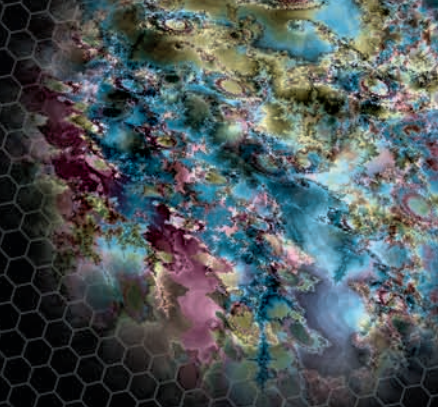
THE EXPERT'S VOICE® IN .NET



# Expert F#

*Take functional programming to the next level while enjoying the benefits of a language based on the .NET Framework*

Don Syme, Adam Granicz, and Antonio Cisternino

*Foreword by Erik Meijer*

Apress®

# Expert F#

Don Syme, Adam Granicz, and
Antonio Cisternino

Apress®

**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 Petrícek
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, or visit http://www.springeronline.com.

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

The 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

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

**INDEX** ..... 571

# Contents

Foreword . . . . . xxi
About the Authors . . . . . xxiii
About the Technical Reviewer . . . . . xxv
Acknowledgments . . . . . xxvii

**CHAPTER 1 Introduction** . . . . . 1

The Genesis of F# . . . . . 1
About This Book . . . . . 2
Who This Book Is For . . . . . 5

**CHAPTER 2 Getting Started with F# and .NET** . . . . . 7

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

CHAPTER 3 Introducing Functional Programming ..... 27

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** . . . . . 101

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

CHAPTER 7 Encapsulating and Packaging Your Code ..... 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

CHAPTER 8 Mastering F#: Common Techniques ..... 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** . . . . . 317

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** . . . . . 355

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** . . . . . 393

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

**CHAPTER 16 Lexing and Parsing** ..... 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

**CHAPTER 17 Interoperating with C and COM** ..... 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

**CHAPTER 18 Debugging and Testing F# Programs** ..... 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

CHAPTER 19 **Designing F# Libraries** . . . . . 545

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

APPENDIX **F# Brief Language Guide** . . . . . 563

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

INDEX . . . . . 571

# Foreword

According 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 FORTAN, 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 PETRÍCEK** 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

We 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 Petrícek, 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.

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#.

888bba9af12226c8bc6011165b8042d4

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`.

---

CHAPTER 2

# Getting Started with F# and .NET

In this chapter, we cover some simple interactive programming with F# and .NET. By now you should have downloaded and installed a version of the F# distribution as described in Chapter 1. In the sections that follow, we use F# Interactive, a tool you can use to execute fragments of F# code interactively and a convenient way to explore the language. Along the way, you'll see examples of the most important F# language constructs and many important libraries.

## Creating Your First F# Program

Listing 2-1 shows your first complete F# program. You may not follow it all at first glance, but we explain it piece by piece after the listing.

**Listing 2-1.** *Analyzing a String for Duplicate Words*

```
#light
/// Analyze a string for duplicate words
let wordCount text =
    let words = String.split [' '] text
    let wordSet = Set.of_list words
    let nWords = words.Length
    let nDups  = words.Length - wordSet.Count
    (nWords,nDups)

let showWordCount text =
    let nWords,nDups = wordCount text
    printfn "--> %d words in the text" nWords
    printfn "--> %d duplicate words" nDups
```

You can paste this program into F# Interactive, which you can start either by using the command line, by running `fsi.exe` from the F# distribution, or by using an interactive environment such as Visual Studio. If running from the command line, remember to enter `;;` to terminate the interactive entry:

```
C:\Users\dsyme\Desktop> fsi.exe
MSR F# Interactive, (c) Microsoft Corporation, All Rights Reserved

NOTE: See 'fsi --help' for flags
NOTE:
NOTE: Commands: #r <string>;;    reference (dynamically load) the given DLL.
NOTE:           #I <string>;;    add the given search path for referenced DLLs.

NOTE:           #use <string>;;  accept input from the given file.
NOTE:           #load <string> ...<string>;;
NOTE:                            load the given file(s) as a single unit.
NOTE:           #quit;;          exit.
NOTE:
NOTE: Visit the F# website at http://research.microsoft.com/fsharp.
NOTE: Bug reports to fsbugs@microsoft.com. Enjoy!

> <paste in the earlier program here> ;;
val wordCount : string -> int * int
val showWordCount : string -> unit
```

Here F# Interactive has reported the type of the functions `wordCount` and `showWordCount` (you'll learn more about types in a moment). The keyword `val` stands for *value*; in F# programming, functions are just values, a topic we return to in Chapter 3. Also, sometimes F# Interactive will show a little more information than we show in this book (such as some internal details of the generated values); if you're trying out these code snippets, then you can just ignore that additional information. For now let's just use the `wordCount` function interactively:

```
> let (nWords,nDups) = wordCount "All the king's horses and all the king's men";;
val nWords : int
val nDups  : int

> nWords;;
val it : int = 9

> nDups;;
val it : int = 2

> nWords - nDups;;
val it : int = 7
```

This code shows the results of executing the function `wordCount` and binding its two results to the names `nWords` and `nDups`, respectively. You can examine the values by just entering each as a single expression, which assigns the result to a value called `it` and displays the value.

Examining the values shows that the given text contains nine words: two duplicates and seven words that occur only once. `showWordCount` prints the results instead of returning them as a value:

```
> showWordCount "Couldn't put Humpty together again";;
--> 5 words in the text
--> 0 duplicate words
```

From the output you can more or less see what the code does. Now that you've done that, we'll go through the program in detail.

■ **Tip** You can start F# Interactive in Visual Studio by selecting Tools ➤ Add-in Manager and then selecting F# Interactive for Visual Studio in the Add-in Manager dialog box. A tool window will then appear, and you can send text to F# Interactive by selecting the text and pressing Alt+Return.

## Turning On the Lightweight Syntax Option

The first line of the file simply turns on the F# lightweight syntax option. This option is assumed throughout this book; in other words, you should have `#light` at the head of all your source files:

```
#light
```

This option allows you to write code that looks and feels simpler by omitting recurring F# tokens such as `in`, `done`, `;` (semicolon), `;;` (double semicolon), `begin`, and `end`. The option instructs the F# compiler and F# Interactive to use the indentation of F# code to determine where constructs start and finish. The indentation rules are very intuitive, and we discuss them in the Appendix, which is a guide to the F# syntax. Listing 2-2 shows a fully qualified version of the first function.

**Listing 2-2.** *A Version of the `wordCount` Function That Doesn't Use the `#light` Syntax Option*

```
/// Analyze a string for duplicate words
let wordCount text =
    let words = String.split [' '] text in
    let wordSet = Set.of_list words in
    let nWords = words.Length in
    let nDups  = words.Length - wordSet.Count in
    (nWords,nDups)
```

Double semicolons (`;;`) are still required to terminate entries to F# Interactive even when using the `#light` syntax option. However, if you're using an interactive development environment such as Visual Studio, then the environment typically adds this automatically when code is selected and executed. We show the double semicolons in the interactive code snippets used this book, though not in the larger samples.

■ **Tip** We recommend that you use four-space indentation for F# code. Tab characters cannot be used in `#light` code, and the F# tools will give an error if they are encountered. In Visual Studio, selecting Tools ➤ Options reveals an Options tab for controlling the options used by F# Interactive at start-up; for example, you can use `--light` to turn on the lightweight syntax option automatically on start-up.

## Documenting Code Using XMLDocs

The first real line of the program in Listing 2-1 is not code but a comment:

```
/// Analyze a string for duplicate words
```

Comments are either lines starting with `//` or blocks enclosed by `(*` and `*)`. Comment lines beginning with three slashes (`///`) are XMLDoc comments and can, if necessary, include extra XML tags and markup. The comments from a program can be collected into a single `.xml` file and processed with additional tools or can be converted immediately to HTML by the F# command-line compiler (`fsc.exe`). We cover using the F# command-line compiler in more detail in Chapter 7.

■ **Tip** The F# command-line compiler (`fsc.exe`) options for generating HTML documentation are `--generate-html` and `--html-output-directory`. To generate an XMLDoc file, use `-doc`.

## Understanding Scope and Using "let"

The next two lines of the program in Listing 2-1 introduce the start of the definition of the function `wordCount` and define the local value `words`, both using the keyword `let`:

```
let wordCount text =
    let words = ...
```

`let` is the single most important keyword you'll use in F# programming: it is used to define data, computed values, functions, and procedures. The left of a `let` binding is often a simple identifier but can also be a pattern. (See the "Using Tuples" section for some simple examples.) It can also be a function name followed by a list of argument names, as in the case of `wordCount`, which takes one argument: `text`. The right of a `let` binding (after the `=`) is an expression.

