F# for Scientists

Jon Harrop
Flying Frog Consultancy Ltd.

Foreword by Don Syme
F# for Scientists
This Page Intentionally Left Blank
F# for Scientists

Jon Harrop
Flying Frog Consultancy Ltd.

Foreword by Don Syme
To my family
# Contents in Brief

1. Introduction  
2. Program Structure  
3. Data Structures  
4. Numerical Analysis  
5. Input and Output  
6. Simple Examples  
7. Visualization  
8. Optimization  
9. Libraries  
10. Databases  
11. Interoperability  
12. Complete Examples  

Bibliography
## CONTENTS

- Foreword xix
- Preface xxii
- Acknowledgments xxiii
- List of Figures xxv
- List of Tables xxxi
- Acronyms xxxiii

### 1 Introduction 1

- 1.1 Programming guidelines 2
- 1.2 A brief history of F# 2
- 1.3 Benefits of F# 3
- 1.4 Introducing F# 3
  - 1.4.1 Language overview 4
  - 1.4.2 Pattern matching 15
  - 1.4.3 Equality 24
  - 1.4.4 Sequence expressions 26

- 15
CONTENTS

1.4.5 Exceptions
1.5 Imperative programming
1.6 Functional programming
  1.6.1 Immutability
  1.6.2 Recursion
  1.6.3 Curried functions
  1.6.4 Higher-order functions

2 Program Structure
  2.1 Nesting
  2.2 Factoring
    2.2.1 Factoring out common subexpressions
    2.2.2 Factoring out higher-order functions
  2.3 Modules
  2.4 Objects
    2.4.1 Augmentations
    2.4.2 Classes
  2.5 Functional design patterns
    2.5.1 Combinators
    2.5.2 Maps and folds
  2.6 F# development
    2.6.1 Creating an F# project
    2.6.2 Building executables
    2.6.3 Debugging
    2.6.4 Interactive mode
    2.6.5 C# interoperability

3 Data Structures
  3.1 Algorithmic complexity
    3.1.1 Primitive operations
    3.1.2 Complexity
  3.2 Arrays
    3.2.1 Array literals
    3.2.2 Array indexing
    3.2.3 Array concatenation
    3.2.4 Aliasing
    3.2.5 Subarrays
    3.2.6 Creation
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.7 Iteration</td>
<td>72</td>
</tr>
<tr>
<td>3.2.8 Map</td>
<td>73</td>
</tr>
<tr>
<td>3.2.9 Folds</td>
<td>73</td>
</tr>
<tr>
<td>3.2.10 Sorting</td>
<td>75</td>
</tr>
<tr>
<td>3.2.11 Pattern matching</td>
<td>75</td>
</tr>
<tr>
<td>3.3 Lists</td>
<td></td>
</tr>
<tr>
<td>3.3.1 Sorting</td>
<td>76</td>
</tr>
<tr>
<td>3.3.2 Searching</td>
<td>76</td>
</tr>
<tr>
<td>3.3.3 Filtering</td>
<td>78</td>
</tr>
<tr>
<td>3.3.4 Maps and folds</td>
<td>78</td>
</tr>
<tr>
<td>3.3.5 Pattern matching</td>
<td>80</td>
</tr>
<tr>
<td>3.4 Sets</td>
<td></td>
</tr>
<tr>
<td>3.4.1 Creation</td>
<td>82</td>
</tr>
<tr>
<td>3.4.2 Insertion</td>
<td>82</td>
</tr>
<tr>
<td>3.4.3 Cardinality</td>
<td>83</td>
</tr>
<tr>
<td>3.4.4 Set-theoretic operations</td>
<td>83</td>
</tr>
<tr>
<td>3.4.5 Comparison</td>
<td>84</td>
</tr>
<tr>
<td>3.5 Hash tables</td>
<td></td>
</tr>
<tr>
<td>3.5.1 Creation</td>
<td>84</td>
</tr>
<tr>
<td>3.5.2 Searching</td>
<td>85</td>
</tr>
<tr>
<td>3.5.3 Insertion, replacement and removal</td>
<td>86</td>
</tr>
<tr>
<td>3.5.4 Higher-order functions</td>
<td>87</td>
</tr>
<tr>
<td>3.6 Maps</td>
<td></td>
</tr>
<tr>
<td>3.6.1 Creation</td>
<td>87</td>
</tr>
<tr>
<td>3.6.2 Searching</td>
<td>88</td>
</tr>
<tr>
<td>3.6.3 Higher-order functions</td>
<td>89</td>
</tr>
<tr>
<td>3.7 Choosing a data structure</td>
<td>90</td>
</tr>
<tr>
<td>3.8 Sequences</td>
<td></td>
</tr>
<tr>
<td>3.9 Heterogeneous containers</td>
<td>91</td>
</tr>
<tr>
<td>3.10 Trees</td>
<td></td>
</tr>
<tr>
<td>3.10.1 Balanced trees</td>
<td>100</td>
</tr>
<tr>
<td>3.10.2 Unbalanced trees</td>
<td>101</td>
</tr>
<tr>
<td>3.10.3 Abstract syntax trees</td>
<td>110</td>
</tr>
<tr>
<td>4 Numerical Analysis</td>
<td>113</td>
</tr>
<tr>
<td>4.1 Number representation</td>
<td>113</td>
</tr>
<tr>
<td>4.1.1 Machine-precision integers</td>
<td>113</td>
</tr>
<tr>
<td>4.1.2 Machine-precision floating-point numbers</td>
<td>114</td>
</tr>
</tbody>
</table>
4.2 Algebra 117
4.3 Interpolation 119
4.4 Quadratic solutions 120
4.5 Mean and variance 122
4.6 Other forms of arithmetic 123
  4.6.1 Arbitrary-precision integer arithmetic 123
  4.6.2 Arbitrary-precision rational arithmetic 124
  4.6.3 Adaptive precision 125

