

# Expert F#



Don Syme, Adam Granicz, and  
Antonio Cisternino

## **Expert F#**

**Copyright © 2007 by Don Syme, Adam Granicz, and Antonio Cisternino**

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-59059-850-4

ISBN-10: 1-59059-850-4

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editors: Jim Huddleston, Jonathan Hassell

Technical Reviewer: Tomáš Petríček

Editorial Board: Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jason Gilmore, Kevin Goff, Jonathan Hassell, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Sofia Marchant

Copy Editor: Kim Wimpsett

Associate Production Director: Kari Brooks-Copony

Production Editor: Ellie Fountain

Compositor: Susan Glinert

Proofreader: April Eddy

Indexer: Present Day Indexing

Artist: Kinetic Publishing Services, LLC

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](mailto: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](mailto:info@apress.com), or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.expert-fsharp.com>.

*This book is dedicated to the memory of James Huddleston,  
the editor at Apress who initiated this book project and encouraged  
the authors with his insights, loyalty, enthusiasm, and humor.*

*Jim passed away in February 2007, an enormous loss  
to his family, Apress, and the authors.*



# Contents at a Glance

Foreword .....	xxi
About the Authors .....	xxiii
About the Technical Reviewer .....	xxv
Acknowledgments .....	xxvii
■ <b>CHAPTER 1</b> Introduction .....	1
■ <b>CHAPTER 2</b> Getting Started with F# and .NET .....	7
■ <b>CHAPTER 3</b> Introducing Functional Programming .....	27
■ <b>CHAPTER 4</b> Introducing Imperative Programming .....	69
■ <b>CHAPTER 5</b> Mastering Types and Generics .....	101
■ <b>CHAPTER 6</b> Working with Objects and Modules .....	125
■ <b>CHAPTER 7</b> Encapsulating and Packaging Your Code .....	155
■ <b>CHAPTER 8</b> Mastering F#: Common Techniques .....	181
■ <b>CHAPTER 9</b> Introducing Language-Oriented Programming .....	211
■ <b>CHAPTER 10</b> Using the F# and .NET Libraries .....	255
■ <b>CHAPTER 11</b> Working with Windows Forms and Controls .....	275
■ <b>CHAPTER 12</b> Working with Symbolic Representations .....	317
■ <b>CHAPTER 13</b> Reactive, Asynchronous, and Concurrent Programming .....	355
■ <b>CHAPTER 14</b> Building Web Applications .....	393
■ <b>CHAPTER 15</b> Working with Data .....	431
■ <b>CHAPTER 16</b> Lexing and Parsing .....	461
■ <b>CHAPTER 17</b> Interoperating with C and COM .....	491
■ <b>CHAPTER 18</b> Debugging and Testing F# Programs .....	523
■ <b>CHAPTER 19</b> Designing F# Libraries .....	545
■ <b>APPENDIX</b> F# Brief Language Guide .....	563
■ <b>INDEX</b> .....	571



# Contents

Foreword .....	xxi
About the Authors .....	xxiii
About the Technical Reviewer .....	xxv
Acknowledgments .....	xxvii
<b>CHAPTER 1 Introduction .....</b>	<b>1</b>
The Genesis of F# .....	1
About This Book .....	2
Who This Book Is For .....	5
<b>CHAPTER 2 Getting Started with F# and .NET .....</b>	<b>7</b>
Creating Your First F# Program .....	7
Turning On the Lightweight Syntax Option .....	9
Documenting Code Using XMLDocs .....	10
Understanding Scope and Using “let” .....	10
Understanding Types .....	13
Calling Functions .....	14
Using Data Structures .....	15
Using Properties and the Dot-Notation .....	16
Using Tuples .....	17
Using Imperative Code .....	19
Using .NET Libraries from F# .....	20
Using open to Access Namespaces and Modules .....	21
Using new and Setting Properties .....	22
Fetching a Web Page .....	23
Summary .....	25

