



F# for Scientists

Jon Harrop

Flying Frog Consultancy Ltd.

Foreword by Don Syme



WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

This Page Intentionally Left Blank

F# for Scientists

This Page Intentionally Left Blank

F# for Scientists

Jon Harrop

Flying Frog Consultancy Ltd.

Foreword by Don Syme



WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2008 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic format. For information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Harrop, Jon D.

F# for scientists / Jon Harrop.

p. cm.

Includes index.

ISBN 978-0-470-24211-7 (cloth)

1. F# (Computer program language) 2. Functional programming (Computer science) 3. Science—Data processing. I. Title.

QA76.73.F163H37 2008

005.1'14—dc22

2008009567

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To my family

This Page Intentionally Left Blank

Contents in Brief

1	Introduction	1
2	Program Structure	37
3	Data Structures	63
4	Numerical Analysis	113
5	Input and Output	127
6	Simple Examples	141
7	Visualization	173
8	Optimization	199
9	Libraries	225
10	Databases	249
11	Interoperability	267
12	Complete Examples	281
	Bibliography	325

This Page Intentionally Left Blank

CONTENTS

Foreword	xix
Preface	xxi
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
	ix

1.4.5	Exceptions	27
1.5	Imperative programming	29
1.6	Functional programming	31
1.6.1	Immutability	31
1.6.2	Recursion	32
1.6.3	Curried functions	33
1.6.4	Higher-order functions	35
2	Program Structure	37
2.1	Nesting	38
2.2	Factoring	38
2.2.1	Factoring out common subexpressions	39
2.2.2	Factoring out higher-order functions	39
2.3	Modules	42
2.4	Objects	44
2.4.1	Augmentations	44
2.4.2	Classes	46
2.5	Functional design patterns	49
2.5.1	Combinators	49
2.5.2	Maps and folds	52
2.6	F# development	53
2.6.1	Creating an F# project	54
2.6.2	Building executables	54
2.6.3	Debugging	56
2.6.4	Interactive mode	56
2.6.5	C# interoperability	58
3	Data Structures	63
3.1	Algorithmic complexity	64
3.1.1	Primitive operations	64
3.1.2	Complexity	65
3.2	Arrays	69
3.2.1	Array literals	69
3.2.2	Array indexing	70
3.2.3	Array concatenation	70
3.2.4	Aliasing	71
3.2.5	Subarrays	72
3.2.6	Creation	72

3.2.7	Iteration	72
3.2.8	Map	73
3.2.9	Folds	73
3.2.10	Sorting	75
3.2.11	Pattern matching	75
3.3	Lists	75
3.3.1	Sorting	76
3.3.2	Searching	76
3.3.3	Filtering	78
3.3.4	Maps and folds	78
3.3.5	Pattern matching	80
3.4	Sets	82
3.4.1	Creation	82
3.4.2	Insertion	83
3.4.3	Cardinality	83
3.4.4	Set-theoretic operations	83
3.4.5	Comparison	84
3.5	Hash tables	84
3.5.1	Creation	85
3.5.2	Searching	86
3.5.3	Insertion, replacement and removal	86
3.5.4	Higher-order functions	87
3.6	Maps	87
3.6.1	Creation	88
3.6.2	Searching	89
3.6.3	Higher-order functions	90
3.7	Choosing a data structure	91
3.8	Sequences	92
3.9	Heterogeneous containers	92
3.10	Trees	93
3.10.1	Balanced trees	100
3.10.2	Unbalanced trees	101
3.10.3	Abstract syntax trees	110
4	Numerical Analysis	113
4.1	Number representation	113
4.1.1	Machine-precision integers	113
4.1.2	Machine-precision floating-point numbers	114

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	158
6.4	List related	159
6.4.1	count	159
6.4.2	positions	160
6.4.3	fold_to	160
6.4.4	insert	161
6.4.5	chop	161
6.4.6	dice	162
6.4.7	apply_at	163
6.4.8	sub	163
6.4.9	extract	163
6.4.10	shuffle	164
6.4.11	transpose	165
6.4.12	combinations	165
6.4.13	distribute	166
6.4.14	permute	167
6.4.15	Power set	167
6.5	Array related	168
6.5.1	rotate	168
6.5.2	swap	168
6.5.3	except	169
6.5.4	shuffle	169
6.6	Higher-order functions	169
6.6.1	Tuple related	170
6.6.2	Generalized products	170
7	Visualization	173
7.1	Windows Forms	174
7.1.1	Forms	174
7.1.2	Controls	175
7.1.3	Events	175
7.1.4	Bitmaps	176
7.1.5	Example: Cellular automata	177
7.1.6	Running an application	179
7.2	Managed DirectX	180
7.2.1	Handling DirectX devices	180
7.2.2	Programmatic rendering	183
7.2.3	Rendering an icosahedron	188

7.2.4	Declarative rendering	191
7.2.5	Spawning visualizations from the F# interactive mode	192
7.3	Tessellating 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	233
9.4	Random numbers	234
9.5	Regular expressions	234
9.6	Vectors and matrices	235
9.7	Downloading from the Web	236
9.8	Compression	237
9.9	Handling XML	237
9.9.1	Reading	237
9.9.2	Writing	238
9.9.3	Declarative representation	238
9.10	Calling native libraries	239
9.11	Fourier transform	240
9.11.1	Native-code bindings	240
9.11.2	Interface in F#	242
9.11.3	Pretty printing complex numbers	243
9.11.4	Example use	244
9.12	Metaprogramming	245
9.12.1	Emitting IL code	245
9.12.2	Compiling with LINQ	247
10	Databases	249
10.1	Protein data bank	250
10.1.1	Interrogating the PDB	250
10.1.2	Pretty printing XML in F# interactive sessions	251
10.1.3	Deconstructing XML using active patterns	251
10.1.4	Visualization in a GUI	253
10.2	Web services	254
10.2.1	US temperature by zip code	255
10.2.2	Interrogating the NCBI	256
10.3	Relational databases	258
10.3.1	Connection to a database	259
10.3.2	Executing SQL statements	259
10.3.3	Evaluating SQL expressions	261
10.3.4	Interrogating the database programmatically	261
10.3.5	Filling the database from a data structure	263
10.3.6	Visualizing the result	263
10.3.7	Cleaning up	264

11	Interoperability	267
11.1	Excel	267
11.1.1	Referencing the Excel interface	268
11.1.2	Loading an existing spreadsheet	268
11.1.3	Creating a new spreadsheet	269
11.1.4	Referring to a worksheet	269
11.1.5	Writing cell values into a worksheet	270
11.1.6	Reading cell values from a worksheet	271
11.2	MATLAB	272
11.2.1	Creating a .NET interface from a COM interface	272
11.2.2	Using the interface	273
11.2.3	Remote execution of MATLAB commands	273
11.2.4	Reading and writing MATLAB variables	273
11.3	Mathematica	275
11.3.1	Using .NET-link	275
11.3.2	Example	277
12	Complete Examples	281
12.1	Fast Fourier transform	281
12.1.1	Discrete Fourier transform	282
12.1.2	Danielson-Lanczos algorithm	283
12.1.3	Bluestein's convolution algorithm	285
12.1.4	Testing and performance	287
12.2	Semi-circle law	288
12.2.1	Eigenvalue computation	289
12.2.2	Injecting results into Excel	290
12.2.3	Results	291
12.3	Finding n^{th} -nearest neighbors	291
12.3.1	Formulation	292
12.3.2	Representing an atomic configuration	295
12.3.3	Parser	295
12.3.4	Lexer	297
12.3.5	Main program	297
12.3.6	Visualization	299
12.4	Logistic map	301
12.5	Real-time particle dynamics	303
	Appendix A: Troubleshooting	311

A.1	Value restriction	311
A.2	Mutable array contents	312
A.3	Negative literals	313
A.4	Accidental capture	313
A.5	Local and non-local variable definitions	313
A.6	Merging lines	314
A.7	Applications that do not die	314
A.8	Beware of “it”	315
Glossary		317
Bibliography		325
Index		329

This Page Intentionally Left Blank

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.

Preface

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 proteomic 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

Acknowledgments

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 Baltopoulos and Enrique Nell for proofreading this book and giving essential feedback.

J. D. H.

This Page Intentionally Left Blank

List of Figures

2.1	The <code>fold_range</code> function can be used to accumulate the result of applying a function <code>f</code> to a contiguous sequence of integers, in this case the sequence <code>[1, 9)</code> .	40
2.2	Developing an application written entirely in F# using Microsoft Visual Studio 2005.	53
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 <code>x</code> in the source code brings up a tooltip giving the inferred type of <code>x</code> .	54
2.4	A project's properties page allows the compiler to be controlled.	55
2.5	The Add-in Manager is used to provide the F# interactive mode.	57
2.6	Creating a new C# class library project called <code>ClassLibrary1</code> inside a new solution called <code>Interop</code> .	59
2.7	Creating a new F# project called <code>Project1</code> also inside the <code>Interop</code> solution.	59

- 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 $\forall i \in \{0 \dots n - 1\}$ 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 $i \in \{0 \dots n - 1\}$ 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 $l_i \in \{0 \dots n - 1\}$ into the head element h and the tail list $t_i \in \{0 \dots n - 2\}$. 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 <code>counted_ptree_of_tree</code> function.	96
3.12	An unbalanced binary tree with the remaining depth stored in every node.	98
3.13	A maximally-unbalanced binary tree of depth $x = 7$ containing $2x + 1 = 15$ nodes, including the <i>root</i> node and $x + 1 = 8$ leaf nodes.	100
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.	106
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 $t = t_{\text{tree}}/t_{\text{array}}$ relative to the array-based method.	109
4.1	Values i of the type <code>int</code> , called <i>machine-precision integers</i> , are an exact representation of a consecutive subset of the set of integers $i \in [l \dots u] \subset \mathbb{Z}$ where l and u are given by <code>min_int</code> and <code>max_int</code> , respectively.	114
4.2	Values of the type <code>float</code> , called <i>double-precision floating-point numbers</i> , are an approximate representation of real-valued numbers, showing: a) full-precision (normalized) numbers (black), and b) denormalized numbers (gray).	115
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).	119
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.	132
6.1	The first seven rows of Pascal's triangle.	149
7.1	A blank Windows form.	175
7.2	A form with a single control, a button.	176
7.3	A thousand generations of the rule 30 cellular automaton.	179
7.4	A DirectX viewer that clears the display to a single color (called "coral").	183

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 11×11 board.	204
8.2	Measured performance (time t in seconds) of <code>mem</code> 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 <code>List.of_array</code> , <code>Array.copy</code> and <code>Set.of_array</code> data structures containing n elements.	208
8.5	Measured performance (time t in seconds per element) of <code>iter</code> functions over list, array and set data structures containing n elements.	209
8.6	Measured performance (time t in seconds per element) of the <code>fold_left</code> functions over list and array data structures containing n elements.	210
8.7	Measured performance (time t in seconds per element) of the <code>fold_right</code> 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 <code>f</code> over a list <code>l</code> twice, and b) mapping the composite function $f \circ f$ over the list <code>l</code> once.	214