Local values such as `words` and `wordCount` can't be accessed outside their *scope*. In the case of variables defined using `let`, the scope of the value is the entire expression that follows the definition, though not the definition itself. Here are two examples of invalid definitions that try to access variables outside their scope. As you can see, `let` definitions follow a sequential, top-down order, which helps ensure that programs are well-formed and free from many bugs related to uninitialized values:

```
let badDefinition1 =
    let words = String.split text
                           ^^^^  error: text is not defined
    let text = "We three kings"
    words.Length

let badDefinition2 = badDefinition2+1
                     ^^^^^^^^^^^^^^  error: badDefinition2 is not defined
```

Sometimes it is convenient to write `let` definitions on a single line, even when using the `#light` syntax option. You can do this by separating the expression that follows a definition from the definition itself using `in`. For example:

```
let powerOfFour n =
    let nSquared = n * n in nSquared * nSquared
```

Here's an example use of the function:

```
> powerOfFour 3;;
val it : int = 81
```

Indeed, `let` *pat* `=` *expr1* `in` *expr2* is the true primitive construct in the language, where *pat* stands for *pattern* and *expr1* and *expr2* stand for *expressions*. The `#light` syntax option simply provides a veneer that lets you optionally omit the `in` if *expr2* is column-aligned with the `let` keyword on a subsequent line, and a preprocessing stage inserts the `in` token for you.

Within function definitions, values can be *outscoped* by declaring another value of the same name. For example, the following function computes `(n*n*n*n)+2`:

```
let powerOfFourPlusTwo n =
    let n = n * n
    let n = n * n
    let n = n + 2
    n
```

This code is equivalent to the following:

```
let powerOfFourPlusTwo n =
    let n1 = n * n
    let n2 = n1 * n1
    let n3 = n2 + 2
    n3
```

Outscoping a value doesn't change the original value; it just means the name of the value is no longer accessible from the current scope.

Because `let` bindings are just one kind of expression, you can use them in a nested fashion. For example:

```
let powerOfFourPlusTwoTimesSix n =
    let n3 =
        let n1 = n * n
        let n2 = n1 * n1
        n2 + 2
    let n4 = n3 * 6
    n4
```

In the previous example, `n1` and `n2` are values defined locally by `let` bindings within the expression that defines `n3`. These local values are not available for use outside their scope. For example, the following code gives an error:

```
let invalidFunction n =
    let n3 =
        let n1 = n + n
        let n2 = n1 * n1
        n1 * n2
    let n4 = n1 + n2 + n3     // Error! n3 is in scope, but n1 and n2 are not!
    n4
```

Local scoping is used for many purposes in F# programming, especially to hide implementation details that you don't want revealed outside your functions or other objects. We cover this topic in more detail in Chapter 7.

### VALUES AND IMMUTABILITY

In other languages, a local *value* is called a local *variable*. However, in F# you can't change the immediate value of locals after they've been initialized, unless the local is explicitly marked as `mutable`, a topic we return to in Chapter 4. For this reason, F# programmers and the language specification tend to prefer the term *value* to *variable*.

As you'll see in Chapter 4, data indirectly referenced by a local value can still be mutable even if the local value is not; for example, a local value that is a handle to a hash table cannot be changed to refer to a different table, but the contents of the table itself can be changed by invoking operations that add and remove elements from the table. However, many values and data structures in F# programming are *completely immutable*; in other words, neither the local value nor its contents can be changed through external mutation. These are usually just called *immutable values*. For example, all basic .NET types such as integers, strings, and `System.DateTime` values are immutable, and the F# library defines a range of immutable data structures such as `Set` and `Map`, based on binary trees.

Immutable values bring many advantages. At first it might seem strange to define values you can't change. However, knowing a value is immutable means you rarely need to think about the *object identity* of these values—you can pass them to routines and know they won't be mutated. You can also pass them between multiple threads without worrying about unsafe concurrent access to the values, discussed in Chapter 14. You can find out more about programming with immutable data structures at `http://www.expert-fsharp.com/Topics/FunctionalDataStructures`.

## Understanding Types

F# is a typed language, so it's reasonable to ask what the *type* of `wordCount` is, and indeed F# Interactive has shown it already:

```
val wordCount : string -> int * int
```

This indicates that `wordCount` takes one argument of type `string` and returns `int * int`, which is F#'s way of saying "a pair of integers." The keyword `val` stands for *value*, and the symbol `->` represents a function. No explicit type has been given in the program for `wordCount` or its argument `text`, because the full type for `wordCount` has been "inferred" from its definition. We discuss type inference further in the "What Is Type Inference?" sidebar and in more detail in later chapters.

Types are significant in both F# and .NET programming more generally for reasons that range from performance to coding productivity and interoperability. Types are used to help structure libraries, to guide the programmer through the complexity of an API and to place constraints on code to ensure it can be implemented efficiently. However, unlike many other typed languages, the type system of F# is both simple and powerful because it uses orthogonal, composable constructs such as tuples and functions to form succinct and descriptive types. Furthermore, type inference means you almost never have to write types in your program, though doing so can be useful. Table 2-1 shows some of the most important type constructors. We discuss all these types in more detail in Chapter 3 and Chapter 4.

**Table 2-1.** *Some Important Types, Type Constructors, and Their Corresponding Values*

| Family of Types | Examples | Description |
|---|---|---|
| `int` | `int` | 32-bit integers. For example: `-3`, `0`, `127`. |
| *type* `option` | `int option`, `option<int>` | A value of the given type or the special value `None`. For example: `Some 3`, `Some "3"`, `None`. |
| *type* `list` | `int list`, `list<int>` | An immutable linked list of values of the given type. All elements of the list must have the same type. For example: `[]`, `[3;2;1]`. |
| *type1* `->` *type2* | `int -> string` | A function type, representing a value that will accept values of the first type and compute results of the second type. For example: `(fun x -> x+1)`. |
| *type1* `*` ... `*` *typeN* | `int * string` | A tuple type, such as a pair, triple, or larger combination of types. For example: `(1,"3")`, `(3,2,1)`. |
| *type* `[]` | `int[]` | An array type, indicating a flat, fixed-size mutable collection of values of type *type*. |
| `unit` | `unit` | A type containing a single value `()`, akin to `void` in many imperative languages. |
| `'a`, `'b` | `'a`, `'b`, `'Key`, `'Value` | A variable type, used in generic code. |

Some type constructors such as `list` and `option` are *generic*, which means they can be used to form a range of types by instantiating the generic variables, such as `int list`, `string list`, `int list list`, and so on. Instantiations of generic types can be written using either prefix notation (such as `int list`) or postfix notation (such as `list<int>`). Variable types such as `'a` are placeholders for any type. We discuss generics and variable types in more detail in Chapter 3 and Chapter 5.

### WHAT IS TYPE INFERENCE?

*Type inference* works by analyzing your code to collect constraints. These are collected over the scope of particular parts of your program, such as each file for the F# command-line compiler and each chunk entered in F# Interactive. These constraints must be consistent, thus ensuring your program is well-typed, and you'll get a type error if not. Constraints are collected from top to bottom, left to right, and outside in, which is important because long identifier lookups, method overloading, and some other elements of the language are resolved using the normalized form of the constraint information available at the place where each construct is used.

Type inference also *automatically generalizes* your code, which means that when your code is reusable and generic in certain obvious ways, then it will be given a suitable generic type without you needing to write the generic type down. Automatic generalization is the key to succinct but reusable typed programming. We discuss automatic generalization in Chapter 5.

## Calling Functions

Functions are at the heart of most F# programming, and it's not surprising that the first thing you do is call a library function, in this case, `String.split`:

```
let wordCount text =
    let words = String.split [' '] text
```

The function `String.split` takes two arguments. F# Interactive reveals the type of `String.split` as follows:

```
> String.split;;
val it: char list -> string -> string list
```

To understand this type, let's first investigate `String.split` by running F# Interactive:

```
> String.split [' '] "hello world";;
val it : string list = [ "hello"; "world" ]

> String.split ['a';'e';'i';'o';'u'] "hello world";;
val it : string list = [ "h"; "ll"; " w"; "rld" ]
```

You can see that `String.split` breaks the given text into words using the given characters as delimiters. The first argument is the list of delimiters, and the second is the string to split.

`String.split` takes two arguments, but the arguments are given in a style where the arguments come sequentially after the function name, separated by spaces. This is quite common in F# coding and is mostly a stylistic choice, but it also means functions can be *partially applied* to fewer arguments, leaving a residue function, which is a useful technique you'll look at more closely in Chapter 3.

In the earlier code, you can also see examples of the following:

- Literal characters such as `' '` and `'a'`
- Literal strings such as `"hello world"`
- Literal lists of characters such as `['a';'e';'i';'o';'u']`
- Literal lists of strings such as the returned value `[ "hello"; "world" ]`

We cover literals and lists in detail in Chapter 3. Lists are an important data structure in F#, and you'll see many examples of their use in this book.

### WHAT IS "STRING" IN "STRING.SPLIT"?

The name `String` references the F# module `Microsoft.FSharp.Core.String` in the F# library. This contains a set of simple operations associated with values of the `string` type. It is common for types to have a separate module that contains associated operations. All modules under the `Microsoft.FSharp` namespaces `Core`, `Collections`, `Text`, and `Control` can be referenced by simple one-word prefixes, such as `String.split` and `open String`. Other modules under these namespaces include `List`, `Option`, and `Array`.

Since `String` is a standard .NET type, you can also use functions provided by the .NET Framework runtime located under `System.String` and other important namespaces such as `System.Text.RegularExpresions`. Throughout this book, we use both the .NET Framework library and the F# additions extensively. We give an overview of the most commonly used .NET and F# libraries in Chapter 10.