5 Input and Output 127
  5.1 Printing 127
    5.1.1 Generating strings 128
  5.2 Generic printing 129
  5.3 Reading from and writing to files 130
  5.4 Serialization 131
  5.5 Lexing and parsing 132
    5.5.1 Lexing 133
    5.5.2 Parsing 137

6 Simple Examples 141
  6.1 Functional 141
    6.1.1 Nest 142
    6.1.2 Fixed point 142
    6.1.3 Within 142
    6.1.4 Memoize 143
    6.1.5 Binary search 147
  6.2 Numerical 147
    6.2.1 Heaviside step 147
    6.2.2 Kronecker δ-function 148
    6.2.3 Gaussian 148
    6.2.4 Binomial coefficients 149
    6.2.5 Root finding 151
    6.2.6 Grad 151
    6.2.7 Function minimization 152
    6.2.8 Gamma function 154
    6.2.9 Discrete wavelet transform 155
  6.3 String related 157
    6.3.1 Transcribing DNA 157
6.3.2  Word frequency  
6.4  List related  
6.4.1  count  
6.4.2  positions  
6.4.3  fold_to  
6.4.4  insert  
6.4.5  chop  
6.4.6  dice  
6.4.7  apply_at  
6.4.8  sub  
6.4.9  extract  
6.4.10  shuffle  
6.4.11  transpose  
6.4.12  combinations  
6.4.13  distribute  
6.4.14  permute  
6.4.15  Power set  
6.5  Array related  
6.5.1  rotate  
6.5.2  swap  
6.5.3  except  
6.5.4  shuffle  
6.6  Higher-order functions  
6.6.1  Tuple related  
6.6.2  Generalized products  

7  Visualization  
7.1  Windows Forms  
7.1.1  Forms  
7.1.2  Controls  
7.1.3  Events  
7.1.4  Bitmaps  
7.1.5  Example: Cellular automata  
7.1.6  Running an application  
7.2  Managed DirectX  
7.2.1  Handling DirectX devices  
7.2.2  Programmatic rendering  
7.2.3  Rendering an icosahedron
7.2.4 Declarative rendering 191
7.2.5 Spawning visualizations from the F# interactive mode 192
7.3 Tesselating objects into triangles 194
7.3.1 Spheres 194
7.3.2 3D function plotting 196

