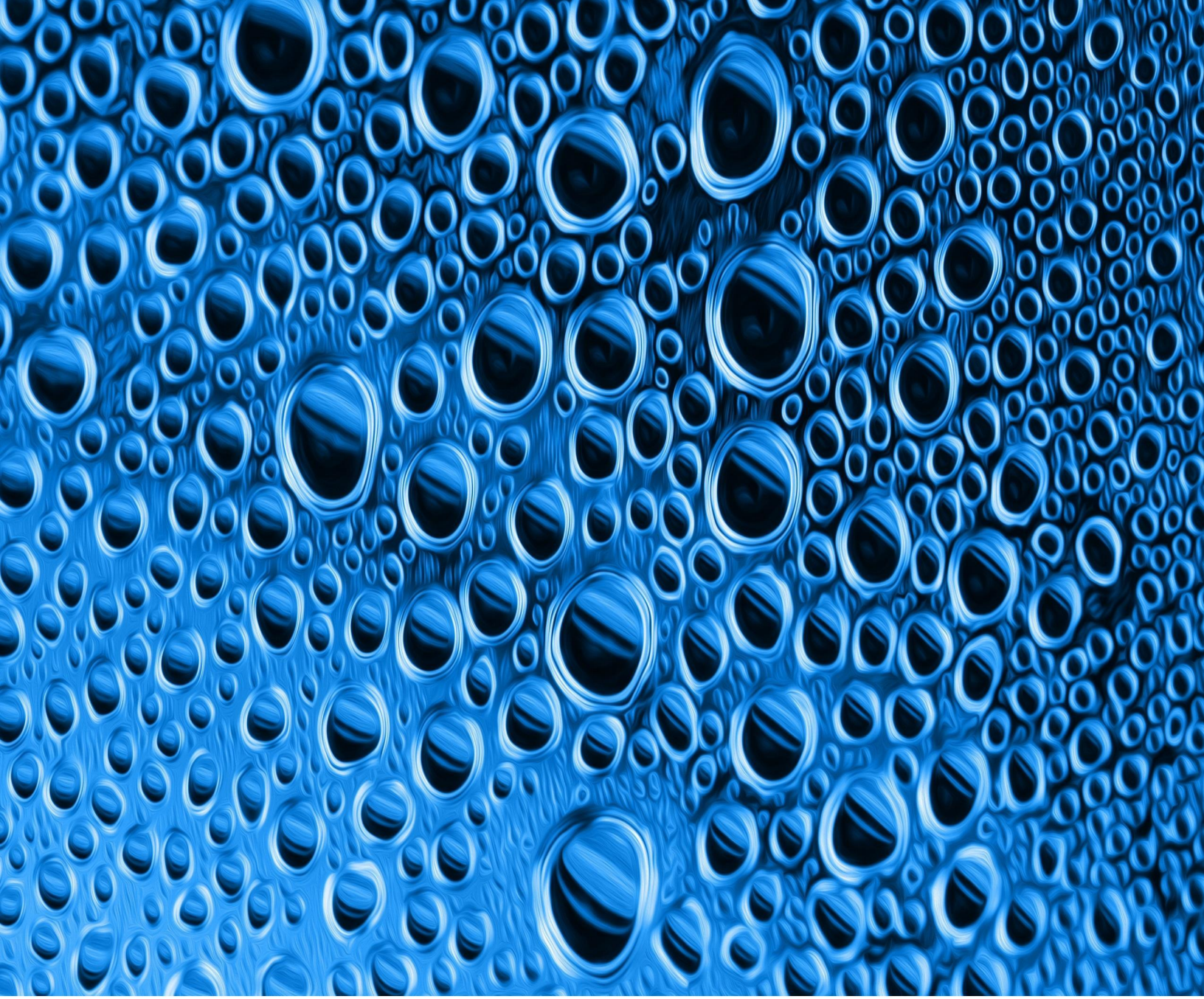

Essential F#

SUCCINCT | ROBUST | PERFORMANT

Ian Russell

Essential F#

Ian Russell

This book is for sale at http://leanpub.com/essential-fsharp

This version was published on 2022-07-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface **1**
Contents 2
F# Software Foundation 4
About the author 4

Acknowledgments **5**

Getting Started **6**
Setting up your environment 6
One Last Thing 7

1 - An Introductory Domain Modelling Exercise **8**
The Problem 8
Getting Started 9
Making the Implicit Explicit 16
Going Further 18
Alternative Approaches (1 of 2) 21
Alternative Approaches (2 of 2) 22
Summary 24
Postscript 25

2 - Functions **26**
Getting Started 26
Theory 26
In Practice 27
Unit 30
Anonymous Functions 31
Multiple Parameters 32
Partial Application (Part 1) 33
The Forward Pipe Operator 33
Partial Application (Part 2) 36
Summary 38

3 - Null and Exception Handling **39**
Null Handling 39

Interop With .NET 42
Handling Exceptions 43
Function Composition With Result 45
Summary 51

4 - Organising Code and Testing 52
Getting Started 52
Solutions and Projects 52
Adding a Source File 52
Namespaces and Modules 53
Writing Tests 56
Writing Real Tests 57
Using FsUnit for Assertions 61
Summary 62

5 - Introduction to Collections 63
Before We Start 63
The Basics 63
Core Functionality 63
Folding 68
Grouping Data and Uniqueness 69
Solving a Problem in Many Ways 70
Working Through a Practical Example 71
Summary 79

6 - Reading Data From a File 80
Setting Up 80
Loading Data 80
Parsing Data 83
Testing the Code 85
Final Code 87
Summary 89

7 - Active Patterns 90
Setting Up 90
Partial Active Patterns 90
Parameterized Partial Active Patterns 92
Multi-Case Active Patterns 96
Single-Case Active Patterns 97
Using Active Patterns in a Practical Example 99
Summary 102

8 - Functional Validation 103
Setting Up 103

Solving the Problem . . . 105
Where are we now? . . . 110
Functional Validation the F# Way . . . 114
Summary . . . 117

9 - Single-Case Discriminated Union . . . 118
Setting Up . . . 118
Solving the Problem . . . 118
Using Modules . . . 125
A Few Minor Improvements . . . 127
Using a Record Type . . . 129
Summary . . . 129

10 - Object Programming . . . 130
Setting Up . . . 130
Class Types . . . 130
Interfaces . . . 132
Object Expressions . . . 134
Encapsulation . . . 136
Equality . . . 138
Summary . . . 140

11 - Recursion . . . 141
Setting Up . . . 141
Solving The Problem . . . 141
Tail Call Optimisation . . . 142
Expanding the Accumulator . . . 143
Using Recursion to Solve FizzBuzz . . . 144
Quicksort using recursion . . . 145
Recursion with Hierarchical Data . . . 146
Other Uses Of Recursion . . . 152
Summary . . . 152

12 - Computation Expressions . . . 153
Setting Up . . . 153
Introduction . . . 153
The Result Computation Expression . . . 156
Introduction to Async . . . 158
Compound Computation Expressions . . . 159
Debugging Code . . . 164
Further Reading . . . 164
Summary . . . 165

13 - Introduction to Web Programming with Giraffe . . . 166

Getting Started . . . 166
Using Giraffe . . . 167
Creating an API Route . . . 170
Creating a Custom HttpHandler . . . 170
Creating a View . . . 171
Adding Subroutes . . . 172
Reviewing the Code . . . 174
Summary . . . 176

14 - Creating an API with Giraffe . . . 177
Getting Started . . . 177
Our Task . . . 177
Sample Data . . . 177
Routes . . . 178
Handlers . . . 180
Using the API . . . 182
Summary . . . 183

15 - Creating Web Pages with Giraffe . . . 184
Getting Started . . . 184
Configuration . . . 184
Adding a Master Page . . . 185
Creating the Todo List View . . . 186
Loading Data on Startup . . . 188
Summary . . . 190

Summary . . . 191
F# Software Foundation . . . 192
Resources . . . 192
And Finally . . . 193

Appendix 1 . . . 194
Creating Solutions and Projects in VS Code . . . 194

Appendix 2 . . . 196
CSS and JavaScript for Chapter 15 . . . 196

Preface

This book is targeted at folks wanting to learn F# and assumes that the reader has no real knowledge of F# or functional programming. Some programming experience, particularly in C# or VB.NET, may be useful but not absolutely necessary.

Although F# has often been portrayed as a functional programming language, it isn't, and neither is this book purely about functional programming. You are not going to learn about Category Theory, Lambda Calculus, or even Monads. You will learn things about programming in a functional-first style as a by-product of how we solve problems with F#. What you will learn is how F#'s functional-first approach to programming empowers everyone to write succinct, robust, and performant code[1]. You will learn a lot of new terminology as you read this book but only enough to understand the features of F# that allow us to solve real-world problems with clearly expressed and concise code. What you will take away most from this book is an understanding of how F# supports us by generally making it easier for us to write code the F# way than it is to be a functional programming purist.

For many historical reasons, functional programming has been viewed as difficult to learn and not very practical by most .NET developers. Functional programming is too often regarded as being too steeped in mathematics and academia to be of any use for writing the everyday line of business applications that most of us work on. This book is designed to help dispel those impressions. So the obvious question to ask is:

> What is F# good for?

Whilst it's great for many things like web programming, cloud programming, Machine Learning, AI, and Data Science, the most accurate answer comes from a long-standing F# Community member Dave Thomas[2]:

> "F# is good for programming"

F# is a general-purpose language for the .NET platform along with C# and VB.NET. F# has been bundled with Visual Studio, and now the .NET SDK, since 2010. The language has been quite stable since then. Code written in 2010 is not only still valid today but is also likely to be stylistically similar to current standards.

Why would you choose F# over another language? I noticed a question from cancelerx[3] on Twitter:

> "I would like to do a lightning talk on F#. My team is primarily PHP and Python devs. What should I showcase? I want to light a spark."

[1] https://fsharp.org/
[2] https://twitter.com/7sharp9_
[3] https://twitter.com/ndy40

My response was:

> The expressive type system, composition with the forward pipe operator, pattern matching, collections, and the REPL.

I've been thinking about my answer and whilst I still think that the list is a good one, I think that I should have said that it is how the various features work together that makes F# so special, not the individual features themselves. It feels like a well-thought-out language that doesn't have features just because other languages have them.

F# supports many paradigms, such as imperative and object-oriented programming but we will concentrate on how it encourages functional-first programming. My view of the choices F# has made in its language design can be summed up quite easily:

> I enjoy functional programming but I like programming in F# even more.

I'm not alone in feeling like this about F#. I hope that this book is a useful step on a similar journey to the one that I've been on.

Contents

This book covers the core features and practices that a developer needs to know to work effectively on the types of F# Line of Business (LOB) applications we work on at Trustbit. It starts with the basics of types, function composition, pattern matching, and testing, and ends with an example of a simple website and API built using the wonderful Giraffe[4] library.

The original source for this book is two series of blog posts that I wrote on the Trustbit blog[5] in 2020/21. They have been significantly re-written to clarify and expand the explanations, to ensure that the code works on VS Code using the ionide F# extension, and to take advantage of some new F# features introduced in versions 5 and 6.

During the course of reading this book, you are going to be introduced to a wide range of features that are the essentials of F#:

Chapter 1 - An Introductory Domain Modelling Exercise

In the opening chapter, we take a typical business problem and look at how we can use some of the features of F# to solve it.

Chapter 2 - Functions

Chapter 2 is all about functions - What they are and how we use composition to build more useful functions.

[4] https://github.com/giraffe-fsharp/Giraffe
[5] https://trustbit.tech/blog

Chapter 3 - Null and Exception Handling

In this chapter, we look at the features F# offers that can eliminate null errors and reduce the need to throw exceptions.

Chapter 4 - Organising Code and Testing

In Chapter 4, we will look at how we should structure our code and files plus we get our first look at testing in F#.

Chapter 5 - Introduction to Collections

F# has amazing support for handling collections of data. In this chapter, we will get an introduction to some of the features it offers and see how easy it is to compose collection functions into pipelines to transform data from one format to another.

Chapter 6 - Reading Data From a File

A short chapter where we see how to load data from a CSV file and parse the data into F# types.

Chapter 7 - Active Patterns

Pattern matching is a very powerful tool for the F# developer and active patterns extend that power by allowing us to build custom matchers that can be used to simplify our code and make it more readable.

Chapter 8 - Functional Validation

In this chapter, we will use the tools from the previous chapter to add validation to the imported data from Chapter 6. We will also be introduced to one of the more unique features of F#, the computation expression.

Chapter 9 - Single-Case Discriminated Unions

In this chapter, we will see how we can reduce our reliance on primitive values in data structures and make our code more domain-centric.

Chapter 10 - Object Programming

F# is a functional-first language but it does have excellent support for object programming. This chapter will introduce you to some of the features available.

Chapter 11 - Recursion

Recursion is a powerful technique. In this chapter, we will see some of the ways that recursion can help us in our F# journey.

Chapter 12 - Computation Expressions

In this chapter, we look more deeply into computation expressions, which are the generic way that F# handles working with effects like Option, Result, and Async.

Chapter 13 - Introduction to Web Programming With Giraffe

In this chapter, we will go from nothing to having an API route and a web page using a really nice F# library called Giraffe and its partner, the Giraffe View Engine.

Chapter 14 - Creating an API With Giraffe

In this chapter, we will extend the API part of the website.

Chapter 15 - Creating Web Pages With Giraffe

A short chapter where we will look at more of the features offered by the Giraffe View Engine.

F# Software Foundation

You should join the F# Software Foundation[6]; It's free. By joining, you will be able to access the dedicated Slack channel where there are excellent tracks for beginners and beyond.

About the author

Ian Russell has over 25 years of experience as a software developer in the UK. He has held many technical roles over the years but made the decision many years ago that he could do the most good by remaining 'just a software developer'. Ian works remotely from the UK for Trustbit, a software solutions provider based in Vienna, on a cloud-based GPS aggregator for a logistics company written mostly in F#. Ian's .NET journey started with C# 1.1 in 2003 and he started playing in his own time with F# in 2010. He has been a regular speaker at UK user groups and conferences since 2009.

[6] https://foundation.fsharp.org/join

Acknowledgments

Thanks to the reviewers of this revised edition (Gianni Bossini, Yeray Cabello, Flavio Colavecchia, David Dawkins, Mauricio Magni, Almir Mesic, Simon Painter, Sanket Patel, and Eric Potter) for helping to add some sparkle to the rough diamond that I created.

Thanks to the many mentees that I've worked with over the last 18 months. Your dedication to learning and willingness to give feedback has been a real joy to me. I hope that you all go on to enjoy F# as much as I have over the last 12 years.