## Using Data Structures

The next portion of the code is as follows:

```
let wordCount text =
    let words = String.split [' '] text
    let wordSet = Set.of_list words
```

This gives you your first taste of using data structures from F# code, and the last of these lines lies at the heart of the computation performed by `wordCount`. It uses the function `Set.of_list` from the F# library to convert the given words to a concrete data structure that is, in effect, much like the mathematical notion of a set, though internally it is implemented using a data structure based on trees. You can see the results of converting data to a set by using F# Interactive:

```
> Set.of_list ["b";"a";"b";"b";"c" ];;
val it : Set<string> = set [ "a"; "b"; "c" ]

> Set.to_list (Set.of_list ["abc"; "ABC"]);;
val it : string list = [ "ABC"; "abc" ]
```

Here you can see several things:

- F# Interactive prints the contents of structured values such as lists and sets.
- Duplicate elements are removed by the conversion.
- The elements in the set are ordered.
- The default ordering on strings used by sets is case sensitive.

## Using Properties and the Dot-Notation

The next two lines of the `wordCount` function compute the result we're after—the number of duplicate words. This is done by using two properties, `Length` and `Count`, of the values you've computed:

```
let nWords = words.Length
let nDups = words.Length - wordSet.Count
```

F# performs resolution on property names at compile time (or interactively when using F# Interactive, where there is no distinction between compile time and run time). This is done using compile-time knowledge of the type of the expression on the left of the dot—in this case, `words` and `wordSet`. Sometimes a type annotation is required in your code in order to resolve the potential ambiguity among possible property names. For example, the following code uses a type annotation to note that `inp` refers to a list. This allows the F# type system to infer that `Length` refers to a property associated with values of the `list` type:

```
let length (inp : 'a list) = inp.Length
```

Here the `'a` indicates that the length function is generic; that is, it can be used with any type of list. We cover generic code in more detail in Chapter 3 and Chapter 5. Type annotations can be useful documentation and, when needed, should generally be added at the point where a variable is declared.

As you can see from the use of the dot-notation, F# is both a functional language and an object-oriented language. In particular, properties are a kind of *member*, a general term used for any functionality associated with a type or value. Members referenced by prefixing a type name are called *static members*, and members associated with a particular value of a type are called *instance members*; in other words, instance members are accessed through an object on the left of the dot. We discuss the distinction between values, properties, and methods later in this chapter, and we discuss members in full in Chapter 6.

Sometimes explicitly named functions play the role of members. For example, we could have written the earlier code as follows:

```
let nWords = List.length words
let nDups = List.length words - Set.size wordSet
```

You will see both styles in F# code. Some F# libraries don't use members at all or use them only sparingly. However, judiciously using members and properties can greatly reduce the need for trivial get/set functions in libraries, can make client code much more readable, and can allow programmers who use environments such as Visual Studio to easily and intuitively explore the primary features of libraries they write.

If your code does not contain enough type annotations to resolve the dot-notation, you will see an error such as the following:

```
> let length inp = inp.Length;;

  let length inp = inp.Length;;
  ----------------^^^^

stdin(1,17): error: Lookup on object of indeterminate type. A type annotation may
be needed prior to this program point to constrain the type of the object. This
may allow the lookup to be resolved.
```

You can resolve this simply by adding a type annotation as shown earlier.

## Using Tuples

The final part of the `wordCount` function returns the results `nWords` and `nDups` as a *tuple*.

```
let nWords = words.Length
let nDups  = words.Length - wordSet.Size
(nWords,nDups)
```

Tuples are the simplest but perhaps most useful of all F# data structures. A tuple expression is simply a number of expressions grouped together to form a new expression:

```
let site1 = ("www.cnn.com",10)
let site2 = ("news.bbc.com",5)
let site3 = ("www.msnbc.com",4)
let sites = (site1,site2,site3)
```

Here the inferred types of `site1` and `sites` are as follows:

```
val site1 : string * int
val sites : (string * int) * (string * int) * (string * int)
```

Tuples can be decomposed into their constituent components in two ways. For pairs—that is, tuples with two elements—you can explicitly call the functions `fst` and `snd`, which, as their abbreviated names imply, extract the first and second parts of the pair:

```
> fst site1
val it : string = "www.cnn.com"

> let relevance = snd site1
val relevance : int

> relevance;;
val it : int = 10
```

The functions `fst` and `snd` are defined in the F# library and are always available for use by F# programs—here are their simple definitions:

```
let fst (a,b) = a
let snd (a,b) = b
```

More commonly tuples are decomposed using *patterns*, as in the following code:

```
let url,relevance = site1
let site1,site2,site3 = sites
```

In this case, the names in the tuples on the left of the definitions are bound to the respective elements of the tuple value on the right, so again `url` gets the value `"www.cnn.com"` and `relevance` gets the value `10`.

Tuple values are typed, and strictly speaking there are an arbitrary number of families of tuple types, one for pairs holding two values, one for triples holding three values, and so on. This means if you try to use a triple where a pair is expected, then you'll get a type-checking error before your code is run:

```
> let a,b = (1,2,3);;
error: this pattern matches values of type 'int * int' but is here used
with values of type 'int * int * int'. The tuples have different lengths.
```

Tuples are often used to return multiple values from functions, as in the `wordCount` example earlier. They are also often used for multiple arguments to functions, and frequently the tupled output of one function becomes the tupled input of another function. Here is an example that shows a different way of writing the `showWordCount` function defined and used earlier:

```
let showResults (nWords,nDups) =
    printfn "--> %d words in the text" nWords
    printfn "--> %d duplicate words" nDups
let showWordCount text = showResults (wordCount text)
```

The function `showResults` accepts a pair as input, decomposed into `nWords` and `nDups`, matching the results of `wordCount`.

### VALUES AND OBJECTS

In F# everything is a *value*. In some other languages everything is an *object*. In practice, you can use the words largely interchangeably, though F# programmers tend to reserve *object* for special kinds of values:

- Values whose observable properties change as the program executes, usually through the explicit mutation of underlying in-memory data or through external state changes
- Values that refer to data or state that reveal an identity, such as a unique integer stamp or the underlying .NET object identity, where that identity may differ from otherwise identical values
- Values that can be queried to reveal additional functionality, through the use of casts, conversions, and interfaces

F# thus supports objects, but not all values are referred to as objects. We discuss identity and mutation further in Chapter 4.

## Using Imperative Code

The `showWordCount` and `showResults` functions defined in the previous section output the results using a library function called `printfn`:

```
printfn "--> %d words in the text" nWords
printfn "--> %d duplicate words" nDups
```

For those familiar with OCaml, C and C++ `printfn` will look familiar as a variant of `printf`—`printfn` also adds a newline character at the end of printing. Here the pattern `%d` is a placeholder for an integer, and the rest of the text is output verbatim to the console. F# also supports related functions such as `printf`, `sprintf`, and `fprintf`, which are discussed further in Chapter 4. Unlike C/C++, `printf` is a type-safe text formatter, where the F# compiler checks that the subsequent arguments match the requirements of the placeholders. There are also other ways to format text with F#. For example, you could have used the .NET libraries directly:

```
System.Console.WriteLine("--> {0} words in the text", box(nWords))
System.Console.WriteLine("--> {0} duplicate words", box(nDups))
```

Here `{0}` acts as the placeholder, though no checks are made that the arguments match the placeholder before the code is run. The use of `printfn` also shows how you can use sequential expressions to cause effects in the outside world.

As with `let ... in ...` expressions, it is sometimes convenient to write sequential code on a single line. You can do this by separating two expressions by a semicolon (`;`), and again this is the primitive construct of the language. The first expression is evaluated (usually for its side effects), its result is discarded, and the overall expression evaluates to the result of the second. Here is a simpler example of this construct:

```
let two = (printfn "Hello World"; 1+1)
let four = two + two
```

When executed, this code will print `Hello World` precisely once, when the right side of the definition of `two` is executed. F# does not have statements as such: the fragment (`printfn "Hello World"; 1+1`) is an expression, but when evaluated, the first part of the expression causes a side effect, and its result is discarded. It is also often convenient to use parentheses to delimit sequential code. The code from the script could in theory be parenthesized with a semicolon added to make the primitive constructs involved more apparent:

```
(printfn "--> %d words in the text" nWords;
 printfn "--> %d duplicate words" nDups)
```

---

■ **Note** The token `;` is used to write sequential code within expressions, and `;;` is used to terminate interactions with the F# Interactive session. Semicolons are optional when the `#light` syntax option is used and the individual fragments of your sequential code are placed on separate lines beginning at the same column position.

---

# Using .NET Libraries from F#

The true value of F# lies not just in what you can do inside the language but in what you can connect to outside the language. For example, F# does not come with a GUI library. Instead, F# is connected to .NET and via .NET to most of the significant programming technologies available on major computing platforms. To emphasize this, our second sample uses two of the powerful libraries that come with the .NET Framework: `System.Net` and `System.Windows.Forms`. The full sample is in Listing 2-3 and is a script for use with F# Interactive.