8 Optimization 199
8.1 Timing 200
8.1.1 Absolute time 200
8.1.2 CPU time 201
8.1.3 Looping 201
8.1.4 Example timing 202
8.2 Profiling 202
8.2.1 8-queens problem 202
8.3 Algorithmic optimizations 205
8.4 Lower-level optimizations 206
8.4.1 Benchmarking data structures 207
8.4.2 Compiler flags 211
8.4.3 Tail-recursion 212
8.4.4 Avoiding allocation 213
8.4.5 Terminating early 217
8.4.6 Avoiding higher-order functions 220
8.4.7 Use mutable 220
8.4.8 Specialized functions 221
8.4.9 Unboxing data structures 222
8.4.10 Eliminate needless closures 223
8.4.11 Inlining 224
8.4.12 Serializing 224

9 Libraries 225
9.1 Loading .NET libraries 226
9.2 Charting and graphing 226
9.3 Threads 227
9.3.1 Thread safety 228
9.3.2 Basic use 229
9.3.3 Locks 231
9.3.4 The thread pool 232
9.3.5 Asynchronous delegates 233
9.3.6 Background threads
9.4 Random numbers
9.5 Regular expressions
9.6 Vectors and matrices
9.7 Downloading from the Web
9.8 Compression
9.9 Handling XML
  9.9.1 Reading
  9.9.2 Writing
  9.9.3 Declarative representation
9.10 Calling native libraries
9.11 Fourier transform
  9.11.1 Native-code bindings
  9.11.2 Interface in F#
  9.11.3 Pretty printing complex numbers
  9.11.4 Example use
9.12 Metaprogramming
  9.12.1 Emitting IL code
  9.12.2 Compiling with LINQ

10 Databases

10.1 Protein data bank
  10.1.1 Interrogating the PDB
  10.1.2 Pretty printing XML in F# interactive sessions
  10.1.3 Deconstructing XML using active patterns
  10.1.4 Visualization in a GUI
10.2 Web services
  10.2.1 US temperature by zip code
  10.2.2 Interrogating the NCBI
10.3 Relational databases
  10.3.1 Connection to a database
  10.3.2 Executing SQL statements
  10.3.3 Evaluating SQL expressions
  10.3.4 Interrogating the database programmatically
  10.3.5 Filling the database from a data structure
  10.3.6 Visualizing the result
  10.3.7 Cleaning up
11 Interoperability

11.1 Excel
11.1.1 Referencing the Excel interface
11.1.2 Loading an existing spreadsheet
11.1.3 Creating a new spreadsheet
11.1.4 Referring to a worksheet
11.1.5 Writing cell values into a worksheet
11.1.6 Reading cell values from a worksheet
11.2 MATLAB
11.2.1 Creating a .NET interface from a COM interface
11.2.2 Using the interface
11.2.3 Remote execution of MATLAB commands
11.2.4 Reading and writing MATLAB variables
11.3 Mathematica
11.3.1 Using .NET-link
11.3.2 Example

12 Complete Examples

12.1 Fast Fourier transform
12.1.1 Discrete Fourier transform
12.1.2 Danielson-Lanczos algorithm
12.1.3 Bluestein’s convolution algorithm
12.1.4 Testing and performance
12.2 Semi-circle law
12.2.1 Eigenvalue computation
12.2.2 Injecting results into Excel
12.2.3 Results
12.3 Finding \( n \)th-nearest neighbors
12.3.1 Formulation
12.3.2 Representing an atomic configuration
12.3.3 Parser
12.3.4 Lexer
12.3.5 Main program
12.3.6 Visualization
12.4 Logistic map
12.5 Real-time particle dynamics

Appendix A: Troubleshooting
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Value restriction</td>
<td>311</td>
</tr>
<tr>
<td>A.2 Mutable array contents</td>
<td>312</td>
</tr>
<tr>
<td>A.3 Negative literals</td>
<td>313</td>
</tr>
<tr>
<td>A.4 Accidental capture</td>
<td>313</td>
</tr>
<tr>
<td>A.5 Local and non-local variable definitions</td>
<td>313</td>
</tr>
<tr>
<td>A.6 Merging lines</td>
<td>314</td>
</tr>
<tr>
<td>A.7 Applications that do not die</td>
<td>314</td>
</tr>
<tr>
<td>A.8 Beware of “it”</td>
<td>315</td>
</tr>
</tbody>
</table>