Thanks to the reviewers of the original release of the book (Yeray Cabello, Emiliano Conti, Martin Fuß, Erle Granger II, Viacheslav Koryagin, and Cédric Rup) without whom it would have been a longer and more tortuous delivery.

Thanks to Dustin Moris Gorski for Giraffe[7], Krzysztof Cieślak and Chet Husk for their work on ionide-fsharp[8], the various people who have worked on the language and compilers over the years, and finally, last but not least, Don Syme[9] for creating the wonder that is the F# language. All Hail Don Syme!

Finally, thanks to Trustbit and all its employees for putting up with me over the last three years.

[7] https://github.com/giraffe-fsharp/Giraffe
[8] https://ionide.io/
[9] https://twitter.com/dsymetweets

Getting Started

Setting up your environment

1. Install F#. The best and simplest way to do this is to install the latest .NET SDK[10] (6.0.x at time of publishing).
2. Install Visual Studio Code (VS Code).
3. Install the ionide F# extension for VS Code. You may need to reload the environment. The installation page for the extension will tell you.
4. Create a folder for the book and then folders for each book chapter to hold the code you will write.

F# Interactive (FSI)

If you are coming to F# from C#, the F# Interactive window is similar to the C# Interactive Window. The primary difference is that rather than relying on line-by-line debugging, F# developers will use the FSI instead to validate code. I never debug any code with breakpoints when doing F# development.

FSI appears in the VS Code Terminal window. In addition to the .NET Terminal, you will also have access to an F# Interactive window. You can type directly into FSI but we will send code from our files directly to FSI to compile and run.

> We run code in FSI by selecting the code that we want to run and pressing ALT+ENTER. The code will be copied into FSI, compiled, and run. If you don't change that code, you can reuse it later as FSI keeps your compiled code in memory.

If you type directly into FSI, you will need to end your input with two semi-colons `;;` to tell FSI to execute your code.

Script Files vs Code Files

F# supports two types of files: Script (.fsx) and Code (.fs). We will use both types throughout this book. The primary differences in VS Code are that only .fs files are compiled into the output .exe or .dll and .fsx files are primarily used to try things out with F# Interactive.

F# code from any type of file can be run in F# Interactive by sending it using ALT + ENTER.

There is a command-line version of FSI that can be used to run script files but that is outside the scope of this book.

[10] https://dotnet.microsoft.com/download

Getting to know VS Code and Ionide

Whilst the VS Code and ionide environment is not as fully featured as the full IDEs (Visual Studio and Jet Brains Rider), it is not devoid of features. You don't actually need that many features to easily work on F# codebases, even large ones.

Compositional-IT[11] have released a number of videos on YouTube that give an introduction to VS Code + F# shortcuts[12].

Ionide[13] is much more than just the F# extension for VS Code. Take some time to have a look at what they offer to help your F# experience.

One Last Thing

This book contains lots of code. It will be tempting to copy and paste the code from the book but you may learn more by actually typing the code out as you go. I definitely learn more by doing it this way.

The time for talking has finished. Now it is time to dive into some F# code. In chapter 1 we will work through a common business use case using F#.

[11] https://www.compositional-it.com/
[12] https://www.youtube.com/playlist?list=PLlzAi3ycg2x27lPUwp3Z-M2m44n681cED
[13] https://ionide.io/index.html

1 - An Introductory Domain Modelling Exercise

This book will introduce you to the world of functional-first programming in F#. Rather than start with theory or a formal definition, I thought that I'd start with a typical business problem and look at how we can use some of the functional programming features of F# to solve it.

> **F# Interactive**
>
> We will be using F# Interactive to compile and run our code in this chapter. If you are not aware of how to use this, please read the section on **Setting up your environment** in the Introduction.

Using VS Code, create a new folder to store the code from this chapter. Add a new file called *part1.fsx*. The .fsx file is an F# script file. We are going to use script files and run the code using F# Interactive rather than creating and running a console application via the dotnet CLI.

The Problem

This problem comes from a post by Chris Roff[14] where he looks at using F# and Behaviour Driven Development[15] together.

```
Feature: Applying a discount
Scenario: Eligible Registered Customers get 10% discount when they spend £100 or more

Given the following Registered Customers
|Customer Id|Is Eligible|
|John       |true       |
|Mary       |true       |
|Richard    |false      |

When [Customer Id] spends [Spend]
Then their order total will be [Total]

Examples:
```

[14] https://medium.com/@bddkickstarter/functional-bdd-5014c880c935
[15] https://cucumber.io/docs/bdd/

```
|Customer Id|   Spend|   Total|
|Mary       |   99.00|   99.00|
|John       |  100.00|   90.00|
|Richard    |  100.00|  100.00|
|Sarah      |  100.00|  100.00|

Notes:
Sarah is not a Registered Customer
Only Registered Customers can be Eligible
```

Along with some examples showing how you can verify that your code is working correctly, there are a number of domain-specific words and concepts that we may want to represent in our code. We will start with a simple but naive solution and then we'll see how F#'s types can help us make it much more domain-centric and as an added benefit, be less susceptible to bugs.

Getting Started

Along with simple datatypes like string, decimal, and boolean, F# has a powerful Algebraic Type System (ATS). Think of these types as simple data structures that you can use to compose larger data structures. We define a type using the `type` keyword as shown with this *tuple* type:

```
type Customer = string * bool * bool
```

This definition states that a *Customer* type is a tuple that consists of a *string* and two *booleans*. This means that a *tuple* is defined as an *AND* type. Tuples in F# do not have named parts, so a consumer of your type would have to guess the meaning of the data. Note the use of the capital letter at the start of the name of the type, commonly referred to as *Pascal Case*. Most things other than type definitions use *Camel Case*, where the first letter is in lower case.

We create an instance of the *Customer* type using the `let` keyword:

```
// string * bool * bool
let fred = ("Fred", true, true)
```

The parentheses in this example are optional. Notice the difference between the definition which uses `*` and the usage which makes use of `,` to separate the parts of the *tuple*.

Using `let`, we have bound the data `("Fred", true, true)` to the identifier `fred`. By default, all data is immutable in F#, so you shouldn't think of `fred` as a variable.

We can explicitly state the type of data that the `fred` identifier is bound to by adding it to our *let binding*:

```
// Customer
let fred : Customer = ("Fred", true, true)
```

We can also use the `let` keyword to decompose the tuple:

```
let (id, isEligible, isRegistered) = fred
```

We have bound the identifiers `id`, `isEligible`, and `isRegistered` to the data in the `fred` binding.

Rather than use a *tuple*, we will create a *Record* type that solves the property name issue. We can define our initial customer record type like this:

```
type Customer = { Id:string; IsEligible:bool; IsRegistered:bool }
```

The record type, like the tuple, is an *AND* type, so in this example, a *Customer* consists of an Id which is a string value, and two boolean values called *IsEligible* and *IsRegistered*. Record types, like most of the types we will see through this book, are immutable, that is they cannot be changed once created. This means that all of the data to create a *Customer* record must be supplied when an instance is created.

Instead of defining the record type on a single line, we can also put each field on a separate line.

> WARNING: Tabs are not supported by F#, so your IDE/editor needs to be able to convert tabs to spaces, which most do including VS code.

```
type Customer = {
    Id : string
    IsEligible : bool
    IsRegistered : bool
}
```

Notice how the properties are aligned. F# makes use of *Significant Whitespace*, so alignment is important for scope. Alignment issues are one of the first things you should look at if the compiler warns you of an error in your code.

If you put the named values on separate lines, you no longer need to use the semi-colon separator. Spaces between the name value's label and type are allowed and they do not need to be consistent.

To create an instance of a customer we would write the following **below the type definition**:

```
let fred = { Id = "Fred"; IsEligible = true; IsRegistered = true } // Customer
```

The F# compiler has inferred the type of the value bound to the *fred* identifier to be a Customer. By using the let keyword, we have bound the identifier *fred* to that instance of a Customer.

Another option that we have is to use the following style which is better when you have more properties to make your code easier to read:

```
let fred = {
    Id = "Fred"
    IsEligible = true
    IsRegistered = true
}
```

Note that we no longer need to use semi-colons to separate the property definitions.

You can annotate the type on the value binding if you like:

```
let fred : Customer = { Id = "Fred"; IsEligible = true; IsRegistered = true }
```

If there are multiple record types defined with the same structure, the compiler will not be able to infer the type, so we will need to annotate the identifier with the type.

We are not going to use the *fred* binding, so you can delete that line of code.

If you have used languages like C# or Java, you may have been surprised by the strict ordering of the code. The F# compiler works from the top of the file, downwards. It might seem a little odd at first but you soon get used to it. It has some really nice advantages for how we write and verify our code as well as making the compiler's job easier. This ordering applies at the project level too, so your source code (*.fs*) files need to be ordered in the same way, not alphabetically.

Below the *Customer* type, we need to create a function to calculate the total that takes a *Customer* and a *Spend* (decimal) as input parameters and returns the *Total* (decimal) as output:

```
// Customer -> decimal -> decimal
let calculateTotal (customer:Customer) (spend:decimal) : decimal =
    let discount =
        if customer.IsEligible && spend >= 100.0M
        then (spend * 0.1M) else 0.0M
    let total = spend - discount
    total
```

The M suffix at the end of a number tells the compiler that the number is a decimal.

There are a few things to note about functions:

- We have used `let` again to define the function and inside the function to define the discount and total bindings.
- The discount and total bindings are scoped to the function and are not visible outside the function.
- There is no container, such as a class, because functions are first-class citizens.
- The return type is to the right of the input arguments.
- No return keyword is needed as the last line is returned automatically.

- Nesting using significant whitespace is used to define scope. Tabs are not allowed.
- We used an `if` expression that must return the same type for true and false routes.

> **Expressions** are used throughout F# programming since they always return an output. This makes them easy to compose with other expressions and easy to test. In contrast, statements do not return an output, so are not composable.

The comment above the function definition shows the function signature, which is **Customer -> decimal -> decimal**. The item at the end of the signature, after the last arrow, is the return type of the function. This function signature reads as *this function takes a Customer and a decimal as inputs and returns a decimal as output*. This is not strictly true as you will discover in the next chapter. Your IDE will normally show you the function signature or you can see it by hovering your mouse over the function name.

> **Function Signatures** are **very important**, as you will discover in the next chapter; get used to looking at them.

The input parameters of the *calculateTotal* function are in curried form. Curried parameters means that there is more than one of them. If a function signature has more than one arrow, then you have curried parameters. We can also arrange them in tupled form:

```
// (Customer * decimal) -> decimal
let calculateTotal (customer:Customer, spend:decimal) : decimal =
    let discount =
        if customer.IsEligible && spend >= 100.0M
        then (spend * 0.1M) else 0.0M
    let total = spend - discount
    total
```

Note the change in the function signature from **Customer -> decimal -> decimal** to **(Customer * decimal) -> decimal**. The body of the function remains the same, it's just the input parameters that have changed and become a single tupled parameter. There are significant benefits to using the curried form for most F# functions, so that is the style that you will mostly see used in this book. We will discover some of the benefits and where the name came from in the next chapter.

The F# Compiler supports a feature called **Type Inference** which means that most of the time it can determine types through usage without you needing to explicitly define them. As a consequence, we can re-write the function as:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        if customer.IsEligible && spend >= 100.0M then spend * 0.1M
        else 0.0M
    spend - discount
```

The IDE should still display the function signature as **Customer -> decimal -> decimal**. I also removed the total binding as I don't think it adds anything to the readability of the function.

> Don't forget to highlight the code you've written so far and press ALT + ENTER to run it in F# Interactive (FSI).

Whilst type inference works extremely well, there are some situations where it doesn't and you will need to provide a type to a parameter. One of the cases is where you are using a .NET function like `DateTime.TryParse`. The problem here is that that function has two overrides and the compiler can't determine which one it needs to use.

Now that we have finished making changes to the function, we need to create a customer from our specification below the *calculateTotal* function and run in FSI:

```
let john = { Id = "John"; IsEligible = true; IsRegistered = true }
```

Rather than write a formal test, we can use FSI to run simple verifications for us. We will look at writing proper unit tests in chapter 4. Write the following after the *john* binding:

```
let assertJohn = (calculateTotal john 100.0M = 90.0M)
```

Highlight all of the code and press ALT + ENTER to run the code in FSI. What you should see is the following:

```
val assertJohn : bool = true
```

Add in the other users and test cases from the specification in the same way:

```
let john = { Id = "John"; IsEligible = true; IsRegistered = true } // Customer
let mary = { Id = "Mary"; IsEligible = true; IsRegistered = true } // Customer
let richard = { Id = "Richard"; IsEligible = false; IsRegistered = true } // Customer
let sarah = { Id = "Sarah"; IsEligible = false; IsRegistered = false } // Customer

let assertJohn = (calculateTotal john 100.0M = 90.0M) // bool
let assertMary = (calculateTotal mary 99.0M = 99.0M) // bool
let assertRichard = (calculateTotal richard 100.0M = 100.0M) // bool
let assertSarah = (calculateTotal sarah 100.0M = 100.0M) // bool
```

Multi-purpose = operator

There is no such thing as == or even === in F#. We use = for binding, setting property values, and equality. To update mutable bindings, we use the <- operator:

```
let mutable myInt = 0
myInt = 1 // false [equality check]
myInt <- 1 // myInt = 1 [assignment]
myInt = 1 // true
```

Notice that we have to explicitly define a binding as mutable.

Highlight the new code and press ALT + ENTER. You should see the following in FSI.

```
val assertJohn : bool = true
val assertMary : bool = true
val assertRichard : bool = true
val assertSarah : bool = true
```

We don't actually need to use the parentheses:

```
let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

To add clarity, we could add a new function that would perform the equality check for us:

```
// 'a -> 'a -> bool (' means generic)
let areEqual expected actual =
        actual = expected

let assertJohn = areEqual 90.0M (calculateTotal john 100.0M)
let assertMary = areEqual 99.0M (calculateTotal mary 99.0M)
let assertRichard = areEqual 100.0M (calculateTotal richard 100.0M)
let assertSarah = areEqual 100.0M (calculateTotal sarah 100.0M)
```

The *areEqual* function is generic because the compiler has determined that any two values of the same type could be compared using the equality operator.