**Listing 2-3.** *Using the .NET Framework Windows Forms and Networking Libraries from F#*

```
open System.Windows.Forms

let form = new Form(Visible=true,TopMost=true,Text="Welcome to F#")

let textB = new RichTextBox(Dock=DockStyle.Fill, Text="Here is some initial text")
form.Controls.Add(textB)

open System.IO
open System.Net

/// Get the contents of the URL via a web request
let http(url: string) =
    let req = System.Net.WebRequest.Create(url)
    let resp = req.GetResponse()
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    let html = reader.ReadToEnd()
    resp.Close()
    html
```

```
let google = http("http://www.google.com")
textB.Text <- http("http://news.bbc.co.uk")
```

This example uses several important .NET libraries and will help you to explore some interesting F# language constructs. We walk you through this listing in the following sections.

## Using open to Access Namespaces and Modules

The first thing you see in the sample is the use of `open` to access functionality from the namespace `System.Windows.Forms`:

```
open System.Windows.Forms
```

We discuss namespaces in more detail in Chapter 7. The earlier declaration simply means you can access any content under this path without quoting the long path. If it had not used `open`, you would have to write the following, which is obviously a little verbose:

```
let form = new System.Windows.Forms.Form(Visible=true,TopMost=true,
                                         Text="Welcome to F#")
```

You can also use `open` to access the contents of an F# module such as `Microsoft.FSharp.Core.String` without using long paths. We discuss modules in more detail in Chapter 7.

### MORE ABOUT OPEN

Using `open` is an easy way to access the contents of namespaces and modules. However, there are some subtleties. For example, `open` doesn't actually load or reference a library—instead, it reveals functionality from already-loaded libraries. Libraries themselves are loaded by referring to a particular DLL using `#r` in a script or `-r` as a command-line option. Libraries and namespaces are orthogonal concepts: multiple libraries can contribute functionality to the same namespace, and each library can contribute functionality to multiple namespaces. Often one particular library contributes most of the functionality in a particular namespace. For example, most of the functionality in the `System.Windows.Forms` namespace comes from a library called `System.Windows.Forms.dll`. As it happens, this library is automatically referenced by F#, which is why you haven't needed an explicit reference to the library so far. You can place your code in a namespace by using a `namespace` declaration at the top of your file, discussed further in Chapter 7.

In an earlier example, you saw that `String` in `String.split` referenced a value in the module `Microsoft.FSharp.Core.String`. By default, all F# code is interpreted with an implicit open of the following namespaces and modules:

- `Microsoft.FSharp.Core`: Contains modules such as `String`, `Int32`, `Int64`, and `Option` and contains types such as `'a option` and `'a ref`
- `Microsoft.FSharp.Core.Operators`: Contains values such as `+`, `-`, `*`, `box`, `unbox`, `using`, and `lock`
- `Microsoft.FSharp.Collections`: Contains modules such as `List`, `Seq`, `HashSet`, `Map`, and `Set` and contains types such as `'a list`

- `Microsoft.FSharp.Control`: Contains modules such as `Lazy`, `Async`, and `IEvent` and contains types such as `'a lazy` and `IEvent<'a>`
- `Microsoft.FSharp.Text`: Contains modules such as `Printf`

If two namespaces have types, subnamespaces, and/or modules with identical names, then when you open these, you can access the contents of both using the same shortened paths. For example, the namespace `System` contains a type `String`, and the namespace `Microsoft.FSharp.Core` contains a module `String`. In this case, long identifier lookups such as `String.split` search the values and members under both of these, preferring the most recently opened if there is an ambiguity.

Finally, if you ever have name collisions, you can define your own short aliases for modules and types, such as by using `module MyString = My.Modules.String` and `type SysString = System.String`.

## Using new and Setting Properties

The next lines of the sample script use the keyword `new` to create a top-level window (called a *form*) and set it to be visible. If you run this code in F# Interactive, you will see a top-level window appear with the title text *Welcome to F#*.

```
let form = new Form(Visible=true,TopMost=true,Text="Welcome to F#")
```

Here, `new` is shorthand for calling a function associated with the type `System.Windows.Forms.Form` that constructs a value of the given type—these functions are called *constructors*. Not all F# and .NET types use constructors; you will also see values being constructed using names such as `Create` or via one or more functions in a related module such as `String.create` or `Array.init`. You'll see examples of each throughout this book.

A form is an *object*; that is, its properties change during the course of execution, and it is a handle that mediates access to external resources (the display device, mouse, and so on). Sophisticated objects such as forms often need to be configured, either by passing in configuration parameters at construction or by adjusting properties from their default values after construction. The arguments `Visible=true`, `TopMost=true`, and `Text="Welcome to F#"` set the initial values for three properties of the form. The labels `Visible`, `TopMost`, and `Text` must correspond to either named arguments of the constructor being called or properties on the return result of the operation. In this case, all three are object properties, and the arguments indicate initial values for the object properties.

Most properties on graphical objects can be adjusted dynamically. You set the value of a property dynamically using the notation `obj.Property <- value`. For example, you could also have constructed the form object as follows:

```
open System.Windows.Forms
let form = new Form()
form.Visible <- true
form.TopMost <- true
form.Text <- "Welcome to F#"
```

Likewise, you can watch the title of the form change by running the following code in F# Interactive:

```
form.Text <- "Programming is Fun!"
```

Setting properties dynamically is frequently to configure objects, such as forms, that support many potential configuration parameters that evolve over time.

The object created here was bound to the name `form`. Binding this value to a new name doesn't create a new form; rather, two different handles now refer to the same object (they are said to *alias* the same object). For example, the following code sets the title of the same form, despite it being accessed via a different name:

```
let form2 = form
form2.Text <- "F# Forms are Fun"
```

### VALUES, METHODS, AND PROPERTIES

Here are the differences between values, methods, and properties:

- *Simple values*: Functions, parameters, and top-level items defined using `let` or pattern matching. Examples: `form`, `text`, `wordCount`.
- *Methods*: Function values associated with types. Interfaces, classes, and record and union types can all have associated methods. Methods can be overloaded (see Chapter 6) and must be applied immediately to their arguments. Examples: `System.Net.WebRequest.Create` and `resp.GetResponseStream`.
- *Properties*: A shorthand for invoking method members that read or write underlying data. Interfaces, classes, and record and union types can all have associated properties. Examples: `System.DateTime.Now` and `form.TopMost`.
- *Indexer properties*: A property can take arguments, in which case it is an *indexer property*. Indexer properties named `Item` can be accessed using the `.[_]` syntax. Examples: `vector.[3]` and `matrix.[3,4]`.

The next part of the sample creates a new `RichTextBox` control and stores it in a variable called `textB`. A control is typically an object with a visible representation, or more generally, an object that reacts to operating system events related to the windowing system. A form is one such control, but there are many others. A `RichTextBox` control is one that can contain formatted text, much like a word processor.

```
let textB = new RichTextBox(Dock= DockStyle.Fill)
form.Controls.Add(textB)
```

## Fetching a Web Page

The second half of Listing 2-3 uses the `System.Net` library to define a function `http` to read HTML web pages. You can investigate the operation of the implementation of the function by entering the following lines into F# Interactive:

```
> open System;;
> open System.IO;;
> open System.Net;;

> let req = WebRequest.Create("http://www.microsoft.com");;
val req : WebRequest

> let resp = req.GetResponse();;
val resp : WebResponse

> let stream = resp.GetResponseStream();;
val stream : Stream

> let reader = new StreamReader(stream);;
val reader : StreamReader

> let html = reader.ReadToEnd();;
val html : string

> textB.Text <- html;;
```

The final line will set the contents of the text box form to the HTML contents of the Microsoft home page. Let's take a look at this code line by line.

The first line of the code creates a `WebRequest` object using the static method `Create`, a member of the type `System.Net.WebRequest`. The result of this operation is an object that acts as a handle to a running request to fetch a web page—you could, for example, abandon the request or check to see whether the request has completed. The second line calls the instance method `GetResponse`. The remaining lines of the sample get a stream of data from the response to the request using `resp.GetResponseStream()`, make an object to read this stream using `new StreamReader(stream)`, and read the full text from this stream. We cover .NET I/O in more detail in Chapter 4, but for now you can test by experimentation in F# Interactive that these actions do indeed fetch the HTML contents of a web page. The inferred type for `http` that wraps up this sequence as a function is as follows:

```
val http : string -> string
```

■ **Note** *Static members* are items qualified by a concrete type or module. Examples include `System.String.Compare`, `System.DateTime.Now`, `List.map`, and `String.split`. *Instance members* are methods, properties, and values qualified by an expression. Examples include `form.Visible`, `resp.GetResponseStream()`, and `cell.contents`.

### XML HELP IN VISUAL STUDIO

In a rich editor such as Visual Studio 2005, you can easily find out more about the functionality of .NET libraries by hovering your mouse over the identifiers in your source code. For example, if you hover over `Dock` in `textB.Dock`, you'll see the XML help shown here:

# Summary

In this chapter, you took a look at some simple interactive programming with F# and .NET. Along the way, you met many of the constructs you'll use in your day-to-day F# programming. In the next chapter, you'll take a closer look at these and other constructs that are used to perform compositional and succinct functional programming in F#.

---

■ **Note** In Chapter 3 you'll use some of the functions you defined in this chapter, so if you're using F# Interactive, you might want to leave your session open as you proceed.

---

CHAPTER 3

# Introducing Functional Programming

F# is effective and productive primarily because it is built on the tried and tested constructs of functional programming. In this chapter, we cover the core building blocks of functional programming with F#, including simple types and function values, pattern matching, lists, options, and sequences, as well as how to declare some of your own simple types. We cover imperative programming, generics, and object-oriented programming in Chapters 4 through 6.