Glossary                                      | 317  |

Bibliography                                 | 325  |

Index                                         | 329  |
Foreword

Computational science is one of the wonders of the modern world. In almost all areas of science the use of computational techniques is rocketing, and software has moved from being a supporting tool to being a key site where research activities are performed. This has meant a huge increase in the importance of controlling and orchestrating computers as part of the daily routine of a scientific laboratory, from large teams making and running the computers performing global climate simulations to the individual scientist/programmer working alone. Across this spectrum, the productivity of teams and the happiness of scientists depends dramatically on their overall competency as programmers, as well as on their skills as researchers within their field. So, in the last 30 years we have seen the continued rise of that new profession: the scientific programmer. A good scientific programmer will carry both epithets with pride, knowing that programming is a key foundation for a successful publication record.

However, programming cultures differ widely, and, over time, gaping divides can emerge that can be to the detriment of all. In this book, Dr. Harrop has taken great steps forward to bridging three very different cultures: managed code programming, scientific programming and functional programming. At a technical level, each has its unique characteristics. Managed code programming, epitomized by .NET and Java, focuses on the productivity of the (primarily commercial) programmer. Scientific programmers focus on high performance computations, data manipulation, numerical
computing and visualization. Functional programming focuses on crisp, declarative solutions to problems using compositional techniques. The challenge, then, is to bring these disparate worlds together in a productive way.

The language F#, which Dr. Harrop uses in this book, itself bridges two of these cultures by being a functional language for the .NET platform. F# is an incredibly powerful language: the .NET libraries give a rich and solid foundation of software functionality for many tasks, from routine programming to accessing web services and high performance graphics engines. F# brings an approach to programming that routinely makes even short programs powerful, simple, elegant and correct. However Dr. Harrop has gone a step further, showing how managed code functional programming can revolutionize the art of scientific programming itself by being a powerful workhorse tool that unifies and simplifies many of the tasks scientific programmers face.

But what of the future? The next 20 years will see great changes in scientific programming. It is customary to mention the ever-increasing challenges of parallel, concurrent, distributed and reactive programming. It is widely expected that future micro-processors will use ever-increasing transistor counts to host multiple processing cores, rather than more sophisticated microprocessor designs. If computations can be parallelized and distributed on commodity hardware then the computing resources that can be brought can be massively increased. It is well known that successful concurrent and distributed computing requires a combination of intelligent algorithm design, competent programming, and core components that abstract some details of concurrent execution, e.g. databases and task execution libraries. This needs a language that can interoperate with key technologies such as databases, and parallelism engines. Furthermore, the ability to declaratively and crisply describe solutions to concurrent programming problems is essential, and F# is admirably suited to this task.

The future, will, however, bring other challenges as well. Truly massive amounts of data are now being generated by scientific experiments. Web-based programming will become more and more routine for scientific teams: a good web application can revolutionize a scientific field. Shared databases will soon be used in almost every scientific field, and programmatic access to these will be essential. F# lends itself to these challenges: for example, it is relatively easy to perform sophisticated and high-performance analysis of these data sources by bringing them under the static type discipline of functional programming, as shown by some of the samples in this book.

You will learn much about both programming and science through this book. Dr. Harrop has chosen the style of F# programming most suited to the individual scientist: crisp, succinct and efficient, with a discursive presentation style reminiscent of Mathematica. It has been a pleasure to read, and we trust it will launch you on a long and productive career as a managed code, functional scientific programmer.
The face of scientific computing has changed. Computational scientists are no longer writing their programs in Fortran and competing for time on supercomputers. Scientists are now streamlining their research by choosing more expressive programming languages, parallel processing on desktop machines and exploiting the wealth of scientific information distributed across the internet.

The landscape of programming languages saw a punctuation in its evolution at the end of the 20th century, marked by the advent of a new breed of languages. These new languages incorporate a multitude of features that are all designed to serve a single purpose: to make life easier. Modern programming languages offer so much more expressive power than traditional languages that they even open up new avenues of scientific research that were simply intractable before.

The next few years will usher in a new era of computing, where parallelism becomes ubiquitous. Few approaches to programming will survive this transition, and functional programming is one of them.