<b>CHAPTER 3</b>	<b>Introducing Functional Programming</b>	<b>27</b>
	Getting Started with F# Arithmetic	27
	Basic Literals	27
	Arithmetic Operators	28
	Bitwise Operations	29
	Arithmetic Conversions	30
	Arithmetic Comparisons	31
	Overloaded Math Functions	31
	Introducing Simple Strings	31
	Working with String Literals and Primitives	32
	Building Strings	33
	Working with Lists and Options	34
	Using F# Lists	34
	Using F# Option Values	37
	Using Option Values for Control	39
	Working with Conditionals: && and	39
	Defining Recursive Functions	40
	Introducing Function Values	42
	Using Anonymous Function Values	43
	Computing with Aggregate Operators	44
	Composing Functions with >>	45
	Building Functions with Partial Application	46
	Using Local Functions	47
	Using Functions As Abstract Values	48
	Iterating with Aggregate Operators	49
	Abstracting Control with Functions	49
	Using .NET Methods As First-Class Functions	50
	Getting Started with Pattern Matching	51
	Matching on Structured Values	53
	Guarding Rules and Combining Patterns	54
	Getting Started with Sequences	55
	Using Range Expressions	55
	Iterating a Sequence	56
	Transforming Sequences with Aggregate Operators	56
	Which Types Can Be Used As Sequences?	57
	Using Lazy Sequences from External Sources	58



Using Sequence Expressions .....	59
Creating Sequence Expressions Using for .....	60
Enriching Sequence Expressions with Additional Clauses .....	60
Enriching Sequence Expressions to Specify Lists and Arrays .....	61
Exploring Some Simple Type Definitions .....	62
Defining Type Abbreviations .....	62
Defining Records .....	63
Handling Non-unique Record Field Names .....	64
Cloning Records .....	64
Defining Discriminated Unions .....	65
Using Discriminated Unions As Records .....	67
Defining Multiple Types Simultaneously .....	67
Summary .....	68

## ■ CHAPTER 4    **Introducing Imperative Programming** ..... 69

Imperative Looping and Iterating .....	70
Simple for loops .....	70
Simple while loops .....	71
More Iteration Loops Over Sequences .....	71
Using Mutable Records .....	72
Mutable Reference Cells .....	73
Avoiding Aliasing .....	74
Hiding Mutable Data .....	75
Using Mutable Locals .....	76
Working with Arrays .....	77
Generating and Slicing Arrays .....	79
Two-Dimensional Arrays .....	80
Introducing the Imperative .NET Collections .....	80
Using Resizeable Arrays .....	80
Using Dictionaries .....	81
Using Dictionary's TryGetValue .....	82
Using Dictionaries with Compound Keys .....	83
Some Other Mutable Data Structures .....	84
Exceptions and Controlling Them .....	84
Catching Exceptions .....	86
Using try ... finally .....	86
Defining New Exception Types .....	87

Having an Effect: Basic I/O .....	88
Very Simple I/O: Reading and Writing Files .....	88
.NET I/O via Streams .....	89
Some Other I/O-Related Types .....	91
Using System.Console .....	91
Using printf and Friends.....	91
Generic Structural Formatting.....	93
Cleaning Up with IDisposable, use, and using.....	93
Working with null Values .....	94
Some Advice: Functional Programming with Side Effects .....	95
Consider Replacing Mutable Locals and Loops with Recursion.....	96
Separate Pure Computation from Side-Effecting Computations.....	96
Separating Mutable Data Structures .....	97
Not All Side Effects Are Equal .....	98
Avoid Combining Imperative Programming and Laziness .....	98
Summary .....	100

## ■ CHAPTER 5    **Mastering Types and Generics** .....

Understanding Generic Type Variables .....	101
Writing Generic Functions .....	102
Understanding Some Important Generic Functions .....	103
Generic Comparison .....	103
Generic Hashing .....	105
Generic Pretty-Printing.....	105
Generic Boxing and Unboxing .....	106
Generic Binary Serialization via the .NET Libraries.....	107
Making Things Generic .....	108
Generic Algorithms Through Explicit Arguments.....	108
Generic Algorithms Through Abstract Object Types.....	110
Understanding .NET Types .....	112
Reference Types and Value Types .....	112
Other Flavors of .NET Types .....	113
Understanding Subtyping .....	113
Casting Up Statically.....	114
Casting Down Dynamically .....	114
Performing Type Tests via Pattern Matching.....	115
Using Flexible # Types .....	116
Knowing When Upcasts Are Applied Automatically .....	117