## Getting Started with F# Arithmetic

We first cover the most common base types of data manipulated in F# code, beginning with the basics of F# arithmetic.

### Basic Literals

Table 3-1 lists the basic numeric types used in F# code and their corresponding literal forms. We've also listed the non-numeric types `bool` and `unit`.

**Table 3-1.** *Basic Types and Literals*

| Type | Description | Sample Literals | .NET Name |
|---|---|---|---|
| `bool` | True/false values | `true`, `false` | `System.Boolean` |
| `byte` | 8-bit unsigned integers | `0uy`, `19uy`, `0xFFuy` | `System.Byte` |
| `sbyte` | 8-bit signed integers | `0y`, `19y`, `0xFFy` | `System.SByte` |
| `int16` | 16-bit signed integers | `0s`, `19s`, `0x0800s` | `System.Int16` |
| `uint16` | 16-bit unsigned integers | `0us`, `19us`, `0x0800us` | `System.UInt16` |
| `int`, `int32` | 32-bit signed integers | `0`, `19`, `0x0800`, `0b0001` | `System.Int32` |
| `uint32` | 32-bit unsigned integers | `0u`, `19u`, `0x0800u` | `System.UInt32` |
| `int64` | 64-bit signed integers | `0L`, `19L`, `0x0800L` | `System.Int64` |

**Table 3-1.** *Basic Types and Literals (Continued)*

| Type | Description | Sample Literals | .NET Name |
|---|---|---|---|
| `uint64` | 64-bit unsigned integers | `0UL`, `19UL`, `0x0800UL` | `System.UInt64` |
| `nativeint` | Machine-sized signed integers | `0n`, `19n`, `0x0800n` | `System.IntPtr` |
| `unativeint` | Machine-sized unsigned integers | `0un`, `19un`, `0x0800un` | `System.UIntPtr` |
| `single`, `float32` | 32-bit IEEE floating-point | `0.0f`, `19.7f`, `1.3e4f` | `System.Single` |
| `double`, `float` | 64-bit IEEE floating-point | `0.0`, `19.7`, `1.3e4` | `System.Double` |
| `decimal` | High-precision decimal values | `0M`, `19M`, `19.03M` | `System.Decimal` |
| `bigint` | Arbitrarily large integers | `0I`, `19I` | `Math.BigInt` |
| `bignum` | Arbitrary-precision rationals | `0N`, `19N` | `Math.BigNum` |
| `unit` | The type with only one value | `()` | `Core.Unit` |

## Arithmetic Operators

Table 3-2 lists the most commonly used arithmetic operators. These are overloaded to work with all the numeric types listed in Table 3-1.

**Table 3-2.** *Arithmetic Operators and Examples*

| Operator | Description | Sample Use on `int` | Sample Use on `float` |
|---|---|---|---|
| `+` | Unchecked addition | `1 + 2` | `1.0 + 2.0` |
| `-` | Unchecked subtraction | `12 - 5` | `12.3 - 5.4` |
| `*` | Unchecked multiplication | `2 * 3` | `2.4 * 3.9` |
| `/` | Division | `5 / 2` | `5.0 / 2.0` |
| `%` | Modulus | `5 % 2` | `5.4 % 2.0` |
| `-` | Unary negation | `-(5+2)` | `-(5.4+2.4)` |

The behavior of these and other operators can be extended for user-defined types, a topic we cover in Chapter 6. In F#, addition, subtraction, and multiplication over integers are unchecked; that is, if overflow or underflow occurs beyond the representable range, then wraparound occurs. For example, `2147483647` is the largest representable 32-bit integer of the `int` type:

```
> 2147483647+1;;
val it : int = -2147483648
```

Checked versions of arithmetic operators that raise `System.OverflowException` exceptions can be accessed by opening the `Microsoft.FSharp.Core.Operators.Checked` module. If avoiding

overflow is a priority, then using the `decimal`, `bigint`, and `bignum` types is recommended. Division by zero raises a `System.DivideByZeroException` exception, except in the case of floating-point numbers where it returns one of the special floating point numbers `Infinity` or `-Infinity`. Operator overloading interacts with type inference—if a use of an overloaded operator is not otherwise constrained to work on a particular type, then F# will assume it works on 32-bit integers. To constrain a use of an operator to a particular type, you must give a type annotation that has the effect of telling the compiler the type on the left of the two arguments to the binary operator. For example, in the absence of additional type information, the following function is assumed to work with integers:

```
> let doubleAndAdd a b = a * a + b;;
val doubleAndAdd: int -> int -> int
```

A single type annotation on `a` is sufficient to indicate that `a * a` is an operation on `float` values, and thus returns a `float` value, and that `a * a + b` is also an operation on `float`:

```
> let doubleAndAdd (a:float) b = a * a + b;;
val doubleAndAdd: float -> float -> float
```

If you want, you can also give full type annotations for the arguments and return type of a function:

```
> let doubleAndAdd (a:float) (b:float) : float = a * a + b;;
val doubleAndAdd: float -> float -> float
```

## Bitwise Operations

All the integer types listed in Table 3-1 support bitwise manipulations on their underlying representations. Table 3-3 shows the bitwise manipulation operators.

**Table 3-3.** *Bitwise Arithmetic Operators and Examples*

| Operator | Description | Sample Use | Result |
|---|---|---|---|
| `&&&` | Bitwise “and” | `0x65 &&& 0x0F` | `0x05` |
| `\|\|\|` | Bitwise “or” | `0x65 \|\|\| 0x18` | `0x7D` |
| `^^^` | Bitwise “exclusive or” | `0x65 ^^^ 0x0F` | `0x6A` |
| `~~~` | Bitwise negation | `~~~0x65` | `0xFFFFFF9a` |
| `<<<` | Left shift | `0x01 <<< 3` | `0x08` |
| `>>>` | Right shift (arithmetic if signed) | `0x65 >>> 3` | `0x0C` |

The following sample shows how to use these operators to encode 32-bit integers into 1, 2, or 5 bytes, represented by returning a list of integers. Integers in the range 0 to 127 return a list of length 1:

```
let encode (n: int32) =
    if   (n >= 0    && n <= 0x7F)   then [ n ]
    elif (n >= 0x80 && n <= 0x3FFF) then [ (0x80 ||| (n >>> 8)) &&& 0xFF;
                                           (n &&& 0xFF) ]
    else  [ 0xC0; ((n >>> 24) &&& 0xFF);
                  ((n >>> 16) &&& 0xFF);
                  ((n >>> 8)  &&& 0xFF);
                   (n         &&& 0xFF) ]
```

Here's an example of the function in action:

```
> encode 32;;
val it : int32 list = [32]

> encode 320;;
val it : int32 list = [129; 64]

> encode 32000;;
val it : int32 list = [192; 0; 0; 125; 0]
```

## Arithmetic Conversions

Numeric types are not implicitly converted—conversions between different numeric types must be made explicitly. You do this by using overloaded conversion operators. These work in the same way as overloaded infix operators such as + and *. Table 3-4 shows the primary conversion operators.

**Table 3-4.** *Overloaded Arithmetic Conversions and Examples*

| Operator | Description | Sample Use | Result |
|---|---|---|---|
| `sbyte` | Convert/truncate to `sbyte` | `sbyte (-17)` | `-17y` |
| `byte` | Convert/truncate to `byte` | `byte 255` | `255uy` |
| `int16` | Convert/truncate to `int16` | `int16 0` | `0s` |
| `uint16` | Convert/truncate to `uint16` | `uint16 65535` | `65535us` |
| `int/int32` | Convert/truncate to `int32` | `int 17.8` | `17` |
| `uint32` | Convert/truncate to `uint32` | `uint32 12` | `12u` |
| `int64` | Convert/truncate to `int64` | `int64 (-100.4)` | `-100L` |
| `uint64` | Convert/truncate to `uint64` | `uint64 1` | `1UL` |
| `float32` | Convert to `float32/single` | `float32 65` | `65.0f` |
| `float` | Convert to `float/double` | `float 65` | `65.0` |

These conversions are all unchecked in the sense that they will not raise exceptions. Again, the `Microsoft.FSharp.Core.Operators.Checked` module has corresponding definitions of these operators. An alternative is to use the .NET static methods contained in the type `System.Convert`, such as `System.Convert.ToDouble( )`. These do perform checking, which means they raise an exception if the source number can't be represented within the numeric range of the target type. As with many .NET constructs, uses of `System.Convert` methods may require type annotations to resolve overloading, discussed further in Chapter 5 and Chapter 6.

## Arithmetic Comparisons

When used with numeric values, the binary comparison operators `=`, `<>`, `<`, `<=`, `>`, `>=`, `min`, and `max` perform comparisons according to the natural ordering for each particular numeric type. You can also use these operators on other data types, such as to compare lists of integers, and you can customize their behavior for new types you define. We discuss generic comparison in detail in Chapter 5, and we discuss customizing generic comparison in Chapter 8.

When used with floating-point values, these operators implement the IEEE semantics for `NaN` (Not a Number) values. For example, `(NaN = NaN)` is false, as is `(NaN <= NaN)` and `(NaN < NaN)`.

## Overloaded Math Functions

The module `Microsoft.FSharp.Core.Operators` includes the definition of a number of useful overloaded math operators. These are shown in Table 3-5 and are overloaded either on a suitable range of integer types or on the basic floating-point types.