Using the first version of the asserts, our code should now look like this:

```
type Customer = {
    Id : string
    IsEligible : bool
    IsRegistered : bool
}

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        if customer.IsEligible && spend >= 100.0M then spend * 0.1M
        else 0.0M
    spend - discount

let john = { Id = "John"; IsEligible = true; IsRegistered = true }
let mary = { Id = "Mary"; IsEligible = true; IsRegistered = true }
let richard = { Id = "Richard"; IsEligible = false; IsRegistered = true }
let sarah = { Id = "Sarah"; IsEligible = false; IsRegistered = false }

let assertJohn = (calculateTotal john 100.0M = 90.0M)
let assertMary = (calculateTotal mary 99.0M = 99.0M)
let assertRichard = (calculateTotal richard 100.0M = 100.0M)
let assertSarah = (calculateTotal sarah 100.0M = 100.0M)
```

Whilst this code works, having boolean properties representing domain concepts is not the most robust approach. In addition, it is possible to be in an invalid state where a customer could be eligible but not registered. To prevent this, we could have added a check for `customer.IsRegistered = true` but it is easy to forget to do this. Instead, we will take advantage of the F# type system and make domain concepts like *Registered* and *Unregistered* explicit.

Making the Implicit Explicit

Create a new file called 'part2.fsx' and copy the final code from part1.fsx into it. We are going to modify the code in part2.fsx in this section.

Firstly, we create specific record types for *Registered* and *Unregistered* customers.

```
type RegisteredCustomer = {
    Id : string
    IsEligible : bool
}

type UnregisteredCustomer = {
        Id : string
}
```

To represent the fact that a customer can be either *Registered* or *Unregistered*, we will use another of the built-in types in the Algebraic Type System: the Discriminated Union (DU). We define the Customer type like this:

```
type Customer =
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
```

This reads as "a customer can either be *Registered* of type *RegisteredCustomer* or a *Guest* of type *UnregisteredCustomer*". The items are called Union Cases and consist of a Case Identifier, in this example *Registered*/*Guest*, and some optional Case Data. You can optionally attach any type or mixture of types to a union case.

Discriminated unions are closed sets, so only the cases described in the type definition are available. The only place that cases can be defined is in the type definition.

The easiest way to understand a discriminated union is to use them! We have to make changes to the users that we have defined. Firstly the *UnregisteredCustomer*:

```
// Guest of UnregisteredCustomer
let sarah = Guest { Id = "Sarah" } // Customer
```

In this example, *sarah* is still a Customer but you cannot create a Customer, you have to be one of the cases: Registered or Guest.

Look at how the definition in the discriminated union compares to the case definition.

Now let's make the required changes to the *RegisteredCustomer* bindings:

```
// Registered of RegisteredCustomer
let john = Registered { Id = "John"; IsEligible = true }
let mary = Registered { Id = "Mary"; IsEligible = true }
let richard = Registered { Id = "Richard"; IsEligible = false }
```

Changing the *Customer* type to a discriminated union from a record also has an impact on the *calculateTotal* function. We will modify the discount calculation using another F# feature: Pattern Matching using a match expression:

```
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered c ->
            if c.IsEligible && spend >= 100.0M then spend * 0.1M else 0.0M
        | Guest _ -> 0.0M
    spend - discount
```

We are using a match expression to determine the case of the *Customer*.

> **Expressions** produce an output value. Pretty much everything in F# including values, functions, and control flows is an expression. This allows them to be easily composed.

To understand what the pattern match is doing, compare the match `Registered c` with how we constructed the users: `Registered { Id = "John"; IsEligible = true }`. In this case, `c` is a placeholder for the *RegisteredCustomer* case data. The underscore in the *Guest* pattern match is a wildcard that implies that we don't need access to that case data. Pattern matching against discriminated unions is exhaustive; that is, every case must be handled by the match expression. If you don't handle every case, you will get a warning from the compiler saying 'incomplete pattern match'.

We can simplify the logic with a guard clause but it does mean that we need to account for non-eligible Registered customers otherwise the match is incomplete:

```
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered c when c.IsEligible && spend >= 100.0M -> spend * 0.1M
        | Registered _ -> 0.0M
        | Guest _ -> 0.0M
    spend - discount
```

We can simplify the last two matches using a wildcard like this:

```
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered c when c.IsEligible && spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

> Be careful when using the wildcard like this because it may prevent the compiler from warning you of additions to the discriminated union and could result in unexpected outcomes.

The tests don't need to change.

This is much better than the naive version we had before. It's easier to understand the logic and you can no longer create data in an invalid state. Does it get better if we make eligibility explicit as well? Let's see!

Going Further

Create a new file called *part3.fsx* and copy the final code from part2 into it. We are going to modify the code in *part3.fsx* in this section.

Now we are going to make eligibility a real domain concept. We remove the *IsEligible* flag from *RegisteredCustomer* and add *Eligible* to the *Customer* type.

```
type RegisteredCustomer = {
    Id : string
}

type UnregisteredCustomer = {
    Id : string
}

type Customer =
    | Eligible of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
```

We need to make a change to our function.

```
let calculateTotal customer spend =
    let discount =
        match customer with
        | Eligible _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

We no longer need to test for IsEligible and we also no longer need access to the instance, so we can replace the 'c' with an underscore (wildcard).

We need to make some minor changes to two of our customer bindings:

```
let john = Eligible { Id = "John" }
let mary = Eligible { Id = "Mary" }
```

Run your code in FSI to check all is still OK.

The state of our code after all of our improvements is:

```
type RegisteredCustomer = {
        Id : string
}

type UnregisteredCustomer = {
    Id : string
}

type Customer =
    | Eligible of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer

let calculateTotal customer spend =
    let discount =
        match customer with
        | Eligible _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount

let john = Eligible { Id = "John" }
let mary = Eligible { Id = "Mary" }
let richard = Registered { Id = "Richard" }
let sarah = Guest { Id = "Sarah" }
```

```
let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

I think that this is a worthwhile improvement over where we started as readability has improved and we have prevented getting into invalid states. However, we can still do better by replacing primitives with domain concepts; we will revisit this in a later chapter. Another improvement we will make is to add unit testing with XUnit where we can make use of the helpers and assertions we've already written.

Now that the *RegisteredCustomer* and *UnregisteredCustomer* records are really simple, we could remove them and add the string Id value directly to the case values in the Customer discriminated union:

```
type Customer =
    | Eligible of Id:string
    | Registered of Id:string
    | Guest of Id:string

let calculateTotal customer spend =
    let discount =
        match customer with
        | Eligible _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount

let john = Eligible "John"
let mary = Eligible "Mary"
let richard = Registered "Richard"
let sarah = Guest "Sarah"

let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

I chose not to do this as it's likely that the *RegisteredCustomer* case data would contain more data than we currently see. That doesn't stop you from doing it only to the *UnregisteredCustomer* record type.

This was one approach to modelling this simple domain but we can do it in a few other styles as well.

Alternative Approaches (1 of 2)

Create a new file called *part4.fsx* and copy the final code from *part3.fsx* into it. We are going to modify the code in *part4.fsx* in this section.

An alternative approach would be to use a discriminated union as a type in a property of a record type:

```
type CustomerType =
    | Registered of IsEligible:bool
    | Guest

type Customer = { Id:string; Type:CustomerType }

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer.Type with
        | Registered (IsEligible = isEligible) when isEligible && spend >= 100.0M ->\
 spend * 0.1M
        | _ -> 0.0M
    spend - discount

let john = { Id = "John"; Type = Registered (IsEligible = true) }
let mary = { Id = "Mary"; Type = Registered (IsEligible = true) }
let richard = { Id = "Richard"; Type = Registered (IsEligible = false) }
let sarah = { Id = "Sarah"; Type = Guest }

let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

This is perfectly valid but it means introducing language that isn't in our current understanding of the domain.

We can simplify the filter like this:

```
let calculateTotal customer spend =
    let discount =
        match customer.Type with
        | Registered (IsEligible = true) when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

It solves the illegal state issue nicely but I don't like having to invent names. I much prefer to make my code as domain-centric as possible.

Alternative Approaches (2 of 2)

Remember when we discounted tuples earlier as they don't allow the individual parts to be named? Discriminated unions have their own data structures that solve that. Although they look like tuples, the data structures of discriminated unions are not tuples.

Create a new file called *part5.fsx* and copy the final code from *part2.fsx* into it. We are going to modify the code in *part5.fsx* in this section.

Let's start by modifying the *Customer* type and the *calculateTotal* function:

```
type Customer =
    | Registered of Id:string * IsEligible:bool
    | Guest of Id:string

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered (id, isEligible) when isEligible && spend >= 100.0M -> spend * \
0.1M
        | _ -> 0.0M
    spend - discount
```

You don't need to touch the asserts but we do need to change the let bindings to support the new Customer data structure:

```
let john = Registered (Id = "John", IsEligible = true)
let mary = Registered (Id = "Mary", IsEligible = true)
let richard = Registered (Id = "Richard", IsEligible = false)
let sarah = Guest (Id = "Sarah")
```

Highlight all of the code in this script file and run it in FSI to verify that the asserts all return true.

If we don't need the value of the Id property, we can use a wildcard to ignore it:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered (_, isEligible) when isEligible && spend >= 100.0M -> spend * 0\
.1M
        | _ -> 0.0M
    spend - discount
```

It is also possible to simplify the filter as we have done in previous versions:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered (IsEligible = true) when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

If you need the value of the Id property, we can ask for it at the same time as applying the `IsEligible = true` filter:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered (Id = id; IsEligible = true) when spend >= 100.0M -> spend * 0.\
1M
        | _ -> 0.0M
    spend - discount
```

If you are looking at some of the filters we've written and are wondering if they could be extracted out to some kind of re-usable code, then you are in luck! The feature you are looking for is called Active Patterns. We will spend a whole chapter looking at them later in the book but this is an example of one being used in this version of our code:

```
// Customer -> unit option
let (|IsEligible|_|) customer =
    match customer with
    | Registered (IsEligible = true) -> Some ()
    | _ -> None

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | IsEligible when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

You will meet Option, Some, None, and Unit in chapter 3 and Active Patterns in chapter 7. I just wanted to give you a sneak preview of some of the goodies you are going to encounter later in the book.

My guiding principle for domain modelling is to be as domain-centric as possible, whilst making illegal states unrepresentable[16]. Modelling is an inexact task at best. Try out a number of versions to see if there is a better way than you first thought of.

Summary

In this chapter, we have combined types in many possible ways to solve a simple business problem. The types available in the F# Algebraic Type System, despite being simple AND (record and tuple) and OR (discriminated union) structures, can be combined together to build complicated data structures. Even more importantly, it can build incredibly expressive structures that model almost any kind of domain.

Despite the relative simplicity of the code we have created, we have covered quite a lot in this chapter:

- F# Interactive (FSI)
- Algebraic Type System
  - Tuples
  - Record Types
  - Discriminated Union
  - Type Composition
- Pattern Matching
  - Match Expression

[16] https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable/

    - Guard clause
- Let bindings
- Functions
- Function Signatures

In the next chapter, we will start to look at function composition: Composing bigger functions out of smaller ones.

Postscript

To illustrate the portability of the functional programming concepts we have covered in this post, one of my colleagues, Daniel Weller, wrote a Scala version of one of the solutions:

```
sealed trait Customer

case class Registered(id : String) extends Customer
case class EligibleRegistered(id : String) extends Customer
case class Guest(id: String) extends Customer

def calculateTotal(customer: Customer)(spend: Double) = {
    val discount = customer match {
        case EligibleRegistered(_) if spend >= 100.0 => spend * 0.1
        case _ => 0.0
    }
    spend - discount
}

val john = EligibleRegistered("John")
val assertJohn = (calculateTotal (john) (100.0)) == 90.0
```

As you can see, it is fairly easy to see our solution in this unfamiliar Scala code.

You can find his code here -> https://gist.github.com/frehn

2 - Functions

In this chapter, we are going to concentrate on functions. We will introduce the practice of composing pipelines of functions into functions that can perform more complicated tasks, commonly referred to as function composition. We will also see why understanding function signatures is very important in F# programming.

Getting Started

Functions in F# are fundamentally very simple:

> **Functions have one rule: They take one input and return one output.**

In the previous chapter, we wrote a function that had two input parameters. Later in this chapter, we will discover why that fact and the one input parameter rule for functions are not actually conflicting.

In this chapter, we are going to concentrate on a special subset of functions: Pure functions.

> A **Pure Function** has the following rules:
>
> - They are deterministic. Given the same input, they will always return the same output.
> - They generate no side effects.
>
> **Side Effects** are activities such as talking to a database, sending email, handling user input, and random number generation or using the current date and time. It is highly unlikely that you will ever write a program that is free of side effects.

Functions that satisfy these rules have many benefits: They are easy to test, cacheable, and parallelizable.

You can do a lot in a single function but you can have better code reusability by combining smaller functions together: We call this Function Composition.

Theory

The composition of two functions relies on the output of the first one matching the input of the next.

If we have two functions (f1 and f2) that look like this pseudocode:

```
// The ' implies a generic (any) type
f1 : 'a -> 'b
f2 : 'b -> 'c
```

As the output of f1 matches the input of f2, we can combine them together to create a new function (f3):

```
f3 : f1 >> f2 // 'a -> 'c
```

Treat the composition operator >> as a general-purpose operator for composing any two functions.

What happens if the output of f1 does not match the input of f2?:

```
f1 : 'a -> 'b
f2 : 'c -> 'd
where 'b is not the same type as 'c
```

To resolve this, we would create an adaptor function, or use an existing one, that we can plug in between f1 and f2:

```
f3 : 'b -> 'c
```

After plugging f3 in, we get a new function f4:

```
f4 : f1 >> f3 >> f2 // 'a -> 'd
```

Any number of functions can be composed together in this way.

Let's look at a concrete example.

In Practice

I've taken and slightly simplified some of the code from Jorge Fioranelli's excellent F# Workshop[17]. Once you've finished this chapter, I suggest that you download the workshop (it's free!) and complete it.

This example has a simple record type and three functions that we can compose together because the function signatures match up.

[17] http://www.fsharpworkshop.com/

```
type Customer = {
    Id : int
    IsVip : bool
    Credit : decimal
}

// Customer -> (Customer * decimal)
let getPurchases customer =
    let purchases = if customer.Id % 2 = 0 then 120M else 80M
    (customer, purchases) // Parentheses are optional

// (Customer * decimal) -> Customer
let tryPromoteToVip purchases =
    let (customer, amount) = purchases
    if amount > 100M then { customer with IsVip = true }
    else customer

// Customer -> Customer
let increaseCreditIfVip customer =
    let increase = if customer.IsVip then 100M else 50M
    { customer with Credit = customer.Credit + increase }
```

There are a few things in this sample code that we haven't seen before. The *getPurchases* function returns a tuple. Tuples can be used for transferring small chunks of data around, generally as the output from a function. Notice the difference between the definition of the tuple `(Customer * decimal)` and the usage `(customer, amount)` when we decompose it into its constituent parts.

The other new feature is the copy-and-update record expression. This allows you to create a new record instance based on an existing record, usually with some modified data. Records and their properties are immutable by default.

These are some of the ways that F# supports to compose functions together:

```
// Composition operator
let upgradeCustomerComposed = // Customer -> Customer
    getPurchases >> tryPromoteToVip >> increaseCreditIfVip

// Nested
let upgradeCustomerNested customer = // Customer -> Customer
    increaseCreditIfVip(tryPromoteToVip(getPurchases customer))

// Procedural
let upgradeCustomerProcedural customer = // Customer -> Customer
    let customerWithPurchases = getPurchases customer
```

```
    let promotedCustomer = tryPromoteToVip customerWithPurchases
    let increasedCreditCustomer = increaseCreditIfVip promotedCustomer
    increasedCreditCustomer

// Forward pipe operator
let upgradeCustomerPiped customer = // Customer -> Customer
    customer
    |> getPurchases
    |> tryPromoteToVip
    |> increaseCreditIfVip
```

They all have the same function signature and will produce the same result with the same input.

The *upgradeCustomerPiped* function uses the forward pipe operator (|>). It is equivalent to the *upgradeCustomerProcedural* function but without having to specify the intermediate values. The value from the line above gets passed down as the last input argument of the next function. The difference between the function composition operator `>>` and the forward pipe operator `|>` is that the function composition operator sits between two functions, whereas the forward pipe sits between a value and a function.

> Use the forward pipe operator (|>) as your default style.

It is quite easy to verify the output of the upgrade functions using FSI.