Troubleshooting Type Inference Problems .....	118
Using a Visual Editing Environment .....	118
Using Type Annotations .....	118
Understanding the Value Restriction .....	119
Working Around the Value Restriction .....	120
Understanding Generic Overloaded Operators .....	123
Summary .....	123

## CHAPTER 6 Working with Objects and Modules ..... 125

Getting Started with Objects and Members .....	125
Using Constructed Classes .....	128
Adding Further Object Notation to Your Types .....	131
Working with Indexer Properties .....	131
Adding Overloaded Operators .....	132
Using Named and Optional Arguments .....	133
Using Optional Property Settings .....	134
Adding Method Overloading .....	135
Defining Object Types with Mutable State .....	137
Getting Started with Object Interface Types .....	139
Defining New Object Interface Types .....	140
Implementing Object Interface Types Using	
Object Expressions .....	140
Implementing Object Interface Types Using Concrete Types ....	142
Using Common Object Interface Types from the	
.NET Libraries .....	142
Understanding Hierarchies of Object Interface Types .....	143
More Techniques to Implement Objects .....	144
Combining Object Expressions and Function Parameters .....	144
Defining Partially Implemented Class Types .....	146
Using Partially Implemented Types via Delegation .....	146
Using Partially Implemented Types via	
Implementation Inheritance .....	147
Using Modules and Static Members .....	148
Extending Existing Types and Modules .....	150
Working with F# Objects and .NET Types .....	151
Structs .....	152
Delegates .....	152
Enums .....	153
Summary .....	153

<b>CHAPTER 7</b>	<b>Encapsulating and Packaging Your Code</b>	155
	Hiding Things Away	156
	Hiding Things with Local Definitions	156
	Hiding Things with Accessibility Annotations	158
	Using Namespaces and Modules	161
	Putting Your Code in a Namespace	162
	Using Files As Modules	163
	Using Signature Types and Files	164
	Using Explicit Signature Types and Signature Files	164
	When Are Signature Types Checked?	166
	Creating Assemblies, DLLs, and EXEs	166
	Compiling EXEs	166
	Compiling DLLs	167
	Mixing Scripting and Compiled Code	168
	Choosing Optimization Settings	169
	Generating Documentation	170
	Building Shared Libraries and the Using Global Assembly Cache	171
	Using Static Linking	172
	Packaging Applications	173
	Packaging Different Kinds of Code	173
	Using Data and Configuration Settings	174
	Building Installers	177
	Deploying Web Applications	178
	Summary	179
 <b>CHAPTER 8</b>	 <b>Mastering F#: Common Techniques</b>	 181
	Equality, Hashing, and Comparison	181
	Efficient Precomputation and Caching	184
	Precomputation and Partial Application	184
	Precomputation and Objects	185
	Memoizing Computations	187
	Lazy Values	189
	Other Variations on Caching and Memoization	190
	Cleaning Up Resources	190
	Cleaning Up with use	191
	Managing Resources with More Complex Lifetimes	193
	Cleaning Up Internal Objects	194
	Cleaning Up Unmanaged Objects	196

Cleaning Up in Sequence Expressions . . . . .	197
Using using . . . . .	198
Stack As a Resource: Tail Calls and Recursion . . . . .	198
Tail Recursion and List Processing . . . . .	200
Tail Recursion and Object-Oriented Programming . . . . .	202
Tail Recursion and Processing Unbalanced Trees . . . . .	203
Using Continuations to Avoid Stack Overflows . . . . .	204
Another Example: Processing Syntax Trees . . . . .	206
Events and Wiring . . . . .	207
Events As First-Class Values . . . . .	208
Creating and Publishing Events . . . . .	209
Summary . . . . .	210

## ■ CHAPTER 9 Introducing Language-Oriented Programming . . . . . 211

Using XML As a Concrete Language Format . . . . .	212
Using the System.Xml Namespace . . . . .	212
From Concrete XML to Abstract Syntax . . . . .	214
Working with Abstract Syntax Representations . . . . .	217
Abstract Syntax Representations: “Less Is More” . . . . .	217
Processing Abstract Syntax Representations . . . . .	218
Transformational Traversals of Abstract Syntax Representations . . . . .	219
Using On-Demand Computation with Abstract Syntax Trees . . . . .	220
Caching Properties in Abstract Syntax Trees . . . . .	221
Memoizing Construction of Syntax Tree Nodes . . . . .	222
Introducing Active Patterns . . . . .	224
Converting the Same Data to Many Views . . . . .	225
Matching on .NET Object Types . . . . .	227
Defining Partial and Parameterized Active Patterns . . . . .	228
Hiding Abstract Syntax Implementations with Active Patterns . . . . .	228
Embedded Computational Languages with Workflows . . . . .	230
An Example: Success/Failure Workflows . . . . .	232
Defining a Workflow Builder . . . . .	235
Workflows and “Untamed” Side Effects . . . . .	238
Example: Probabilistic Workflows . . . . .	239
Combining Workflows and Resources . . . . .	244
Recursive Workflow Expressions . . . . .	244
Using F# Reflection . . . . .	245
Reflecting on Types . . . . .	245
Schema Compilation by Reflecting on Types . . . . .	245

Using F# Quotations .....	249
Example: Using F# Quotations for Error Estimation .....	251
Resolving Top Definitions .....	253
Summary .....	254

## ■ CHAPTER 10 Using the F# and .NET Libraries ..... 255

A High-Level Overview .....	255
Namespaces from the .NET Framework .....	256
Namespaces from the F# Libraries.....	258
Using the System Types .....	259
Using Regular Expressions and Formatting .....	261
Matching with System.Text.RegularExpressions .....	261
Formatting Strings Using .NET Formatting .....	265
Encoding and Decoding Unicode Strings .....	266
Encoding and Decoding Binary Data .....	266
Using Further F# and .NET Data Structures .....	266
System.Collections.Generic and Other .NET Collections .....	267
Introducing Microsoft.FSharp.Math .....	268
Using Matrices and Vectors.....	268
Using Operator Overloads on Matrices and Vectors .....	269
Supervising and Isolating Execution .....	270
Further Libraries for Reflective Techniques .....	270
Using General Types.....	270
Using Microsoft.FSharp.Reflection .....	271
Some Other .NET Types You May Encounter .....	272
Some Other .NET Libraries .....	273
Summary .....	274

## ■ CHAPTER 11 Working with Windows Forms and Controls ..... 275

Writing “Hello, World!” in a Click .....	275
Understanding the Anatomy of a Graphical Application .....	276
Composing User Interfaces .....	277
Drawing Applications .....	282
Writing Your Own Controls .....	287
Developing a Custom Control .....	287
Anatomy of a Control .....	290

Displaying Samples from Sensors .....	291
Building the GraphControl: The Model .....	292
Building the GraphControl: Style Properties and Controller .....	294
Building the GraphControl: The View .....	298
Putting It Together .....	302
Creating a Mandelbrot Viewer .....	303
Computing Mandelbrot. ....	304
Setting Colors .....	305
Creating the Visualization Application .....	308
Creating the Application Plumbing .....	310
Summary .....	315

## ■ CHAPTER 12 Working with Symbolic Representations .....

Symbolic Differentiation and Expression Rendering .....	318
Modeling Simple Algebraic Expressions .....	318
Implementing Local Simplifications .....	320
A Richer Language of Algebraic Expressions .....	321
Parsing Algebraic Expressions .....	323
Simplifying Algebraic Expressions .....	325
Symbolic Differentiation of Algebraic Expressions .....	328
Rendering Expressions. ....	329
Building the User Interface .....	335
Verifying Circuits with Propositional Logic .....	338
Representing Propositional Logic .....	339
Evaluating Propositional Logic Naively. ....	340
From Circuits to Propositional Logic. ....	343
Checking Simple Properties of Circuits .....	346
Representing Propositional Formulae Efficiently Using BDDs ....	347
Circuit Verification with BDDs .....	350
Summary .....	353

## ■ CHAPTER 13 Reactive, Asynchronous, and Concurrent Programming .....

Introducing Some Terminology .....	356
Using and Designing Background Workers .....	357
Building a Simpler Iterative Worker .....	359
Raising Additional Events from Background Workers .....	362
Connecting a Background Worker to a GUI .....	363

Introducing Asynchronous Computations .....	365
Fetching Multiple Web Pages Asynchronously .....	365
Understanding Thread Hopping.....	367
Under the Hood: What Are Asynchronous Computations? .....	369
File Processing Using Asynchronous Workflows.....	371
Running Asynchronous Computations .....	374
Common I/O Operations in Asynchronous Workflows .....	375
Under the Hood: Implementing a Primitive	
Asynchronous Step .....	376
Under the Hood: Implementing Async.Parallel .....	377
Understanding Exceptions and Cancellation .....	378
Passing and Processing Messages .....	379
Introducing Message Processing .....	379
Creating Objects That React to Messages.....	381
Scanning Mailboxes for Relevant Messages .....	384
Example: Asynchronous Web Crawling .....	385
Using Shared-Memory Concurrency .....	388
Creating Threads Explicitly .....	388
Shared Memory, Race Conditions, and the .NET	
Memory Model.....	389
Using Locks to Avoid Race Conditions .....	390
Using ReaderWriterLock .....	391
Some Other Concurrency Primitives.....	392
Summary .....	392

## ■ CHAPTER 14 Building Web Applications .....

Serving Static Web Content .....	393
Serving Dynamic Web Content with ASP.NET .....	396
Understanding the Languages Used in ASP.NET.....	397
A Simple ASP.NET Web Application .....	399
Deploying and Running the Application .....	402
Using Code-Behind Files .....	404
Using ASP.NET Input Controls .....	406
Displaying Data from Databases .....	409
Going Further with ASP.NET .....	412
ASP.NET Directives.....	412
Server Controls .....	413
Debugging, Profiling, and Tracing .....	415



Understanding the ASP.NET Event Model .....	416
Maintaining the View State .....	418
Understanding the Provider Model .....	419
Creating Custom ASP.NET Server Controls .....	421
Building Ajax Rich Client Applications .....	422
More on F# Web Tools .....	423
Using Web Services .....	424
Consuming Web Services .....	425
Calling Web Services Asynchronously .....	427
Summary .....	429

## CHAPTER 15 Working with Data ..... 431

Querying In-Memory Data Structures .....	431
Select/Where/From Queries Using Aggregate Operators .....	432
Using Aggregate Operators in Queries .....	433
Accumulating Using “Folding” Operators .....	434
Expressing Some Queries Using Sequence Expressions .....	435
Using Databases to Manage Data .....	436
Choosing Your Database Engine .....	438
Understanding ADO.NET .....	438
Establishing Connections to a Database Engine .....	439
Creating a Database .....	440
Creating Tables, Inserting, and Fetching Records .....	442
Using Untyped Datasets .....	444
Generating Typed Datasets Using xsd.exe .....	446
Using Stored Procedures .....	448
Using Data Grids .....	449
Working with Databases in Visual Studio .....	450
Creating a Database .....	450
Visual Data Modeling: Adding Relationships .....	450
Accessing Relational Data with F# LinqToSql .....	452
Generating the Object/Relational Mapping .....	453
Building the DataContext Instance .....	453
Using LinqToSql from F# .....	454
Working with XML As a Generic Data Format .....	455
Constructing XML via LINQ .....	457
Storing, Loading, and Traversing LinqToXml Documents .....	458
Querying XML .....	459
Summary .....	459

<b>CHAPTER 16</b>	<b>Lexing and Parsing</b>	461
	Processing Line-Based Input	462
	On-Demand Reading of Files	463
	Using Regular Expressions	463
	Tokenizing with FsLex	464
	The fslex Input in More Detail	467
	Generating a Simple Token Stream	468
	Tracking Position Information Correctly	470
	Handling Comments and Strings	471
	Recursive-Descent Parsing	473
	Limitations of Recursive-Descent Parsers	477
	Parsing with FsYacc	477
	The Lexer for Kitty	478
	The Parser for Kitty	480
	Parsing Lists	482
	Resolving Conflicts, Operator Precedence, and Associativity	483
	Putting It Together	484
	Binary Parsing and Pickling Using Combinators	486
	Summary	489
<b>CHAPTER 17</b>	<b>Interoperating with C and COM</b>	491
	Common Language Runtime	491
	Memory Management at Run Time	494
	COM Interoperability	496
	Platform Invoke	507
	Getting Started with PInvoke	508
	Data Structures	510
	Marshalling Strings	513
	Function Pointers	516
	PInvoke Memory Mapping	517
	Wrapper Generation and Limits of PInvoke	520
	Summary	522
<b>CHAPTER 18</b>	<b>Debugging and Testing F# Programs</b>	523
	Debugging F# Programs	524
	Using Advanced Features of the Visual Studio Debugger	526
	Instrumenting Your Program with the System.Diagnostics	
	Namespace	528
	Debugging Concurrent and Graphical Applications	531

Debugging and Testing with F# Interactive .....	533
Controlling F# Interactive .....	534
Some Common F# Interactive Directives .....	535
Understanding How F# Interactive Compiles Code .....	535
Unit Testing .....	537
Summary .....	543
 <b>CHAPTER 19 Designing F# Libraries .....</b>	<b>545</b>
Designing Vanilla .NET Libraries .....	546
Understanding Functional Design Methodology .....	551
Understanding Where Functional Programming Comes From. . .	551
Understanding Functional Design Methodology .....	552
Applying the .NET Design Guidelines to F# .....	554
Some Recommended Coding Idioms .....	560
Summary .....	562
 <b>APPENDIX F# Brief Language Guide .....</b>	<b>563</b>
Comments and Attributes .....	563
Basic Types and Literals .....	563
Types .....	564
Patterns and Matching .....	564
Functions, Composition, and Pipelining .....	564
Binding and Control Flow .....	565
Exceptions .....	565
Tuples, Arrays, Lists, and Collections .....	566
Operators .....	567
Type Definitions and Objects .....	568
Namespaces and Modules .....	569
Sequence Expressions and Workflows .....	569
 <b>INDEX .....</b>	<b>571</b>



# Foreword

**A**ccording to Wikipedia, “Scientists include theoreticians who mainly develop new models to explain existing data and experimentalists who mainly test models by making measurements—though in practice the division between these activities is not clear-cut, and many scientists perform both.” The domain-specific language that many scientists use to define their models is mathematics, and since the early days of computing science, the holy grail has been to close the semantic gap between scientific models and executable code as much as possible. It is becoming increasingly clear that all scientists are practicing applied mathematics, and some scientists, such as theoretical physicists, are behaviorally indistinguishable from pure mathematicians. The more we can make programming look like mathematics, the more helpful we make it to scientists and engineers.

John Backus wrote the design for the “IBM Mathematical Formula Translating System,” which later became the language FORTRAN, in the early 1950s. Still today, FORTRAN is popular within the scientific community for writing efficient numeric computations. The second oldest programming language, Lisp, was invented by John McCarthy in 1958. Like FORTRAN, Lisp was also inspired by mathematics, in this case by Alonzo Church’s lambda calculus. Today, Lisp is still popular in the scientific community for writing high-level symbolic computations.

Interestingly, despite their common roots in mathematics, one can consider FORTRAN as the mother of all imperative and object-oriented languages and Lisp as the mother of all declarative languages. Their differences can be accounted to point of view: FORTRAN starts close to the machine with numbers and moves upward toward the mathematics, adding layers of abstraction where possible. Lisp starts with the mathematics with symbols and grows downward to the machine, peeling off layers of abstraction when necessary. But just as the previous quote remarks that the division between theoretical and experimental scientists is not clear-cut, in practice many programming problems require both imperative and declarative aspects.

Functional programming today is a close-kept secret amongst researchers, hackers, and elite programmers at banks and financial institutions, chip designers, graphic artists, and architects. As the grandchildren of Lisp, functional programming languages allow developers to write concise programs that are extremely close to the mathematical models they develop to understand the universe, the human genome, the pricing of options, the location of oil, the serving of advertisements on web pages, or the writing of fault-tolerant distributed systems. However, to the uninformed developer, functional programming seems a cruel and unnatural act, *effete mumbo jumbo*. The academic and mathematical origins of functional programming plays up in scary big words such as *type inference*, *structural types*, *closures*, *currying*, *continuations*, *principal types*, *monads*, *inference*, *impredicative higher-ranked types*, and so on. Even worse, most pure functional languages today are not well integrated with mainstream professional tools and IDEs, libraries, and frameworks.