**Table 3-5.** *Overloaded Math Functions and Examples*

| Function | Description | Sample Use | Result |
|---|---|---|---|
| `abs` | Absolute value of signed numeric types | `abs (-10.0f)` | `10.0f` |
| `cos`, `sin`, `tan` | Trigonometric functions | `cos 0.0` | `1.0` |
| `cosh`, `sinh`, `tanh` | Hyperbolic trigonometric functions | `cos 1.0` | `1.543080635` |
| `acos`, `asin`, `atan`, `atan2` | Inverse trigonometric functions | `acos 1.0` | `0.0` |
| `ceil`, `floor` | Round up, round down | `ceil 1.001` | `2.0` |
| `truncate` | Round toward zero | `truncate 8.9` | `8.0` |
| `exp`, `log`, `log10` | Exponent, logarithm, base-10 logarithm | `exp 1.0` | `2.718281828` |
| `( ** )` | Power | `2.0 ** 4.0` | `16.0` |

# Introducing Simple Strings

The F# type `string` is an abbreviation for .NET type `System.String` and represents a sequence of Unicode UTF-16 characters. In the following sections, we briefly introduce strings and the most useful functions for formatting them.

## Working with String Literals and Primitives

Table 3-6 shows the different forms for writing string literals.

**Table 3-6.** *String and Character Literals*

| Example | Kind | Type |
|---|---|---|
| `"Humpty Dumpty"` | String | `string` |
| `"c:\\Program Files"` | String | `string` |
| `@"c:\Program Files"` | Verbatim string | `string` |
| `"xyZy3d2"B` | Literal byte array | `byte[]` |
| `'c'` | Character | `char` |

Table 3-7 shows the escape characters that you can use in strings and characters.

**Table 3-7.** *Escape Characters in Nonverbatim Strings*

| Escape | Character | ASCII/Unicode Value | Examples |
|---|---|---|---|
| `\n` | New line | 10 | `"\n"` |
| `\r` | Carriage return | 13 | `"\r"` |
| `\t` | Tab | 9 | `"\t"` |
| `\b` | Backspace | 8 | |
| `\`*`NNN`* | Trigraph | NNN | `"\032"` *(space)* |
| `\u`*`NNNN`* | Unicode character | NNNN | `"\u00a9"` (©) |
| `\U`*`NNNNNNNN`* | Long Unicode character | NNNN NNNN | `"\U00002260"` (≠) |

As shown in Table 3-6, a literal form is also available for arrays of bytes: the characters are interpreted as ASCII characters, and non-ASCII characters can be embedded by escape codes. This can be useful when working with binary protocols:

```
> "MAGIC"B;;
val it : byte [] = [|77uy; 65uy; 71uy; 73uy; 67uy|]
```

Verbatim string literals are particularly useful for file and path names that contain the backslash character (\):

```
> let dir  = @"c:\Program Files";;
val dir : string
```

You can also use multiline string literals:

```
> let s = "All the kings horses
- and all the kings men";;
val s : string
```

The operator `.[]` is used to access the elements of a string, and the property `.Length` retrieves its length:

```
> let s = "Couldn't put Humpty";;
val s : string

> s.Length;;
val it : int = 19

> s.[13];;
val it : char = 'H'
```

Strings are immutable; that is, a string value cannot be modified once built. For example, the `Substring` method on the `string` type doesn't modify the original string but returns a new string representing the result. As mentioned in Chapter 2, immutability is a key concept for many F# values, and you'll encounter it at many places in this book. If you attempt to mutate a string, you will get an error like the one shown here:

```
> let s = "Couldn't put Humpty";;
val s : string = "Couldn't put Humpty"

> s.[13] <- 'h';;

  s.[13] <- 'h';;
  ^^
stdin(75,0): error: FS0001: Type error in the use of the overloaded operator
'set_Item'. The type 'string' does not support any operators named 'set_Item'
```

## Building Strings

The simplest way to build strings is via concatenation using the + operator:

```
> "Couldn't put Humpty" + " " + "together again";;
val it : string = "Couldn't put Humpty together again"
```

You can also build strings using objects of the .NET type `System.Text.StringBuilder`. These objects are mutable buffers that you can use to accumulate and modify text, and they are more efficient than repeated uses of the + operator. Here's an example:

```
> let buf = new System.Text.StringBuilder();;
val buf : System.Text.StringBuilder

> buf.Append("Humpty Dumpty");;

> buf.Append(" sat on the wall");;

> buf.ToString();;
val it : string = "Humpty Dumpty sat on the wall"
```

■ **Note** For compatibility with OCaml, the ^ operator can also be used for string concatenation, though it is generally used only when cross-compiling code with OCaml.

# Working with Lists and Options

Some of the foundational data structures of F# coding are tuples, lists, and options. In the following sections, we discuss these and some related topics by example.

## Using F# Lists

F# lists are a common data structure used in functional programming. You saw some examples of concrete lists when using the results of the `String.split` function in Chapter 2. Table 3-8 shows the primitive constructs for building lists.

**Table 3-8.** *Some List-Related Language Constructs and Operators*

| Operator/Expression | Description | Examples |
|---|---|---|
| `[]` | The empty list | `[]` |
| *expr* `::` *expr* | "Cons" an element with a list | `1 :: [2; 3]` |
| `[`*expr*`; ...;` *expr*`]` | A list value | `[1; 2; 3]` |
| `[`*expr* `..` *expr*`]` | A range of integers | `[1 .. 99]` |
| `[ for x in list ->` *expr* `]` | A generated list (see end of chapter) | `[ for x in 1..99 -> x * x ]` |
| *expr* `@` *expr* | Concatenates two lists | `[1; 2] @ [3]` |

Here are some basic list values:

```
let oddPrimes = [3; 5; 7; 11]
let morePrimes = [13; 17]
let primes = 2 :: (oddPrimes @ morePrimes)
```

The value and type of `primes` are as follows:

```
val primes : int list = [2; 3; 5; 7; 11; 13; 17]
```

It is important to note that lists are immutable: the "cons" `::` and "append" `@` operations do not modify the original lists; instead, they create new lists. You can see this in the following interactive session:

```
> let people = [ "Adam"; "Dominic"; "James" ];;
val people : string list

> people;;
val it : string list = [ "Adam"; "Dominic"; "James" ]

> "Chris" :: people;;
val it : string list = [ "Chris"; "Adam"; "Dominic"; "James" ]

> people;;
val it : string list = [ "Adam"; "Dominic"; "James" ]
```

Note that `people` has not been changed by the construction of a new list using the "cons" operator. That is, lists and tuples are unchangeable, immutable values. F# lists are represented in memory as linked lists, and each F# list value is a "cons" cell containing a value plus a pointer to the next chain in the list, or else it is a special "nil" object. When you create a new list using the `::` operator, then the tail of the new list will point to the old list, which ensures that the inner memory associated with lists is often reused as part of multiple list values. You can decompose lists from the head downward by using *pattern matching*. You saw some simple examples of pattern matching on tuples in Chapter 2, and we'll look at pattern matching in more detail in "Getting Started with Pattern Matching" later in this chapter. Here is an example of using pattern matching with lists:

```
let printFirst primes =
    match primes with
    | h :: t -> printfn "The first prime in the list is %d" h
    | [] -> printfn "No primes found in the list"
```

```
> printFirst oddPrimes;;
The first prime in the list is 3
val it : unit = ()
```

The first line after the `match` is a pattern-matching rule that matches the input `primes` against the pattern `h :: t`. If `primes` is a nonempty list, then the match will be successful, and the first `printfn` will be executed with `h` bound to the head of the list and `t` to its tail. The second line considers the case where `primes` is an empty list. Note that the `::` and `[]` symbols can be used both to build up lists in expressions and to decompose them in pattern matching. The F# library also includes a module `List` that contains some useful functions related to programming with lists. You'll be seeing many of these functions in the next section and throughout this book. Table 3-9 shows some of these.

F# lists are not appropriate for all circumstances; for example, very large data structures should probably be represented using arrays or other data structures or even managed by an external tool such as a relational database. We discuss a number of immutable data structures in the "Some Common Immutable Data Structures" sidebar.

