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 make 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.