```
let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

let assertVIP =
    upgradeCustomerComposed customerVIP = {Id = 1; IsVip = true; Credit = 100.0M }
let assertSTDtoVIP =
    upgradeCustomerComposed customerSTD = {Id = 2; IsVip = true; Credit = 200.0M }
let assertSTD =
    upgradeCustomerComposed { customerSTD with Id = 3; Credit = 50.0M } = {Id = 3; I\
sVip = false; Credit = 100.0M }
```

Record types use **structural equality** which means that if the records look the same, they are equal.

> If two records contain the same data, they have structural equality through the equality operator (=).

Try replacing the *upgradeCustomerComposed* function with any of the other three functions in the asserts to confirm that they produce the same results in FSI.

Unit

All functions must have one input and one output. To solve the problem of a function that doesn't require an input value or produce any output, F# has a special type called unit.

> Any function taking unit as the only input or returning it as the output is probably causing side effects

```
open System

// unit -> System.DateTime
let now () = DateTime.UtcNow

// 'a -> unit
let log msg =
    // Log message or similar task that doesn't return a value
    ()
```

Unit appears in the function signature as `unit` but in code, you will use `()`.

If you call the *now* function without the unit () parameter you will get a result similar to this:

```
val it : (unit -> DateTime) = <fun:it@30-4>
```

This looks really strange but what you actually get is a function as the output that takes unit and returns a DateTime value as shown by the signature. To execute the function, you need to supply the input parameter. This is an important F# feature called Partial Application, which we will explain later in this chapter.

If you forget to add the unit parameter `()` when you define the binding, you get a fixed value from the first time the line is executed:

```
let fixedNow = DateTime.UtcNow

let theTimeIs = fixedNow
```

Wait a few seconds and run the *theTimeIs* binding again and you'll notice that the date and time haven't changed from when the binding was first used.

> A function binding always has at least one parameter, even if it is just unit, otherwise, it is a value binding. You can tell if it's a function by checking whether the signature has an arrow (->) or not.

Anonymous Functions

So far, we have only dealt with named functions but we can also create functions without names, commonly called anonymous functions. We'll start with a simple named function:

```
// int -> int -> int
let add x y = x + y

let sum = add 1 4
```

We can rewrite the `add` function to use a lambda (`->`):

```
// int -> int -> int
let add = fun x y -> x + y

let sum = add 1 4
```

Notice that the signatures remain the same (`int -> int -> int`). If you're wondering why the compiler has inferred that the inputs and output are integers, it is because the default for `+` usage is with integers.

Functions are first-class citizens in F# which means that we can use them like other values as input parameters for functions. In the following function, `f` is a function:

```
// ('a -> 'b -> 'c) -> 'a -> 'b -> 'c
let apply f x y = f x y
```

The compiler has decided that this is a generic function that can take any function with two input parameters. Let's call the `apply` function with the `add` function and two input values:

```
let sum = apply add 1 4
```

We can also pass an anonymous function with the same signature as `add` into the function:

```
let sum = apply (fun x y -> x + y) 1 4
```

Anonymous functions are used throughout F# because they save us having to write lots of small functions to do simple, one-off tasks.

We will look much more closely at this in Chapter 3 when we will be introduced to Higher-Order functions.

Although the example we used is a bit artificial, anonymous functions are very useful in many ways. If we create a function that returns a random number between 0 and 99:

```
open System

// unit -> int
let rnd () =
    let rand = Random()
    rand.Next(100)

List.init 50 (fun _ -> rnd())
```

The underscore (_) tells the compiler that we can ignore that value because we are not going to use it.

This will create a new instance of the Random class every time that it is called when the list is created.

The version below uses scoping rules to re-use the same instance of the Random class each time you call the *rnd* function:

```
open System

// unit -> int
let rnd =
    let rand = Random()
    fun () -> rand.Next(100)

List.init 50 (fun _ -> rnd())
```

Multiple Parameters

At the start of the chapter, I stated that it is a rule that all functions must have one input and one output but in the last chapter, we created a function with multiple input parameters:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend = ...
```

Let's write the function signature using an anonymous function:

```
// Customer -> (decimal -> decimal)
let calculateTotal customer =
        fun spend -> (...)
```

The *calculateTotal* function takes a Customer as input and returns a function with the signature (decimal -> decimal) as output; That function takes a decimal as input and returns a decimal as output. The ability to automatically chain single input/single output functions together like this is called **Currying** after Haskell Curry, a US Mathematician. It allows you to write functions that appear to have multiple input arguments but are actually a chain of one input/one output functions. This seemingly odd behaviour that F# performs under the hood for us, opens the way to another very powerful functional concept called **Partial Application**.

Partial Application (Part 1)

We will use the *calculateTotal* function to illustrate how partial application works. We can ignore the actual implementation of the function as we only care about the input parameters:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend = ...
```

If I only provide the first parameter, I get a new function as output with signature (Decimal -> Decimal):

```
// Decimal -> Decimal
let partial = calculateTotal john
```

If we then supply the final parameter, the original function completes and returns the expected output:

```
// Decimal
let complete = partial 100.0M
```

Run the code in FSI and look at the outputs from both lines of code.

You must add the input parameters in strict left to right order, in ones or multiples. Trying to add a parameter out of order will not compile if the types don't match:

```
let doesNotWork = calculateTotal 100M // Does not compile
```

You may wonder why you would want to do this but the partial application of function parameters has some very important uses such as allowing the forward pipe operator `|>` we used earlier in this chapter.

The Forward Pipe Operator

As the forward pipe operator is something that we will be using a lot, it's worth spending some time investigating what it is and how it works under the covers. Hint: It relies on partial application.

We saw that we could take the `calculateTotal` function:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend = ...
```

By only applying the first input parameter, we can create a partially applied function which we can complete by applying the missing input parameter:

```
// Decimal -> Decimal
let partial = calculateTotal john

// Decimal
let complete = partial 100.0M
```

We can do the same thing without the need for the named function by using the forward pipe operator:

```
let complete = 100.0M |> calculateTotal john
```

The value to the left of the forward pipe operator is applied as the missing parameter to the partially applied function on the right of the operator.

The simple asserts that we wrote in the last chapter, can be made more readable than this example:

```
let assertJohn = calculateTotal john 100.0M = 90.0M
```

We added a simple helper function and used that:

```
// 'a -> 'a -> bool (' means generic in F#)
let areEqual expected actual =
        expected = actual

let assertJohn = areEqual 90.0M (calculateTotal john 100.0M)
```

This works nicely but it would be nice if the assertion read as `calculateTotal john 100.0M` is equal to `90.0M`. We can't quite do this but using the forward pipe operator we can get quite close:

```
// 'a -> 'a -> bool
let isEqualTo expected actual =
    expected = actual

let assertJohn = calculateTotal john 100.0M |> isEqualTo 90.0M
```

You don't have to change the name of the helper function but we did in this case as it makes it read more like an English sentence.

In this example, `isEqualTo` is partially applied as part of the forward pipe because only one of the two input parameters that it needs is provided. The other parameter is passed through by the forward pipe operator. In this case, we are passing the result of a function to the right-hand side.

So what is the forward pipe operator (`|>`)? It is defined as part of FSharp.Core but it looks something like this:

```
// 'a -> ('a -> 'b) -> 'b
let (|>) v f = f v
```

Despite this being very short, there is a lot to unpack here.

This looks like a function and it is but it's a special category called a custom operator. The name of the function is `|>` but the parentheses surrounding it are required because it can be used in two different ways which we will see shortly.

The operator takes a value of type *'a* as the first parameter and a function that takes an *'a* and returns a *'b* as the second parameter. In the scope of the forward pipe, the function is considered to be partially applied because it has not had all of its required parameters applied until the one is passed from the left of the forward pipe.

This operator is a higher-order function because it takes a function as a parameter.

> **Higher-Order Functions** take one or more functions as parameters and/or return a function as its output.

When the value on the left of the forward pipe operator is applied to the partially applied function on the right, it has all of the parameters it needs and can execute to return a value of type *'b*.

Let's use the operator in our assert in the defined prefix form:

```
// decimal -> (decimal -> bool) -> bool
let assertJohn = (|>) (calculateTotal john 100.0M) (isEqualTo 90.0M)
```

Normally we would skip this step as it looks worse that our original version but we can convert our operator from the current prefix form into the infix form like this:

```
// decimal -> (decimal -> bool) -> bool
let assertJohn = (calculateTotal john 100.0M) |> (isEqualTo 90.0M)
```

Notice that as an infix operator, you don't use the parentheses.

The last part is to remove the unnecessary parentheses:

```
// decimal -> (decimal -> bool) -> bool
let assertJohn = calculateTotal john 100.0M |> isEqualTo 90.0M
```

The `isEqualTo` function has a signature of `decimal -> decimal -> bool`, so if we apply the first input parameter, it now has the signature of `decimal -> bool`. When you apply the decimal result from the `calculateTotal` function, the `isEqualTo` function has all of its input parameters and executes producing the `bool` result.

Understanding how the forward pipe operator works is important as we will be using it throughout the book. In reality, to use it, you only need to remember that the value on the left of the operator is applied as the last parameter to the function on the right of the operator.

Partial Application (Part 2)

Now that we have a better understanding of partial application, let's have a look at another example. Create a function that takes the log level as a discriminated union and a string for the message:

```
type LogLevel =
    | Error
    | Warning
    | Info

// LogLevel -> string -> unit
let log (level:LogLevel) message =
    printfn "[%A]: %s" level message
    ()
```

Every function in F# must return an output. In this case, we added an extra line to output *unit*. However, *printfn* returns *unit*, so we can remove the additional row:

```
// LogLevel -> string -> unit
let log (level:LogLevel) message =
    // or string interpolation printfn $"[%A{level}]: %s{message}"
    printfn "[%A]: %s" level message
```

Both *printfn* and string interpolation have support for format specifiers. If the types supplied don't match the format specifier, it will result in a compile error rather than a runtime error. You don't need to supply format specifiers for string interpolation but then you don't get type safety from the format specifiers.

```
// LogLevel -> string -> unit
let log (level:LogLevel) message =
    printfn "[%A]: %s" level message

// LogLevel -> string -> unit
let log (level:LogLevel) message =
    printfn $"[%A{level}]: %s{message}"

// LogLevel -> 'a -> unit
let log (level:LogLevel) message =
    printfn $"[{level}]: {message}"
```

To partially apply the *log* function, I'm going to define a new function that only takes the *log* function and its level argument but not the message:

```
let logError = log Error // string -> unit
```

The name *logError* is bound to a function that takes a *string* and returns *unit*. So now, we can use the *logError* function instead:

```
let m1 = log Error "Curried function"

let m2 = logError "Partially Applied function"
```

As the return type is *unit*, you don't have to let bind the result of the function to a value:

```
log Error "Curried function"

logError "Partially Applied function"
```

Partial application is a very powerful concept that is only made possible because of curried input parameters. It is not possible with a single, tupled parameter as you need to supply the whole parameter at once:

```
type LogLevel =
    | Error
    | Warning
    | Info

// (LogLevel * string) -> unit
let log (level:LogLevel, message:string) =
    printfn "[%A]: %s" level message
```

There's so much more that we could cover here but we've already covered a lot in this chapter and we've got the rest of the book to fit them in!

Summary

In this chapter, we have covered:

- Pure functions
- Anonymous functions
- Function composition
- Tuples
- Copy-and-update record expression
- Curried and tupled parameters
- Currying and partial application

We have now covered the fundamental building blocks of programming in F#: Composition of types and functions, expressions, and immutability.

In the next chapter, we will investigate the handling of null and exceptions in F#.

3 - Null and Exception Handling

In this chapter, we will investigate how we handle nulls and exceptions in F#. In addition, we'll be extending our understanding of function composition that we looked at in the previous chapter and will start looking at higher-order functions.

> **Higher-Order Functions** take one or more functions as arguments and/or return a function as its output.

Null Handling

Most of the time, you will not have to deal with null in your F# code as it has a built-in type called Option that you will use instead. It looks very similar to this:

```
type Option<'T> =
    | Some of 'T
    | None
```

It is a discriminated union with two cases to handle whether there is data or not. The single tick (') before the T is the F# way of showing that T is a Generic type. As a consequence, any type can be treated as optional.

> You must either delete this Option type definition or comment it out as it already exists in the F# language and it will interfere with the correct compilation of the code if you don't.

Create a new file in your folder called *option.fsx*.

Don't forget to highlight and run the code examples in this chapter in F# Interactive (FSI).

We'll start by creating a function to try to parse a string as a DateTime:

```
open System

// string -> Option<DateTime>
let tryParseDateTime (input:string) =
    let (success, value) = DateTime.TryParse input
    if success then Some value else None
```

We need to make an import declaration to the top of the page using the `open` keyword to provide access to the .NET DateTime functionality.

If you are wondering how `DateTime.TryParse` returns a tuple, the answer is quite simple; the F# language does not support *out* parameters, so the clever folks behind F# made it so that during interop, *out* parameters are added to the function output, which generally results in a tuple.