Imperative programming today is the tool of choice for scientific programmers who simulate fluid dynamics, chemical reactions, mechanical models, and commercial software developers who write operating systems, enterprise applications, and shrink-wrapped software such as word processors, spreadsheets, games, media players, and so on. Imperative languages typically have great tool support, debuggers, profilers, refactoring editors, unit test frameworks, and so on, and large standard numeric libraries that have been perfected over decades by domain experts. As grandchildren of FORTRAN, they focus on machine operations first and build abstractions upward. Compared to functional languages, their syntax is unnecessarily verbose, and they lack modern features emerging from the mathematics of computing itself, such as closures, type inference, anonymous and structural types, and pattern matching. These features are essential for the kind of compositional development that makes functional programming so powerful.

F# is unique amongst both imperative and declarative languages in that it is the golden middle road where these two extremes converge. F# takes the best features of both paradigms and tastefully combines them in a highly productive and elegant language that both scientists and developers identify with. F# makes programmers better mathematicians and mathematicians better programmers.

Eric Meijer

# About the Authors

■ **DON SYME** is the main designer of F# and has been a functional programmer since 1989. Since joining Microsoft Research in 1998, he has been a seminal contributor to a wide variety of leading-edge projects, including generics in C# and the .NET Common Language Runtime. He received a Ph.D. from the University of Cambridge Computer Laboratory in 1999.

■ **ADAM GRANICZ** is the founder of IntelliFactory, a consultancy firm providing F# expertise. He has done research on extensible functional compilers, formal environments, and domain-specific languages. Adam has consulted for EPAM Systems, the leading software outsourcing company in CE Europe, and he is an industry domain expert in gambling, airline and travel package distribution, reverse logistics, and insurance/health care. He has a M.Sc. from the California Institute of Technology.

■ **ANTONIO CISTERNINO** is assistant professor in the Computer Science Department of the University of Pisa. His primary research is on meta-programming and domain-specific languages on virtual-machine-based execution environments. He has been active in the .NET community since 2001, and he recently developed annotated C#, an extension of C#, and Robotics4.NET, a framework for programming robots with .NET. Antonio has a Ph.D. in computer science from the University of Pisa.





# About the Technical Reviewer

■ **TOMÁŠ PETRÍČEK** is a graduate student of Charles University in Prague. Tomas is active in the .NET community, and he has been Microsoft MVP since 2004, awarded for his technical articles and presentations. Recently, he spent three months as an intern at Microsoft Research working with the F# team, and he also developed the F# WebTools project. His articles about F# and various other topics can be found at his website at <http://tomasp.net>.



# Acknowledgments

**W**e would like to thank Jonathan Hassell, our editor, and Sofia Marchant, our project manager, for their guidance and flexible schedules to keep us on track for publication. Likewise, we thank Tomáš Petríček, the primary technical reviewer, whose comments were invaluable in ensuring that the book is a comprehensive and reliable source of information. We also thank Chris Barwick, our original technical reviewer, and Dominic Cooney and Joel Pobar, who both helped with planning the early structure of the book. Any remaining mistakes are of course our own responsibility.

The various drafts of the chapters were read and commented on by many people, and the book has benefited greatly from their input. In particular, we would like to thank Ashley Feniello whose meticulous reviews have proved invaluable and uncovered numerous errors and inconsistencies, as well as John Bates, Nikolaj Bjorner, Laurent Le Brun, Richard Black, Chris Brumme, Jason Bock, Dominic Cooney, Can Erten, Thore Graepel, György Gyurica, Jon Hagen, Jon Harrop, Andrew Herbert, Ralf Herbrich, Jason Hogg, Anders Janmyr, Paulo Janotti, Pouya Larjani, Julien Laugel, James Margetson, Richard Mortier, Enrique Nell, Gregory Neverov, Ravi Pandya, Robert Pickering, Darren Platt, Joel Pobar, Andy Ray, Mark Shields, Guido Scatena, Mark Staples, Phil Trelford, Dave Waterworth, Dave Wecker, and Onno Zoeter, to name but a few.

We also thank Microsoft Research, without which neither F# nor this book would have been possible, and we are very grateful for the help and support given to F# by other language designers, including Anders Hejlsberg, Xavier Leroy, Simon Marlow, Erik Meijer, Malcom Newey, Martin Odersky, Simon Peyton Jones, Mads Torgersen, and Phil Wadler.

Finally, we thank our families and loved ones for their long-suffering patience. It would have been impossible to complete this book without their unceasing support.





# 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>.

---