Seamlessly interoperating with computers across the world is of pivotal importance not only because of the breadth of information now available on-line but also because this is the only practicable way to interrogate the enormous amount of data available. The amount of genomic and proteinomic data published every year continues to grow exponentially, as each generation of technology fuels the next.
Only one mainstream programming language combines awesome expressive power, interoperability and performance: F#. This book introduces all of the aspects of the F# programming language needed by a working scientist, emphasizing aspects not covered by existing literature. Consequently, this book is the ideal complement to a detailed overview of the language itself, such as the F# manual or the book Expert F#[25].

Chapters 1–5 cover the most important aspects of F# programming needed to start developing useful F# programs. Chapter 6 ossifies this knowledge with a variety of enlightening and yet simple examples. Chapters 7–11 cover advanced topics including real-time visualization, interoperability and parallel computing. Chapter 12 concludes the book with a suite of complete working programs relevant to scientific computing.

The source code from this book is available from the following website:
http://www.ffconsultancy.com/products/fsharp_for_scientists/

J. D. Harrop

Cambridge, UK
June, 2008
I would like to thank Don Syme, the creator of F#, for pioneering research into programming languages and for thrusting the incredibly powerful ML family of languages into the limelight of mainstream software development.

Xavier Leroy, everyone at projet Cristal and the Debian package maintainers for making OCaml so practically useful.

Stephen Elliott and Sergei Taraskin and their group at the University of Cambridge for teaching me how to be a research scientist and letting me pursue crazy ideas when I should have been working.

Ioannis Balthopoulos and Enrique Nell for proofreading this book and giving essential feedback.

J. D. H.
2.1 The fold_range function can be used to accumulate the result of applying a function \( f \) to a contiguous sequence of integers, in this case the sequence \( \{1, 9\} \).

2.2 Developing an application written entirely in F# using Microsoft Visual Studio 2005.

2.3 Visual Studio provides graphical throwback of the type information inferred by the F# compiler: hovering the mouse over the definition of a variable \( r \) in the source code brings up a tooltip giving the inferred type of \( r \).

2.4 A project’s properties page allows the compiler to be controlled.

2.5 The Add-in Manager is used to provide the F# interactive mode.

2.6 Creating a new C# class library project called ClassLibrary1 inside a new solution called Interop.

2.7 Creating a new F# project called Project1 also inside the Interop solution.
2.8 Setting the startup project of the Interop solution to the F# project Project1 rather than the C# project ClassLibrary1 as a DLL cannot be used to start an application. 60

3.1 Complexities of the ipow_1 and ipow_2 functions in terms of the number $T(n)$ of multiplies performed. 67

3.2 Complexities of the ipow_2 function in terms of the number of multiplies performed, showing: exact complexity $T(n)$ (dots) and lower- and upper-bounds algorithmic complexities $\log_2(n) - 1 \leq T(n) \leq 2(1 + \log_2 n)$ for $n > 1$ (lines). 68

3.3 Measured performance of the ipow_1 and ipow_2 functions which have asymptotic algorithmic complexities of $\Theta(n)$ and $\Theta(ln n)$, respectively. 68

3.4 Arrays are the simplest data structure, allowing fast, random access (reading or writing) to the $i$th element $\forall i \in \{0 \ldots n - 1\}$ where $n$ is the number of elements in the array. Elements cannot be added or removed without copying the whole array. 69

3.5 The higher-order Array.init function creates an array $a_i = f(i)$ for $i \in \{0 \ldots n - 1\}$ using the given function $f$. 72

3.6 The higher-order Array.map function creates an array containing the result of applying the given function $f$ to each element in the given array $a$. 73

3.7 The higher-order Array.fold_left function repeatedly applies the given function $f$ to the current accumulator and the current array element to produce a new accumulator to be applied with the next array element. 74

3.8 Lists are the simplest, arbitrarily-extensible data structure. Decapitation splits a list $l_i i \in \{0 \ldots n - 1\}$ into the head element $h$ and the tail list $t_i i \in \{0 \ldots n - 2\}$. 76

3.9 Measured performance (time $t$ in seconds) for inserting key-value pairs into hash tables and functional maps containing $n - 1$ elements. Although the hash table implementation results in better average-case performance, the $O(n)$ time-complexity incurred when the hash table is resized internally produces much slower worst-case performance by the hash table. 88

3.10 A perfectly-balanced binary tree of depth $x = 3$ containing $2^{x+1} - 1 = 15$ nodes, including the root node and $2^x = 8$ leaf nodes. 94
3.11 The result of inserting an integer counter into each node of the tree depicted in figure 3.10 using the counted_ptree_of_tree function.

3.12 An unbalanced binary tree with the remaining depth stored in every node.

3.13 A maximally-unbalanced binary tree of depth \( x = 7 \) containing \( 2x + 1 = 15 \) nodes, including the root node and \( x + 1 = 8 \) leaf nodes.

3.14 An unbalanced binary tree used to partition the space \( r \in [0, 1) \) in order to approximate the gravitational effect of a cluster of particles in a system.

3.15 Measured performance of the tree-based approach relative to a simple array-based approach for the evaluation of long-range forces showing the resulting fractional error \( \delta = |O - E|/E \) vs time taken \( t = t_{tree}/t_{array} \) relative to the array-based method.

4.1 Values \( i \) of the type \texttt{int}, called \textit{machine-precision integers}, are an exact representation of a consecutive subset of the set of integers \( i \in [l \ldots u] \subset \mathbb{Z} \) where \( l \) and \( u \) are given by \texttt{min_int} and \texttt{max_int}, respectively.

4.2 Values of the type \texttt{float}, called \textit{double-precision floating-point numbers}, are an approximate representation of real-valued numbers, showing: a) full-precision (normalized) numbers (black), and b) denormalized numbers (gray).

4.3 Accuracy of two equivalent expressions when evaluated using floating-point arithmetic: a) \( f_1(x) = \sqrt{1 + x} - 1 \) (solid line), and b) \( f_2(x) = x/(1 + \sqrt{1 + x}) \) (dashed line).

5.1 Parsing character sequences often entails lexing into a token stream and then parsing to convert patterns of tokens into grammatical constructs represented hierarchically by a tree data structure.

6.1 The first seven rows of Pascal’s triangle.

7.1 A blank Windows form.

7.2 A form with a single control, a button.

7.3 A thousand generations of the rule 30 cellular automaton.

7.4 A DirectX viewer that clears the display to a single color (called “coral”).
7.5  Abutting triangles can be amortised into triangle fans and strips to reduce the number of vertices required to describe a geometry.  187

7.6  A triangle rendered programmatically and visualized using an orthographic projection.  189

7.7  A DirectX viewer that draws an icosahedron.  190

7.8  Progressively more refined uniform tessellations of a sphere, obtained by subdividing the triangular faces of an icosahedron and normalising the resulting vertex coordinate vectors to push them onto the surface of a sphere.  195

7.9  3D surface plot of $y = \sin(r + 3x)/r$ where $r = 5\sqrt{x^2 + z^2}$.  197

8.1  Profiling results generated by the freely-available NProf profiler for a program solving the queens problem on an $11 \times 11$ board.  204

8.2  Measured performance (time $t$ in seconds) of mem functions over set, list and array data structures containing $n$ elements.  206

8.3  Relative time taken $t = t_s/t_a$ for testing membership in a set ($t_s$) and an array ($t_a$) as a function of the number of elements $n$ in the container, showing that arrays are up to $2\times$ faster for $n < 35$.  208

8.4  Measured performance (time $t$ in seconds per element) of List.of_array, Array.copy and Set.of_array data structures containing $n$ elements.  208

8.5  Measured performance (time $t$ in seconds per element) of iter functions over list, array and set data structures containing $n$ elements.  209

8.6  Measured performance (time $t$ in seconds per element) of the fold_left functions over list and array data structures containing $n$ elements.  210

8.7  Measured performance (time $t$ in seconds per element) of the fold_right functions over list, array and set data structures containing $n$ elements.  210

8.8  Controlling the optimization flags passed to the F# compiler by Visual Studio 2005.  211

8.9  Deforestation refers to methods used to reduce the size of temporary data, such as the use of composite functions to avoid the creation of temporary data structures illustrated here: a) mapping a function $f$ over a list $l$ twice, and b) mapping the composite function $f \circ f$ over the list $l$ once.  214