You can also pattern match the result of the function with a match expression instead of using an if expression:

```
// string -> Option<DateTime>
let tryParseDateTime (input:string) =
    match DateTime.TryParse input with
    | true, result -> Some result
    | false, _ -> None
```

The wildcard symbol _ implies the value can be anything.

```
// string -> Option<DateTime>
let tryParseDateTime (input:string) =
    match DateTime.TryParse input with
    | true, result -> Some result
    | _, _ -> None
```

The wildcard match `_, _` can be simplified to a single wildcard:

```
// string -> Option<DateTime>
let tryParseDateTime (input:string) =
    match DateTime.TryParse input with
    | true, result -> Some result
    | _ -> None
```

Any of these versions are acceptable but in this case, I would choose either the if expression or the first pattern match as they are the easiest to discern their intent.

Run the following examples in FSI with your chose version of the function:

```
let isDate = tryParseDateTime "2019-08-01" // Some 01/08/2019 00:00:00

let isNotDate = tryParseDateTime "Hello" // None
```

You will see that the string that can be parsed into a valid date returns Some of the valid date and the non-date string returns None.

Another way that the Option type can be used is for optional data like a person's middle name as not everyone has one:

```
type PersonName = {
    FirstName : string
    MiddleName : Option<string>
    LastName : string
}
```

If the person doesn't have a middle name, you set it to None and if they do you set it to Some "name", as shown here:

```
let person = { FirstName = "Ian"; MiddleName = None; LastName = "Russell"}

let person2 = { person with MiddleName = Some "????" }
```

Notice that we have used the copy-and-update record expression we met in the last chapter.

So far, we have used the `Option<'T>` style but there is an `option` keyword that we can use instead:

```
type PersonName = {
    FirstName : string
    MiddleName : string option
    LastName : string
}
```

You will tend to see this style used but neither is wrong.

Sadly, there is one area where nulls can sneak into your codebase and that is through interop with code/libraries written in other .Net languages such as C# including most of the .NET platform.

> **What is.NET?**
>
> .NET is a free, cross-platform, open-source developer platform for building many different types of applications.

Interop With .NET

If you are interacting with code written in C#, there is a chance that you will have some null issues. In addition to the Option type, F# also offers the *Option* module that contains some very useful helper functions to make life easier.

Let's create a null for both a Reference type and a Nullable primitive:

```
open System

// Reference type
let nullObj:string = null

// Nullable type
let nullPri = Nullable<int>()
```

Run the code in FSI to prove that they are both null.

To convert from .Net to an F# Option type, we can use the `Option.ofObj` and `Option.ofNullable` functions:

```
let fromNullObj = Option.ofObj nullObj

let fromNullPri = Option.ofNullable nullPri
```

To convert from an Option type to .Net types, we can use the `Option.toObj` and `Option.toNullable` functions.

```
let toNullObj = Option.toObj fromNullObj

let toNullPri = Option.toNullable fromNullPri
```

Run the code in FSI to show that this works correctly:

What happens if you want to convert from an *Option* type to something that doesn't support null but instead expects a placeholder value? You could use pattern matching as Option is a discriminated union or you can use the `Option.defaultValue` function:

```
let resultPM input =
    match input with
    | Some value -> value
    | None -> "------"

let resultDV = Option.defaultValue "------" fromNullObj
```

If the *Option* value is Some, then the value wrapped by Some is returned, otherwise, the default value, in these cases, "——", is returned.

You could also use the forward pipe operator (|>):

```
let resultFP = fromNullObj |> Option.defaultValue "------"

let resultFPA =
    fromNullObj
    |> Option.defaultValue "------"
```

If you use this a lot, you may find that using Partial Application might make the task more pleasurable. We create a function that takes the default but not the Option value:

```
// (string option -> string)
let setUnknownAsDefault = Option.defaultValue "????"

let result = setUnknownAsDefault fromNullObj
```

Or using the forward pipe operator:

```
let result = fromNullObj |> setUnknownAsDefault
```

As you can see, handling of null and optional values is handled very nicely in F#. If you are diligent, you should never see a NullReferenceException in a running F# application.

Handling Exceptions

Create a new file called *result.fsx* in your folder.

We will create a function that does simple division but returns an exception if the divisor is 0:

```
open System

// decimal -> decimal -> decimal
let tryDivide (x:decimal) (y:decimal) =
    try
        x/y
    with
    | :? DivideByZeroException as ex -> raise ex
```

Whilst this code is perfectly valid, the function signature is lying to you; It doesn't always return a decimal. The only way I would know this is by looking at the implementation or getting the error when the code is executed. This goes against the general ethos of F# coding; you should be able to trust function signatures.

Most functional languages implement a type that offers a choice between success and failure and F# is no exception. This is an example of a potential implementation:

```
type Result<'TSuccess,'TFailure> =
    | Success of 'TSuccess
    | Failure of 'TFailure
```

Unsurprisingly, there is one built into the language (from F# 4.1) but rather than Success/Failure, it uses Ok/Error. We can delete or comment out our Result type as we will use the one in the F# language.

Let's use the Result type in our `tryDivide` function:

```
// decimal -> decimal -> Result<decimal,exn>
let tryDivide (x:decimal) (y:decimal) =
    try
        Ok (x/y)
    with
    | :? DivideByZeroException as ex -> Error ex
```

The `try...with` expression is used to handle exceptions and is analogous to Try-Catch in C#. In our case, we are only expecting one specific error type: `System.DivideByZeroException`. The cast operator `:?` is a pattern matching feature to match a specified type or subtype, in this case, if the error can be cast as `System.DivideByZeroException`. The `as ex` gives us access to the actual exception instance which we then pass as the case data to construct the *Error* case. Any unexpected exception gets passed up the call chain until something else handles it or it crashes the application, just like the rest of .NET.

Run the code in FSI to see what you get from an error and success cases:

```
// Error "System.DivideByZeroException: Attempted to divide by zero..."
let badDivide = tryDivide 1M 0M

let goodDivide = tryDivide 1M 1M // Some 1M
```

Next, we are going to look at how we can incorporate the Result type into the composition code we used in the last chapter.

Function Composition With Result

I have modified the `getPurchases` and `increaseCreditIfVip` functions to return *Result* types but have left the `tryPromoteToVip` function alone so that we see the impact the changes have on function composition:

```
open System

type Customer = {
    Id : int
    IsVip : bool
    Credit : decimal
}

// Customer -> Result<(Customer * decimal),exn>
let getPurchases customer =
    try
        // Imagine this function is fetching data from a Database
        let purchases =
            if customer.Id % 2 = 0 then (customer, 120M)
            else (customer, 80M)
        Ok purchases
    with
    | ex -> Error ex

// Customer * decimal -> Customer
let tryPromoteToVip purchases =
    let customer, amount = purchases
    if amount > 100M then { customer with IsVip = true }
    else customer

// Customer -> Result<Customer,exn>
let increaseCreditIfVip customer =
    try
```

```
        // Imagine this function could cause an exception
        let increase =
            if customer.IsVip then 100.0M else 50.0M
        Ok { customer with Credit = customer.Credit + increase }
    with
    | ex -> Error ex

let upgradeCustomer customer =
    customer
    |> getPurchases
    |> tryPromoteToVip // Compiler problem
    |> increaseCreditIfVip

let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

let assertVIP =
        upgradeCustomer customerVIP = Ok {Id = 1; IsVip = true; Credit = 100.0M }
let assertSTDtoVIP =
        upgradeCustomer customerSTD = Ok {Id = 2; IsVip = true; Credit = 200.0M }
let assertSTD =
        upgradeCustomer { customerSTD with Id = 3; Credit = 50.0M } = Ok {Id = 3; IsVip = f\
alse; Credit = 100.0M }
```

There is a problem in the `upgradeCustomer` function on the call to the `tryPromoteToVip` function because the function signatures don't match up any longer. To solve this, we are going to use Higher-Order functions.

> **Higher Order Functions** take one or more functions as arguments and/or return a function as its output.

Scott Wlaschin[18] visualises composition with the Result type as two parallel railway tracks which he calls Railway Oriented Programming (ROP), with one track for Ok and one for Error. You travel on the Ok track until you have an error and then you switch to the Error track. He defines the `tryPromoteToVip` function as a one-track function because it doesn't output a Result type and will only execute on the Ok track. This causes an issue because the output from the previous function and input to the next function doesn't match.

The first thing that we need to do is to use a match expression in an anonymous function to unwrap the tuple from the Ok part of the Result returned from the `getPurchases` function:

[18] https://fsharpforfunandprofit.com/rop/

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> fun result ->
        match result with
        | Ok x -> Ok (tryPromoteToVip x)
        | Error ex -> Error ex
    |> increaseCreditIfVip // Compiler problem
```

We have to return a Result from this code, so we wrap the call to `tryPromoteToVip` in an Ok.

We now do the same for the `increaseCustomerCreditIfVip` function. In this case, we don't need to add the Ok as it already returns a result:

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> fun result ->
        match result with
        | Ok x -> Ok (tryPromoteToVip x)
        | Error ex -> Error ex
    |> fun result ->
        match result with
        | Ok x -> increaseCreditIfVip x
        | Error ex -> Error ex
```

It is possible to simplify this code slightly like this:

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> function
        | Ok x -> Ok (tryPromoteToVip x)
        | Error ex -> Error ex
    |> function
        | Ok x -> increaseCreditIfVip x
        | Error ex -> Error ex
```

Either style is fine to use.

What we need to do now is to convert our anonymous functions into named functions. We start with the `tryToPromoteToVip` block. Our new function has the following signature:

```
(Customer * decimal -> Customer) -> Result<Customer * decimal, exn> -> Result<Custom\
er, exn>
```

We are going to call this function `map` for reasons that will become clear later in the section:

```
// (Customer * decimal -> Customer) -> Result<Customer * decimal, exn>
// -> Result<Customer, exn>
let map (tryPromoteToVip:Customer * decimal -> Customer) (result:Result<Customer * d\
ecimal, exn>)
        : Result<Customer, exn> =
    match result with
    | Ok x -> Ok (tryPromoteToVip x)
    | Error ex -> Error ex
```

The `map` function is a higher-order function because it takes the `tryPromoteToVip` function as an input parameter.

We don't use the types specific to `tryPromoteToVip` or the result, so we can make this function take generic parameters:

```
// ('a -> 'b) -> Result<'a,'c> -> Result<'b,'c>
let map (f:'a -> 'b) (result:Result<'a, 'c>) : Result<'b, 'c> =
    match result with
    | Ok x -> Ok (f x)
    | Error ex -> Error ex
```

Although we have left the Error case data identifier as `ex`, it is important to know that now that it is generic, it doesn't have to be an exception and could just as easily be a string or a discriminated union. In fact, it can be any F# type that we choose.

We can remove the types from the function and the compiler will confirm that it is generic:

```
// ('a -> 'b) -> Result<'a,'c> -> Result<'b,'c>
let map f result =
    match result with
    | Ok x -> Ok (f x)
    | Error ex -> Error ex
```

We can easily show this is true by trying the new map function with some code that matches the same pattern, such as the `tryParseDateTime` function we met earlier:

```
// ('a -> 'b) -> Result<'a,'c> -> Result<'b,'c>
let map f result =
    match result with
    | Ok x -> Ok (f x)
    | Error ex -> Error ex

// string -> DateTime option
let tryParseDateTime (input:string) =
    let success, value = DateTime.TryParse input
    if success then Some value else None

// Result<string, exn>
let getResult =
    try
        Ok "Hello"
    with
    | ex -> Error ex

// Result<DateTime option, exn>
let parsedDT = getResult |> map tryParseDateTime
```

If we use the `map` function in our `upgradeCustomer` function we get this:

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> map tryPromoteToVip
    |> fun result ->
        match result with
        | Ok x -> increaseCreditIfVip x
        | Error ex -> Error ex
```

Now we do a similar thing for the `increaseCreditIfVip` function. This slightly different to the map function as we don't need to wrap the output in a result as the `increaseCreditIfVip` already does that. The function will be called `bind` for reasons we will see very soon:

```
// (Customer -> Result<Customer, exn>) -> Result<Customer, exn> -> Result<Customer, \
exn>
let bind (increaseCreditIfVip:Customer -> Result<Customer, exn>) (result:Result<Cust\
omer, exn>) : Result<Customer, exn> =
        match result with
    | Ok x -> increaseCreditIfVip x
    | Error ex -> Error ex
```

As with the `map` function, `bind` is also a higher order function because it takes the `increaseCreditIfVip` function as an input parameter.

We can make `bind` generic as we don't use any of the parameters in the function:

```
// ('a -> Result<'b, 'c>) -> Result<'a, 'c> -> Result<'b, 'c>
let bind (f:'a -> Result<'b, 'c>) (result:Result<'a, 'c>) : Result<'b, 'c> =
        match result with
    | Ok x -> f x
    | Error ex -> Error ex
```

And we can remove the parameter types:

```
// ('a -> Result<'b, 'c>) -> Result<'a, 'c> -> Result<'b, 'c>
let bind f result =
        match result with
    | Ok x -> f x
    | Error ex -> Error ex
```

Let's plug the `bind` function into `upgradeCustomer`:

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> map tryPromoteToVip
    |> bind increaseCreditIfVip
```

The code should now have no compiler warnings. Run the code in FSI and then the asserts to verify the code works as expected.

Written as procedural code, the `upgradeCustomer` function looks like this:

```
let upgradeCustomer customer =
    let purchasedResult = getPurchases customer
    let promotedResult = map tryPromoteToVip purchasedResult
    let increaseResult = bind increaseCreditIfVip promotedResult
    increaseResult
```

The reason for using `map` and `bind` as function names is because the *Result* module in F# has them built in as it is a common requirement. Let's update the `upgradeCustomer` function to use the *Result* module functions rather than our own.

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> Result.map tryPromoteToVip
    |> Result.bind increaseCreditIfVip
```

We can delete our `map` and `bind` functions as we don't need them any longer.

If you want to learn more about this style of programming, I highly recommend Scott's book Domain Modelling Made Functional[19]. Not only does it cover Railway Oriented Programming (ROP) but also lots of very useful Domain-Driven Design information. It's my favourite technical book!

Summary

We've completed another chapter and have covered a lot of additional features and concepts that build upon our existing knowledge.

- Null handling
- Option type and module
- Exception handling
- Result type and module
- Higher Order Functions

In the next chapter, we'll start looking at organising your code into projects and unit testing.