**Table 3-9.** *Some Sample Functions in the List Module*

| Function | Type | Description |
|---|---|---|
| `List.length` | `: 'a list -> int` | Returns the length of the list. |
| `List.hd` | `: 'a list -> 'a` | Returns the first element of a nonempty list. |
| `List.tl` | `: 'a list -> 'a list` | Returns all the elements of a nonempty list except the first. |
| `List.init` | `: int -> (int -> 'a) -> 'a list` | Returns a new list of length given by the first parameter and elements generated by the second function parameter. |
| `List.append` | `: 'a list -> 'a list -> 'a list` | Returns a new list containing the elements of the first list followed by the elements of the second list. |
| `List.filter` | `: ('a -> bool) -> 'a list -> 'a list` | Returns a new list containing only those elements of the original list where the function returns `true`. |
| `List.map` | `: ('a -> 'b )  -> 'a list -> 'b list` | Returns a new list where the function has been applied to each element of the list. |
| `List.iter` | `: ('a -> unit) -> 'a list -> unit` | Executes the given function for each element of the list. |
| `List.unzip` | `: ('a * 'b) list -> 'a list * 'b list` | Returns two new lists containing the first and second elements of the pairs in the input list. |
| `List.zip` | `: 'a list -> 'b list -> ('a * 'b) list` | Returns a new list containing the elements of the two input lists combined pairwise. The input lists must be the same length; otherwise, an exception is raised. |
| `List.to_array` | `: 'a list -> 'a[]` | Converts the list to an array. |
| `List.of_array` | `: 'a[]    -> 'a list` | Converts the array to a list. |

Here are examples of how to use some of the functions from Table 3-9. The last two examples use *function values*, which we cover in more detail in "Introducing Function Values" later in this chapter.

```
> List.hd [5; 4; 3];;
val it : int = 5

> List.tl [5; 4; 3];;
val it : int list = [ 4; 3 ]

> List.map (fun x -> x*x) [1; 2; 3];;
val it : int list = [ 1; 4; 9 ]

> List.filter (fun x -> x % 3 = 0) [2; 3; 5; 7; 9];;
val it : int list = [ 3; 9 ]
```

### SOME COMMON IMMUTABLE DATA STRUCTURES

Data structures are generally divided between *mutable* and *immutable*, a distinction touched upon in Chapter 2 and covered in more detail in Chapter 4. Immutable data structures are sometimes called *persistent* or simply *functional*. Here are some of the immutable data structures commonly used with F#:

- *Tuple values and option values*: These are immutable and are basic workhorses of F# programming.
- *Immutable linked lists of type* `'a list`: These are cheap to access from the left end. They are inefficient for random access lookup because the list must be traversed from the left for each lookup, that is, random access lookup is O(*n*) where *n* is the number of elements in the collection. The full name of this type is `Microsoft.FSharp.Collections.List<'a>`.
- *Immutable sets based on balanced trees*: We show some example uses of immutable sets in Chapter 2, and an implementation is provided via the type `Set<'a>` in the F# library namespace `Microsoft.FSharp.Collections`. These are cheap to add, access, and union, with O(log(*n*)) access times, where *n* is the number of elements in the collection. Because the type is immutable, internal nodes can be shared between different sets.
- *Immutable maps based on balanced trees*: These are similar to immutable sets but associate keys with values (that is, they are immutable dictionaries). One implementation of these is provided via the F# library type `Map<'key,'value>` in `Microsoft.FSharp.Collections`. As with sets, these have O(log(*n*)) access times.

We cover imperative programming and mutable data structures in Chapter 4.

## Using F# Option Values

Like lists and tuples, option values are simple constructs frequently used as the workhorses in F# coding. An option is simply either a value `Some(v)` or the absence of a value `None`. For example,

options are useful for returning the value of a search where you might or might not have a result. You shall see in "Defining Discriminated Unions" that the `option` type is defined in the F# library as follows:

```
type 'a option =
    | None
    | Some of 'a
```

The following is a data structure that uses options to represent the (optional) parents of some well-known characters:

```
> let people = [ ("Adam", None);
                 ("Eve" , None);
                 ("Cain", Some("Adam","Eve"));
                 ("Abel", Some("Adam","Eve")) ];;
val people : (string * (string *string) option) list
```

Pattern matching is frequently used to examine option values:

```
> let showParents (name,parents) =
      match parents with
      | Some(dad,mum) -> printfn "%s has father %s, mother %s" name dad mum
      | None          -> printfn "%s has no parents!" name;;
val showParents : (string * (string * string) option) -> unit

> showParents ("Adam",None);;
Adam has no parents
val it : unit = ()
```

The F# library also includes a module `Option` that contains some useful functions for programming with options. Table 3-10 shows some of these. Although it is easy to code these by hand using pattern matching, it can also be useful to learn and rely on the standard definitions.

**Table 3-10.** *Some Sample Functions in the Option Module*

| Function | Type | Description |
|---|---|---|
| `Option.get` | `: 'a option -> 'a` | Returns the value of a `Some` option. |
| `Option.is_some` | `: 'a option -> bool` | Returns `true` for a `Some` option. |
| `Option.map` | `: ('a -> 'b ) -> 'a option -> 'b option` | Given `None`, returns `None`. Given `Some(x)`, returns `Some(f x)`, where `f` is the given function. |
| `Option.iter` | `: ('a -> unit) -> 'a option -> unit` | Applies the given function to the value of a `Some` option; otherwise, does nothing. |

## Using Option Values for Control

You can use option values for both data and control; they're often used to represent the success or failure of a computation. This can be useful when catching an exception, as shown in the following sample (this sample uses the function `http` from Chapter 2):

```
let fetch url =
    try Some(http(url))
    with :? System.Net.WebException -> None
```

We describe exceptions in more detail in Chapter 4—what matters here is that if a network error occurs during the HTTP request, then the exception will be caught, and the result of the `fetch` function will be the value `None`. Successful web page requests will return a `Some` value. Option values can then be discriminated and decomposed using pattern matching, as shown here:

```
> match (fetch "http://www.nature.com") with
  | Some(text) -> printfn "text = %s" text
  | None -> printfn "**** no web page found";;
text = <HTML> ... </HTML>  (note: the HTML is shown here if connected to the web)
val it : unit = ()
```

# Working with Conditionals: && and ||

A basic control construct in F# programming is `if/then/elif/else`. Here's an example:

```
let round x =
    if x >= 100 then 100
    elif x < 0 then 0
    else x
```

Conditionals are really shorthand for pattern matching; for example, the previous code could have been written like this:

```
let round x =
    match x with
    | _ when x >= 100 -> 100
    | _ when x < 0    -> 0
    | _               -> x
```

Conditionals are always guarded by a Boolean-valued expression. You can build them using && and || (the "and" and "or" operators) as well as any library functions that return Boolean values:

```
let round2 (x,y) =
    if x >= 100 || y >= 100 then 100,100
    elif x < 0 || y < 0 then 0,0
    else x,y
```

The operators && and || have the usual "shortcut" behavior in that the second argument of && is evaluated only if the first evaluates to `true`, and likewise, the second argument of || is evaluated only if the first evaluates to `false`.

---

■ **Note** If you don't use the `#light` lightweight syntax option, then when you combine conditionals with imperative code, you will sometimes need to use parentheses, as in `(...)`, or `begin`/`end` to delimit the regions covered by each branch of the conditional, for example, `if x > 100 then (...)`. If you use the `#light` syntax option, then these are optional, as long as the branches are correctly indented from the `if`, `elif`, and `else` tokens of the construct.

---

# Defining Recursive Functions

One of the fundamental building blocks of computation in F# is recursion. The following code shows a simple well-known recursive function:

```
> let rec factorial n = if n <= 1 then 1 else n * factorial (n-1);;
val factorial : int -> int

> factorial 5;;
val it : int = 120
```

This example shows that a recursive function is simply one that can call itself as part of its own definition. Recursive functions are introduced by `let rec`. Functions are not recursive by default, since it is wise to isolate recursive functions to help you control the complexity of your algorithms and keep your code maintainable. It may help to visualize the execution of `factorial 5` in the following way (though note that in reality F# executes the function using efficient native code):

```
factorial 5
= 5 * factorial 4
= 5 * (4 * factorial 3)
= 5 * (4 * (3 * factorial 2))
= 5 * (4 * (3 * (2 * factorial 1 )))
= 5 * (4 * (3 * (2 * 1)))
= 5 * (4 * (3 * 2))
= 5 * (4 * 6)
= 5 * 24
= 120
```

As with all calls, the execution of the currently executing instance of the function is suspended while a recursive call is made.

Many of the operators you have encountered so far can be coded as recursive functions. For example, the following is one possible implementation of `List.length`:

```
let rec length l =
    match l with
    | [] -> 0
    | h :: t -> 1 + length t
```

Likewise, many functions such as `List.map` are implemented using recursive functions.

Recursion is sometimes used as a means of programming particular patterns of control. This is usually used in contexts where functions have deliberate side effects. For example, the following code repeatedly fetches the HTML for a particular web page, printing each time it is fetched:

```
let rec repeatFetch url n =
    if n > 0 then
        let html = http url
        printfn "fetched <<< %s >>> on iteration %d" html n
        repeatFetch url (n-1)
```

Recursion is powerful but not always the ideal way to encode either data manipulations or control constructs, at least if other techniques are readily available. For example, the previous program could be implemented using a `for` loop, as explained in Chapter 4, which would be clearer. Likewise, explicit recursion should typically be avoided if an operator is available that captures the pattern of recursion being used. For example, many explicit loops and recursive functions can be replaced by uses of functions such as `List.map` and `Array.map`.

A typical error with recursion is to forget to decrement a variable at the recursive call. For example, the author incorrectly entered the following nonterminating function when writing this chapter:

```
let rec badFactorial n = if n <= 0 then 1 else n * badFactorial n
```

You should always check your recursive calls to ensure that the function is tending toward termination, that is, that the arguments are approaching the base case. This is called *well-founded recursion*.

