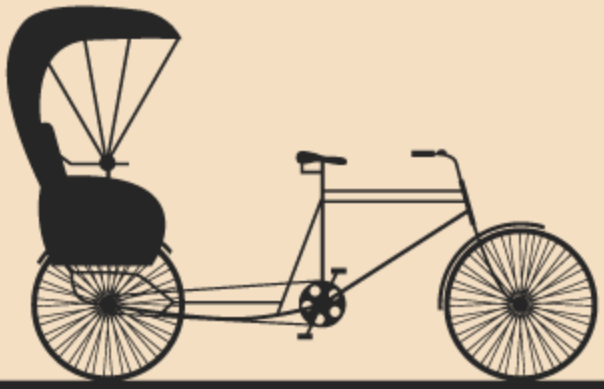


Imperative to Functional Programming Succinctly

by Marc Clifton

Syncfusion | Technology Resource Portal

Imperative to Functional Programming Succinctly

By

Marc Clifton

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Chris Brinkley
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Marc Clifton lives in a farming community near Harlemville, NY, providing consulting services to companies across the country in application development. He specializes in software architecture and frequently gets his hands dirty in implementation as well. In addition to being a multi-year Microsoft MVP and Code Project MVP, Marc has forayed into the wonderful world of Ruby on Rails. He is also interested in Anthroposophy, alternative education, and promoting local economies. He's learning how to play the lyre and is a mentor in the Albany Executive Mentorship Program, which connects young professionals with seasoned business leaders (Marc is supposedly the latter!).

Introduction

My management always thinks it's imperative that my programming is functional yesterday. –Gary R. Wheeler

About this book

F# is not just another programming language. It is designed mainly for functional programming, which both invites and requires a different way of thinking about computer programs. Functional programming is based on the very simple principle that behavior is strictly determined by inputs, so that the same inputs always produce the same behavior. Functions act like the functions in high school algebra, but may operate with a wide range of structured data types including other functions and computations—not just with numbers. This simple idea has many consequences for programming practice. For instance, in imperative languages, we rarely pass functions to other functions, and we are often working with mutable objects that exhibit (intentionally or not) side effects. When using a functional programming language, the opposite is true: functions are often passed as parameters to other functions, objects are immutable, and there are no side effects unless intentionally implemented.

This book explores "how to think functionally," which is very different from thinking imperatively. Functional programming requires thinking about a computer program in terms of algebra-like computations built up only from simpler algebra-like computations. It isn't enough just to learn F# syntax. For that reason, I will spend very little time discussing the F# language itself—there are many resources for learning the syntax of F#, for example *F# Succinctly*[1] and Wikibooks' *F# Programming*.[2] If you are completely new to F#, I strongly encourage you to read through those two resources first.

Instead, this book will discuss functional programming concepts such as:

- Currying
- Partial application
- Immutability
- First-class functions
- Higher-order functions
- Function pipelines
- Function composition
- Recursion
- Map, filter, and reduce
- Continuations
- Continuation passing style
- Monads (computation expressions)

[1] F# Succinctly: http://www.syncfusion.com/resources/techportal/ebooks/fsharp

[2] Wikibooks F# Programming: http://en.wikibooks.org/wiki/F_Sharp_Programming

Understanding these concepts and how they change the way you think about programming are critical in the path to becoming a proficient functional programmer.

Even if you are not immediately planning on using a functional programming language, many of these concepts are finding their way into imperative languages such as C# and C++. Becoming familiar with these concepts as an imperative language programmer will strengthen your imperative programming skills as well.

This book is divided into four chapters:

- Chapter 1 – Basic Vocabulary and Concepts
- Chapter 2 – Changing Your Thinking
- Chapter 3 – Going Deeper
- Chapter 4 – Imperative/Functional Interaction

The intention is to introduce concepts in a particular order and at a reasonable pace such that, by the time we get to Chapter 3, you will have a good understanding of how to work with F#-specific concepts.

What you should already know

You should already have some exposure to F# or some other language that supports functional programming, such as Python, Haskell, or Clojure, to name three. Some of the examples will compare F# with C#, so a strong knowledge in C#, especially with C# 3.0 and higher, is recommended.

Resources used

The code examples in this book are written with Visual Studio 2012 and also use Syncfusion Essential Studio and Microsoft SQL Server Express. In particular, many of the code examples in Chapters 2 and 3 require Microsoft's sample database AdventureWorks. The correct version of AdventureWorks depends on the version of SQL Server; please consult the Microsoft documentation for the AdventureWorks database to determine the correct version for your database. If you do not have SQL Server installed, you can download the free edition of SQL Server Express from Microsoft to complete the demonstration.

The history of functional programming

Functional programming has its roots in lambda calculus, which is a "formal system in mathematical logic and computer science for expressing computation based on function

abstraction and application using variable binding and substitution."[3] Lambda calculus was introduced by mathematician Alonzo Church[4] in the 1930s.

For our purposes, the salient point of lambda calculus is that it makes functions first-class objects, which is to say they can be used in all the same ways as regular values or objects. The concept of first-class functions was made explicit in the 1960s by Christopher Strachey,[5] who also coined the term *currying*,[6] which is a fundamental concept of functional programming. Incidentally, the term *currying* is also sometimes called *schönfinkeling,* named after Moses Schönfinkel[7] for originating "…the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument…".[8] We commonly use the term *currying*, named after Haskell Curry[9] who further developed the concepts of combinatorial logic based on Schönfinkel's paper.

One of the first languages that had functional characteristics was Lisp,[10] developed in 1958 by John McCarthy.[11] It is interesting to note that Lisp pioneered many of the ideas we take for granted in modern programming languages: tree data structures, dynamic typing, conditionals, and recursion, to name a few.

Support for functional programming has been slowly adopted by mainstream languages. Python,[12] in 1994, added support for lambda expressions and the collection operations filter, map, and reduce (also called "fold"). First-class functions were introduced in C# 3.0 in 2007.

Lambda expressions in C#

We see the term "lambda expression" used in many recent programming languages, including C#.[13] In C#, lambda expressions are anonymous functions often used to write LINQ query expressions. For example, C#'s collection objects provide selector functions, which you can use to select specific elements of the collection:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int sumOfOddNumbers = numbers.Sum(x => x % 2 == 1 ? x : 0);
Console.WriteLine("Sum of odd numbers 1..5: " + sumOfOddNumbers);
```

The lambda expression:

---

[3] Lambda calculus: http://en.wikipedia.org/wiki/Lambda_calculus
[4] Alonzo Church: http://en.wikipedia.org/wiki/Alonzo_Church
[5] Christopher Strachey: http://en.wikipedia.org/wiki/Christopher_Strachey
[6] Currying: http://en.wikipedia.org/wiki/Currying
[7] Moses Schönfinkel: http://en.wikipedia.org/wiki/Moses_Sch%C3%B6nfinkel
[8] Currying: http://en.wikipedia.org/wiki/Currying
[9] Haskell Curry: http://en.wikipedia.org/wiki/Haskell_Curry
[10] Lisp: http://en.wikipedia.org/wiki/Lisp_%28programming_language%29
[11] John McCarthy: http://en.wikipedia.org/wiki/John_McCarthy_%28computer_scientist%29
[12] Python: http://en.wikipedia.org/wiki/Python_%28programming_language%29
[13] Lambda Expressions: http://msdn.microsoft.com/en-us/library/bb397687.aspx

```
x => x % 2 == 1 ? x : 0
```

returns **x** if x is odd; otherwise it returns 0.

Functional programming in C#

If you've ever used C#'s **Action<T>** or **Func<T, TResult>** delegates, you are already writing in a functional programming style (technically, if you've ever used delegates and events, you are already doing some functional programming).

For example, you can specify a method as a **Func** (as long as it matches the expected return type and parameter types) and use it as a function to qualify the operation of another function. In the following example, we specify a method (a function taking an **int** and returning an **int**) to filter values in an array that we want to sum:

```
static int EvenOrZero(int x)
{
  return x % 2 == 0 ? x : 0;
}

int[] numbers = { 1, 2, 3, 4, 5 };
int sumOfEvenNumbers = numbers.Sum((Func<int, int>)EvenOrZero);
```

In this example, the cast is necessary to resolve the ambiguity between **Func<int, int>** and **Func<int, int?>**.

Alternatively, you can instantiate a function using a lambda expression (this eliminates the issue of the cast):

```
Func<int, int> evenOrZero = x => x % 2 == 0 ? x : 0;
int[] numbers = { 1, 2, 3, 4, 5 };
int sumOfEvenNumbers2 = numbers.Sum(evenOrZero);
```

In either case, the use of **Action<T>** and **Func<T, TResult>** delegates and their variants leverages C#'s delegate capability in making functions first-class citizens, from which higher-order functions can also be composed.

LINQ and functional programming

In the previous examples, what we were really doing was using the method syntax of LINQ (Language INtegrated Query) and providing a predicate, or a lambda expression. You will often end up with a mixture of LINQ syntax and lambda expressions, so the lines get blurry when using an imperative language like C# with LINQ and lambda expressions. A pure LINQ example would read as follows:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int sumOfOdds = (from s in numbers where s % 2 == 1 select s).Sum();
```

And in fact:

- The **LambdaExpression** class is a member of the **System.Linq.Expressions.LambdaExpressions** namespace.
- Aggregate functions such as **Sum** are not available without including **System.Linq**.

As the previous example illustrates, there are several different ways of approaching functional programming in imperative languages such as C#.

Chapter 1 Basic Vocabulary and Concepts

It is a good idea to have a solid understanding of the terms *imperative* and *functional* before proceeding further with how to think like a functional programmer. But before we do that, we're going to dive right into two of the terms in functional programming most frequently misunderstood, *currying* and *partial application*. A solid grasp of these terms is critical to understanding more advanced concepts developed later in the book. Fasten your seatbelts and get ready for that first 500-foot drop on the functional programming rollercoaster.

Currying vs. partial function application

If you've read anything about functional programming, you will have undoubtedly come across the term currying. In the 1960s, Christopher Strachey coined the term currying[14] for what became an important technique in functional programming language implementation:

"In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument (partial application)."[15]

By contrast, partial application is a process of binding values to parameters:

"In computer science, partial application (or partial function application) refers to the process of fixing a number of arguments to a function, producing another function of smaller arity."[16] ("Arity" means the number of arguments that a function accepts.)

Currying and partial function application are often conflated.[17] Some of the confusion over currying has to do with use of the term, which is sometimes used to refer to things the programmer does and other times used to refer to things the language implementation does. As J. Storrs Hall wrote in 2007: "I suspect the confusion arises because originally currying was a technique to model multiple-argument functions in a single-argument framework and was a meta-operation. In ML-like languages, the functions are typically *already curried*, so the only operation you see being done is partial application."[18] In this quotation, currying refers to something the language implementation does.

On the other hand, F# architect Don Syme et al. say "*Currying* is the name used when functions take arguments in the iterated form—that is, when the functions can be partially applied" (*Expert F# 2.0*, p. 559). Multi-argument functions represented as such in the programmer's code are curried by default in the language implementation, and can be partially applied. By contrast, a multi-argument function could also be represented as a function of a tuple, in which case the

---

[14] Currying: http://en.wikipedia.org/wiki/Currying
[15] Currying: http://en.wikipedia.org/wiki/Currying
[16] Partial application: http://en.wikipedia.org/wiki/Partial_application
[17] Currying and partial function application: http://en.wikipedia.org/wiki/Currying#Contrast_with_partial_function_application
[18] Currying vs. partial application: http://lambda-the-ultimate.org/node/2266#comment-33625

tuple will be treated by the language implementation as a single argument, which makes the function a single-argument function. This means that the function cannot be partially applied, because there is no other argument. The choice of representation is up to the programmer, and Syme et al. refer to the programmer's choice of the representation that is curried by default in the language implementation as currying, in contrast to the earlier quotation which refers to what the language implementation does when the programmer makes that choice. But to say that curried functions *can be* partially applied is not to identify currying with partial application—it is only to say that the possibility of partial application depends on the language implementation's support for currying.

1. Since partial application depends on currying, the two often occur together, but they are different because with partial application, you can "bind" more than one parameter to a value, and to evaluate the resulting function all you need are the remaining parameters. There are three things I just stated that define the difference between partial application and currying. Partial application is a process of binding values to parameters that results in a function with fewer parameters. By contrast, currying is a process that replaces a single multi-parameter function with a nested set or chain of single-parameter functions.
2. Partial application can be performed by binding more than one parameter at a time. By contrast, currying always works with one parameter at a time.
3. To fully evaluate a partially applied function, it is only necessary to bind the parameters that have not already been bound. By the definition of partial application, one or more parameters must already have been bound. By contrast, with a curried function, full evaluation requires binding of all parameters, and zero or more parameters may already have been bound.

Partial application

The following example demonstrates partial application (FSI console):

```
> let Add5 a b c d e = a + b + c + d + e;;

val Add5 : a:int -> b:int -> c:int -> d:int -> e:int -> int

> Add5 1 2 3 4 5;;
val it : int = 15
> let Add2More = Add5 1 2 3;;

val Add2More : (int -> int -> int)

> Add2More 4 5;;
val it : int = 15
```

The function **`Add2More`** is a function resulting from the partial application of **`Add5 1 2 3`**. **`Add2More`** can be evaluated as I did in the example by binding the remaining two parameters. Partial application works because **`Add5`** is a curried function. This illustrates the dependency of partial application on currying (noted by Syme et al. in the passage quoted previously). Incidentally, C#'s "partial methods" have nothing to do with partial function application.

Currying

Currying is handled by the F# language implementation and therefore essentially invisible to the programmer, but you can recognize where it will occur. Currying occurs when you call a multi-parameter function in F# without having to do anything special. The following example is a definition of a multi-parameter function, followed by a call to it:

```
> let Add5 a b c d e = a + b + c + d + e;;

val Add5 : a:int -> b:int -> c:int -> d:int -> e:int -> int

// the call site:

> Add5 1 2 3 4 5;;
val it : int = 15
```

When used at the call site, **`Add5 1 2 3 4 5`** is calling the curried function **`Add5 a b c d e`**. F# calls do not use parentheses around the arguments or commas between them. (If the function were defined instead by **`let Add5 (a, b, c, d, e) = a + b + c + d + e`**, it would still yield the same result, but the function would have a different type. This would be taken by the language implementation as defining a *single-parameter* function taking as input a single tuple with five elements, where the parentheses tell the language implementation to evaluate what is inside as a single possibly complex value, and the commas distinguish independent components of that value.)

In the previous example, the language implementation recognizes that **`Add5 1 2 3 4 5`** matches the signature of the previously defined function **`Add5 a b c d e`**, and therefore evaluates **`Add5 1 2 3 4 5`** as a function call.

Multi-parameter functions are automatically curried by the language implementation. To see the currying, we have to go beneath the covers.

Let's use dotPeek[19] to decompile this simple F# program:

```
open System.Security.Cryptography
open System.Text

[<EntryPoint>]
let main argv =
    let Add5 a b c d e = a + b + c + d + e
    let q = Add5 1 2 3 4 5
    printfn "%i" q
    0
```

Note how the function **`Add5`** is cast as curried functions (as in, each function has one parameter, ending with the final function where the second parameter is a literal):

[19] dotPeek - http://www.jetbrains.com/decompiler/

```
(FSharpFunc<int, FSharpFunc<int, FSharpFunc<int, FSharpFunc<int, FSharpFunc<int, int>>>>>)
```

And there you have it, **`Add5 a b c d e`** has been curried to the type **`FSharpFunc<'T, 'U'>`**[20] such that it can be used in the form *value, function*.

What is important to understand here is that F# syntax is defined so that any function that specifies its parameters with white space is automatically curried, and merely by using this syntax you are using the currying built into the language implementation.

Another example: Since **`Add x y`** will be evaluated as a curried function (its parameters are separated by white space), we can use partial application directly or by pipelining, which more explicitly creates partial function applications:

```
> Add 1 (Add 2 (Add 3 (Add 4 (5))));;   // Looks familiar to the decompiled F# code!
val it : int = 15

// - or, using pipelining –

> 1 |> (2 |> (3 |> (4 |> (5 |> Add) |> Add) |> Add) |> Add);;
val it : int = 15
```

Lessons

- The F# language implementation creates a curried function whenever you represent multiple parameters individually rather than as a tuple or other complex argument.
- You are creating a partial function application when you bind values for only the first *n* parameters of a multi-parameter function.

What is imperative programming?

Now that we have demystified one of the most commonly misunderstood terms in functional programming, let's take a big step back and survey the landscape of imperative and functional programming.

"In computer science, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state…imperative programs define sequences of commands for the computer to perform."[21] In an object-oriented language context, I will specialize this definition slightly—"imperative programming is a programming paradigm that describes computation in terms of statements that change the *object state.*"

This is a fundamental feature of imperative programming. For example, this C# code alters the state of Point *p*:

[20] FSharpFunc<'T, 'U> Class: http://msdn.microsoft.com/en-us/library/ee340302.aspx

[21] Imperative programming: http://en.wikipedia.org/wiki/Imperative_programming

```
Point p = new Point();
p.X = 1;                // Mutation
p.Y = 2;                // Mutation
p.Offset(11, 12);       // Function with side effects.
Console.WriteLine(p.ToString());  // ToString is for illustration purposes only.
```

This code, because of its simplicity, is an excellent example of how often we use mutable objects that exhibit side effects.

What are classes?

One way to think about a class is that it is nothing more than a container for fields, some of which are exposed to the outside world, and methods which return results, affect the state of the class' fields, or both. This is the definition of encapsulation. When we write object-oriented code, every class we write:

- Describes everything necessary to maintain the state of the fields in the class.
- Describes how state is altered using methods such as property getters and setters.
- Describes computations (methods) that utilize the current state, and those computations may have side effects resulting in state change.

This may seem like an overly narrow description because it ignores other OO features like inheritance and polymorphism, and the concept that a class describes a type, but fundamentally, the description is true for every imperative class, whether it's a super-class or a derived class.

The C# example at the end of the previous section illustrates each of these points:

- The class **`Point`** consists of two fields, X and Y.
- The statements **`p.X = 1`** and **`p.Y = 2`** are property setter functions that change the state of **`p`** and illustrate that the object is mutable.
- The statement **`p.ToString()`** can be considered a computation on the current state, converting X and Y to a string representation.
- The statement **`p.Offset(11, 12)`** is a computation that alters the current state, illustrating the side effect of the computation, in that it mutates the object's state.

This illustrates two features of imperative programming: mutability and side effects.

- Mutability is the ability to change an object's state directly and usually explicitly.
- In a side effect, the object is mutated (its state changes) indirectly and usually implicitly. Side effects however are not limited to changing the object's state—side effects can also change the state of another object, a data store, and so forth.

A good example of a useful mutable object with side effects is a file reader that needs to keep track of the position in the data stream and set an end-of-file flag when the end of the file is reached. This creates problems for pure functional programming languages and is handled in interesting ways, which we'll look at later.

In addition to the previous narrow description, object-oriented programming, and specifically classes, offer a couple additional features:

- Unifies behavior patterns with polymorphism (including default values).
- Associates with other classes to create a rich composition of objects utilizing the object-oriented principles of inheritance and membership.

These illustrate the strengths of imperative programming, and in "impure" functional programming languages such as F#, one can continue to utilize these features, but carefully.

Lessons

- Imperative code has mutability "coded" into it very naturally.
- Imperative code very frequently has side effects.
- Bonus lesson: Imperative methods are not curried—the compiler does not support implicit partial application; you have to do this yourself.

What is functional programming?

"In computer science, functional programming is a programming paradigm, a style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions and avoids state and mutable data. Functional programming emphasizes functions that produce results that depend only on their inputs and not on the program state—i.e. pure mathematical functions."[22]

In functional programming we are defining terms, not giving orders. Terms are defined by equations that may, for instance, explain how to construct a complex structure from simpler structures. The work of programming is essentially setting up systems of equations so they can be solved automatically by appropriate substitution during execution.

Immutability

Functional programming, at its core, emphasizes immutability. Mathematical definitions, after all, do not change. Without mutable objects, there are no side effects. This is achieved by creating a new object every time there is a "state change"—the old object remains unaffected and a new object is created instead to reflect the result of the computation.

In F#, the preferred implementation of the previous C# example would be immutable. The = sign in a functional language indicates an equation, not an assignment.

```
type Point =
  {x: int;
   y: int;}

let Offset p dx dy = {x = p.x + dx; y = p.y + dy}
```

[22] Functional programming: http://en.wikipedia.org/wiki/Functional_programming

```
let p = {x=1; y=2}
let p2 = Offset p 11 12
;;
```

In this example, **p2** is not the same instance as **p** nor is it equal to **p**. The reason is because the function **Offset** does not change (mutate) **p**; it creates a new instance of a point.

Let's encapsulate the function **Offset** in an F# class type, again preserving immutability:

```
type Point(x : int, y: int) =
    member self.X = x
    member self.Y = y
    member self.Offset dx dy = new Point(self.X + dx, self.Y + dy)

let p = new Point(1, 2)
let p2 = p.Offset 11 12
```

(From here on, we'll omit the terminator **;;** in many examples. Just add it at the end as in the previous example to get the examples to evaluate in the console.)

We can see that the **Offset** function explicitly constructs a new **Point** object, thus **p <> p2**.

The only way to update the **X** and **Y** members is to explicitly declare them to be mutable using:

- The **member val** keywords to automatically create a backing store.
- The **with get, set** keywords to create the property getter and setter.

```
type Point(x : int, y: int) =
    member val X = x with get, set
    member val Y = y with get, set
    member self.Offset dx dy =
        self.X <- self.X + dx
        self.Y <- self.Y + dy
        self

let p = new Point(1, 2)
let p2 = p.Offset 11 12
```

In this example, we have created the equivalent of the C# **Point** class, and the result is that **p2** equals **p** and in fact, **p2** is the same instance of *p* because the **Offset** function returns itself.

This is not functional programming, so if you're going to program like this, stick with C# or some other imperative programming language. It does however illustrate why F# is called an "impure" functional programming language.

First-class and higher-order functions

"In computer science, a programming language is said to have first-class functions if it treats functions as first-class citizens. Specifically, this means the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures."[23]

Higher-order functions[24] are functions that can either take other functions as arguments or return them as results. Higher-order functions are a key feature of a functional programming language. "A defining characteristic of functional programming languages is the elevation of functions to first-class status. You should be able to do with a function whatever you can do with values of the other built-in types, and be able to do so with a comparable degree of effort."[25] What makes first-class functions "first class" is that the language implementation allows them to be treated as a kind of value, so you can do with them all the things you can do with other values.

In imperative languages, we usually don't pass functions as parameters or return functions. Traditionally, where functions were supported, imperative language implementations did not allow them to be treated as values. Only recently, for example with C#'s **`Action`** and **`Func`** classes and lambda support, are there explicit language capabilities (beyond delegates) for passing functions as parameters and returning functions. These features are in fact borrowed from functional languages. Older imperative languages such as Pascal, Fortran, BASIC, and so forth have no facilities for passing functions as parameters or returning functions. For example:

Functions in structures

With C# and the **`Func<T, U>`** class, we can create a list of functions having the same signature:

```
static int SquareValue(int n) { return n * n; }
static int DoubleValue(int n) { return n + n; }

List<Func<int, int>> funcList = new List<Func<int, int>>() { SquareValue, DoubleValue };
```

(For the C# audience, I am intentionally refraining from using constructs like **`var funcList`**.)

In F#, we can implement this more naturally:

```
let squareValue n = n * n
let doubleValue n = n * n
let funcList = [squareValue; doubleValue]
```

Functions as parameters

We can pass functions to other functions. Again, in C#, this might be written like this:

---

[23] First-class functions: http://en.wikipedia.org/wiki/First-class_function

[24] Higher-order functions: http://en.wikipedia.org/wiki/Higher-order_function

[25] Functions as First-Class Values: http://msdn.microsoft.com/en-us/library/dd233158.aspx

```
static int Operation(Func<int, int> f, int value) { return f(value); }

int result1 = Operation(SquareValue, 5); // Call using defined function.
Int result2 = Operation((n) => n * n, 6); // Call using lambda expression.
```

In F#, this can be written as:

```
let operation f value = f value
let result1 = operation squareValue 5
let result2 = operation (fun n -> n * n)  6
```

Note the explicit use of the **fun** keyword to define lambda expressions.

Functions returning functions

In C#, we can return functions:

```
static Func<int, int> GetMyOperation() { return SquareValue; }

int result3 = GetMyOperation()(10);
```

And in F#:

```
let getMyOperation = squareValue
let result3 = getMyOperation 10
```

Does this mean C# is a functional programming language?

The previous example illustrates how C# supports functional programming behaviors, so the answer is “sort of, yes.” We will see later that there are distinctions between imperative languages that support function programming constructs, and functional languages that support imperative programming constructs.

C# falls in the first category, and F# falls in the second category. A “pure” functional programming language will fall outside of both categories.

Where C# and F# start to diverge

We can do some interesting things with function binding and functions as parameters, like writing “generic” operators. In C#, this would look like:

```
static T Adder<T>(Func<T, T, T> add, T a, T b) { return add(a, b); }
```

```
var sum1 = Adder((a, b) => a + b, 1, 2);         // 3
var sum2 = Adder((a, b) => a + b, 10.5, 12.6);   // 23.1
var sum4 = Adder((a, b) => a + b, "a", "b");     // “ab”
```

Here the compiler infers the type from the values, which must all be of the same type. We can use it with different types because the lambda expression is evaluated when needed, and hence the type is at that point known. We can do something similar in F# using the **`inline`** keyword:

```
let inline adder (a : 'T) (b : 'T) : 'T = a + b

let sum1 = adder 1 2
let sum2 = adder 10.5 12.6
let sum3 = adder "a" "b"
```

Here **`adder`** is a function that takes two parameters, **`a`** and **`b`** of type **`T`**, and returns a type **`T`**.

There is an interesting difference: in C# we cannot create a generic delegate that is assigned the lambda expression without specifying the type when the delegate is assigned. For example, this does not work:

```
delegate T del<T>(T a, T b);
static del<T> adder = (a, b) => a + b;  // <<-- del<T> results in an error
static T DlgtAdder<T>(del<T> add, T a, T b) { return add(a, b); }
var sum5 = DlgtAdder(adder, 3, 4);
```

If we specify **`del<int>`**, then the previous code works. Note that in F#, we can definitely assign the function to a name and use that function later with type inference.

We can go even further, providing the operation to perform at the call site:

```
let inline operation op (a : 'T) (b : 'T) : 'T = op a b
let sum1 = operation (+) 1 2     // 3
let diff1 = operation (-) 10 4   // 6
```

This is something you cannot do in an imperative language—not even in C#, without some significant workarounds. Granted, my example is a bit esoteric, but it does well to serve as an example of where a functional language like F# enables you to do things you otherwise could not do at all.

Lessons

- Functional programming avoids mutability (which can be annoying at times).
- Immutable objects cannot be affected by side effects. (Because F# is “impure,” though, an object with all immutable fields could still have methods that cause side effects somewhere else, e.g., for display.)

- At some point, the syntax of C# simply becomes too unwieldy and F# starts to look syntactically more attractive.
- At some point the support for functional programming fails in C#, whereas F#, being a functional programming language, supports “the last mile.”

Chapter 2 Change Your Thinking

Functional programming requires a different way of thinking about the architecture of your software as compared to imperative programming. The first order of business is to stop thinking in terms of statement-based programming and start thinking in terms of expression-based programming. This means thinking in terms of systems of definitions and what can be derived from what, rather than sequential cause and effect. It will lead naturally to thinking in terms of immutable objects rather than the (mostly unconscious) mutable programming we're all used to. Expression-based programming and immutable objects have direct consequences for a successful functional programmer and the ease with which we think about algorithms in a functional programming language.

Expression-based programming

What makes an imperative language imperative? Earlier I wrote:

"In computer science, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state…imperative programs define sequences of commands for the computer to perform."[26]

In the previous quote is the key word "statement". "In computer programming a statement is the smallest standalone element of an imperative programming language…[Statements do] not return results and are executed solely for their side effects."[27] Obviously, in modern imperative languages, statements can also return values but they don't have to.

Conversely, "an expression-oriented programming language is a programming language where every (or nearly every) construction is an expression and thus yields a value…All functional programming languages are expression-based."[28]

A statement implicitly says "trust me and do this, I claim it's the right thing to do," whereas an expression says "here's a way to derive a particular kind of result." We can see the difference from our earlier examples.

A C# *statement*:

```
p.Offset(11, 12);
```

An F# *expression*:

[26] Imperative programming: http://en.wikipedia.org/wiki/Imperative_programming
[27] Statement-oriented programming: http://en.wikipedia.org/wiki/Statement_(programming)
[28] Expression-oriented programming: http://en.wikipedia.org/wiki/Expression-oriented_programming_language

```
let p2 = p.Offset 11 12
```

Void return in F#

In order to work with classes defined in the .NET framework, F# has to allow for statement-based programming and therefore allows a **void** return type. This opens a Pandora's box of mutable functions; why would you call a function that returns nothing unless it has some side effect? So, for example, in C# we often write methods with a **void** return type:

```
public void PrintSomething(string s)
{
  Console.WriteLine(s);
}
```

Because F# is an impure language and needs to support void returns in order to work with the .NET framework and other imperative .NET languages, we can also write, in F# (FSI console):

```
let PrintSomething s =
  printfn "%s" s
  ();;

val PrintSomething : s:string -> unit
```

We are told this is a function that takes a string and returns a "unit," F#'s word for "void." The side effect is that something is emitted to the console window (its state changes). One should mostly avoid writing functions that return "unit," as these imply that the function has side effects. There are perfectly valid reasons, such as persisting data to a database, but of course that is a side effect!

Eager expression evaluation

"In computer programming, eager evaluation or greedy evaluation is the evaluation strategy used by most traditional programming languages. In eager evaluation, an expression is evaluated as soon as it is bound to a variable. The alternative to eager evaluation is lazy evaluation, where expressions are only evaluated when evaluating a dependent expression. Imperative programming languages, where the order of execution is implicitly defined by the source code organization, almost always use eager evaluation."[29]

Statements behave differently in statement-based and expression-based programming. In a statement-based language, at least within a small context such as a block, in the absence of other control logic, statements execute sequentially *in the order in which they are defined*. On the other hand, an impure expression-based language like F# allows statements to be embedded in expressions. In this case, the embedded statement will execute at the time the expression is evaluated, whenever that is. This is often difficult to wrap one's head around at

[29] Eager evaluation: http://en.wikipedia.org/wiki/Eager_evaluation

first, and results in some interesting behaviors. For example, let's say you have a function that prints something and returns a literal (FSI console):

```
> let printAndReturn =
  printfn "Hi!"
  5;;
Hi!

val printAndReturn : int = 5

> printAndReturn;;
val it : int = 5
```

Note how *Hi!* is emitted immediately—F# performs "eager" expression evaluation by default. Furthermore, when we later call the function, because the expression has already been evaluated, the **`printfn`** statement embedded in it will not execute and *Hi!* is no longer emitted.

Conversely (FSI console):

```
> let Adding a b =
  printfn "Adding %d %d" a b
  a+b
;;

val Adding : a:int -> b:int -> int

> Adding 1 2;;
Adding 1 2
val it : int = 3
> Adding 3 4;;
Adding 3 4
val it : int = 7
```

Here, **`Adding`** cannot be evaluated until its parameters have been supplied. Therefore we get the console message *Adding…* each time we evaluate at the call site with **`Adding 1 2`** or **`Adding 3 4`**.

Statements embedded in expressions can lead to a lot of confusion, for example when using **`printfn`** for debugging purposes. The expression will evaluate as soon as it can, not necessarily in the order that you would expect, especially coming from an imperative, statement-based, programming language.

Expression-based programming has other nuances. There are two fundamental evaluation strategies for expressions: eager and lazy. Let's say you want to evaluate some expression based on whether a value is true or false:

```
> let Add a b =
    printfn "Computing a + b"
    a + b
```

```
let Subtract a b =
    printfn "Computing a - b"
    a – b

let Compute operation sum difference =
    match operation with
    | "Add" -> sum
    | "Subtract" -> difference
    | _ -> 0

Compute "Add" (Add 1 2) (Subtract 5 4)
;;
Computing a + b
Computing a - b

val Add : a:int -> b:int -> int
val Subtract : a:int -> b:int -> int
val Compute : operation:string -> sum:int -> difference:int -> int
val it : int = 3
```

Notice how the expressions **`(Add 1 2)`** and **`(Subtract 5 4)`** are both evaluated regardless of which expression result we want. This can potentially affect performance, so instead, we want to explicitly use lazy expression evaluation, which is discussed in the next section.

Lazy expression evaluation

"In programming language theory, lazy evaluation, or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations…Delayed evaluation has the advantage of being able to create calculable infinite lists without infinite loops or size matters interfering in computation."[30]

Note that lazy evaluation was introduced as a part of lambda calculus, which if you recall, is the foundation for functional programming, so it is interesting that F# employs an eager evaluation scheme. However, it does support explicit lazy evaluation. The ML language family, on which F# is based, historically used eager evaluation by default, because its properties are more easily provable and its performance is more predictable. In the Haskell.NET experiment that preceded work on F#, Don Syme and Simon Peyton-Jones encountered so much difficulty trying to get a good implementation of laziness by default on top of .NET's fundamentally eager approach that they basically gave up, and turned instead to a language with the same eager bias as .NET.

Let's modify the previous example slightly, changing how we call Compute by using the "lazy" type[31]:

```
let Compute operation sum difference =
    match operation with
    | "Add" -> sum
```

[30] Lazy evaluation: http://en.wikipedia.org/wiki/Lazy_evaluation

[31] Lazy Computations: http://msdn.microsoft.com/en-us/library/dd233247.aspx

```
    | "Subtract" -> difference
    | _ -> lazy 0

let result = Compute "Add" (lazy (Add 1 2)) (lazy (Subtract 5 4))
;;

val Add : a:int -> b:int -> int
val Subtract : a:int -> b:int -> int
val Compute :
  operation:string -> sum:Lazy<int> -> difference:Lazy<int> -> Lazy<int>
val result : Lazy<int> = Value is not created.

> result.Force();;
Computing a + b
val it : int = 3
```

Notice now that when we force the expression to evaluate, only the expression **`(Add 1 2)`** is evaluated! By the way, also notice how the expression **`0`** had to be "lazy evaluated" in the match statement; we must keep type consistency in all the return values.

Functional programming requires that you have an understanding of when an expression evaluates and, keeping that in mind, consider when you might want to explicitly use lazy evaluation to improve the performance of the code. Note that some functional programming languages, such as Haskell, use "lazy evaluation" by default.

Lessons

- Imperative programming is statement-based.
- Functional programming is expression-based.
- Avoid functions that return *unit* ("void"), as this is a serious indicator that you are not coding in a "functional" manner and that your function has side effects.
- Expressions evaluate "eagerly" in F#, so for performance reasons, you may need to explicitly instruct the compiler that you want lazy evaluation.

Think immutable

It can be very difficult to learn how to think in terms of an immutable world. Things in everyday life are mutable. When we get in our car to drive somewhere, we turn the engine on. When we arrive at our destination, we turn the engine off. We are clearly changing the state of the car. We do not "copy" the car so that we now have one car with the engine off and another "identical" car but with the engine on! We live in a stateful world and one of the fundamental characteristics of things in our world is their state.

On the other hand, mathematics and logic deal primarily with immutable things, so functional programming can benefit from being much closer to natural ways of working in math and logic. Database design and significant areas of digital logic design also work with immutable definitions. Immutability gives us enormous power to reason about behavior of programs from a compile-time perspective—power to potentially *know* and be able to *prove* that a program is correct. Scientific understanding of the world is, after all, based largely on mathematics. To have

a scientific understanding is to be able to explain outputs in terms of inputs, without magic or unfounded assumptions, just like in functional programming. Much of the work of developing a program or scientific theory is formalization—not so much writing code in a syntax, as getting from a vague understanding to a sharp one.

Shifting to a concept of immutability can be mentally difficult, but there are certain concrete structural things that you can do when writing functional code that help to make immutability more automatic. These are:

- Reduce encapsulation by creating associations explicitly as maps.
- Create smaller functions that do just one thing.

Reducing encapsulation

Thinking in terms of immutability means changing how you think about encapsulation. Often, we use encapsulation to collect many different concepts, or attributes, under a single umbrella. In imperative programming, this often appears reasonable but eventually leads to entanglement and complexity. In a functional programming environment, encapsulating seemingly related but actually distinct concepts creates undo “noise” in a given type and makes it difficult to work with immutable objects. What we need to do instead is tease apart the associations, using separate objects to interrelate them. Among other positive effects, this makes working with types much easier because a type doesn’t include attributes that fall into the “has a” relationship. A type should always express exactly what it *is* (that is to say, what *defines* it), not what it *has* (that is to say, what things are associated with it).

Think relational database architecture

The idea of “what defines an object” has its equivalence in database design and the concept of a unique key—given a table, each record is defined by what fields make that record unique. Yes, there may be other attributes, but the salient point is that you can always identify the unique record by its unique key. The other fields in the table can’t do this. This helps us identify core types in functional programming.

Another key aspect of database architecture is that associations are represented as separate tables through the use of foreign keys. When reducing encapsulation, think about how you would represent the class in a database. What are the separate tables you will need, and how are they associated? When you have gone through this exercise, you will be ready to implement the types correctly in F#.

Another key aspect of good database architecture is normalization, specifically, not repeating the same data. We will also use this practice as a guideline for creating better F# code.

An example of reducing encapsulation

We’ll work next with an example and see how we can relate this new way of thinking to a very imperative implementation of the C# class **`User`**:

```
public class User
{
```

```
        private List<string> roles = new List<string>();
        private byte[] passwordHash;

        public string Username { get; set; }
        public string EmailAddress { get; set; }
        public string PasswordHash
        {
                get { return Encoding.UTF8.GetString(passwordHash); }
        }

        public void AddRole(string roleName)
        {
                roles.Add(roleName);
        }

        public void RemoveRole(string roleName)
        {
                roles.Remove(roleName);
        }

        public void SetPassword(string newPassword, HashAlgorithm hashAlgo)
        {
                byte[] bytes = Encoding.UTF8.GetBytes(newPassword);
                passwordHash = hashAlgo.ComputeHash(bytes);
        }
}

// Example usage:

User user = new User();
user.SetPassword("Foobar", new SHA1Managed());
Console.WriteLine("Password Hash: " + user.PasswordHash);
```

This simple class serves the purpose of illustrating mutability and side effects in a typical imperative style. This class requires some rework to be a well-behaved set of F# types and functions. Note how even in the use of the terms *types* and *functions*, we are decomposing the properties of a class into their constituent components. If implemented as an immutable object in F#, it would require cloning all the properties, including the collection of roles. This is something we want to optimize when a user's configuration changes.

If we apply our "design a class like you would design a database" principle, we can conclude that:

- A user is *defined* by the combination of username and password or email and password, so the properties username, email, and password are a natural grouping. The additional roles collection is not characteristic of defining a user and should be removed from the type that defines **`User`**.
- Roles can also be considered a foreign key association, giving further motivation to the idea that it should be its own "first class" type and that a separate type that manages associations between users and roles is needed.
- This also "normalizes" the data by eliminating the duplication of role names across many users.

We can now define our types in F#, starting with **`User`**:

```
type User =
    {
        username : string;
        emailAddress : string;
        passwordHash : byte[];
    }

// Example Instantiation:

let u =
  {
    username = "Marc";
    emailAddress = "marc.clifton@gmail.com";
    passwordHash = null;
  }
```

We also need a simple **`UserRole`** type:

```
type UserRole =
    {
        rolename : string;
    }
```

And lastly, a type that associates a user with a list of roles:

```
type UserRoles =
    {
        user : User;
        roles : List<Role>;
    }
```

We can now instantiate a user with a couple roles, for example:

```
let u =
    {
        username = "Marc";
        emailAddress = "marc.clifton@gmail.com";
        passwordHash = null;
    }

let role1 = {rolename = "Administrator"}
let role2 = {rolename = "SuperUser"}

let marc = {user = u; roles = [role1; role2]}
```

Notice that by reducing the complexity of the encapsulation illustrated in the C# **`User`** class, we have created three different types. Granted, there is nothing that prevents us from doing this in an imperative language; it's just that we typically gravitate towards complexity when working

with classes. The types we've created here are much simpler structural compositions, which ultimately facilitates being able to do more with these simpler structures. Using these new types, we will next explore the "functional" way of operating on these types in an immutable context.

Lessons

- Reduce object complexity by decoupling the implicit associations of types.
- To do this, think more in terms of a normalized relational database architecture:
    - What are the fields that uniquely define a record?
    - What are the associations between records?
- Create small types (records), as these are more easily associated with other types (records).

Reduce complex functions

Let's look again at the **`SetPassword`** method in C#:

```
public void SetPassword(string newPassword, HashAlgorithm hashAlgo)
{
       byte[] bytes = Encoding.UTF8.GetBytes(newPassword);
       passwordHash = hashAlgo.ComputeHash(bytes);
}
```

This method is very simple, but from a functional programming perspective, it has several problems:

1. The encoding protocol is hard-coded into the method body—anyone using this method cannot change the encoding protocol.
2. The hash algorithm must be known when the method is called.
3. The method does not return the hashed password; it instead sets the internal field to the resulting hash value—a side effect.
4. This method cannot be extended—for example, adding password salt—without deriving from the entire class.
5. Since the programmer didn't specify that this method is virtual, the method in fact cannot be overridden, forcing the programmer to take more drastic measures to create the desired function. The code as written is therefore not very extensible, and any workarounds that the next programmer creates probably results in code duplication, introduces potential errors if the programmer changes the original method, and contributes to code brittleness.

Certainly the last point can again be easily addressed by better OO design, and it is certainly possible to write bad functional programs as well; however, functional programming tends to promote the thinking "do one thing only." We therefore decompose the **`SetPassword`** method into two functions.

For the following examples, we need to include a couple .NET namespaces:

```
open System.Security.Cryptography
open System.Text
```

We can still define the encoding function as:

```
let encodeUTF8 password = Encoding.UTF8.GetBytes(password : string)
```

And the hash computation algorithm as (one way of doing this is):

```
let getHash hashAlgo bytes = (hashAlgo : HashAlgorithm).ComputeHash(bytes :
byte[])
```

Because we're working with a .NET base class (**`HashAlgorithm`**) that has many overloaded **`Compute`** methods, we have to specify the type information for the class containing the **`Compute`** method and the parameter type.

Another way of defining this function is to specify the required types in the parameters of the **`getHash`** method:

```
let getHash (hashAlgo : HashAlgorithm) (bytes : byte[]) =
hashAlgo.ComputeHash(bytes)
```

This second form allows the compiler to select the correct **`Compute`** method because it already knows the type for **`hashAlgo`** as well as the type for **`bytes`**.

By separating the C# method into two simpler functions, we are moving toward working with more core behaviors, giving the programmer the ability to change the behavior of the program without running into the problems I've listed previously.

We can define a third function, which, given a password, computes the password hash:

```
let pwdHash pwd = getHash (new SHA1Managed()) (encodeUTF8 pwd)
```

And we use it like this (FSI console):

```
> pwdHash "abc";;
val it : byte [] =
  [|169uy; 153uy; 62uy; 54uy; 71uy; 6uy; 129uy; 106uy; 186uy; 62uy; 37uy;
    113uy; 120uy; 80uy; 194uy; 108uy; 156uy; 208uy; 216uy; 157uy|]
```

Not only is this a fairly ugly F# function, but it still has an imperative "feel" to it. A more functional approach would be to use function pipelining.

Lessons

- Create functions that do one thing and one thing only.
- Smaller functions are more composable (more on this next).
- Smaller functions make it easier for you to change the behavior of the application without breaking other parts of the code.

Function pipelining, partial application, and function composition

While it can be considered a mantra to write smaller functions, even in imperative languages, at the end of the day all those little functions must still be put together to do something bigger. Imperative programming promotes a very tight coupling of "sequence" and requires that all the information be available right now in order to execute the sequence of instructions.

Functional programming has several different ways of putting together all those little functions:

- Function pipelining: By supplying the first value, the resulting value(s) of one function feed directly into the next function, resulting in a final value.
- Partial application: Creating a function that has some of its initial parameters bound to values, allowing the rest of the parameters to be supplied later.
- Function composition: Creating a function that is composed of several other functions, where, similar to pipelining, the resulting value of one function supplies the first parameter of the next function.

Function pipelining

One of the important features in functional programming is the idea of a function pipeline. "The pipeline operator, **|>**, is one of the most important operators in F#."[32] The pipeline operator "can be thought of as threading an argument through a series of functions."[33] Nested function calls can be difficult to read. A function pipeline with *n* stages provides an easy, left-to-right readable alternate syntax for a nested series of function calls *n* levels deep.

If we rewrite the previous F# code using the pipeline operator **|>**, the code becomes more readable:

```
let pwdHash pwd = pwd |> encodeUTF8 |> getHash(new SHA1Managed())
```

[32] Function Pipelining: http://en.wikibooks.org/wiki/F_Sharp_Programming/Higher_Order_Functions#The_.7C.3E_Operator

[33] Function Pipelining: http://blogs.msdn.com/b/ashleyf/archive/2011/04/21/programming-is-pointless.aspx

Here we are telling the compiler:

1. Create a function called **pwdHash** that takes a single parameter, which is inferred to be a string.
2. Call the function **encodeUTF8**, passing the password.
3. The evaluation is passed to the function **GetHash**, along with the parameter **new SHA1Managed()**.

Partial application

"In computer science, partial application (or partial function application) refers to the process of fixing a number of arguments to a function, producing another function of smaller arity."[34] ("Arity" means the number of arguments that a function accepts.)

We can further "functionalize" the previous code by deferring the hash algorithm, simply by creating a partial application:

```
let pwdHash2 pwd = pwd |> encodeUTF8 |> getHash2
```

Here, we are defining a function that takes a password but returns a function that expects a hash algorithm to be provided. However, this also requires that we switch the parameter order of the **getHash** function, which is why I previously called it **getHash2** (more on this later):

```
let getHash2 (bytes : byte[]) (hashAlgo : HashAlgorithm) = hashAlgo.ComputeHash(bytes)
```

Why? Because we are providing the first parameter in the pipeline and expecting the caller to later provide the second parameter, which is the hash algorithm.

In the **pwdHash2** function, we are "piping" the password parameter into the byte encoder and then "piping" the resulting encoded **byte[ ]** value into the **getHash2** function, which returns a function that expects an encoding algorithm and returns a **byte[ ]** of encoded values.

We use it like this (FSI console):

```
> let myEncPwd2 = ("abc" |> pwdHash2)(new SHA1Managed());;

val myEncpwd2 : byte [] =
  [|169uy; 153uy; 62uy; 54uy; 71uy; 6uy; 129uy; 106uy; 186uy; 62uy; 37uy;
    113uy; 120uy; 80uy; 194uy; 108uy; 156uy; 208uy; 216uy; 157uy|]
```

Observe how we created the partial application **pwdHash2** that, on evaluation, encodes our password with UTF8 encoding, but lets us defer the selection of the actual hash algorithm. We assign the encoded password by piping in the raw password string, completing the function application by providing the hash algorithm object.

[34] Partial application: http://en.wikipedia.org/wiki/Partial_application

The syntax of this is a little messy because we're mixing in .NET objects rather than working with pure F# functions. However, I believe this represents the real world of what you'll be dealing with when programming in F#, so I'd rather illustrate this syntax than a pure F# implementation. It's all part of thinking functionally but realizing that you still have to work with imperative style frameworks that don't support function pipelining and currying.

Lessons

- When you write functions, consider which parameters are most likely to be partially applied and put those parameters first.
- If the programmer didn't provide the parameters in the order that you want, write a function that flips the parameters around that you can use to create a partial application function.

Revisiting function pipelining and introducing tuples

We can now consider how to add different behaviors, for example, adding some "salt" to the password. This further illustrates how the behavior of our application can be easily changed as a result of simple, composable functions. First, let's look at the encoding function this way:

```
let myEncPwd2 = ("abc" |> encodeUTF8 |> getHash2)(new SHA1Managed())
```

We'll be appending an encoded *salt* before passing the result to the **`getHash2`** function:

```
let mySaltedPassword = (("abc" |> encodeUTF8, "salt" |> encodeUTF8) ||> Array.append |>
getHash2)(new SHA1Managed())
```

Notice the **`||>`** operator. This operator takes a tuple and pipes it to a function that takes two parameters. The tuple is constructed from the password and **`salt`**:

```
("abc" |> encodeUTF8, "salt" |> encodeUTF8)
```

Next, it's piped into the **`Array.append`** function:

```
let mySaltedPassword = ("abc" |> encodeUTF8, "salt" |> encodeUTF8) ||> Array.append
```

The result is then piped to the **`getHash2`** function with the desired hash encoder. If you're curious, there's also the triple pipeline operator **`|||>`**, which takes a tuple of three values and passes it to a function that takes three parameters.

Function composition

Pipelining requires that the input values are provided to "initialize" the pipe. For example, this function:

```
encodeUTF8 pwd
```

Is rewritten using pipelining as:

```
pwd |> encodeUTF8
```

However, we may not know the value, and this is where function composition, using the **>>** operator, comes into play. The example used earlier:

```
let myEncPwd2 = ("abc" |> encodeUTF8 |> getHash2)(new SHA1Managed())
```

Can be rewritten without the initial value of **abc**:

```
let encodeUTF8 password = Encoding.UTF8.GetBytes(password : string)
let getHash (hashAlgo : HashAlgorithm) (bytes : byte[]) = hashAlgo.ComputeHash(bytes)
let pwdEncoder = encodeUTF8 >> getHash(new SHA1Managed())
```

Here we have created a function **pwdEncoder** that is a composition of the **encoder** and **hashing** functions. We use it like this (FSI console):

```
> pwdEncoder "abc";;
val it : byte [] =
  [|169uy; 153uy; 62uy; 54uy; 71uy; 6uy; 129uy; 106uy; 186uy; 62uy; 37uy;
    113uy; 120uy; 80uy; 194uy; 108uy; 156uy; 208uy; 216uy; 157uy|]
```

Or like this:

```
“abc” |> pwdEncoder
```

Lessons

- In order to take advantage of functional composition, it’s best to have functions that take only one parameter.
- Your functions in a composition most likely will be partial function applications!

Functions vs. literals: the key difference between pipeline and composition

Function composition, as the name implies, is a composition of functions that is itself a function. First-class functions are a kind of value, but values in general are not functions. So for example, I cannot write:

```
let array1 = [|1;2;3|]
let array2 = [|4;5;6|]
let array3 = array1 >> Array.append array2
```

The compiler complains that it is expecting **array1** to be a function, but instead it is an **int[ ]**; in other words, a literal.

We can, however, write:

```
let t1 = encodeUTF8 >> Array.append (encodeUTF8 "salt")
> “abc” |> t1;;
val it : byte [] = [|115uy; 97uy; 108uy; 116uy; 97uy; 98uy; 99uy|]
```

We can do this because **encodeUTF8** is a function, not a literal, and we can use functions anywhere we need values. Once we supply the function **t1** with the initial literal value *abc*, then **t1** evaluates, passing the result of **“abc” |> encodeUTF8** to the **Array.append** function, which has already received as its first parameter the value resulting from evaluating **encodeUTF8 “salt”**.

Consider this partial function application in which we do not provide the encoded *salt* parameter:

```
let t1 = encodeUTF8 >> Array.append
```

Which can be used as follows:

```
("abc" |> t1) (encodeUTF8 "salt")
// - or –
(encodeUTF8 "salt") |> ("abc" |> t1)
```

Notice how we are combining pipelining (starting with an initial value) with function composition (a function that encodes a value and passes the result to the partial application of the append function.) It is very important to know whether you are working with a literal or a “function as a value,” as this guides you on how to work with function pipelining and function composition.

Partial application and pipelining

It’s also important to understand the interaction between partial application and pipelining, as there can be unintended consequences.

Given (FSI console):

```
let encodeUTF8 (password :string) = Encoding.UTF8.GetBytes(password)
let getHash (hashAlgo : HashAlgorithm) (bytes : byte[]) =
hashAlgo.ComputeHash(bytes)

> ((encodeUTF8 "abc", encodeUTF8 "salt") ||> Array.append) |> getHash(new
SHA1Managed());;
val it : byte [] =
  [|153uy; 25uy; 141uy; 252uy; 72uy; 224uy; 52uy; 198uy; 99uy; 86uy; 24uy;
    63uy; 133uy; 78uy; 179uy; 34uy; 246uy; 7uy; 193uy; 221uy|]
```

Here, **`getHash(new SHA1Managed())`** returns a partial application function, having had the first parameter "fixed" or "bound." Having been supplied the first parameter, this leaves the **`byte[ ]`** parameter to be supplied by the pipeline.

Compare this with:

```
> ((encodeUTF8 "abc", encodeUTF8 "salt") ||> Array.append) |> getHash;;

  ((encodeUTF8 "abc", encodeUTF8 "salt") ||> Array.append) |> getHash;;
  ------------------------------------------------------------^^^^^^^

stdin(134,61): error FS0001: The type 'byte []' is not compatible with the type
'HashAlgorithm'
```

Here we can clearly see that pipelining tries to supply the first parameter, but since **`getHash`** is a partial application in which no parameters have been supplied, it fails with a type mismatch because the pipeline is attempting to provide the first parameter, which is of type **`HashAlgorithm`**, with a **`byte[ ]`**!

One correct usage would look like this:

```
(new SHA1Managed(), ((encodeUTF8 "abc", encodeUTF8 "salt") ||> Array.append)) ||> getHash;
```

Value confusion

It becomes clear from the previous code that the partial application **`getHash(new SHA1Managed())`** takes precedence over the pipeline operator. When we didn't correctly create the partial function application, we were able to identify this problem during coding because the **`getHash`** function has parameters of different types and the compiler or IDE told us we had a problem. However, if the parameters are the same type, we can easily create computational errors that are only discovered at run time. We can illustrate this point more clearly with the following code (FSI console):

```
let Sub x y = x - y;;

val Sub : x:int -> y:int -> int

> Sub 5 3;;
val it : int = 2     // We expect 5 – 3 = 2
> let t = 5 |> Sub;; // Here we create a partial application "Sub 5"

val t : (int -> int)

> t 3;;              // And we expect that t(3) = (Sub 5)(3) = 2
val it : int = 2
> let t = 5 |> Sub 3;;    // But what does this do? Do we expect this to = 2 ???

val t : int = -2   // This is 3 – 5 !!!
```

Because F# evaluates from left to right, it would seem logical that in the code **`5 |> Sub 3`** that **5** is assigned to **x**, resulting in the partial function application **`Sub 5`**, and then **3** is applied as the **y** parameter, resulting in an answer of **2**. This is not the case! If we want this behavior, we need to specify that **`5 |> Sub`** is to be evaluated first by parenthesizing the expression:

```
> let t = (5 |> Sub) 3;;

val t : int = 2
```

Here we clearly see that the partial application **`Sub 3`** takes priority, making 3 the **x** and 5 the **y**.

Putting it all together the right way

The previous code still misses what we want to accomplish:

- We want to be able to specify the encoder and hash algorithm to be used outside of the actual hashing function—in other words, we want to abstract out the actual encoder and hash algorithm that the application might want to use.
- We'd like to be able to cleanly append byte streams so we can work with passwords and salt.

Using everything we've learned about function composition, partial function application, and function pipelining, we're now ready to put together a much cleaner implementation, demonstrating (hopefully) the unique features of F# and functional programming in general.

Implementation

We start with the two functions that interface to the .NET API:

```
let encodeUTF8 password = Encoding.UTF8.GetBytes(password : string)
let getHash (hashAlgo : HashAlgorithm) (bytes : byte[]) =
hashAlgo.ComputeHash(bytes)
```

Next, we want to generalize the idea of appending byte arrays, so we'll write a general purpose "encode string and append" function:

```
let encodeStringAndAppendFunction (f : string -> byte []) = f >> Array.append
```

The reason I call this **`encodeStringAndAppendFunction`** is because, when given an encoding algorithm that converts a string to a byte[ ], it returns a partial function application—only the first parameter of the **`Array.append`** function has been supplied.

This allows us to chain together encode-and-append operations. We can do this in separate **`let`** statements, concluding with the final "append an empty array."

```
let a = "abc" |> encodeStringAndAppendFunction encodeUTF8
let b = ("salt" |> encodeStringAndAppendFunction encodeUTF8) >> a
```

```
let c = [||] |> b
```

Or we can put the entire expression inline:

```
let encoded = [||] |>
   ("abc" |> encodeStringAndAppendFunction encodeUTF8 <<
   ("salt" |> encodeStringAndAppendFunction encodeUTF8))
```

Important: Notice that I am using the right-to-left composition operator, **<<**. This is because I would like my salt be appended to my password, and therefore the salt must be the second parameter of the **`Array.append`** function.

For our purposes, we will create a function that simply encodes a password and salt:

```
let encodePwdAndSalt encoder pwd salt =
  ((encoder, pwd) ||> encodeStringAndAppendFunction <<
  ((encoder, salt) ||> encodeStringAndAppendFunction))
```

Here we use the double-pipe operator to provide the encoder along with the password, and then apply it again with the salt, creating a function composition that evaluates when we pipe in the empty array, which we'll see next.

What's nice about this function is that it provides a clear template for how to extend the function with additional strings that we might want to add to the byte[ ] before it is hashed. Later on, in the discussion on recursion, we'll generalize this even further.

We then create a function whose parameters have been carefully arranged such that it is suitable for partial function application:

- Because the encoder and hash algorithms probably won't vary for a set of passwords and salt, we want these to be the first parameters.
- Furthermore, the encoder and hash algorithm are always "together," so we can specify them as a tuple rather than in curried form (separated by white space).

```
let createHash (encoder, hasher) pwd salt =
    [||] |> encodePwdAndSalt encoder pwd salt |> getHash (hasher)
```

We can use this function directly (FSI console):

```
> createHash (encodeUTF8, (new SHA1Managed())) "abc" "salt";;
val it : byte [] =
  [|153uy; 25uy; 141uy; 252uy; 72uy; 224uy; 52uy; 198uy; 99uy; 86uy; 24uy;
    63uy; 133uy; 78uy; 179uy; 34uy; 246uy; 7uy; 193uy; 221uy|]
```

But because I've judiciously placed the encoder-hasher tuple first, I can create partial function applications with different encoding and hashing algorithms (FSI console):

```
// Create a UTF32 encoder.
let encodeUTF32 password = Encoding.UTF32.GetBytes(password : string)

// Here's my UTF8, SHA1 encoder-hasher.
let UTF8SHA1HashFunction = createHash (encodeUTF8, (new SHA1Managed()))

// Here's my UTF32, SHA256 encoder-hasher.
let UTF32SHA256HashFunction = createHash (encodeUTF32, (new SHA256Managed()))
;;

val encodeUTF32 : password:string -> byte []
val UTF8SHA1HashFunction : (string -> string -> byte [])
val UTF32SHA256HashFunction : (string -> string -> byte [])
```

We now have a very flexible system of hashing salted passwords with different encoding and hashing approaches, using partial application to create a re-usable hashing function.

```
// Now let's create a couple hashes using our different partial applications:
let hashedPassword1 = UTF8SHA1HashFunction "abc" "salt"
let hashedPassword2 = UTF32SHA256HashFunction "abc" "salt"
;;

// And here are the results:

val hashedPassword1 : byte [] =
  [|153uy; 25uy; 141uy; 252uy; 72uy; 224uy; 52uy; 198uy; 99uy; 86uy; 24uy;
    63uy; 133uy; 78uy; 179uy; 34uy; 246uy; 7uy; 193uy; 221uy|]

val hashedPassword2 : byte [] =
  [|130uy; 93uy; 100uy; 179uy; 200uy; 148uy; 12uy; 22uy; 159uy; 233uy; 81uy;
    22uy; 180uy; 27uy; 241uy; 140uy; 151uy; 56uy; 7uy; 210uy; 55uy; 254uy; 5uy;
    115uy; 8uy; 86uy; 11uy; 242uy; 12uy; 71uy; 131uy; 171uy|]
```

Lessons

- Wrap .NET functions so that you can use partial application constructs.
- Put the invariant parameters first in a function so that the function is suitable for partial application.
- Partial application is very powerful, allowing you to create function compositions that reveal useful patterns, giving you clues as to further abstraction for creating robust applications.
- Partial application is one of functional programming's "reuse" strategies. Code for this strategy.
- Not to be repetitive, but make your functions as small as possible!

Recursion is the new iteration

In imperative languages, we often (not as often anymore, but you still see it) write iteration like this (C#):

```
int sum = 0;

for (int i = 0; i < 10; i++)
{
       sum += i;
}

Console.WriteLine(sum);  // 45
```

With the advent of lambda expressions, we can write (C#):

```
int sum = 0;
Array.ForEach(Enumerable.Range(0, 10).ToArray(), n => sum += n);
Console.WriteLine(sum);
```

Or, if you wish to implement an extension method:

```
public static class EntensionMethods
{
       public static void ForEach<T>(this IEnumerable<T> source, Action<T> action)
       {
              foreach (T element in source) action(element);
       }
}

// ...

int sum = 0;
Enumerable.Range(0, 10).ForEach(n => sum += n);
Console.WriteLine(sum);
```

All of these C# examples require mutability—the value of sum is being incremented and is a side effect of the computation expression.

Recursion is used in F# to avoid mutability. Yes, we could write this example in F# as (FSI console):

```
> let summer =
    let mutable sum = 0
    for i in 1 .. 10 do
        sum <- sum + i
    sum
;;

val summer : int = 55
```

Notice we have to explicitly state that the sum is mutable.

Instead, the “functional programming” way of working with iteration is to use recursion (FSI console):

```
> let rec rec_summer n acc =
    match n with
    | 0 -> acc
    | _ -> rec_summer (n-1) (acc+n)

rec_summer 9 0;;

val rec_summer : n:int -> acc:int -> int
val it : int = 45
```

This is an example of tail recursion. “In computer science, a tail call is a subroutine call that happens inside another procedure as its final action; it may produce a return value which is then immediately returned by the calling procedure. The call site is then said to be in tail position, i.e. at the end of the calling procedure. If any call that a subroutine performs, such that it might eventually lead to this same subroutine being called again down the call chain, is in tail position, such a subroutine is said to be tail-recursive which is a special case of recursion. Tail calls need not be recursive—the call can be to another function—but tail recursion is particularly useful, and often easier to handle in implementations...Tail calls are significant because they can be implemented without adding a new stack frame to the call stack...[I]n functional programming languages, tail call elimination is often guaranteed by the language standard, and this guarantee allows using recursion, in particular tail recursion, in place of loops.”[35]

There are two things that make tail recursion difficult coming from imperative programming:

- How do you convert a loop into a tail recursion?
- How do you know you’ve implemented tail recursion correctly?

Converting loops to tail recursion

One could roughly put tail recursion into two categories:

- Those in which an “accumulator” needs to be threaded through the recursion.
- Those that operate on lists without needing an accumulator.

The previous example illustrates how to manually thread an accumulator through a recursion. The F# list classes have many functions that do this for you automatically. For example, one would normally have written (FSI console):

```
> let sumList list = List.fold (fun acc elem -> acc + elem) 0 list
sumList [1..9];;

val sumList : list:int list -> int
val it : int = 45
```

[35] Tail call: http://en.wikipedia.org/wiki/Tail_call

Or:

```
> let sumList2 list = List.reduce (fun acc elem -> acc + elem) list
sumList2 [1..9];;

val sumList2 : list:int list -> int
val it : int = 45
```

These examples take advantage of the **`List.fold`**[36] and **`List.reduce`**[37] functions. Realistically though, sometimes multiple accumulators are required, or the operation on each item in a list may require a manual tail recursion implementation for some reason.

The second kind of operation, operating on a list without needing an accumulator, often works with the "head" and "tail" of a list. For example, simply printing the numbers in a list, we use a match statement to test for an empty list; otherwise we use the syntax **`hd :: tl`** to extract the head of the list and the remainder of the list, the tail (FSI console):

```
> let rec printList list =
  match list with
  | [] -> ()
  | hd :: tl ->
    printfn "%i" hd
    printList tl

printList [1..3];;
1
2
3
```

As a side note, notice that since this function has a side effect (it merely prints numbers to the console), when the list is empty, we are returning unit, expressed by the "()".

Implementing tail recursion correctly

"'Tail Recursion' is a special recursive function which does not [include] any other execution after the recursive call, meaning there is no 'pending operation.'"[38]

If we use dotPeek to decompile the IL generated by the previous function, we see that it is implemented as an iteration with a **`while (true)`** loop:

```
internal class printList : FSharpFunc<FSharpList<int>, Unit>
  {
    internal printList()
    {
```

[36] List.fold: http://msdn.microsoft.com/en-us/library/ee353894.aspx
[37] List.reduce: http://msdn.microsoft.com/en-us/library/ee370239.aspx
[38] Tail Recursion: http://liangwu.wordpress.com/2010/07/17/the-basic-of-recursive-function-in-f/

```
    }

    public override Unit Invoke(FSharpList<int> list)
    {
      while (true)
      {
        if (fsharpList1.get_TailOrNull() != null)
        {
          // ...
        }
        else
          break;
      }
      return (Unit) null;
    }
  }
```

If instead we have a pending operation (here created by printing the head last):

```
let rec printListNoTail list =
      match list with
      | [] -> ()
      | hd :: tl ->
        printListNoTail tl
        printfn "%i" hd
```

Besides printing the list in reverse order, we note from the decompiled code that the compiler has implemented it as a recursive call:

```
internal class printListNoTail : FSharpFunc<FSharpList<int>, Unit>
  {
    internal printListNoTail()
    {
    }

    public override Unit Invoke(FSharpList<int> list)
    {
      // ...
      this.Invoke(tailOrNull);
      return // ... ;
    }
  }
```

This is not the desired implementation, as it is susceptible to stack overflows and will perform poorly because of all the stack pushes and unwinding that will occur.

In order to implement tail recursion, the recursive call must be the last operation in the code branch of the function or must return nothing (unit) or a value. Sometimes this can be difficult to implement, which is where the interesting subject of continuation passing style (CPS) comes up.

You can read more about CPS at http://codebetter.com/matthewpodwysocki/2008/08/13/recursing-on-recursion-continuation-passing/. We'll be looking at CPS a bit more in Chapter 3.

For and while: when to use iteration

The primary purpose of **`for`** and **`while`** statements is to iterate a fixed number of times independent of the data being processed, or to perform a process forever (or typically for the application's lifetime). F# provides three different looping constructs:

- for … to[39]
- for … in[40]
- while[41]

For example, either of these constructs makes much more sense:

```
for i = 1 to 10 do
    printfn "%d" i

for i in 1..10 do
    printfn "%d" i
```

When compared to a recursive implementation:

```
let rec print n limit =
  match n with
  | q when q > limit -> ()
  | q ->
    printfn "%d" q
    print (q+1) limit

print 1 10
```

With the recursive implementation it's much clearer what it's doing and when, but it becomes much harder to answer the question, "How many times does this loop iterate?" However, except for a fixed number iterations (data independent), one should consider carefully the advantages of using recursion over iteration.

Why use recursion?

Recursion has the advantage of being more mathematically expressive—it declares exactly what conditions result in a recursive call and it declares exactly what ends the recursive call. It also explicitly states what computation is performed during and at the conclusion of the recursion. For this reason, recursion can actually be easier to read than iteration, which may

[39] Loops: for…to Expression: http://msdn.microsoft.com/en-us/library/dd233236.aspx
[40] Loops: for…in Expression: http://msdn.microsoft.com/en-us/library/dd233227.aspx
[41] Loops: while…do Expression: http://msdn.microsoft.com/en-us/library/dd233208.aspx

have control logic such as a return or break embedded somewhere in the iteration, as well as multiple exit processes. Using recursion, these behaviors are almost always either placed outside of the recursive call or are made very explicit, usually by threading state through the recursive call.

Using recursion with imperative collections

One might think that another good use case for iteration is working with imperative collections. For example, this data reader seems perfectly fine to implement with an F# **`while…do`** construct:

```
open System
open System.Data
open System.Data.SqlClient

let openConnection name =
    let connection = new SqlConnection()
    let connectionString =
        "data source=localhost;initial catalog=" + name + ";integrated security=SSPI"
    connection.ConnectionString <- connectionString
    connection.Open()
    connection

let createReader (connection : SqlConnection) sql =
    let command = connection.CreateCommand()
    command.CommandText <- sql
    command.ExecuteReader()

let showDataIter (reader : SqlDataReader) =
    while reader.Read() do
        let id = Convert.ToInt32(reader.["BusinessEntityID"])
        let fname = reader.["FirstName"].ToString()
        let lname = reader.["LastName"].ToString()
        printfn "%d %s %s" id fname lname
    reader.Close()

let db = openConnection "AdventureWorks2008"
let sql =
  "select top 5 BusinessEntityID, FirstName, LastName from Person.Person order by FirstName"
createReader db sql |> showDataIter
```

However, it is just as easily implemented with recursion, which in the opinion of the author has a very nice explicitness to it, especially with regards to closing the reader, which the programmer may otherwise forget (and still might in the F# implementation, but I think it does make it more explicit):

```
// Replacing only showDataIter:

let showDataRec (reader : SqlDataReader) =
    let rec showData (reader : SqlDataReader) =
        match reader.Read() with
        | true ->
            let id = Convert.ToInt32(reader.["BusinessEntityID"])
            let fname = reader.["FirstName"].ToString()
            let lname = reader.["LastName"].ToString()
```

```
            printfn "%d %s %s" id fname lname
            showData reader
        | false -> reader.Close()
    showData reader

createReader db sql |> showDataRec
```

We'll use the database reader example next when we look at working with collections.

Lessons

- Learn the functions in the **`Collections.List`** module,[42] as these will greatly reduce the amount of recursion code you have to write yourself.
- When thinking about recursion, ask yourself, "Do I need to thread an accumulator through the process?" This affects the signature of the recursive function.
- Make sure the last thing your recursive function does is call itself—there should not be any pending operations. Otherwise the compiler will not implement the recursion as a loop and you will incur the overhead of function calls and stack usage and possibly cause a stack overflow.
- Tail recursion is so-called not because of the **`head :: tail`** syntax when working with lists, but because the recursive call is in the "tail position" of the function.[43]

Collections

In order to ensure that the programmer doesn't accidentally mutate a collection, F# provides a completely separate implementation of collections. In C#, we have the **`System.Collections`** and **`System.Collections.Generic`** namespaces. F# gives us the **`Microsoft.FSharp.Collections`**[44] namespace, which implements immutable **`List`**, **`Array`**, **`seq`**, **`Map`**, and **`Set`** collections.

Building collections

When building a list of items, the new item always appears first when using the "cons" (prepend) **`::`** operator. The reason is straightforward once you think about it. Given a list:

```
let list = [1; 2; 3]
let list2 = list :: 4           // Not a valid operation!
```

If we were to append an item to this list, we would be modifying the current list: the "next item" for the entry **`3`** would need to be modified to now link to item **`4`**.

Instead, we have to write:

---

[42] Collections.List Module: http://msdn.microsoft.com/en-us/library/ee353738.aspx
[43] Tail calls in F#: http://blogs.msdn.com/b/fsharpteam/archive/2011/07/08/tail-calls-in-fsharp.aspx
[44] F# Collection Types: http://msdn.microsoft.com/en-us/library/hh967652.aspx

```
let list = [1; 2; 3]
let list2 = 4 :: list
```

This ensures that **`list`** doesn't mutate. In order to append an item to a list, we have to use the concatenation operator (**`@`**) and put our new item into a second list:

```
let list = [1; 2; 3]
let list2 = list @ [4]
```

This can be a bit confusing to work with when coming from the mutable collection classes in the **`System.Collections.Generic`** namespace, where we can operate directly on the list in a willy-nilly fashion.

The real reason for recursion

Let's look at how we can go about creating a list of records using the query I created previously. Here's the record:

```
type Person =
    {
        PersonID: int;
        FirstName: string;
        LastName: string;
    }
```

We will first look at the iterative code in the previous section that reads the People data.

```
let createPersonListIter (reader : SqlDataReader) =
    let mutable list = []
    while reader.Read() do
        let id = Convert.ToInt32(reader.["BusinessEntityID"])
        let fname = reader.["FirstName"].ToString()
        let lname = reader.["LastName"].ToString()
        let person = {PersonID = id; FirstName = fname; LastName = lname}
        list <- person :: list
    reader.Close()
    list

let db = openConnection "AdventureWorks2008"
let sql = "select top 5 BusinessEntityID, FirstName, LastName from Person.Person
order by FirstName"
let people = createReader db sql |> createPersonListIter
```

Notice in the previous code that the list variable must be mutable when we use iteration. Iteration requires constantly pre-pending (we could have used concatenation also) the list with new items and updating the variable that holds the list, forcing us to use a mutable list variable. This is functional programming code smell.

The only way to solve this is with recursion where the list that we're building is threaded through the recursive calls:

```
let createPersonListRec (reader : SqlDataReader) =
    let list = []
    let rec getData (reader : SqlDataReader) list =
        match reader.Read() with
        | true ->
            let id = Convert.ToInt32(reader.["BusinessEntityID"])
            let fname = reader.["FirstName"].ToString()
            let lname = reader.["LastName"].ToString()
            let person = {PersonID = id; FirstName = fname; LastName = lname}
            getData reader (person :: list)
        | false ->
            reader.Close()
            list
    getData reader list

let db = openConnection "AdventureWorks2008"
let sql = "select top 5 BusinessEntityID, FirstName, LastName from Person.Person
order by FirstName"
let people = createReader db sql |> createPersonListRec
```

Here we see why recursion is so important when working in a functional programming language—it allows us to preserve immutable behavior by threading the list through the recursive function calls, such that each new call is actually a new list. It can take some time getting used to constructing collections in an immutable manner, and recursion plays a significant role in that process. I would strongly recommend that you take some simple iteration code samples from your favorite imperative language and practice rewriting them in an immutable, recursive manner.

Working with collections: map, reduce, and filter

Working with immutable collections requires a certain degree of mental gymnastics. Recursion is one of the fundamental tools in the programmer's toolkit for managing collections. The other toolkit items are the three fundamental operations on collections: **`map`**, **`reduce`** (equivalent to "fold" in some other languages), and **`filter`**.[45]

- **`Map`** applies a function to every item and returns a list of the results of each function call.
- **`Reduce`** applies two arguments cumulatively such that the result is a single value determined by the provided function.
- **`Filter`** returns a list qualified by a test function that returns true or false.

We perform these operations implicitly all the time in imperative programming. With functional programming, there are specific functions provided by the collection classes that implement the map, reduce, and filter behaviors. With the advent of C# 3.0, we also have these functions

[45] Map/Reduce/Filter in C#: http://www.25hoursaday.com/weblog/2008/06/16/FunctionalProgrammingInC30HowMapReduceFilterCanRockYourWorld.aspx

available to any collection that implements **`IEnumerable`** as **`Select`**, **`Aggregate`**, and **`Where`**, respectively, and you might have already used them in LINQ expressions, for example.

Regardless of your familiarity with them in imperative .NET languages, you should take the time to think about your list manipulations in terms of these three behavior patterns. You will frequently see them combined using pipelining to return complex computational results.

Using the data returned from this SQL query (I removed the “top 5”):

```
let sql = "select BusinessEntityID, FirstName, LastName from Person.Person order by
FirstName"
```

We can see how we can apply filtering, mapping, and several other list operations:

```
let selectNames = createReader db sql
                  |> createPersonListRec
                  |> List.filter (fun r -> r.LastName.StartsWith("Cl"))
                  |> List.map (fun r -> r.LastName + ", " + r.FirstName)
                  |> List.sort
                  |> Seq.distinct
                  |> List.ofSeq
```

Here we are pipelining several operations:

- Filtering only the last names that start with **Cl**.
- Mapping the record to a comma delimited last name, first name string.
- Sorting by the resulting string.
- Selecting only the unique names.
- Converting the sequence back to a list.

This is a nice illustration of the readability and conciseness of F#.

Lessons

- Recursion is a key tool when working with immutable collections.
- Learn the map, reduce, and filter functions provided by the F# collection classes.

Some further, minor points

Be expressive

The reader should not have to worry about “how” you are doing something as long you express “what” you are doing adequately. The problem with imperative code is that the “what” is often lost in the “how,” and functional programming can help to bring that to light.

Use **`Collection`** features such as **`iter`**—use lambda expressions!

Return something!

Even if you must write a function that results in a mutation or side effect and ideally you don't need to return anything, consider what you might want to return so that you can take advantage of function pipelining. Typically, one simply returns the same parameter that was passed into the function. However, since you don't know how the caller is going to use your function, program "functionally" so that the caller is not limited by your function's implementation.

Chapter 3 Going Deeper

We'll start this section with some easier concepts that have less equivalence in C# (at least without some considerable work) but deepen our understanding of F# and add to our "thinking like an F# programmer" toolbox. We'll begin with closures and discriminated unions, and then move on to active patterns and continuation passing style (CPS). We'll revisit recursion that uses CPS and then take on two of the most difficult topics for developers new to functional programming: monads and computation expressions.

Closures

Closure is a concept you should be familiar with already, as it plays an important role in lambda expressions in C#. "In programming languages, a closure…is a function …together with a referencing environment—a table storing a reference to each of the non-local variables (also called *free variables* or *upvalues*) of that function. A closure—unlike a plain function pointer—allows a function to access those non-local variables even when invoked outside its immediate lexical scope."[46]

For example, in C# we can demonstrate "print the value of the lexical scope" closure:

```
using System;

namespace Closures
{
	delegate void Func();

	class Program
	{
		static Func[] funcs = new Func[10];

		static void CreateFuncs()
		{
			for (int i = 0; i < funcs.Length; i++)
			{
				int j = i;   // Create a lexical scope.
				funcs[i] = () => Console.Write("{0} ", j);
			}
		}

		static void Main(string[] args)
		{
			CreateFuncs();

			for (int i = 0; i < funcs.Length; i++)
			{
```

[46] Closure: http://en.wikipedia.org/wiki/Closure_(computer_programming)

```
                        funcs[i]();
                    }
                }
            }
}
```

Resulting in the output:

```
0 1 2 3 4 5 6 7 8 9
```

We can write something similar in F#:

```
let rec createFuncs i arrayOfFuncs =
    match i with
    | 10 -> arrayOfFuncs
    | n -> createFuncs (n+1) ((fun () -> printfn "%d" n) :: arrayOfFuncs)

let funcs = List.rev (createFuncs 0 [])

List.iter (fun f -> f()) funcs;;
```

The previous code results in the same output (each number on a separate line). Closures are relied upon frequently in functional programming because one is often creating partial function applications that pass in values (or functions) that are part of the current lexical scope. One should become very comfortable with closures in order to be proficient at functional programming.

Discriminated unions…

"Discriminated unions provide support for values that can be one of a number of named cases, possibly each with different values and types. Discriminated unions are useful for heterogeneous data; data that can have special cases, including valid and error cases; data that varies in type from one instance to another; and as an alternative for small object hierarchies. In addition, recursive discriminated unions are used to represent tree data structures."[47]

You can think of discriminated unions as providing additional semantic meaning to a type, by reading a discriminated union as something like "A type that can be an **x**, a **y**, or a **z** (etc.) where **x**, **y**, and **z** are of type **a**, **b**, and **c**, respectively." This does not hold true in all cases, as we will see later.

[47] Discriminated Unions, Visual Studio 2012: http://msdn.microsoft.com/en-us/library/dd233226%28v=vs.110%29.aspx

…Instead of object hierarchies

Discriminated unions have no direct equivalent in imperative, object-oriented languages. The closest one can get is to implement an object hierarchy. Take for example two shapes: a circle and a rectangle. In an object-oriented paradigm, we would typically write an object model:

```
public class Shape
{
  public abstract double Area { get; }
}

public class Circle : Shape
{
  public double Radius { get; set; }
  public override double Area { get { return 3.14 * Radius * Radius; } }
}

public class Rectangle : Shape
{
  public double Height { get; set; }
  public double Width { get; set; }
  public override double Area { get { return Width * Height; } }
}
```

In F#, we can implement a discriminated union:

```
type Shape =
    | Circle of double
    | Rectangle of double * double
```

Note that in Visual Studio 2013, we can assign labels to the types: [48]

```
type Shape =
    | Circle of radius : double
    | Rectangle of width : double * height : double
```

For our purposes, we'll stay with the implementation of discriminated unions supported in Visual Studio 2012.

Note that the closest construct in C# would be a class, as illustrated earlier.

We can then calculate the area of our shapes using a function and pattern matching on the type:

```
let area shape =
    match shape with
```

[48] Discriminated Unions, Visual Studio 2013: http://msdn.microsoft.com/en-us/library/dd233226.aspx

```
    | Circle r -> 3.14 * r * r
    | Rectangle (w, h) -> w * h
```

Since this function only does matching on a single parameter with lambda expressions for each case, we use this shorthand syntax:

```
let area = function
    | Circle r -> 3.14 * r * r
    | Rectangle (w, h) -> w * h
```

We can then compute the areas of our two shapes (F# console):

```
> area (Circle(1.0));;
val it : float = 3.14
> area (Rectangle(2.0, 4.0));;
val it : float = 8.0
```

Comparing this to the C# code earlier, you could say that we have “de-virtualized” the class hierarchy and explicitly implemented the equivalent of a virtual table.

…Can act like enumerations

A discriminated union does not need to specify the “type of” for the case-identifier. We can construct something similar to an “enum” in C# using (F# console):

```
> type Vehicle =
    | Truck
    | Car

let vehicles = [Truck, Car];;

type Vehicle =
  | Truck
  | Car
val vehicles : (Vehicle * Vehicle) list = [(Truck, Car)]
```

The salient difference with C#’s enumerations is that the case-identifiers (here “Truck” and “Car”) are not assigned values—the identifier is just that, an identifier.

…Can be self-referencing

```
type Assembly =
  | Subassembly of string * Assembly
  | Part of string
```

Here I am defining an assembly that can be either a sub-assembly that has a name and further sub-assemblies, or the assembly can be the actual part, having a name.

…Are very useful when working with trees

One finds discriminated unions frequently in describing a node in a tree. For example, the following code defines a binary tree type as being either **`Empty`** (with no type) or having a **`Node`** with a three-element tuple type, being:

- another tree on the left of the same type,
- a value for the node, and
- another tree of the same type on the right.

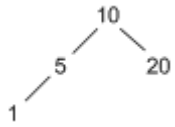
```
type 'a BinaryTree =
  | Empty
  | Node of 'a BinaryTree * 'a * 'a BinaryTree
```

This is one way of defining a binary tree. Note that the case-identifier **`Empty`** does not specify any "of-type" information—it merely serves as a placeholder identifier, allowing us to construct trees like this (F# console):

```
> let tree = Node( Node( Node(Empty, 1, Empty), 5, Empty), 10, Node(Empty, 20, Empty));;

val tree : Tree<int> =
  Node (Node (Node (Empty,1,Empty),5,Empty),10,Node (Empty,20,Empty))
```

With this example we have constructed a small tree that looks like this:



Expression trees are another excellent example of where discriminated unions can be very effective. This example, borrowed from MSDN's description of discriminated unions,[49] illustrates the concept:

```
type Expression =
    | Number of int
    | Add of Expression * Expression
    | Subtract of Expression * Expression
    | Multiply of Expression * Expression
    | Divide of Expression * Expression
    | Variable of string
```

Lessons

- You can use discriminated unions in places where you would otherwise have used object hierarchies.

[49] Discriminated Unions: http://msdn.microsoft.com/en-us/library/dd233226%28v=vs.110%29.aspx

- Discriminated unions are a useful construct for indicating that the “type-name” consists of specific identifiers, each of which can be of a different type.
- Discriminated unions might seem like **`typeof(…)`** in C#, but they’re really not. They’re more like a lookup—given an identifier name, this is its (optional) type.
- Use discriminated unions to attach semantic meaning to a type. The additional semantic meaning is used by the compiler to validate your computations.

Active patterns

Pattern matching is a common technique in functional programming languages. A match expression looks a little like a case statement in an imperative language, but there are significant differences. First, it’s an expression, not a statement. Second, in its most distinctive form, each case represents an alternative way a value of the type of the value being matched on could have been constructed, which provides a powerful generic way to take things of a given type apart. In some languages, the compiler will tell you whether the cases are exhaustive for the type or not. There are various extensions to the basic concept. A noteworthy one specific to F# is active patterns.

“Active patterns enable you to define named partitions that subdivide input data, so that you can use these names in a pattern-matching expression just as you would for a discriminated union. You can use active patterns to decompose data in a customized manner for each partition.”[50]

Similar to a discriminated union in which we create identifiers that may or may not be associated with types, with active patterns we can name “partitions” of a pattern in an *active recognizer*. For example:

```
let (|Prefix|_|) (p:string) (s:string) =
    if s.StartsWith(p) then
        Some(s.Substring(p.Length))
    else
        None
```

This active recognizer will allow us to match the identifier **`Prefix`** when the prefix **`p`** is found at the start of the string “s”. Incidentally this code is a “partial active pattern” because we don’t care what the rest (the “_”) of the string is.

We use the active recognizer in a **`match`** statement:

```
let activePatternTest s =
    match s with
     | Prefix "F#" rest -> printfn "Started with 'F#', rest is '%s'" rest
     | Prefix "Is" rest -> printfn "Started with 'Is', rest is '%s'" rest
     | _ -> printfn "Don't know what you're talking about."
```

[50] Active Patterns: http://msdn.microsoft.com/en-us/library/dd233248.aspx

This allows us to test for a particular pattern as determined by the active recognizer function. So, for example, this is the output of three pattern-matching tests (F# console):

```
> activePatternTest "F# is great!"
activePatternTest "Is F# great?"
activePatternTest "Yes it is!";;

Started with 'F#', rest is ' is great!'
Started with 'Is', rest is ' F# great?'
Don't know what you're talking about.
```

You can use active patterns to decompose a value into different representations. For example, let's say you want to decompose the result of a division into either the result itself or a quotient and its remainder. We can write two active patterns (please note that for the following examples, I am not dealing with division by zero):

```
let (|DivisionResult|) (numerator, divisor) = numerator / divisor
let (|QuotientRemainder|) (numerator, divisor) =
                      (numerator / divisor, numerator % divisor)
```

Now let's say we have two functions, and among other things, they will print the result either as a floating point result or as a quotient and divisor:

```
let printResult numerator divisor =
    match (numerator, divisor) with
    | DivisionResult(r) -> printfn "Result = %f" r

let printQuotientDivisor numerator divisor =
    match (numerator, divisor) with
    | QuotientRemainder(q, d) -> printfn "Quotient = %d, Divisor = %d" q d
```

We can then call these two functions and obtain different results based on how we intend to match the input with the active pattern (F# console):

```
> printResult 1.0 3.0;;
Result = 0.333333
val it : unit = ()

> printQuotientDivisor 1 3;;
Quotient = 0, Divisor = 1
val it : unit = ()
```

These two examples are certainly trivial. The real power of active patterns comes about when combining several options in the match expression using partial active patterns. For example, let's say that we do not want a match in the floating point division if the fractional part (everything to the right of the decimal point) is more than two digits, so we use the quotient-remainder format instead. First, let's write a small helper function to determine the length of the fractional part:

```
let moreThanTwoDigits n =
    let strN = n.ToString()
    let idx = strN.IndexOf('.')
    if idx > 0 then  // Is there a decimal point?
        let remainder = strN.Substring(idx)
        remainder.Length > 3 // Including the '.', returning true or false.
    else
        false  // No decimal point.
```

Next, we write a partial active pattern ending in the wildcard (_). This tells us it's a partial pattern and that we will be returning **None** or **Some**:

```
let (|DivisionResult|_|) (numerator: int, divisor: int) : float option =
    let r = float(numerator) / float(divisor)
    if moreThanTwoDigits r then
        None
    else
        Some r
```

(The casting is used to force the function to accept only integers and to return an optional float.)

The quotient-remainder pattern stays the same (and it also is *not* a partial pattern):

```
let (|QuotientRemainder|) (numerator, divisor) =
                      (numerator / divisor, numerator % divisor)
```

Finally, we can write a function that prints the result based on the match with the active pattern:

```
let printResult numerator divisor =
    match (numerator, divisor) with
    | DivisionResult(r) -> printfn "Result = %f" r
    | QuotientRemainder(q, d) -> printfn "Quotient = %d, Divisor = %d" q d
```

Now let's see the result (F# console):

```
> printResult 2 4;;
Result = 0.500000
val it : unit = ()

> printResult 10 3;;
Quotient = 3, Divisor = 1
val it : unit = ()
```

Here we can see that for 10/3, we fail to match the active pattern **DivisionResult**, but we do match the active pattern **QuotientRemainder**.

Lessons

- As with other concepts that we've already looked at (and will see again soon) the idea is to "invert" the behavior of the function.
- Instead of a complicated **`if…then…else`** statement with all the logic built into it, we have created reusable active patterns that can be applied in specific contexts.
- Again, there's nothing here that can't be done in an imperative language like C#; it's simply that functional programming promotes more of a re-use style to programming (if you use it correctly).

Continuation passing style (CPS)

"[A] continuation represents the remainder of a computation given a point in the computation."[51] Continuations are usually not encountered in imperative programming, but they have definite uses in a functional programming environment, which we'll discuss further later on. Basically, the idea is that a function is provided with another function that is invoked with the result of the first function's computation. So for example, rather than writing the database reader with a pipeline to create the Person records:

```
let selectNames = createReader db sql |> createPersonListRec
```

I could change the implementation of **`createReader`** to accept a continuation function and make a call using CPS:

```
let selectNames = createReaderCps db sql createPersonListRec
```

Notice the pipeline (**`|>`**) token has disappeared and we're providing the function to which we want to pass the result. The **`createReaderCps`** function is then defined as:

```
open System
open System.Data
open System.Data.SqlClient
let createReaderCps (connection : SqlConnection) sql f =
    let command = connection.CreateCommand()
    command.CommandText <- sql
    let result = command.ExecuteReader()
    f result
```

Notice how the last action of the **`createReaderCps`** function is to invoke the provided function **`f`** with the result of the **`ExecuteReader()`** call.

[51] Continuation Passing: http://codebetter.com/matthewpodwysocki/2008/08/13/recursing-on-recursion-continuation-passing/

Continuations as inversion of control

We're used to writing imperative code such that when an invalid condition is discovered, the method handles it in a particular way: throwing an exception, issuing a message, returning null, etc. The "fail" handling is intrinsic to the method. For example, in F# I can create a map of PersonIDs and the records associated with them:

```
let nameMap = createReader db sql
                    |> createPersonListRec
                    |> Seq.map (fun r -> r.PersonID, r)
                    |> Map.ofSeq
```

I could access the record using **`Map.tryFind`** (F# console):

```
> Map.tryFind 1 nameMap;;
val it : Person option = Some {PersonID = 1;
                               FirstName = "Ken";
                               LastName = "Sánchez";}

> Map.tryFind 1234 nameMap;;
val it : Person option = None
```

However, let's use continuations to specify what we want to do once we find the record. For this, we'll create a function that takes two functions (the success and fail) as well as the key and map:

```
let findRecord success fail key map =
    let record = Map.tryFind key map
    match record with
    | None -> fail key
    | _ -> success record
```

Now we can provide the behavior (inversion of control) for the success and failure states. A very trivial example is printing the record value or "not found":

```
> findRecord (fun r -> printfn "%A, %A" r.Value.LastName r.Value.FirstName)
           (fun r -> printfn "not found")
           1
           nameMap;;

"Sánchez", "Ken"
val it : unit = ()

> findRecord (fun r -> printfn "%A, %A" r.Value.LastName r.Value.FirstName)
           (fun r -> printfn "not found")
           1234
           nameMap;;
```

```
not found
val it : unit = ()
```

(In this code I'm de-optioning the result by using the **`Value`** property of the option, which is generally not recommended, but in this case, because I know the function is only being called with a matching record, I know that it is safe to implement in this way.)

A more sophisticated example: caching

Let's say that our records are cached and that if the key isn't found in the cache, we want to hit the database to return the record and add it to our cache. Since caches are inherently mutable, we're going to use the **`System.Collections.Generic`** namespace to instantiate a simple cache that only handles a single table:

```
open System.Collections.Generic
let cache = new Dictionary<int, Person>()
```

Note: Because we are using a mutable structure, it is not thread-safe! The cache dictionary should be locked before updating and retrieving items.

Next, we create a function that updates the cache given a **`Person`** record (which we defined in Chapter 2). Note that this function returns the person record as well, which makes it easier to use with pipelining and binding the record to a variable:

```
let updateCache p =
    cache.[p.PersonID] <- p
    p
```

I'm going to create a helper function for constructing a **`Person`** record from the **`SqlDataReader`**; this is intended to be used when we know we have only one record to read:

```
let createPersonRecord (reader : SqlDataReader) =
    let id = Convert.ToInt32(reader.["BusinessEntityID"])
    let fname = reader.["FirstName"].ToString()
    let lname = reader.["LastName"].ToString()
    {PersonID = id; FirstName = fname; LastName = lname}
```

Next, we write a function that will retrieve a single record from the database or throw an exception. We will use this function when the "get record from the cache" fails. As this is an example only, it uses a bit of brute force:

```
let getPersonFromDatabase id =
    printfn "Querying database"
```

```
    let sql = "select BusinessEntityID, FirstName, LastName from Person.Person
where BusinessEntityID = " + id.ToString()
    let reader = createReader db sql
    match reader.Read() with
    | false -> failwith "record not found"
    | true ->
        let ret = reader |> createPersonRecord |> updateCache
        reader.Close()
        ret
```

Now we can write our continuation function. Note how we require the function **`onFail`** as the first parameter, which, if the record isn't in the cache, we call with the desired ID. This puts it on the caller to provide the implementation for what to do when the record isn't in the cache, thus turning the implementation inside-out—inversion of control.

Also note the call to **`TryGetValue`**. In F#, "out" parameters are returned, along with the method call result, in the form of a tuple. This is a very nice convenience because we can get both the success state of the **`TryGetValue`** as well as the value if successful, without resorting to reference types[52]—or worse, a mutable type to receive the out value.

```
let getRecord onFail id =
    let found, p = cache.TryGetValue(id)
    match found with
    | false -> onFail id
    | true ->
        printfn "From cache"
        p
```

We can now take advantage of partial application by creating two partial functions: one that handles getting data from the database in a "connected" environment, and one that simply throws an exception in a "disconnected" environment:

```
let getRecordConnected = getRecord getPersonFromDatabase
let getRecordDisconnected = getRecord (fun _ -> failwith "not in cache")
```

Let's give it a test drive (F# console):

```
> getRecordConnected 21;;
Querying database
val it : Person = {PersonID = 21;
                   FirstName = "Terry";
                   LastName = "Eminhizer";}
> getRecordConnected 21;;
From cache
val it : Person = {PersonID = 21;
                   FirstName = "Terry";
```

[52] Reference Types: http://msdn.microsoft.com/en-us/library/dd233186.aspx

```
                    LastName = "Eminhizer";}
> getRecordDisconnected 21;;
From cache
val it : Person = {PersonID = 21;
                    FirstName = "Terry";
                    LastName = "Eminhizer";}
> getRecordDisconnected 22;;
System.Exception: not in cache
```

Note that the first call to **`getRecordConnected`** queries the database for **`ID 21`**, and the second call gets the record from the cache. Note also how, in the disconnected state, we can only get records from the cache—otherwise an exception is thrown.

Lessons

- Continuation passing style is a powerful technique for allowing the caller to define how the function continues processing, rather than embedding the processing in the function itself.
- Rethink object hierarchies, where functions are overridden because different behavior is required, using CPS instead.
- Look at your functions and consider what parts could be extracted using CPS to make your program more adaptable to the developer's needs.
- Combine CPS with partial function application to create re-usable behaviors.

CPS and recursion

Recall that for the compiler to translate recursion to iteration, the recursive call must be the last call made in the function. There are times when this is difficult to achieve, especially when working with two or more different "branches" in a recursive process, such as parsing a binary tree.[53] Using our **`Tree`** discriminated union described earlier, we can see CPS is a necessary device to employ when, for example, tail-recursively determining the size of a tree. Let's look at a straightforward recursive implementation that counts non-empty nodes:

```
let getTreeSize tree =
    let rec treeSize tree acc =
        match tree with
        | Empty -> acc
        | Node (left, _, right) -> treeSize left (acc+1) + treeSize right 0

    treeSize tree 0
```

The second half **`treeSize right 0`** is a bit non-intuitive—we increment the accumulator because we have a node. It doesn't matter whether we do this when recursively calling to process a left branch or a right branch; we just don't do it for both left *and* right recursive calls—otherwise we will get double the count.

[53] Binary tree: http://en.wikipedia.org/wiki/Binary_tree

When we call it using our small tree defined earlier, we see that we get four nodes (F# console):

```
let tree = Node(Node(Node(Empty, 1, Empty), 5, Empty), 10, Node(Empty, 20, Empty))
getTreeSize tree;;
val getTreeSize : tree:Tree<'a> -> int
val tree : Tree<int> =
  Node (Node (Node (Empty,1,Empty),5,Empty),10,Node (Empty,20,Empty))
val it : int = 4
```

The problem is, this call is not tail-recursive, so we run the risk of a stack overflow.

To make this tail-recursive, we have to use CPS to pass in the continuation operation:

```
let getTreeSizeRec tree =
    let rec treeSize tree f =
        match tree with
        | Empty -> f 0
        | Node (left, x, right) ->
            treeSize left (fun lacc ->
                treeSize right (fun racc ->
                    f (lacc + racc + 1)
                )
            )
    treeSize tree (fun x -> x)
```

Here we are using double-recursion with left and right accumulators in a CPS to create a tail-recursive function. This is not easy to wrap one's head around, so let's put in some trace statements and see how this is working:

```
let getTreeSizeRec tree =
    let rec treeSize depth caller tree f =
        printfn "Depth: %d   Caller: %s" depth caller
        match tree with
        | Empty ->
            printfn "Empty"
            f 0
        | Node (left, x, right) ->
            printfn "Node: %d" x
            treeSize (depth+1) "left" left (fun lacc ->
                printfn "lacc: %d, x: %d" lacc x
                treeSize (depth+1) "right" right (fun racc ->
                    printfn "lacc: %d, x: %d, racc: %d, returning = %d" lacc x
racc (lacc+racc+1)
                    f (lacc + racc + 1)
                )
            )
    treeSize 0 "root" tree (fun x -> x)
```

Here is the output (F# console):

```
> let tree = Node(Node(Node(Empty, 1, Empty), 5, Empty), 10, Node(Empty, 20, Empty))
let treeSize = getTreeSizeRec tree
;;
Depth: 0   Caller: root
Node: 10
Depth: 1   Caller: left
Node: 5
Depth: 2   Caller: left
Node: 1
Depth: 3   Caller: left
Empty
lacc: 0, x: 1
Depth: 3   Caller: right
Empty
lacc: 0, x: 1, racc: 0, returning = 1
lacc: 1, x: 5
Depth: 2   Caller: right
Empty
lacc: 1, x: 5, racc: 0, returning = 2
lacc: 2, x: 10
Depth: 1   Caller: right
Node: 20
Depth: 2   Caller: left
Empty
lacc: 0, x: 20
Depth: 2   Caller: right
Empty
lacc: 0, x: 20, racc: 0, returning = 1
lacc: 2, x: 10, racc: 1, returning = 4

val tree : int Tree =
  Node (Node (Node (Empty,1,Empty),5,Empty),10,Node (Empty,20,Empty))
val treeSize : int = 4
```

We now have a better understanding of how this function works. Refer to Appendix A for a complete “stack” trace based on the previous output. The “magic” of this code happens in-line with that call to **f** with the sum of the left accumulator + the right accumulator + 1.

```
treeSize left (fun lacc -> treeSize right (fun racc -> f (lacc + racc + 1)
```

A very simple example will hopefully suffice to illustrate this: when the function is called with a single node (**Empty, 10, Empty**):

- The left side is **Empty**, so the function passed in is evaluated with the parameter **0**, resulting in the inner function being called:

```
treeSize right (fun racc -> f (0 + racc + 1)
```

- The right side is **`Empty`**, so the function passed in is evaluated with the parameter **`0`**, resulting in the inner function being called:

```
treeSize right (fun racc -> f (0 + racc + 1)
```

- This "seed" function is the identity function **`x -> x`**; thus **`0 + 0 + 1`** evaluates to **`1`**. A tree with a single node has, well, one node.

With a more complicated tree, **`f`** is the function passed into each recursive call, thus each recursion "nests" the computation for the argument of **`fun lacc`** or **`fun racc`**. Once the function can evaluate, the value becomes the input for the previously nested function. Again, the trace in Appendix A should make this clearer, and I have illustrated the trace using a pseudo-stack for the function—once the innermost function evaluates, the stack is popped and the value is supplied as the parameter to the previous function on the stack.

A generalized tree recursion fold function

One question to ask when writing in a functional programming style is: "Can this computation be passed-in rather than hardwired in the function itself?" Earlier, in discussing higher-order functions, we showed how to write a function that takes a function as a parameter. This is a very powerful technique, and something similar can be done with continuations. For example, the code that determines the size of the tree can be refactored such that the "continuation," in this case **`lacc + racc + 1`**, is passed in by the caller (I've renamed the function to the more generalized **`foldTree`**):

```
let foldTree computation tree =
    let rec folding tree f =
        match tree with
        | Empty -> f 0
        | Node (left, x, right) ->
            folding left (fun lacc ->
                folding right (fun racc ->
                    f (computation lacc x racc)
                )
            )
    folding tree (fun x -> x)
```

This results in a generalized tree **`fold`** operation. The previous computation, counting leaves, can now be passed in as part of the call:

```
let treeSize = foldTree (fun lacc _ racc -> lacc + racc + 1) tree
```

We can pass in other computations, for example, adding all the values of the tree, given that **`a`** is of type **`integer`**:

```
let nodeSum = foldTree (fun lacc x racc -> lacc + x + racc) tree
```

with the result (F# console):

```
> let nodeSum = foldTree tree (fun lacc x racc -> lacc + x + racc);;

val nodeSum : int = 36
```

The “magic” here (besides being an excellent example of higher-order functions) is in the meaning of “accumulator”—in the first case, it’s an incrementing count, and in the second case, it’s a sum.

Lastly, we can employ a partial function application to create re-usable functions:

```
let getTreeSize<'a> = foldTree (fun lacc (x : 'a) racc -> lacc + racc + 1)
let getSumOfNodes = foldTree (fun lacc x racc -> lacc + x + racc)
```

And now we can call our partial applications (F# console):

```
> getTreeSize<int> tree;;
val it : int = 4
> getSumOfNodes tree;;
val it : int = 36
```

Notice in this code example that we have to specify the type for the **`getTreeSize`** function, as the type cannot be inferred since it isn’t used in the computation!

The identity function

This is the identity function:

```
 (fun x -> x)
```

It is an extremely useful construct when “seeding” the call to a recursive function that requires an initial function. This construct is so useful that it is actually an operator:[54]

```
id
```

You can use **`id`** anywhere you would otherwise explicitly write **`(fun x -> x)`**.

Lessons

- CPS is a necessary tool to ensure tail recursion when working with “multi-path” recursion, such as trees.
- Learn to “invert” your thinking about functions—look at what’s inside the function and see if you want to expose it on the outside.

[54] Operators.id<’T’> Function: http://msdn.microsoft.com/en-us/library/ee353607.aspx

- Write your functions with partial application in mind: what are the “re-use” parameters and the “highly varying” parameters? The “re-use” parameters should go first so that partial function applications can be easily defined, therefore promoting function re-use.
- For further fun (pun intended) take a look at “Catamorphisms, part two.”[55]
- Familiarize yourself with the basic F# operators.[56] This can save you a lot of time when trying to figure out someone else’s code.

Computation expressions

The time has come to deal with the “m” word—monads. “Computation expressions in F# provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings. They can be used to provide a convenient syntax for monads, a functional programming feature that can be used to manage data, control, and side effects in functional programs.”[57]

But what is a monad?

“In functional programming, a monad is a structure that represents computations defined as sequences of steps. A type with a monad structure defines what it means to chain operations, or nest functions of that type together. This allows the programmer to build pipelines that process data in steps, in which each action is decorated with additional processing rules provided by the monad. As such, monads have been described as "programmable semicolons"; a semicolon is the operator used to chain together individual statements in many imperative programming languages, thus the expression implies that extra code will be executed between the statements in the pipeline. Monads have also been explained with a physical metaphor as assembly lines, where a conveyor belt transports data between functional units that transform it one step at a time.”[58]

One of the uses for a monad is to retain state without resorting to mutable variables. In this section, we will attempt to wrap our heads around computation expressions and monads as this ends up being a real mind-bender, especially coming from an imperative programming lifestyle. Also, monads have important uses with regard to functional programming and really have no equivalent (except loosely in terms of aspect-oriented programming) in the imperative world.

Continuations, again

Continuations are a key aspect of computation expressions. First off, when we write the following:

```
let a = 5
let b = 6
let c = a + b
```

[55] Catamorphisms, part two: http://lorgonblog.wordpress.com/2008/04/06/catamorphisms-part-two/
[56] Basic F# Operators: http://msdn.microsoft.com/en-us/library/ee353754.aspx
[57] Computation Expressions: http://msdn.microsoft.com/en-us/library/dd233182.aspx
[58] Monads: http://en.wikipedia.org/wiki/Monad_(functional_programming)

We need to realize that this is already a shorthand for the verbose **in** syntax:[59]

```
let a = 5 in
  let b = 6 in
    let c = a + b
    c |> ignore    // Expression required to finish the let block.
```

If we now pipe the value into a function of the same name as the identifier, we can rewrite the previous code using CPS:

```
let n = 5 |> (fun a ->
          6 |> (fun b ->
            a + b |> (fun c -> c)))
```

But now, let's make this more interesting by actually writing a function for the pipeline operator. The pipeline operator is defined as **`|> x f = f x`**. In other words, **f** is called with the parameter **x**. We can write our own pipeline operator as a function easily enough:

```
let pipeFnc x f =
    f x
```

Now, our three **let** statements can be written as:

```
let n = pipeFnc 5 (fun a ->
        pipeFnc 6 (fun b ->
        pipeFnc (a+b) (fun c -> c)))
```

But wait! We can now introduce additional computations in **pipeFnc**! For example, we can log the values of **x**:

```
let pipeFnc x f =
    printfn "x = %d" x
    f x
```

When we now call our converted **let** statements, we get (F# console):

```
let n = pipeFnc 5 (fun a ->
        pipeFnc 6 (fun b ->
        pipeFnc (a+b) (fun c -> c)));;
x = 5
x = 6
x = 11
```

[59] Verbose Syntax: http://msdn.microsoft.com/en-us/library/dd233199.aspx

```
val n : int = 11
```

We have successfully added additional code that does something behind the scenes and represents a very simple example of a computation expression workflow.

A Logger monad

If instead we move these operations into a class and use the F# computation expression syntactical sugar names, we would start with a class type that defines methods that control how the expression executes. This class type is known as a *builder type* and is usually given a *builder name*:

```
// Define the builder type, which is a class type.
type LoggerBuilder() =
    // Define the custom operation.
    let log x = printfn "x = %d" x

    // Called for let! (as well as do!)
    member b.Bind(x, f) =
        log x
        f x

    // Called for "return".
    member b.Return(x) =
        x
```

Using a de-sugared syntax so that the use of our **`LoggerBuilder`** class looks similar to how we were using the **`pipeFnc`** before, we can write:

```
let logger = new LoggerBuilder()
let n = logger.Bind(5, fun a ->
        logger.Bind(6, fun b ->
        logger.Bind(a + b, fun c -> logger.Return(c))))
```

The usage is equivalent to our CPS version, except that we are using member functions of a class and results in the same output, and the result exactly matches our CPS version.

Now let's look at the sugared syntax:

```
// Construct the logger.
let logger = new LoggerBuilder()
let loggedWorkflow =
    logger      // Using the LoggerBuilder...
        {
        let! a = 5          // Bind and compute expression.
        let! b = 6          // Bind and compute expression.
        let! c = a + b      // Bind and compute expression.
        return c            // Return c.
```

```
        }
```

The result is the same (F# console):

```
n = 5
n = 6
n = 11

val loggedWorkflow : int = 11
```

The Maybe monad (aka “May Be”)

Let’s try our hand at something called the *Maybe monad*—a computation expression that stops its computation when an invalid “state” is encountered. A simple example of this is doing a series of division operations where the denominator “may be” 0.

The “builder” class is simple enough (in all examples you will find of this computation expression, you will see it called **`MaybeBuilder`**, but I prefer the more semantically correct **`MayBeBuilder`**, so I will use that in the following examples):

```
// Again we define a class...
type MayBeBuilder() =
    // with the member Bind so our sugared "let!" works.
    member b.Bind(x, f) =
        match x with
        | None -> None      // The value “may be” invalid, in which case we stop.
        | Some n -> f n     // Continue with the workflow if state remains valid

    // and with the member Return so our sugared "return" works.
    member b.Return(x) = Some x
```

We will also need a **`divideBy`** function that returns “None” if dividing by zero, or “Some *n*” to represent the result:

```
let divideBy numerator divisor =
    match divisor with
    | 0 -> None
    | n -> Some (numerator / n)
```

Now we can execute a workflow. For example (F# console):

```
> let mayBeDivided =
    MayBeBuilder
        {
        let! a = (20 |> divideBy) 2
        let! b = (a |> divideBy) 5
        return b
```

```
        };;

val mayBeDivided : int option = Some 2
```

Or, if we try dividing by 0 (F# console):

```
> let mayBeDivided =
    MayBeBuilder
        {
        let! a = (20 |> divideBy) 0
        let! b = (a |> divideBy) 5
        return b
        };;

val mayBeDivided : int option = None
```

Again, this is just syntactical sugar for a continuation passing style of writing a workflow. One interesting thing about the **Maybe** monad that is not to be overlooked is that once the computation is deemed to be invalid (it returns "None"), the workflow stops.

A State monad

A state monad does not solve the overall issue of maintaining state throughout the lifetime of the application. For this, the mutable dictionary from the **System.Collections.Generic** namespace is one solution. Rather, a state monad maintains state within a specific *workflow.* I will take a simple example of generating pseudo-random numbers[60] (manually!), using a "multiply with carry" method. A mutable implementation in F# would look like this:

```
let mutable m_w = 1
let mutable m_z = 2

let getRandom a b =
    m_z <- (36969 * (a &&& 65535) + (a >>> 16))
    m_w <- (18000 * (b &&& 65535) + (b >>> 16))
    uint32((m_z <<< 16) + m_w)
```

Resulting in (F# console):

```
> getRandom 1 2;;
val it : uint32 = 2422836384u

> getRandom m_z m_w;;
val it : uint32 = 1907405312u

> getRandom m_z m_w;;
```

[60] Random number generators:
http://en.wikipedia.org/wiki/Random_number_generation#Computational_methods

```
val it : uint32 = 3696412319u
```

When we convert this to an immutable form, we need to always return the new state along with the new random number:

```
let getRandomImmutable state =
    let m_z = (36969 * (fst state &&& 65535) + (fst state >>> 16))
    let m_w = (18000 * (snd state &&& 65535) + (snd state >>> 16))
    let newRand = uint32((m_z <<< 16) + m_w)
    printfn "%d" newRand
    newRand, (m_z, m_w)
```

Resulting in a tuple of tuples—(random number, (state)):

```
> let ri1 = getRandomImmutable (1, 2);;

val ri1 : uint32 * (int * int) = (2422836384u, (36969, 36000))

> let ri2 = getRandomImmutable (snd ri1);;

val ri2 : uint32 * (int * int) = (1907405312u, (1366706961, 648000000))

> let ri3 = getRandomImmutable (snd ri2);;

val ri3 : uint32 * (int * int) = (3696412319u, (710454127, 820233887))
```

It certainly is inconvenient to have to extract the value from the tuple and pass on the state every time we make this function call.

To create a state monad, we will first define an operator that allows us to chain functions together. Why? Because **`getRandomImmutable`** is a function taking a state and returning a value and a state, and what we're doing in the previous code by always passing in the new state to get the next random number is a manual form of chaining the function calls. Here's our new operator:

```
let (>>=) x f =
    (fun s0 ->
        let v, s = x s0
        f v s)    // Note! Continuation passing style!
```

This operator is borrowed from the functional programming language Haskell, which defines this operator as "Monad sequencing operator with value passing." Here, **`x`** is function that returns a value and a new state given the initial state, **`s0`**. The operator is a partial function application because it is provided with x and the next function in the chain, but it isn't given **`s0`**, the initial state.

Let's look just at what this expression does:

```
let v, s = x s0
```

And let's pretend that **x** is this function (see how it is returning a tuple, a value, and the next state):

```
let p = (fun t -> t * 10, t + 1)
```

So, for example:

```
> let v, s = p 1;;

val v : int = 10
val s : int = 2
```

Here we have a "state system" that returns 10 times the current state value and the state incremented by 1.

The next state in the **>>=** operator is to apply the function **f**, which is on the right-side of the operator. We can do some useful things here:

```
let getState = (fun v s -> s)
let getValue = (fun v s -> v)
```

We can see the results here (F# console):

```
> 1 |> (p >>= getValue);;
val it : int = 10

> 1 |> (p >>= getState);;
val it : int = 2
```

Now, the trick here is that we want the continuation function **f** to have the right signature so that we can continue to use the **>>=** operator in our monadic sequence. Well, what would this be? It's a function that takes a value and returns the function that performs the computation, in other words, **p**. So we can write this:

```
let q v =
    printfn "%d" v
    p
```

Now we can write a monadic sequence (F# console):

```
> 1 |>  (p >>= q >>= q >>= getValue);;
```

```
10
20
val it : int = 30
```

In other words, **p** evaluates with the value of **1**, calls **q** which returns the partial function application of **p**, whose call is "finished" by the caller **f v s**. We could explicitly specify the state parameter as well:

```
let q v s =
    printfn "%d" v
    p s
```

But with partial function application, we don't need to!

Now we can take what we learned about the **>>=** operator and apply it with our random number generator, which already conforms to the form of taking a state and returning a tuple of value and state. All we need to do is write this in a continuation passing style, providing the next function in the chain (F# console):

```
> (1, 2) |> (getRandomImmutable >>=
                (fun a -> getRandomImmutable >>=
                          (fun c -> getRandomImmutable) >>= getValue));;

2422836384
1907405312
3696412319
```

Now let's write this with a **StateBuilder** class:

```
type StateBuilder() =
    member m.Bind(x, f) = x >>= f
    member m.Return a = (fun s -> a, s)
```

The **Bind** function is itself bound to the function we created earlier: a function that takes a state and returns a function that takes a value and a state. Thus, **Bind** is a function that takes a function accepting a state and returning a state and a value, which is passed to a function taking a value and a state.

That last part is where the continuation passing style comes into effect, allowing us to chain the function calls to **getRandomImmutable**.

The de-sugared call using the **StateBuilder** looks like this (note how it's in the same continuous passing style form as all our previous monads):

```
let state = new StateBuilder()

let printRandoms = (1, 2) |>
```

```
                    state.Bind(getRandomImmutable, fun a ->
                    state.Bind(getRandomImmutable, fun b ->
                    state.Bind(getRandomImmutable, fun c -> state.Return 0)));;
```

Here, we start with a “seed” state that is passed to our monad sequence function, which returns a value and a state. The value is “a” (or “b”, or “c”) and the state “value” completes the partial application of the next call, which is again a function that takes a state and returns a value and a state. Again, if we wanted to be explicit about how state is being handled, we could write:

```
let randoms2 = (1, 2) |>
        state.Bind(getRandomImmutable, fun a s ->
        s |> state.Bind(getRandomImmutable, fun b s ->
        s |> state.Bind(getRandomImmutable, fun c s -> s |> state.Return 0)));;
```

But it’s the partial function application syntax that allows us to easily translate the first CPS example to the sugared computation expression:

```
let printRandomValues =
    state {
        let! a = getRandomImmutable
        let! b = getRandomImmutable
        let! c = getRandomImmutable
        return 0
    }
```

The result (F# console):

```
> computeRandomValues (1, 2);;
2422836384
1907405312
3696412319
val it : int * (int * int) = (0, (710454127, 820233887))
```

Observe the final return from the call to **`computeRandomValues`**. It’s a tuple with our return value of **`0`** and the current state of the random number generator. Simply printing a result isn’t very useful (and it’s a side effect, actually), so let’s modify this function slightly to return a list of random values:

```
let computeRandomValues =
    state {
        let! a = getRandomImmutable
        let! b = getRandomImmutable
        let! c = getRandomImmutable
        return a :: b :: c :: []
    };;
```

And now the result (having commented out the **printfn** statement in **getRandomImmutable**) is a list of our three random numbers from the initial seed (F# console):

```
> fst (computeRandomValues (1, 2));;    // Throw away the final state.
val it : uint32 list = [2422836384u; 1907405312u; 3696412319u]
```

Hopefully these exercises have demystified monads and computation expressions. There are several more behaviors that a computation expression "sugars" that you can explore further on the MSDN pages on this topic.

Lessons

- When composing a workflow, the first step is to rewrite the workflow in a continuation passing style. This may involve "helper" functions such as the function defined by the operator **>>=** to make your continuation calls more readable.
- Once the workflow is behaving correctly in a continuation passing style syntax, you can easily translate it to a sugared computation expression.
- Let's review one key part of the definition of a monad: *"A type with a monad structure defines what it means to chain operations, or nest functions of that type together."*
- Computation expressions are a way of writing workflows in a sugared syntax.
- Computation expressions are not monads—they are a sugared syntax for implementing monads in a workflow. This is an important differentiation—often, you will see "monads aka computation expressions" in the online literature, which is not accurate.
- When first encountering the state monad, one might think, "Ah, this is how you preserve state in an F# program." That is not correct; the state monad allows you to preserve state *within a workflow.*
- Monads are a great way to create workflows where you have to manage data, control logic, and state (side effects). Whenever you encounter the need to perform a workflow (such as reading data from a database, a file stream, and so forth) you will most likely want to take advantage of monads and the CPS that is fundamental to the monad pattern. The computation expression syntax provided in F# makes defining and working with monads much more syntactically comprehensible, thus allowing you to write workflows that would otherwise require mutable members to preserve state.

Chapter 4 Imperative and Functional Interaction

This section will discuss how to work with C# and F# code together in one application. We'll create a simple application that has a front-end user interface implemented in C# and its back-end data management implemented in F#. Whether creating Windows Forms or web applications with ASP.NET, mixing both imperative and functional programming styles is probably going to be the most efficient approach—certainly, the tools for designing forms or web pages, XAML or HTML, all have considerably more support for generating C# or VB code. Furthermore, because UIs are constantly managing state, the parts of an application that are intrinsically mutable are quite frankly best implemented in a language that is designed to support mutability. Conversely, the back-end of an application dealing with data persistence and transformation is often well-suited for F#, as immutability is critical for working with asynchronous processes.

Creating multi-language projects

The first step is simple enough: create a new Windows Forms Application project in Visual Studio 2012. I've called mine **fsharp-demo**:

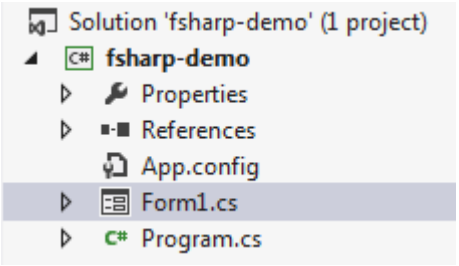


Figure 1: New Windows Forms Project

Next, right-click on the solution and select **Add** > **New Project**. On the left tree, click **Other Languages** to expand the selection list:

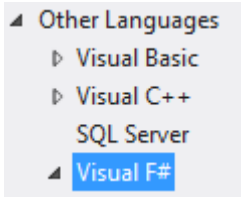


Figure 2: Available Programming Languages

Click **Visual F#** and select **F# Library**. Enter a name, for example, **fsharp-demo-lib**. Note how Visual Studio creates a stub **.fs** file with a default class:

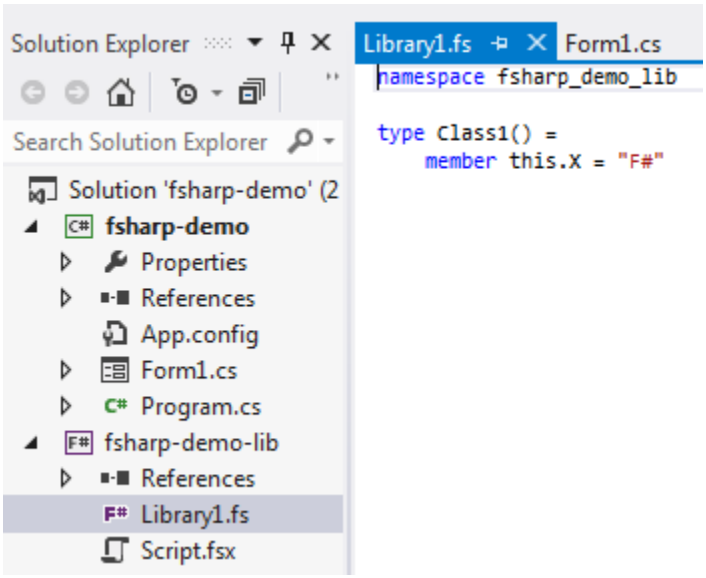


Figure 3: New F# Library

Calling F# from C#

Unfortunately, the code generated in this example is not really what we want—Visual Studio has created an imperative-style class template for us in F#, when what we want is a functional programming template. So, replace the generated code first with a module name. We do this because **`let`** statements, which are static, require a module that is implemented by the compiler as a static class. So instead, we can write, for example:

```
module FSharpLib

let Add x y = x + y
```

We got rid of the stub class as well, and created a simple function that we can call from C#. If you leave off the module name, the module name will default to the name of the file, which in our case would be **Library1**.

If you need to use a namespace to avoid naming conflicts, you can do so like this:

```
namespace fsharp_demo_lib
module FSharpLib =

    let Add x y = x + y
```

Note that the **=** operator is now explicitly required. This is because when combined with a namespace, the module is now local within that namespace rather than top-level.[61] In the C# code, we have to add a **`using fsharp_demo_lib;`** statement or reference the namespace

[61] Modules: http://msdn.microsoft.com/en-us/library/dd233221.aspx

explicitly when resolving the function call, for example **`fsharp_demo_lib.FSharpLib.Add(1, 2);`**. For the remainder of this section, I will not be using namespaces in F#.

Next, you'll want to add a reference in your C# project to the F# project:

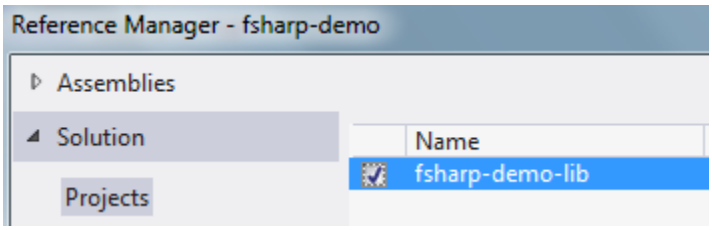


Figure 4: Referencing the C# Project

Now you will notice something interesting—if you go to **Form1.cs** and type the following:

```
public Form1()
{
	int result = FSharpLib.Add(1, 2);   // <<== start typing this
	InitializeComponent();
}
```

You will notice that the IDE flags this as an unknown class and IntelliSense does not work. *You always need to build the solution when making changes in the F# code before those changes will be discovered by the IDE on the C# side.*

Finish the test case in the constructor:

```
public Form1()
{
	int result = FSharpLib.Add(1, 2);
	MessageBox.Show("1 + 2 = " + result);
	InitializeComponent();
}
```

When we run the application, a message box first appears with the result. Congratulations, we've successfully called F# from C#.

Calling C# from F#

We may also want to go the other direction, calling C# from our F# code. We've already seen numerous examples of this, but let's say you want to call some C# code in your own application. As with any multi-project solution, we can't have circular references, so this means that you have to be cognizant of the *structure* of your solution, such that common code shared between C# and F# projects must go in its own project.

The next question is whether you want to call C# code in a static or instance context. A static context is similar to calling F# code. First, simply create a static class and some static members:

```
namespace csharp_demo_lib
{
	public static class StaticClass
	{
		public static void Print(int x, int y)
		{
			MessageBox.Show("Static call, x = " + x + " y = " + y);
		}
	}
}
```

And in the F# code, we add a reference to the C# library and use the **open** keyword (equivalent to the **`using`** keyword in C#) to reference the namespace:

```
module FSharpLib
open csharp_demo_lib

let Add x y =
    StaticClass.Print(x, y)
    x + y
```

Conversely, we can instantiate a C# class and manipulate its members in a mutable context, as well as call methods. For example:

```
namespace csharp_demo_lib
{
    public class InstanceClass
    {
		public void Print()
		{
			MessageBox.Show("Instance Call");
		}
    }
}
```

And in F#:

```
module FSharpLib
open csharp_demo_lib

let Add x y =
    let inst = new InstanceClass()
    inst.Print()
    StaticClass.Print(x, y)
    x + y
```

Database explorer—a simple project

Now that we understand the basics of C# and F# interaction, let's create a simple "database explorer" application. This application will utilize Syncfusion's Essential Studio as the front-end in C#, and F# as the back-end for all the database connections and queries. This user interface will consist of:

- A list control from which the user can pick a table.
- A grid control that will display the contents of the selected table.

The back-end, in F#, will:

- Query the database schema for a list of tables.
- Query the database for selected table data.

We'll be connecting to the AdventureWorks2008 database.

The source code can be viewed and cloned from GitHub at https://github.com/cliftonm/DatabaseExplorer.

The back-end

Let's start by writing the F# back-end and incorporate some simple unit tests[62] (written in F#!) using xUnit[63] in a test-driven development[64] process, as this also illustrates how to write unit tests for F#. We're using xUnit instead of nUnit because, at the time of this writing, nUnit does not support .NET 4.5 assemblies, whereas the **xunit.gui.clr4.exe** test runner does.

Establishing a connection

We'll start with a unit test that verifies that a connection is established to our database, and if we give it a bad connection string, we get a **`SqlException`**:

```
module UnitTests

open System.Data
open System.Data.SqlClient
open Xunit
open BackEnd

type BackEndUnitTests() =
    [<Fact>]
    member s.CreateConnection() =
        let conn = openConnection "data source=localhost;initial
catalog=AdventureWorks2008;integrated security=SSPI"
```

[62] Unit Testing Succinctly: http://www.syncfusion.com/resources/techportal/ebooks/unittesting
[63] xUnit: https://github.com/xunit/xunit
[64] Test-driven development: http://en.wikipedia.org/wiki/Test-driven_development

```
        Assert.NotNull(conn);
        conn.Close

    [<Fact>]
    member s.BadConnection() =
        Assert.Throws<SqlException>(fun () ->
            BackEnd.openConnection("data source=localhost;initial
catalog=NoDatabase;integrated security=SSPI") |> ignore)
```

Note that we have to explicitly pipe the result of a function that returns something to "ignore."

The supporting F# code:

```
module BackEnd

open System.Data
open System.Data.SqlClient

// Opens a SqlConnection instance given the full connection string.
let openConnection connectionString =
    let connection = new SqlConnection()
    connection.ConnectionString <- connectionString
    connection.Open()
    connection
```

Loading the database schema

Next, we'll load the database schema. When interfacing C# and F# code, it is a good practice to stay within the F# namespaces and constructs as much as possible, writing separate functions to convert from F# constructs to the imperative (mutable) structures that C# typically uses. First, let's write a simple unit test that ensures we get some results:

```
[<Fact>]
member s.ReadSchema() =
    use conn = s.CreateConnection()
    let tables = BackEnd.getTables conn
    Assert.True(tables.Length > 0)
    // Verify that some known table exists in the list.
    Assert.True(List.exists(fun (t) -> (t.tableName = "Person.Person")) tables)
```

Note the **`use`** keyword,[65] which will automatically call **`Dispose`** when the variable goes out of scope.

The supporting F# code follows. Note that one of the best practices in writing F# code is to write functions as small as possible and extract behaviors into separate functions:

[65] The "use" Keywords: http://msdn.microsoft.com/en-us/library/dd233240.aspx

```
// A simple record for holding table names.
type TableName = {tableName : string}

// A discriminated union for the types of queries we're going to use.
type Queries =
    | LoadUserTables

// Returns a SQL statement for the desired query.
let getSqlQuery query =
    match query with
    | LoadUserTables -> "select s.name + '.' + o.name as table_name from
sys.objects o left join sys.schemas s on s.schema_id = o.schema_id where type_desc
= 'USER_TABLE'"

// Returns a SqlCommand instance.
let getCommand (conn : SqlConnection) query =
    let cmd = conn.CreateCommand()
    cmd.CommandText <- getSqlQuery query
    cmd

// Reads all records.
let readTableNames (cmd : SqlCommand) =
    let rec read (reader : SqlDataReader) list =
        match reader.Read() with
        | true -> read reader ({tableName = (reader.[0]).ToString()} :: list)
        | false -> list
    use reader = cmd.ExecuteReader()
    read reader []

// Returns the list of tables in the database specified by the connection.
let getTables (conn : SqlConnection) =
    getCommand conn LoadUserTables |> readTableNames |> List.rev
```

In the previous code, we've created:

- A discriminated union so we can have a lookup table for our SQL statements.
- A function that returns the desired SQL statement given the desired type. This could be easily replaced with, say, a lookup from an XML file.
- A function for returning a SqlCommand instance given a connection and the query name.
- A function that implements a recursive reader.
- The **`getTables`** function, which returns a list of table names.

Before we conclude with the reading of the database's user tables, let's write a function that maps the F# list to a **`System.Collections.Generic.List<string>`**, suitable for consumption by C#. Again, we could use the F# types directly in C# by including the **`FSharp.Core`** assembly. This conversion is done here merely as a matter of convenience. Here's a simple unit test:

```
[<Fact>]
```

```
member s.toGenericList() =
    use conn = s.CreateConnection()
    let tables = BackEnd.getTables conn
    let genericList = BackEnd.tableListToGenericList tables
    Assert.Equal(genericList.Count, tables.Length)
    Assert.True(genericList.[0] = tables.[0].tableName)
```

And here's the implementation:

```
// Convert a TableName : list to a System.Collection.Generic.List.
let tableListToGenericList list =
    let genericList = new System.Collections.Generic.List<string>()
    List.iter(fun (e) -> genericList.Add(e.tableName)) list
    genericList
```

Reading a table's data

Next, we want to be able to read any table's data, returning a tuple of the data itself as a list of generic records and a list of column names. The following example is our unit test:

```
[<Fact>]
member s.LoadTable() =
    use conn = s.CreateConnection()
    let data = BackEnd.loadData conn "Person.Person"
    Assert.True((fst data).Length > 0)
    // Assert something we know about the schema.
    Assert.True(List.exists(fun (t) -> (t.columnName = "FirstName")) (snd data))
    // Verify the correct order of the schema.
    Assert.True((snd data).[0].columnName = "BusinessEntityID");
```

To implement this, we now need to extend our SQL query lookup:

```
type Queries =
    | LoadUserTables
    | LoadTableData
    | LoadTableSchema
```

Also, the function that returns the query needs to be smarter, substituting table name for certain queries based on an optional table name parameter:

```
let getSqlQuery query (tableName : string option) =
    match query with
    | LoadUserTables -> "select s.name + '.' + o.name as table_name from
sys.objects o left join sys.schemas s on s.schema_id = o.schema_id where type_desc
= 'USER_TABLE'"
    | LoadTableData ->
        match tableName with
```

```
        | Some name -> "select * from " + name
        | None -> failwith "table name is required."
    | LoadTableSchema ->
        match tableName with
        | Some name ->
            let schemaAndName = name.Split('.')
            "select COLUMN_NAME from information_schema.columns where table_name =
'" + schemaAndName.[1] + "' AND table_schema='" + schemaAndName.[0] + "' order by
ORDINAL_POSITION"
        | None -> failwith "table name is required."
```

Next we need to be able to read the table data:

```
// Returns all the fields for a record.
let getFieldValues (reader : SqlDataReader) =
    let objects = Array.create reader.FieldCount (new Object())
    reader.GetValues(objects) |> ignore
    Array.toList objects

// Returns a list of rows populated with an array of field values.
let readTableData (cmd : SqlCommand) =
    let rec read (reader : SqlDataReader) list =
        match reader.Read() with
        | true -> read reader (getFieldValues reader :: list)
        | false -> list
    use reader = cmd.ExecuteReader()
    read reader []
```

And we need to be able to read the schema for the table (note the required explicit casting to an **`Object[]`**):

```
let readTableSchema (cmd : SqlCommand) =
    let schema = readTableData cmd
    List.map(fun (c) -> {columnName = (c : Object[]).[0].ToString()}) schema |> List.rev
```

And lastly, we have the function that loads the data, returning a tuple of the data and its schema:

```
let loadData (conn : SqlConnection) tableName =
    let data = getCommand conn LoadTableData (Some tableName) |> readTableData
    let schema = (getCommand conn LoadTableSchema (Some tableName) |> readTableSchema)
    (data, schema)
```

Now, if you were paying attention, you'll have noticed that the functions **`readTableNames`** and **`readTableData`** are almost identical. The only difference is how the list is constructed. Let's refactor this into a single reader in which we pass in the desired function for parsing each row to create the final list:

```
// Reads all the records and parses them as specified by the rowParser parameter.
let readData rowParser (cmd : SqlCommand) =
    let rec read (reader : SqlDataReader) list =
        match reader.Read() with
        | true -> read reader (rowParser reader :: list)
        | false -> list
    use reader = cmd.ExecuteReader()
    read reader []
```

We now have a more general-purpose function that allows us to specify how we want to parse a row. We now create a function for returning a **`TableName`** record:

```
// Returns a table name from the current reader position.
let getTableNameRecord (reader : SqlDataReader) =
    {tableName = (reader.[0]).ToString()}
```

This allows us to refactor **`getTables`**:

```
// Returns the list of tables in the database specified by the connection.
let getTables (conn : SqlConnection) =
    getCommand conn LoadUserTables None |> readData getTableNameRecord
```

And we also refactor reading the table schema and the table's records:

```
// Returns a list of ColumnName records representing the field names of a table.
let readTableSchema (cmd : SqlCommand) =
    let schema = readData getFieldValues cmd
    List.map(fun (c) -> {columnName = (c : List<Object>).[0].ToString()}) schema |> List.rev

// Returns a tuple of table data and the table schema.
let loadData (conn : SqlConnection) tableName =
    let data = getCommand conn LoadTableData (Some tableName) |> readData getFieldValues
    let schema = (getCommand conn LoadTableSchema (Some tableName) |> readTableSchema)
    (data, schema)
```

Here we are taking advantage of partial function application—we've easily refactored the reader to be more general purpose in parsing each row. This took about five minutes to do, and with our existing unit tests we were able to verify that our changes didn't break anything.

Converting our F# structures to a DataTable

Unless you want to include the **`FSharp.Core`** assembly in your C# project, you'll want to convert anything being returned to C# into .NET "imperative-style" classes. It simply makes things easier. And certainly, we could have loaded the records directly into a **`DataTable`** by going through a **`DataSet`** reader, but the process we used was more illustrative of keeping things "native" to F# (and besides, one usually wouldn't load up the entire table, but rather implement a pagination scheme of some sort.)

So, the final step is to populate a **`DataTable`** with our F# rows and table schema information, which we'll do in F#. First though, we should create a unit test that ensures some things about our **`DataTable`**:

```
[<Fact>]
member s.ToDataTable() =
    use conn = s.CreateConnection()
    let data = BackEnd.loadData conn "Person.Person"
    let dataTable = BackEnd.toDataTable data
    Assert.IsType<DataTable>(dataTable) |> ignore
    Assert.Equal(dataTable.Columns.Count, (snd data).Length)
    Assert.True(dataTable.Columns.[0].ColumnName = "BusinessEntityID")
    Assert.Equal(dataTable.Rows.Count, (fst data).Length)
```

The implementation in F# involves three functions: setting up the columns, populating the rows, and a function that calls both of those steps and returns a **`DataTable`** instance:

```
// Populates a DataTable given a ColumnName List, returning
// the DataTable instance.
let setupColumns (dataTable : DataTable) schema =
    let rec addColumn colList =
        match colList with
        | hd::tl ->
            let newColumn = new DataColumn()
            newColumn.ColumnName <- hd.columnName
            dataTable.Columns.Add(newColumn)
            addColumn tl
        | [] -> dataTable
    addColumn schema

// Populates the rows of a DataTable from a data list.
let setupRows data (dataTable : DataTable) =
   // Rows:
    let rec addRow dataList =
        match dataList with
        | hd::tl ->
            let dataRow = dataTable.NewRow()
            // Columns:
            let rec addFieldValue (index : int) fieldList =
                match fieldList with
                | fhd::ftl ->
                    dataRow.[index] <- fhd
                    addFieldValue (index + 1) ftl
                | [] -> ()
            addFieldValue 0 hd
            dataTable.Rows.InsertAt(dataRow, 0)
            addRow tl
        | [] -> dataTable
    addRow data

// Return a DataTable populated from our (data, schema) tuple.
```

```
let toDataTable (data, schema) =
    let dataTable = new DataTable()
    setupColumns dataTable schema |> setupRows data
```

All our unit tests pass!

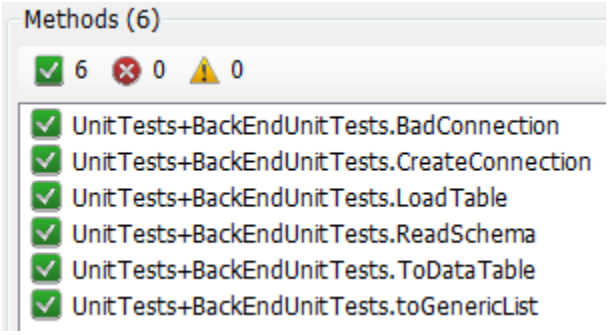


Figure 5: Successful Unit Tests

The front-end

Now we're ready to write the front-end. We'll create a simple layout of two **`GridListControl`** controls:

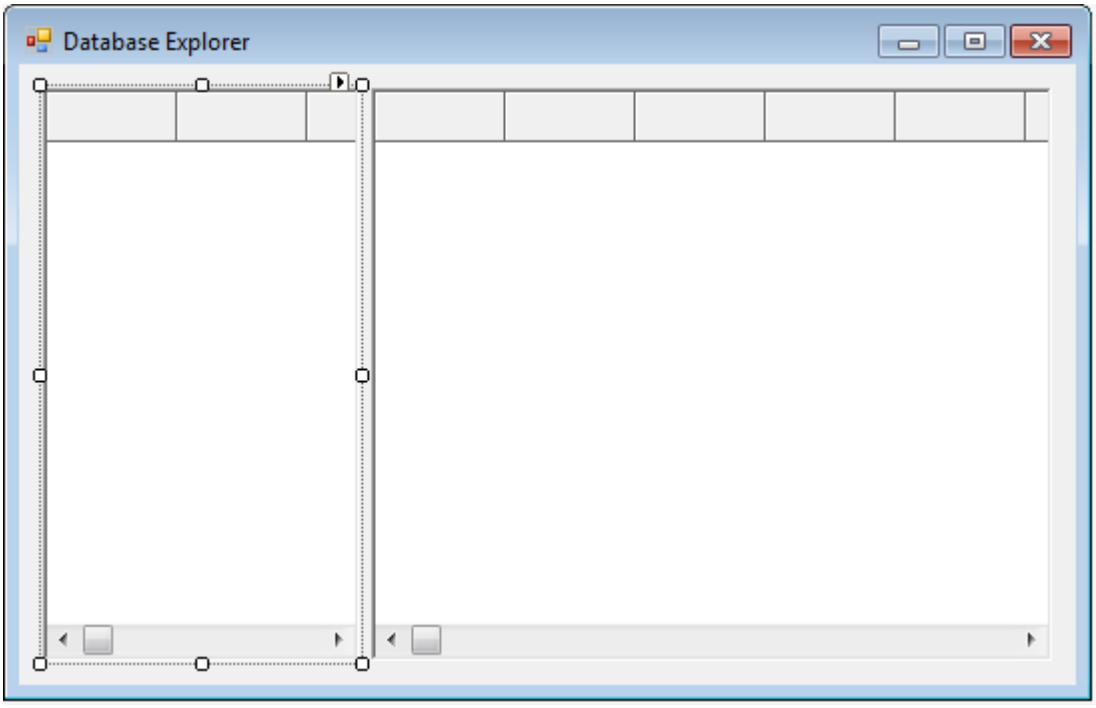


Figure 6: Two GridList Controls

The code-behind consists of loading the list of tables, calling our F# code to get the table list:

```
protected void InitializeTableList()
{
	List<string> tableList;

	using (var conn = BackEnd.openConnection(connectionString))
	{
		tableList = BackEnd.getTablesAsGenericList(conn);
	}

	var tableNameList = new List<TableName>();
	tableList.ForEach(t => tableNameList.Add(new TableName() { Name = t }));
	gridTableList.DisplayMember = "Name";
	gridTableList.ValueMember = "Name";
	gridTableList.DataSource = tableNameList;
}
```

When a table is selected, we get the **`DataTable`** from our F# code and set the grid's **`DataSource`** property.

```
private void gridTableList_SelectedValueChanged(object sender, EventArgs e)
{
	if (gridTableList.SelectedValue != null)
	{
		string tableName = gridTableList.SelectedValue.ToString();
		DataTable dt;

		using (var conn = BackEnd.openConnection(connectionString))
		{
			Cursor = Cursors.WaitCursor;
			var data = BackEnd.loadData(conn, tableName);
			dt = BackEnd.toDataTable(data.Item1, data.Item2);
			Cursor = Cursors.Arrow;
		}

		gridTableData.DataSource = dt;
	}
}
```

This gives us a nice separation between the front-end user interface processes and the back-end database interaction, resulting in a simple database navigator.

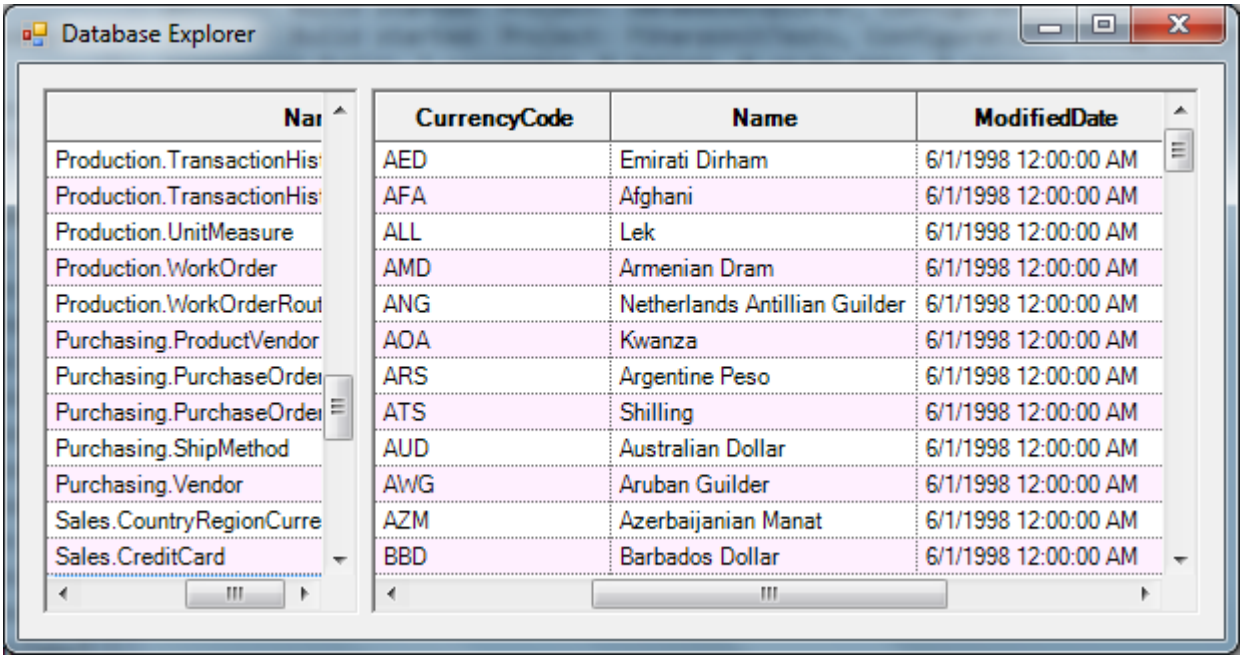


Figure 7: Complete Database Navigator

Conclusion

"Visual F# is a strongly-typed, functional-first programming language for writing simple code to solve complex problems using Visual Studio and the .NET Framework. F# is designed to reduce the time-to-deployment and complexity of software components such as calculation engines and data-rich analytical services in the modern enterprise."[66]

We hear these words "reduce time" and "reduce complexity" so often that they are essentially meaningless now. And rightly so, as there are very few technologies beyond the toaster that actually save time and reduce complexity—most tend to push time and complexity management onto something else, and, as with any programming language, that "something else" is the skill of the developer.

Sure, I was writing my first object-oriented C++ application a couple weeks into reading Bjarne Stroustrup's *The C++ Programming Language*, but was I doing it well? Certainly not. When I started doing C# development, I was eagerly looking forward to features like reflection that didn't exist in C++ and griping about the lack of templates at the same time. But again, the nuances of working with interfaces, events, and so forth were skills I had to learn. Happily, I started working in C# near the point of its inception and so my skills have been able to evolve with the language, writing LINQ, lambda expressions, and other concepts found in C# 4.0 and 5.0.

C# 3.0 was the introduction of functional programming, with the inclusion of lambda expressions and query expressions. And it looks like C# 6.0 will continue adding functional programming style support with the addition of property and method expressions.

No functional programming language is easy to learn. Functional programming languages tend to be very "symbol rich," which sometimes makes it difficult to understand even a single line of code. Reading the definition of a function type is also non-trivial, and while type inference can be a big help, it can also make it confusing.

So if you want to reduce time and complexity in the application development process, you must develop an intimate knowledge of the language. And with functional programming languages like F#, this means not just learning the syntax, but learning to think differently as well. Hopefully this book has provided some guidance on that issue.

There *are* two aspects of functional programming that have the potential to reduce development time, and both are a direct result of immutability. The first, a lack of side effects, means that functions are potentially "provably correct" mathematical equations. Given the same inputs, a function will return the same results. There are no side effects on other functions with which to be concerned, and each function is itself not subject to side effects. In my opinion, side effects, and more generally state management, are some of the most difficult things to adequately test for, resulting in the oft-heard phrase "I have no idea why it did that." The second result of immutability is the ease in which it becomes possible to write multitasking, asynchronous

[66] Visual F# Resources: http://msdn.microsoft.com/en-us/vstudio/hh388569

operations. Mutable structures require synchronization. With languages like F#, synchronization is not an issue because structures are not mutable.

In both of these cases, the language itself promotes a coding style that directly affects implementation, resulting in a natural “correctness” of the application and the ability to easily perform operations asynchronously. As computers are produced with more physical and virtual processors and as programmers start to take advantage of the massive parallel processing streams in a GPU, having a language that implicitly results in “correctness” and supports asynchronous computation is critical for the upcoming tasks of data mining and the live processing of multiple information streams.

This, in my opinion, is the foremost advantage of a functional programming language like F#, and is a sufficient reason to learn how to think “functionally.”

Appendix A

"Stack" trace of tree recursion:

Iteration	"Stack"	Evaluates as:
Depth: 0 Caller: root Node: 10	x -> x	
Depth: 1 Caller: left Node: 5	x -> x lacc -> racc -> f (lacc + racc + 1)	
Depth: 2 Caller: left Node: 1	x -> x lacc -> racc -> f (lacc + racc + 1) lacc -> racc -> f (lacc + racc + 1)	
Depth: 3 Caller: left Empty lacc: 0, x: 1	x -> x lacc -> racc -> f (lacc + racc + 1) lacc -> racc -> f (lacc + racc + 1) lacc -> racc -> f (lacc + racc + 1)	racc -> f (0 + racc + 1)
Depth: 3 Caller: right Empty lacc: 0, x: 1, racc: 0, returning = 1 lacc: 1, x: 5	x -> x lacc -> racc -> f (lacc + racc + 1) lacc -> racc -> f (lacc + racc + 1) racc -> f (0 + racc + 1)	f (0 + 0 + 1) racc -> f (1 + acc + 1)
Depth: 2 Caller: right Empty lacc: 1, x: 5, racc: 0, returning = 2 lacc: 2, x: 10	x -> x lacc -> racc -> f (lacc + racc + 1) racc -> f (1 + racc + 1)	f (1 + 0 + 1)
Depth: 1 Caller: right	x -> x racc -> f (2 + racc + 1)	

Node: 20

Depth: 2 Caller: left	x -> x racc -> f (2 + racc + 1) lacc -> racc -> f (lacc + racc + 1)	
Empty		racc -> f (0 + acc + 1)
lacc: 0, x: 20		

Depth: 2 Caller: right	x -> x racc -> f (2 + racc + 1) racc -> f (0 + racc + 1)	
Empty		f (0 + 0 + 1)
lacc: 0, x: 20, racc: 0, returning = 1	x -> x f (2 + 1 + 1)	4 -> 4

lacc: 2, x: 10, racc: 1, returning = 4