[19] https://pragprog.com/book/swdddf/domain-modeling-made-functional

4 - Organising Code and Testing

In this chapter, we will be looking at how we can use organise our code using .NET Solutions, Projects, Namespaces, and F# Modules. In addition, we'll be writing our first real unit tests in F# with XUnit.

Getting Started

Follow the instructions in the Appendix of this book to create a new .NET solution containing two projects, a console for the code and an XUnit one for the tests.

Once completed, you should see a message like the following in the tests terminal:

```
Passed!  - Failed:     0, Passed:     1, Skipped:     0, Total:     1, Duration: < 1\
 ms - MyProjectTests.dll (net5.0)
```

Open a second Terminal window and navigate to the MyProject folder.

Execute the following command:

```
dotnet run
```

You should see a message like the following in the Terminal:

```
Hello world from F#
```

Solutions and Projects

We have created a Solution to which we added two directories; src for code projects and test for test projects. The test project has to reference the code project to be able to test it.

Adding a Source File

The last thing we need to do is add a new file called *Customer.fs*. We can do this in two ways. You will find it beneficial to know them both when working on F# in VS Code. We can use either the Explorer view (top icon in the toolbar) or the F# Explorer (Look for the F# logo).

Using Explorer:

If you manually add a file, you have to manually include it to the .fsproj file as well so that F# recognises it:

1. Select the *MyProject.fsproj* file.
2. Click on the New File icon and name the file *Customer.fs*.
3. Click on the *MyProject.fsproj* file to edit it. Copy the `<Compile Include="Program.fs" />` entry and paste it above or below the existing entry.
4. Change the file name of the uppermost entry to *Customer.fs* from *Program.fs*.

The *MyProject.fsproj* project file should now contain this:

```
<ItemGroup>
  <Compile Include="Customer.fs" />
  <Compile Include="Program.fs" />
</ItemGroup>
```

> **Ordering is important**
>
> The strict ordering of code within a file is important and so is the ordering of files within a project. Files are compiled from the top down, just as code within a file is.

Using F# Solution Explorer (ionide):

1. Right-click on Program.fs in the MyProject folder.
2. Select the Add File Above menu item and name the file Customer.fs.

Once your new file is created, run 'dotnet build' on the project or solution from the terminal and you will get an error like this:

```
Files in libraries or multiple-file applications must begin with a namespace or modu\
le declaration, e.g. 'namespace SomeNamespace.SubNamespace' or 'module SomeNamespace\
.SomeModule'. Only the last source file of an application may omit any declaration.
```

Namespaces and Modules

Modules are how we logically organise our code in an F# project. Namespaces are used to reduce the chances of naming collisions when referencing code from other projects.

Namespaces

- Can only contain type declarations, import declarations, or modules.
- Span across multiple files.
- Module names must be unique within a namespace even across multiple files.

Modules

- Can contain anything apart from namespaces. You can even nest modules.
- You can have a top-level module instead of a namespace but it must be unique across all files in the project.

To fix our build issue, we are going to go add a namespace to the *Customer.fs* file in the *MyProject* project. Add the namespace declaration to the top of the file:

```
namespace MyProject
```

The name can be anything you like, it doesn't have to match the project name. If you run `dotnet build` in the Terminal again, the error has been fixed.

Let's look at some of the options when using namespaces and modules:

```
namespace MyProject.Customer // Namespace.Namespace

open System

type Customer = {
    Name : string
}

module Domain =

    // string -> Customer
    let create (name:string) =
        { Name = name }

module Db =

    open System.IO

    // Customer -> bool
    let save (customer:Customer) =
        // Imagine this talks to a database
        ()
```

The Customer record type is used by the two modules, so it is defined in the scope of the namespace. The modules have access to the types in the namespace without having to add an import declaration.

There are other combinations of namespace and modules available. Firstly, separate the namespace and module definition:

```
namespace MyApplication // Namespace

open System

module Customer = // Module

    type Customer = {
        Name : string
    }

    module Domain =

        // string -> Customer
        let create (name:string) =
            { Name = name }

    module Db =

        open System.IO

        // Customer -> unit
        let save (customer:Customer) =
            // Imagine this talks to a database
            ()
```

Notice the nesting required for the sub-modules.

We can also use the module keyword at the top level. In this case, *MyApplication* is a namespace and *Customer*, a module:

```
module MyApplication.Customer // Namespace.Module

open System

type Customer = {
    Name : string
}

module Domain =

    // string -> Customer
    let create (name:string) =
        { Name = name }
```

```
module Db =

    open System.IO

    // Customer -> unit
    let save (customer:Customer) =
        // Imagine this talks to a database
        ()
```

A good starting point is to add a namespace to the top of each file using the project name and use modules for everything else.

Do not feel pressured into creating a file per module. It's generally better to keep as much code that changes together in the same file. This means organising your code by feature/domain concept rather than by technical concept as happens in MVC for instance.

Writing Tests

Have a look at the *Tests.fs* file that was created when we generated the test project:

```
module Tests

open System
open Xunit

[<Fact>]
let ``My test`` () =
    Assert.True(true)
```

One of the really nice things about F# for naming things, particularly tests, is that we can use readable sentences if we use the double backticks. When you have test errors, the name of the module and the test appear in the output.

There is generally no need for namespaces in test files as the code is not deployed.

> **WARNING:** If you use the double backtick naming style, do not use any odd chars like [,|-;.]. These used in test names causes the ionide extension to crash. [Correct: 2022-05-20]

Create a new file in the test project called *CustomerTests.fs* and add the following code to it:

```
namespace MyProjectTests

open System
open Xunit

module ``I can group my tests in a module and run`` =

    [<Fact>]
    let ``My first test`` () =
        Assert.True(true)

    [<Fact>]
    let ``My second test`` () =
        Assert.True(true)
```

Run the tests using the Terminal:

```
dotnet test
```

You can also use the test runner built into VS Code.

Change the second test to make it fail.

```
    [<Fact>]
    let ``My second test`` () =
        Assert.True(false)
```

Run the tests and look at the output.

Writing Real Tests

We are going to use the code from chapter 2 to write tests against.

Write the following code into *Customer.fs*:

```
module MyProject.Customer

type Customer = {
    Id : int
    IsVip : bool
    Credit : decimal
}

// Customer -> (Customer * decimal)
let getPurchases customer =
    let purchases = if customer.Id % 2 = 0 then 120M else 80M
    (customer, purchases)

// (Customer * decimal) -> Customer
let tryPromoteToVip purchases =
    let (customer, amount) = purchases
    if amount > 100M then { customer with IsVip = true }
    else customer

// Customer -> Customer
let increaseCreditIfVip customer =
    let increase = if customer.IsVip then 100M else 50M
    { customer with Credit = customer.Credit + increase }

// Customer -> Customer
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> tryPromoteToVip
    |> increaseCreditIfVip
```

We can use the asserts from chapter 2 as the basis of our new tests:

```
let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

let assertVIP =
        upgradeCustomer customerVIP = {Id = 1; IsVip = true; Credit = 100.0M }
let assertSTDtoVIP =
        upgradeCustomer customerSTD = {Id = 2; IsVip = true; Credit = 200.0M }
let assertSTD =
        upgradeCustomer { customerSTD with Id = 3; Credit = 50.0M } = {Id = 3; IsVip = fals\
e; Credit = 100.0M }
```

We should make some minor changes to help make the asserts easier to use as the basis of our tests:

```
let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

let areEqual expected actual =
        actual = expected

let assertVIP =
        let expected = {Id = 1; IsVip = true; Credit = 100.0M }
        areEqual expected (upgradeCustomer customerVIP)
let assertSTDtoVIP =
        let expected = {Id = 2; IsVip = true; Credit = 200.0M }
        areEqual expected (upgradeCustomer customerSTD)
let assertSTD =
        let expected = {Id = 3; IsVip = false; Credit = 100.0M }
        areEqual expected (upgradeCustomer { Id = 3; IsVip = false; Credit = 50.0M })
```

Delete the code from *CustomerTests.fs* and write the following code into the new file:

```
namespace MyProjectTests

open Xunit
open MyProject.Customer

module ``When upgrading customer`` =

    let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
    let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

    // let assertVIP =
    //     let expected = {Id = 1; IsVip = true; Credit = 100.0M }
    //     areEqual expected (upgradeCustomer customerVIP)
    [<Fact>]
    let ``should give VIP customer more credit`` () =
        let expected = { customerVIP with Credit = customerVIP.Credit + 100M }
        let actual = upgradeCustomer customerVIP
        Assert.Equal(expected, actual)

    // let assertSTDtoVIP =
    //     let expected = {Id = 2; IsVip = true; Credit = 200.0M }
    //     areEqual expected (upgradeCustomer customerSTD)
    // let assertSTD =
```

```
    //     let expected = {Id = 3; IsVip = false; Credit = 100.0M }
    //     areEqual expected (upgradeCustomer { customerSTD with Id = 3; Credit = 50\
.0M })
```

Run the tests by running `dotnet test` in the test project Terminal. They should pass.

Add the test replacements for the remaining two asserts.

```
namespace MyProjectTests

open Xunit
open MyProject.Customer

module ``When upgrading customer`` =

    let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
    let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

    // let assertVIP =
    //     let expected = {Id = 1; IsVip = true; Credit = 100.0M }
    //     areEqual expected (upgradeCustomer customerVIP)
    [<Fact>]
    let ``should give VIP customer more credit`` () =
        let expected = { customerVIP with Credit = customerVIP.Credit + 100M }
        let actual = upgradeCustomer customerVIP
        Assert.Equal(expected, actual)

    // let assertSTDtoVIP =
    //     let expected = {Id = 2; IsVip = true; Credit = 200.0M }
    //     areEqual expected (upgradeCustomer customerSTD)
    [<Fact>]
    let ``should convert eligible STD customer to VIP`` () =
        let expected = {Id = 2; IsVip = true; Credit = 200.0M }
        let actual = upgradeCustomer customerSTD
        Assert.Equal(expected, actual)

    // let assertSTD =
    //     let expected = {Id = 3; IsVip = false; Credit = 100.0M }
    //     areEqual expected (upgradeCustomer { customerSTD with Id = 3; Credit = 50\
.0M })
    [<Fact>]
    let ``should not upgrade ineligible STD customer to VIP`` () =
        let expected = {Id = 3; IsVip = false; Credit = 100.0M }
```

```
        let actual = upgradeCustomer { customerSTD with Id = 3; Credit = 50.0M }
        Assert.Equal(expected, actual)
```

Run the tests by running `dotnet test` in the test project Terminal. They should pass.

We are using Xunit as our test framework. There are other testing and assertion libraries available that you may prefer.

Using FsUnit for Assertions

This is an example of how an assertion would look if we used *FsUnit* instead of *XUnit*:

```
open FsUnit

upgraded |> should equal expected
```

Now we can replace our *XUnit* asserts with *FsUnit* ones:

```
    [<Fact>]
    let ``should give VIP customer more credit`` () =
        let expected = { customerVIP with Credit = customerVIP.Credit + 100M }
        let actual = upgradeCustomer customerVIP
        actual |> should equal expected

        [<Fact>]
    let ``should convert eligible STD customer to VIP`` () =
        let customer = { Id = 2; IsVip = false; Credit = 200M }
        let expected = { customer with IsVip = true; Credit = customer.Credit + 100M\
 }
        let actual = upgradeCustomer customer
        actual |> should equal expected

    [<Fact>]
    let ``should not upgrade eligible STD customer to VIP`` () =
        let customer = { Id = 3; IsVip = false; Credit = 50M }
        let expected = { customer with Credit = customer.Credit + 50M }
        let actual = upgradeCustomer customer
        actual |> should equal expected
```

Run the tests by running `dotnet test` in the test project Terminal or the Solution Terminal. They should pass.

Summary

The features we used in this chapter are important if we want to make working on larger F# codebases manageable.

- Solutions, projects, namespaces, and modules
- Unit testing with *XUnit*
- Assertions with *FsUnit*

In the next chapter, we will have an initial look at collections.

5 - Introduction to Collections

In this chapter, we will investigate functional collections and the helper modules that F# supplies out of the box.

Before We Start

Create a new folder for this chapter.

The Basics

Add a new script file called *lists.fsx* for the example code we are going to write in this section.

F# has a hugely deserved reputation for being great for data-centric use cases like finance and data science due in a large part to the power of its support for data structures and collection handling.

There are a number of types of collections in F# that we can make use of but the three primary ones are:

- *Seq* - A lazily evaluated collection.
- *Array* - Great for numerics/data science. There are built-in modules for 2d, 3d, and 4d arrays.
- *List* - Eagerly evaluated, linked list, with immutable structure and data.

> **F# vs .NET Collections**
>
> F# *Seq* is equivalent to .NET *IEnumerable<'T>*.
>
> The .NET *List<'T>* is called *ResizeArray<'T>* in F#.

Each of these types has a supporting module that contains a wide range of functions including the ability to convert to/from each other.

In this chapter, we are going to concentrate on the *List* type and module.

Core Functionality

We will be using *lists.fsx* for this section. Remember to highlight the code and run it in F# Interactive (FSI) using `ALT + ENTER`.

Create an empty list:

```
let items = []
```

Create a list with five integers:

```
let items = [2;5;3;1;4]
```

If they were sorted in ascending order, we could also do this:

```
let items = [1..5]
```

Or we could use a *List Comprehension*:

```
let items = [
        for x in 1..5 do
                yield x
]
```

Since F# 5, we have been able to drop the need for the `yield` keyword in most cases. We can also define it in one line:

```
let items = [ for x in 1..5 do x ]
```

Comprehensions are really powerful. They're also available for the other primary collection types in F#: *Seq* and *Array*, but we are not going to use them in this chapter.

To add an item to a list, we use the cons operator (`::`):

```
// head :: tail
let extendedItems = 6::items // [6;1;2;3;4;5]
```

The original list remains unaffected by the new item as it is immutable. This is very efficient because *extendedItems* is the new item plus a pointer to the original *items* list.

A non-empty list is made up of a single item called the head and a list of items called the tail which could be an empty list `[]`. We can pattern match on a list to show this:

```
let readList items =
    match items with
    | [] -> "Empty list"
    | [head] -> $"Head: {head}" // list containing one item
    | head::tail -> sprintf "Head: %A and Tail: %A" head tail

let emptyList = readList [] // "Empty list"
let multipleList = readList [1;2;3;4;5] // "Head: 1 and Tail: [2;3;4;5]"
let singleItemList = readList [1] // "Head: 1"
```

The labels *head* and *tail* have no meaning; you could use anything such as *h::t* or *fred::ginger*; they are just labels. We also see three ways of outputing a string; directly, string interpolation using `$`, and `sprintf`.