You can define multiple recursive functions simultaneously by separating the definitions with `and`. These are called *mutually recursive functions*. For example:

```
let rec even n = (n = 0u) || odd(n-1u)
and     odd n = (n <> 0u) && even(n-1u)
```

This gives the following types:

```
val even : uint32 -> bool
val odd : uint32 -> bool
```

Of course, a more efficient, nonrecursive implementation of these is available!

```
let even (n:uint32) = (n % 2u) = 0u
let odd  (n:uint32) = (n % 2u) = 1u
```

You must sometimes take care with recursive functions to ensure that they are *tail recursive*, or else the computation stack of your program may be exhausted by large inputs. This is particularly important for library functions or functions that operate over large data structures with very large numbers of recursive calls. Indeed, the implementation of `length` shown previously is not tail recursive. We discuss tail recursion in depth in Chapter 8.

# Introducing Function Values

In this section, we cover the foundational building block of F# functional programming: function values. We begin with a simple and well-known example: using function values to transform one list into another.

One of the primary uses of F# lists is as a general-purpose concrete data structure for storing ordered lists input and ordered results. Input lists are often transformed to output lists using "aggregate" operations that transform, select, filter, and categorize elements of the list according to a range of criteria. These aggregate operations provide an excellent introduction to how to use function values. Let's take a closer look at this in the following code sample, which continues from the definition of `http` in Listing 2-2 in Chapter 2:

```
> let sites = [ "http://www.live.com";
                "http://www.google.com" ];;
val sites : string list

> let fetch url = (url, http url);;
val fetch : string -> string * string

> List.map fetch sites;;
val it : (string * string) list
= [ ("http://www.live.com", "<html>...</html>");
    ("http://www.google.com", "<html>...</html>"); ]
```

The first interaction defines `sites` as a literal list of URLs, and the second defines the function `fetch`. The third calls the aggregate operation `List.map`. This accepts the function value `fetch` as the first argument and the list `sites` as the second argument. The function applies `fetch` to each element of the list and collects the results in a new list.

Types are one useful way to help learn what a function does. Here's the type of `List.map`:

```
> List.map;;
val it: ('a -> 'b) -> 'a list -> 'b list
```

This says `List.map` accepts a function value as the first argument and a list as the second argument, and it returns a list as the result. The function argument can have any type `'a -> 'b`, and the elements of the input list must have a corresponding type `'a`. The symbols `'a` and `'b` are called *type parameters*, and functions that accept type parameters are called *generic*. We discuss type parameters in detail in Chapter 5.

■ **Tip** You can often deduce the behavior of a function from its type, especially if its type involves type parameters. For example, look at the type of `List.map`. Using type parameters, you can observe that the type `'a list` of the input list is related to the type `'a` accepted by the function passed as the first parameter. Similarly, the type `'b` returned by this function is related to the type `'b list` of the value returned by `List.map`. From this it is reasonable to conclude that `List.map` will call the function parameter for items in the list and construct its result using the values returned.

## Using Anonymous Function Values

Function values are so common in F# programming that it is convenient to define them without giving them names. Here is a simple example:

```
> let primes = [2; 3; 5; 7];;
val primes : int list

> let primeCubes = List.map (fun n -> n * n * n) primes;;
val primeCubes: int list

> primeCubes;;
val it : int list = [8; 27; 125; 343]
```

The definition of `primeCubes` uses the *anonymous function value* (`fun n -> n * n * n`). These are similar to function definitions but are unnamed and appear as an expression rather than as a `let` declaration. `fun` is a keyword meaning "function," `n` represents the argument to the function, and `n * n * n` is the result of the function. The overall type of the anonymous function expression is `int -> int`. You could use this technique to avoid defining the intermediary function `fetch` in the earlier sample:

```
let resultsOfFetch = List.map (fun url -> (url, http url)) sites
```

You will see anonymous functions throughout this book. Here is another example:

```
> List.map (fun (_,p) -> String.length p) resultsOfFetch;;
val it : int list = [3932; 2827 ]
```

Here you see two things:

- The argument of the anonymous function is a tuple pattern. Using a tuple pattern automatically extracts the second element from each tuple and gives it the name `p` within the body of the anonymous function.
- Part of the tuple pattern is a wildcard pattern, indicated by an underscore. This indicates you don't care what the first part of the tuple is; you're interested only in extracting the length from the second part of the pair.

## Computing with Aggregate Operators

Functions such as `List.map` are called *aggregate operators*, and they are powerful constructs, especially when combined with the other features of F#. Here is a longer example that uses the aggregate operators `List.filter` and `List.map` to count the number of URL links in an HTML page and then collects stats on a group of pages (this sample uses the function `http` defined in Chapter 2):

```
let delimiters = [ ' '; '\n'; '\t'; '<'; '>'; '=' ]
let getWords s = String.split delimiters s
let getStats site =
    let url = "http://" + site
    let html = http url
    let hwords = html |> getWords
    let hrefs = html |> getWords |> List.filter (fun s -> s = "href")
    (site,html.Length, hwords.Length, hrefs.Length)
```

Here we use the function `getStats` with three web pages:

```
> let sites = [ "www.live.com";"www.google.com";"search.yahoo.com" ];;
val sites : string list

> sites |> List.map getStats;;
val it : (string * int * int * int) list
  = [("www.live.com", 7728, 1156, 10);
     ("www.google.com", 2685, 496, 14);
     ("search.yahoo.com", 10715, 1771, 38)]
```

The function `getStats` computes the length of the HTML for the given website, the number of words in the text of that HTML, and the approximate number of links on that page.

The previous code sample extensively uses the `|>` operator to *pipeline* operations, discussed in the "Pipelining with `|>`" sidebar. The F# library design ensures that a common, consistent set of aggregate operations is defined for each structured type. Table 3-11 shows how the same design patterns occur for the map abstraction.

**Table 3-11.** *A Recurring Aggregate Operator Design Pattern from the F# Library*

| Operator | Type |
|---|---|
| `List.map` | `: ('a -> 'b) -> 'a list -> 'b list` |
| `Array.map` | `: ('a -> 'b) -> 'a[] -> 'b[]` |
| `Option.map` | `: ('a -> 'b) -> 'a option -> 'b option` |
| `Seq.map` | `: ('a -> 'b) -> #seq<'a> -> seq<'b>` |

Frequently, types will also define methods such as `Map` that provide a slightly more succinct way of transforming data. For example, we could have written `sites |> List.map getStats` like so:

```
sites.Map(getStats)
```

Both styles are used in F# programming, depending on the methods and properties that are available for a particular data type.

### PIPELINING WITH |>

The |> "forward pipe" operator is perhaps the most important operator in F# programming. Its definition is deceptively simple:

```
let (|>) x f = f x
```

Here is how to use the operator to compute the cubes of three numbers:

```
[1;2;3] |> List.map (fun x -> x * x * x)
```

This produces `[1;8;27]`, just as if you had written this:

```
List.map (fun x -> x * x * x) [1;2;3]
```

In a sense, |> is just "function application in reverse." However, using |> has distinct advantages:

- *Clarity*: When used in conjunction with operators such as `List.map`, the |> operator allows you to perform the data transformations and iterations in a forward-chaining, pipelined style.
- *Type inference*: Using the |> operator allows type information to be flowed from input objects to the functions manipulating those objects. F# uses information collected from type inference to resolve some language constructs such as property accesses and method overloading. This relies on information being propagated left to right through the text of a program. In particular, typing information to the right of a position is not taken into account when resolving property access and overloads.

For completeness, here is the type of the operator:

```
val (|>) : 'a -> ('a -> 'b) -> 'b
```

## Composing Functions with >>

You saw earlier how to use the |> "forward pipe" operator to pipe values through a number of functions. This was a small example of the process of *computing with functions,* an essential and powerful programming technique in F#. In this section, we cover ways to compute new function values from existing ones using compositional techniques. First let's take a look at function composition. For example, consider the following code:

```
let google = http "http://www.google.com"
google |> getWords |> List.filter (fun s -> s = "href") |> List.length
```

You can rewrite this code using function composition as follows:

```
let countLinks = getWords >> List.filter (fun s -> s = "href") >> List.length
google |> countLinks
```

Let's take a look at this more closely. We have defined `countLinks` as the composition of three function values using the `>>` "forward composition" operator. This operator is defined in the F# library as follows:

```
let (>>) f g x = g(f(x))
```

You can see from the definition that `f >> g` gives a function value that first applies `f` to the `x` and then applies `g`. Here is the type of `>>`:

```
val (>>) : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
```

Note that `>>` is typically applied to only two arguments—those on either side of the binary operator, here named `f` and `g`. The final argument `x` is typically supplied at a later point. F# is good at optimizing basic constructions of pipelines and composition sequences from functions—for example, the function `countLinks` shown earlier will become a single function calling the three functions in the pipeline in sequence. This means sequences of compositions can be used with relatively low overhead.

## Building Functions with Partial Application

Composing functions is just one way to compute interesting new functions. Another useful way is by using *partial application*. Here's an example:

```
let shift (dx,dy) (px,py) = (px + dx, py + dy)
let shiftRight = shift (1,0)
let shiftUp    = shift (0,1)
let shiftLeft  = shift (-1,0)
let shiftDown  = shift (0,-1)
```

The last four functions have been defined by calling `shift` with only one argument, in each case leaving a residue function that expects an additional argument. F# Interactive will report the types as follows:

```
val shift : int * int -> int * int -> int * int
val shiftRight : int * int -> int * int
val shiftLeft  : int * int -> int * int
val shiftUp    : int * int -> int * int
val shiftDown  : int * int -> int * int
```

Here is an example of how to use `shiftRight` and how to apply `shift` to new arguments `(2,2)`: