easy to access hundreds of other libraries with F# that are not covered in this book, including Managed DirectX, Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WWF), Irrlicht, the Mono Unix bindings, the Firebird.NET database bindings, several advanced SQL Server APIs, and mathematical libraries such as Extreme Optimization and NMath. There are also hundreds of open-source projects related to .NET programming, some with a specific focus on F#. F# can also be used with alternative implementations of the CLI such as Mono and Silverlight, topics we address only tangentially in this book. Quotation meta-programming is described only briefly in Chapter 9, and some topics in functional programming such as the design and implementation of applicative data structures are not covered at all. Also, some software engineering issues such as performance tuning are largely omitted. Many of these topics are addressed in more detail in *Foundations of F#* by Robert Pickering, also published by Apress.

Who This Book Is For

We assume you have some programming knowledge and experience. If you don't have experience with F# already, you'll still be familiar with many of the ideas it uses. However, you may also encounter some new and challenging ideas. For example, if you've been taught that object-oriented (OO) design and programming are the only ways to think about software, then programming in F# may be a reeducation. F# fully supports OO development, but F# programming combines elements of both functional and OO design. OO patterns such as implementation inheritance play a less prominent role than you may have previously experienced. Chapter 6 covers many of these topics in depth.

The following notes will help you set a path through this book depending on your background:

C++, C#, Java, and Visual Basic: If you've programmed in a typed OO language, you may find functional programming, type inference, and F# type parameters take a while to get used to. However, you'll soon see how to use these to make you a more productive programmer. Be sure to read Chapters 2, 3, 5, and 6 carefully.

Python, Scheme, Ruby, and dynamically typed languages: F# is statically typed and type-safe. As a result, F# development environments can discover many errors while you program, and the F# compiler can more aggressively optimize your code. If you've primarily programmed in an untyped language such as Python, Scheme, or Ruby, you may think that static types are inflexible and wordy. However, F# static types are relatively nonintrusive, and you'll find the language strikes a balance between expressivity and type safety. You'll also see how type inference lets you recover succinctness despite working in a statically typed language.

Be sure to read Chapters 2 to 6 carefully, paying particular attention to the ways in which types are used and defined.

Typed functional languages: If you are familiar with Haskell, OCaml, or Standard ML, you will find the core of F# readily familiar, with some syntactic differences. However, F# embraces .NET, including the .NET object model, and it may take you a while to learn how to use objects effectively and how to use the .NET libraries themselves. This is best done by learning how F# approaches OO programming in Chapters 6 to 8 and then exploring the applied .NET programming material in Chapters 10 to 19, referring to earlier chapters where necessary. Haskell programmers will also need to learn the F# approach to imperative programming, described in Chapter 4, since many .NET libraries require a degree of imperative coding to create, configure, connect, and dispose of objects.

We strongly encourage you to use this book in conjunction with a development environment that supports F# directly, such as Visual Studio 2005 or Visual Studio 2008. In particular, the interactive type inference in the F# Visual Studio environment is exceptionally helpful for understanding F# code: with a simple mouse movement you can examine the inferred types of the sample programs. These types play a key role in understanding the behavior of the code.

■ **Note** You can download and install F# from `http://research.microsoft.com/fsharp`. Your primary source for information on the aspects of F# explored in this book is `http://www.expert-fsharp.com`, and you can download all the code samples used in this book from `http://www.expert-fsharp.com/CodeSamples`. As with all books, it is inevitable that minor errors may have crept into the text. Adjustments may also be needed to make the best use of versions of F# beyond version 1.9.2, which was used for this book. An active errata and list of updates will be published at `http://www.expert-fsharp.com/Updates`.