> **Strings**
>
> There are multiple ways of outputting strings in F#. The primary two ways are `sprintf` and string interpolation:
>
> $"Head: %A and Tail: %A"
>
> sprintf "Head: %A and Tail: %A" head tail
>
> The `%A` formats the output using the types `toString()` method. You do not have to use formatting with interpolation.

If we remove the pattern match for the single item list, the code still works:

```
let readList items =
    match items with
    | [] -> "Empty list"
    | head::tail -> sprintf "Head: %A and Tail: %A" head tail

let emptyList = readList [] // "Empty list"
let multipleList = readList [1;2;3;4;5] // "Head: 1 and Tail: [2;3;4;5]"
let singleItemList = readList [1] // "Head: 1 and Tail: []"
```

We can join (concatenate) two lists together:

```
let list1 = [1..5]
let list2 = [3..7]
let emptyList = []

let joined = list1 @ list2 // [1;2;3;4;5;3;4;5;6;7]
let joinedEmpty = list1 @ emptyList // [1;2;3;4;5]
let emptyJoined = emptyList @ list1 // [1;2;3;4;5]
```

As lists are immutable, we can re-use them knowing that their values/structure will never change.

We could use the `List.concat` function to do the same job as the `@` operator:

```
let joined = List.concat [list1;list2]
```

We can filter a list using a predicate function with the signature `a -> bool` and the `List.filter` function:

```
let myList = [1..9]

let getEvens items =
    items
    |> List.filter (fun x -> x % 2 = 0)

let evens = getEvens myList // [2;4;6;8]
```

We can add up the items in a list using the `List.sum` function:

```
let sum items =
    items |> List.sum

let mySum = sum myList // 45
```

Other aggregation functions are as easy to use but we are not going to look at them here.

Sometimes we want to perform an operation on each item in a list. If we want to return the new list, we use the `List.map` function:

```
let triple items =
    items
    |> List.map (fun x -> x * 3)

let myTriples = triple [1..5] // [3;6;9;12;15]
```

`List.map` is a higher order function that transforms an `'a list` to a `'b list` using a function that converts `'a ->'b` where `'a` and `'b` could be the same type, like the example shown here which is `int -> int`.

If you have used C#, Select is the closest LINQ equivalent to `List.map`, except it is lazily evaluated. If you need lazy evaluation, you could use `Seq.map` instead.

If we don't want to return a new list, we use the `List.iter` function:

```
let print items =
    items
    |> List.iter (fun x -> (printfn "My value is %i" x))

print myList
```

Let's take a look at a more complicated example using `List.map` that changes the structure of the output list. We will use a list of tuples `(int * decimal)` which might represent quantity and unit price.

```
let items = [(1,0.25M);(5,0.25M);(1,2.25M);(1,125M);(7,10.9M)]
```

To calculate the total price of the items, we can use the `List.map` function to convert a list of tuples of integer and decimal to a list of decimals and then sum the result:

```
let sum items =
    items
    |> List.map (fun (q, p) -> decimal q * p)
    |> List.sum
```

Note the explicit conversion of the integer to a decimal. F# is quite strict about types in calculations and doesn't support implicit conversion between types like this. Notice how we can pattern match in the lambda `(fun (q, p) -> decimal q * p)` to deconstruct the tuple to gain access to the contained values.

In this particular case, there is an easier way to do the calculation in one step with another of the *List* module functions:

```
let sum items =
    items
    |> List.sumBy (fun (q, p) -> decimal q * p)
```

Folding

A very powerful functional concept that we can use to do similar aggregation tasks (and lots more that we won't cover) is the `List.fold` function. If you have used C#, Aggregate is the LINQ equivalent:

Our first example is to sum the numbers from one to ten:

```
[1..10]
|> List.fold (fun acc v -> acc + v) 0
```

`List.fold` is a higher order function that is defined like this:

```
// folder -> initial value -> input list -> output value
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

The folder function first takes the initial value and stores it in the state which we have called `acc` and then loops through the collection adding each value to the state until there are no items left to process, at which point the state is returned.

This can be simplified as:

```
[1..10]
|> List.fold (+) 0
```

The product of `[1..10]` using `List.fold` would be:

```
[1..10]
|> List.fold (fun acc v -> acc * v) 1
```

Notice that we set the initial value to one for multiplication.

This can also be simplified like the sum was:

```
[1..10]
|> List.fold (*) 1
```

We'll now look at how we would write the total price example we looked at earlier using `List.fold`:

```
let items = [(1,0.25M);(5,0.25M);(1,2.25M);(1,125M);(7,10.9M)]

let getTotal items =
    items
    |> List.fold (fun acc (q, p) -> acc + decimal q * p) 0M

let total = getTotal items
```

The folder function uses an accumulator and the deconstructed tuple and simply adds the intermediate calculation to the accumulator. The 0M parameter is the initial value of the accumulator. If we were folding using multiplication, the initial value would have been 1M.

An alternative style is to use another of the forward-pipe operators, (||>). This version supports a pair tupled of inputs:

```
let getTotal items =
    (0M, items) ||> List.fold (fun acc (q, p) -> acc + decimal q * p)

let total = getTotal items
```

There are situations where these additional operators can be useful but in a simple case like this, I prefer the previous version.

Fold is a very nice feature to have but you should try to use simpler, more specific functions like `List.sumBy` first.

Grouping Data and Uniqueness

If we wanted to get the unique numbers from a list, we can do it in many ways. Firstly, we will use the `List.groupBy` function to return a tuple for each distinct value:

```
let myList = [1;2;3;4;5;7;6;5;4;3]

// int list -> (int * int list) list
let gbResult = myList |> List.groupBy (fun x -> x)

// gbResult = [(1,[1]);(2,[2]);(3,[3;3]);(4,[4;4]);(5,[5;5]);(6,[6]);(7,[7])]
```

The `List.groupBy` function returns a list of tuples containing the data you grouped by as the first item and a list of the instances that exist in the original list as the second.

To get the list of unique items from the result list, we can use the `List.map` function:

```
let unique items =
    items
    |> List.groupBy id
    |> List.map (fun (i, _) -> i)

let unResult = unique myList // [1;2;3;4;5;6;7]
```

The anonymous function `fun x -> x` can be replaced by the *identity function* `id`.

Using the `List.groupBy` function is nice but there is a function called `List.distinct` that will do exactly what we want:

```
let distinct = myList |> List.distinct
```

There is a built-in collection type called *Set* that will do this as well:

```
let uniqueSet items =
    items
    |> Set.ofList

let setResult = uniqueSet myList // [1;2;3;4;5;6;7]
```

Most of the collection types have a way of converting from and to each other.

Solving a Problem in Many Ways

Thanks to the work of the F# Language Team, there are a wide range of functions available in the collection modules. This means that there are often multiple ways to solve any given problem. Let's take a simple task like finding the sum of the squares of the odd numbers in a given input list:

```
// Sum of the squares of the odd numbers
let nums = [1..10]

// Step by step
nums
|> List.filter (fun v -> v % 2 = 1)
|> List.map (fun v -> v * v)
|> List.sum

// Using option and choose
nums
|> List.choose (fun v -> if v % 2 = 1 then Some (v * v) else None)
```

```
|> List.sum

// Fold
nums
|> List.fold (fun acc v -> acc + if v % 2 = 1 then (v * v) else 0) 0

// Do not use reduce for this.
// Firstly, reduce is a partial function, so we need to handle empty list
// More importantly, the first item in the list is not processed by the function,
// So the result will probably be incorrect
match nums with
| [] -> 0
| items -> items |> List.reduce (fun acc v -> acc + if v % 2 = 1 then (v * v) else 0)

// The recommended version
nums
|> List.sumBy (fun v -> if v % 2 = 1 then (v * v) else 0)
```

Some of the functions are partial functions which means that they do not work for all possible inputs. In the case of `List.reduce`, it will fail on an empty list as it uses the first item as the initial value, so not having one is an issue. Many of the partial functions have a `try` version that returns an option to solve this but `List.reduce` doesn't.

We will now put our new knowledge to good use by looking at a more practical example using something more substantial than a list of integers.

Working Through a Practical Example

Follow the instructions from the Appendix for creating a solution that we used in the setup of last chapter. The only difference is that you should create a `classlib` project rather than a `console` project.

In this example code, we are going to manage an order that contains an immutable list of items. The functionality we need to add is:

- Add an item
- Increase the quantity of an item
- Remove an item
- Reduce quantity of an item
- Clear all of the items

Create a new file in *MyProject* called *Orders.fs*. You can remove the *Library.fs* file. Add an *OrderTests.fs* file to *MyProjectTests* and add the namespace:

```
namespace MyProject.Orders
```

Add the following record types to *Orders.fs*:

```
type Item = {
    ProductId : int
    Quantity : int
}

type Order = {
    Id : int
    Items : Item list
}
```

Create a module called Domain and put it after the *Order* type definition:

```
module Domain =
```

We need to create a function to add an item to the order. This function needs to cater for products that exist in the order as well as those that don't. Let's think about how we can do this in pseudocode:

```
let addItem item order =
        // 1 - Prepend new item to existing order items
    // 2 - Consolidate each product
    // 3 - Sort items in order by productid to make equality simpler
    // 4 - Update order with new list of items
```

The simplest approach will be to perform steps 1 and 4 first:

```
let addItem item order =
    let items = item::order.Items
    { order with Items = items }
```

Let's create a couple of helpers bindings and some simple asserts to help us test the function in FSI:

```
let order = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
let newItemExistingProduct = { ProductId = 1; Quantity = 1 }
let newItemNewProduct = { ProductId = 2; Quantity = 2 }

addItem newItemNewProduct order =
    { Id = 1; Items = [ { ProductId = 1; Quantity = 1 };{ ProductId = 2; Quantity = \
2 } ] }
addItem newItemExistingProduct order =
    { Id = 1; Items = [ { ProductId = 1; Quantity = 2 } ] }
```

Run this in FSI and do it for every change that we make. If either of the asserts is false, which in this case the second assert will be, run the code to the left of the equals sign on its own to see what the function is actually returning. You should see something like this:

```
val it: Order = { Id = 1
                  Items = [{ ProductId = 1
                             Quantity = 1 }; { ProductId = 1
                                               Quantity = 1 }] }
```

To fix this, we need to consolidate the items per product using `List.groupBy`:

```
// Count how many of each product we have
let addItem item order =
    let items =
        item :: order.Items
        |> List.groupBy (fun i -> i.ProductId) // (int * Item list) list
    { order with Items = items } // Problem
```

This code won't compile as `List.groupBy` returns a tuple and this is not what `order.Items` is expecting.

We need to extend the function by mapping the tuple output to the expected *Item* type:

```
let addItem item order =
    let items =
        item::order.Items
        |> List.groupBy (fun i -> i.ProductId) // (int * Item list) list
        |> List.map (fun (id, items) ->
            { ProductId = id; Quantity = items |> List.sumBy (fun i -> i.Quantity) })
    { order with Items = items }
```

Finally, we need to add a sort to enable the built-in list equality mechanism to work:

```
let addItem item order =
    let items =
        item::order.Items
        |> List.groupBy (fun i -> i.ProductId) // (int * Item list) list
        |> List.map (fun (id, items) ->
            { ProductId = id; Quantity = items |> List.sumBy (fun i -> i.Quantity) })
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }
```

Remove the helper bindings we've used for testing as we are going to create real tests in a new *OrderTests.fs* file in the *MyProjectTests* project.

Add the following to the top of the page:

```
namespace OrderTests

open MyProject.Orders
open MyProject.Orders.Domain
open Xunit
open FsUnit
```

Now we are going to add our first test which verifies that adding items to an empty order works correctly:

```
module ``Add item to order`` =

        [<Fact>]
    let ``when product does not exist in empty order`` () =
        let myEmptyOrder = { Id = 1; Items = [] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 3 } ] }
        let actual = myEmptyOrder |> addItem { ProductId = 1; Quantity = 3 }
        actual |> should equal expected
```

Run the test in the Terminal for the test project or the solution using:

```
dotnet test
```

Add the following tests in the same module and run the tests:

```
    [<Fact>]
    let ``when product does not exist in non empty order`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 };{ Product\
Id = 2; Quantity = 5 } ] }
        let actual = myOrder |> addItem { ProductId = 2; Quantity = 5 }
        actual |> should equal expected

    [<Fact>]
    let ``when product exists in non empty order`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 4 } ] }
        let actual = myOrder |> addItem { ProductId = 1; Quantity = 3 }
        actual |> should equal expected
```

We have created three tests that cover the main cases when adding an item to an existing order.

We can easily add multiple items to an order by changing the cons operator (::) which adds a single item to the list into the concat operator (@) which joins two lists together:

```
// Item list -> Order -> Order
let addItems newItems order =
    let items =
        newItems @ order.Items
        |> List.groupBy (fun i -> i.ProductId)
        |> List.map (fun (id, items) ->
            { ProductId = id; Quantity = items |> List.sumBy (fun i -> i.Quantity) })
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }
```

We should add some tests to *OrderTests.fs* for the `addItems` function:

```
module ``add multiple items to an order`` =

    [<Fact>]
    let ``when new products added to empty order`` () =
        let myEmptyOrder = { Id = 1; Items = [] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 }; { Produc\
tId = 2; Quantity = 5 } ] }
        let actual = myEmptyOrder |> addItems [ { ProductId = 1; Quantity = 1 }; { P\
roductId = 2; Quantity = 5 } ]
        actual |> should equal expected
```

```
    [<Fact>]
    let ``when new products and updated existing to order`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 2 }; { Produc\
tId = 2; Quantity = 5 } ] }
        let actual = myOrder |> addItems [ { ProductId = 1; Quantity = 1 }; { Produc\
tId = 2; Quantity = 5 } ]
        actual |> should equal expected
```

Run the tests to confirm your new code works.

Let's extract the common functionality in `addItem` and `addItems` into a new function:

```
// Item list -> Item list
let recalculate items =
    items
    |> List.groupBy (fun i -> i.ProductId)
    |> List.map (fun (id, items) ->
        { ProductId = id; Quantity = items |> List.sumBy (fun i -> i.Quantity) })
    |> List.sortBy (fun i -> i.ProductId)

let addItem item order =
    let items =
        item :: order.Items
        |> recalculate
    { order with Items = items }

let addItems items order =
    let items =
        items @ order.Items
        |> recalculate
    { order with Items = items }
```

Run the tests to confirm that the changes have been successful.

Removing a product can be easily achieved by filtering out the unwanted item by the *ProductId*:

```
let removeProduct productId order =
    let items =
        order.Items
        |> List.filter (fun x -> x.ProductId <> productId)
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }
```

Again we write some tests to verify our new function works as expected:

```
module ``Removing a product`` =

    [<Fact>]
    let ``when remove all items of existing productid`` () =
        let myEmptyOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [] }
        let actual = myEmptyOrder |> removeProduct 1
        actual |> should equal expected

    [<Fact>]
    let ``should do nothing for non existant productid`` () =
        let myOrder = { Id = 2; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 2; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let actual = myOrder |> removeProduct 2
        actual |> should equal expected
```

Reducing an item quantity is slightly more complex. We reduce the quantity of the item by the specified number, recalculate the items and then filter out any items with a quantity less than or equal to zero:

```
let reduceItem productId quantity order =
    let items =
        { ProductId = productId; Quantity = -quantity } :: order.Items
        |> recalculate
        |> List.filter (fun x -> x.Quantity > 0)
    { order with Items = items }
```

Again we write some tests to verify our new function works as expected:

```
module ``Reduce item quantity`` =

    [<Fact>]
    let ``reduce existing item quantity`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 5 } ] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 2 } ] }
        let actual = myOrder |> reduceItem 1 3
        actual |> should equal expected

    [<Fact>]
    let ``reduce existing item and remove`` () =
        let myOrder = { Id = 2; Items = [ { ProductId = 1; Quantity = 5 } ] }
        let expected = { Id = 2; Items = [] }
        let actual = myOrder |> reduceItem 1 5
        actual |> should equal expected

    [<Fact>]
    let ``reduce item with no quantity`` () =
        let myOrder = { Id = 3; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 3; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let actual = myOrder |> reduceItem 2 5
        actual |> should equal expected

    [<Fact>]
    let ``reduce item with no quantity for empty order`` () =
        let myEmptyOrder = { Id = 4; Items = [] }
        let expected = { Id = 4; Items = [] }
        let actual = myEmptyOrder |> reduceItem 2 5
        actual |> should equal expected
```

Clearing all of the items is really simple. We just set Items to be an empty list:

```
let clearItems order =
    { order with Items = [] }
```

Write some tests to verify our new function works as expected:

```
module ``Empty an order of all items`` =

    [<Fact>]
    let ``order with existing item`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [] }
        let actual = myOrder |> clearItems
        actual |> should equal expected

    [<Fact>]
    let ``order with existing items`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 };{ ProductI\
d = 2; Quantity = 5 } ] }
        let expected = { Id = 1; Items = [] }
        let actual = myOrder |> clearItems
        actual |> should equal expected

    [<Fact>]
    let ``empty order is unchanged`` () =
        let myEmptyOrder = { Id = 2; Items = [] }
        let expected = { Id = 2; Items = [] }
        let actual = myEmptyOrder |> clearItems
        actual |> should equal expected
```

Summary

In this chapter, we have looked at some of the most useful functions within the *List* module. There are similar functions available in the *Seq* and *Array* modules as we will discover in the coming chapters. We have also seen that it is possible to use immutable data structures to provide important business functionality that is succinct and robust.

We have only scratched the surface of what is possible with collections in F#. For more information on the List module, have a look at the F# Docs[20]. Thanks to some awesome community assistance, there are code samples for each of the collection module functions.

In the next chapter, we will look at how to handle a stream of data from a CSV source file.

[20] https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-listmodule.html

6 - Reading Data From a File

In this chapter, we will introduce the basics of reading and parsing external data using sequences and the Seq module and learn how we can isolate code that talks to external services to make our codebase more testable.

Setting Up

In a new folder for this chapter, create a console application by typing the following into a VS Code Terminal window:

```
dotnet new console -lang F#
```

Using the Explorer view, create a folder called *resources* in the code folder for this chapter and then create a new file called *customers.csv*. Copy the following data into the new file:

```
CustomerId|Email|Eligible|Registered|DateRegistered|Discount
John|john@test.com|1|1|2015-01-23|0.1
Mary|mary@test.com|1|1|2018-12-12|0.1
Richard|richard@nottest.com|0|1|2016-03-23|0.0
Sarah||0|0||
```

Remove all the code from *Program.fs* except the following:

```
open System

[<EntryPoint>]
let main argv =
    0
```

As of F# 6, the *EntryPoint* attribute is no longer necessary.

Loading Data

We are going to use the .NET System.IO classes to open the *customers.csv* file:

```
open System.IO
```

To load the data, we need to create a function that takes a path as a string, reads the file contents, and returns a collection of lines as strings from the file. Let's start with reading from a file:

```
let readFile path = // string -> seq<string>
    seq {
        use reader = new StreamReader(File.OpenRead(path))
        while not reader.EndOfStream do
            reader.ReadLine()
    }
```

There are a few new things in this simple function!

The `seq {...}` type is called a Sequence Expression. The code inside the curly brackets will create a sequence of strings. We could have done the same with *list* and *array* but most of the System.IO methods output `IEnumerable<'T>` and `seq {...}` is the F# equivalent to `IEnumerable<'T>`.

The *StreamReader* class implements the `IDisposable<'T>` interface. Just like in C#, we need to make sure the code calls the `Dispose()` method when it has finished using it. To do this, we use the `use` keyword which is the equivalent of `using` in C#. The indenting defines the scope of the disposable; in this case, the `Dispose()` method will be called when the sequence has completed being generated. It is required that we use the `new` keyword when we create instances that implement `IDisposable<'T>`.

We need to write some code in the main function to call our `readFile` function and output the data to the output window;

```
[<EntryPoint>]
let main argv =
    let path = Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    let data = readFile path
    data |> Seq.iter (fun x -> printfn "%s" x)
    0
```

You must leave the `0` at the end of the main function as this tells dotnet that the process has been completed successfully.

We are using a built-in constant, `__SOURCE_DIRECTORY__`, to determine the current source code directory.

The Seq module has a wide range of functions available, similar to the List and Array modules. The `Seq.iter` function will iterate over the sequence, perform an action, and return *unit*.

The code in *Program.fs* should now look like this:

```
open System
open System.IO

// string -> seq<string>
let readFile path =
    seq {
        use reader = new StreamReader(File.OpenRead(path))
        while not reader.EndOfStream do
            reader.ReadLine()
    }

[<EntryPoint>]
let main argv =
    let path = Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    let data = readFile path
    data |> Seq.iter (fun x -> printfn "%s" x)
    0
```

Run the code by typing `dotnet run` in the Terminal.

To handle potential errors from loading a file, we are going to add some error handling to the *readFile* function:

```
// string -> Result<seq<string>,exn>
let readFile path =
    try
        seq {
            use reader = new StreamReader(File.OpenRead(path))
            while not reader.EndOfStream do
                reader.ReadLine()
        }
        |> Ok
    with
    | ex -> Error ex
```

To handle the change in the signature of the readFile function, we will introduce a new function:

```
let import path =
    match path |> readFile with
    | Ok data -> data |> Seq.iter (fun x -> printfn "%A" x)
    | Error ex -> printfn "Error: %A" ex.Message
```

Replace the code in the main function with:

```
[<EntryPoint>]
let main argv =
    let path = Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    import path
    0
```

We should simplify this code whilst we're here:

```
[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import
    0
```

Run the program to check that it still works by typing `dotnet run` in the Terminal.

Sadly, we have introduced a subtle bug here when we have an exception; the exception will not get caught because `seq {...}` is lazily evaluated. If you want to test it out, try a path that does not exist. Thankfully, it is very easy to fix:

```
// string -> Result<seq<string>,exn>
let readFile path =
    try
        File.ReadLines(path)
        |> Ok
    with
    | ex -> Error ex
```

Parsing Data

Now that we can load the data, we should try to convert each line to an F# record:

```
type Customer = {
    CustomerId : string
    Email : string
    IsEligible : string
    IsRegistered : string
    DateRegistered : string
    Discount : string
}
```

It is good practice to put types definitions above the let bindings in a code file.

Create a function that takes a `seq<string>` as input, parses the input data, and returns a `seq<Customer>` as output:

```
// seq<string> -> seq<Customer>
let parse (data:string seq) =
    data
    |> Seq.skip 1 // Ignore the header row
    |> Seq.map (fun line ->
        match line.Split('|') with
        | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
            Some {
                CustomerId = customerId
                Email = email
                IsEligible = eligible
                IsRegistered = registered
                DateRegistered = dateRegistered
                Discount = discount
            }
        | _ -> None
    )
    |> Seq.choose id // Ignore None and unwrap Some
```

There are some new features in this function:

The `Seq.skip` function ignores a number of lines. In this case, the first item in the sequence is a header row and not a Customer, so it should be ignored. There is a potential bug here as `Seq.skip` is a partial function that will raise an exception if the input sequence is empty. We will fix this later in the chapter.

The `Split` function creates an array of strings from a string. We then pattern match the array and get the data which we then use to populate a Customer. If you aren't interested in all of the data, you can use the wildcard (_) for those parts. The number of columns in the array must match the number of identifiers we have used. In this case, we have defined six identifiers as we expect the row to be split into six parts.

We have now met the three primary collection types in F#: List (`[..]`), Seq (`seq {..}`), and Array (`[|..|]`).

The `Seq.choose` function will ignore any item in the sequence that is None and will unwrap the Some items to return a sequence of Customers. Remember that the *id* keyword is the identity function, a built-in value that is equivalent to the anonymous function `(fun x -> x)`.

We need to modify the import function to use the new parse function:

```
let import path =
    match path |> readFile with
    | Ok data -> data |> parse |> Seq.iter (fun x -> printfn "%A" x)
    | Error ex -> printfn "Error: %A" ex.Message
```

The next stage is to extract the code from the `Seq.map` in the `parse` function into its own function:

```
// string -> Customer option
let parseLine (line:string) : Customer option =
    match line.Split('|') with
    | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
        Some {
            CustomerId = customerId
            Email = email
            IsEligible = eligible
            IsRegistered = registered
            DateRegistered = dateRegistered
            Discount = discount
        }
    | _ -> None
```

Modify the `parse` function to use the new `parseLine` function:

```
let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map (fun x -> parseLine x)
    |> Seq.choose id
```

We can simplify this function by removing the lambda, just like a Method Group in C#:

```
let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id
```

Testing the Code

Whilst we have improved the code a lot, it is difficult to test without having to load a file. The signature of the `readFile` function is `string -> Result<seq<string>,exn>` which means that we

could easily convert it to use a webservice or a fake service for testing rather than a path to a file on disk.

To make this testable and extensible, we can make import a higher order function by taking a function as a parameter with the same signature as `readFile`. We should set the name of the input parameter to *dataReader* to reflect the possibility of different sources of data than a file:

```
let output data =
    data
    |> Seq.iter (fun x -> printfn "%A" x)

let import (dataReader:string -> Result<string seq,exn>) path =
    match path |> dataReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex.Message
```

We can now pass any function with this signature `string -> Result<string seq,exn>`into the import function.

This signature is quite simple but they can get quite complex. We can create a Type Abbreviation with the same signature and use that in the parameter instead:

```
type DataReader = string -> Result<string seq,exn>
```

Replace the function signature in `import` with it;

```
let import (dataReader:DataReader) path =
    match path |> dataReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex.Message
```

We can use the type abbreviation like an Interface with exactly one function in the *readFile* function but it does mean modifying our code a little to use an alternate function style:

```
//string -> Result<string seq,exn>
let readFile : DataReader =
    fun path ->
        try
            File.ReadLines(path)
            |> Ok
        with
        | ex -> Error ex
```

We need to make a small change to our call in main to tell it to use the readFile function as we have changed the signature of the import function:

```
[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import readFile
    0
```

If we use import with *readFile* regularly, we can use partial application to create a new function that does that for us:

```
let importWithFileReader = import readFile
```

To use it we would simply call:

```
Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
|> importWithFileReader
```

The payoff for the work we have done to use higher order functions and type abbreviations is that we can easily pass in a fake function with known data for testing:

```
let fakeDataReader : DataReader =
    fun _ ->
        seq {
            "CustomerId|Email|Eligible|Registered|DateRegistered|Discount"
            "John|john@test.com|1|1|2015-01-23|0.1"
            "Mary|mary@test.com|1|1|2018-12-12|0.1"
            "Richard|richard@nottest.com|0|1|2016-03-23|0.0"
            "Sarah||0|0||"
        }
        |> Ok

import fakeDataReader "_"
```

You can use any function that satisfies the *DataReader* function signature.

Final Code

What we have ended up with is the following;

```
open System.IO

type Customer = {
    CustomerId : string
    Email : string
    IsEligible : string
    IsRegistered : string
    DateRegistered : string
    Discount : string
}

type DataReader = string -> Result<string seq,exn>

let readFile : DataReader =
    fun path ->
        try
            File.ReadLines(path)
            |> Ok
        with
        | ex -> Error ex

let parseLine (line:string) : Customer option =
    match line.Split('|') with
    | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
        Some {
            CustomerId = customerId
            Email = email
            IsEligible = eligible
            IsRegistered = registered
            DateRegistered = dateRegistered
            Discount = discount
        }
    | _ -> None

let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id

let output data =
    data
    |> Seq.iter (fun x -> printfn "%A" x)
```

```
let import (dataReader:DataReader) path =
    match path |> dataReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex.Message

[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import readFile
    0
```

In a later chapter, we will extend this code by adding data validation but first, we have some tidying up to do to make our code more succinct and robust.

Summary

In this chapter, we have looked at how we can import data using some of the most useful functions on the Seq module, sequence expressions, and function types. We have also seen how we can use higher order functions to allow us to easily extend functions to take different functions as parameters at runtime and for tests.

In the next chapter, we will look at another exciting F# feature, Active Patterns, to help make our pattern matching more readable.

7 - Active Patterns

In this chapter, we will be extending our knowledge of pattern matching by looking at how we can write our own custom matchers with Active Patterns. There are a number of different Active Patterns types available and we will look at most of them in this chapter.

Setting Up

Create a folder for the code from this chapter.

All of the code in this chapter can be written in an F# script file (.fsx) and run using F# Interactive.

Partial Active Patterns

Partial active patterns return an option as output because only a subset of the possible input values will produce a positive match.

One area where Partial Active Patterns are really helpful is for validation and parsing. This is an example of parsing a string to a DateTime using a function rather than an active pattern:

```
open System

// string -> DateTime option
let parse (input:string) =
    match DateTime.TryParse(input) with
    | true, value -> Some value
    | false, _ -> None

let isDate = parse "2019-12-20" // Some 2019-12-20 00:00:00
let isNotDate = parse "Hello" // None
```

It works but we can't re-use the function body logic directly in a pattern match, except in the guard clause. Let's create a partial active pattern to handle the DateTime parsing for us:

```
open System

// string -> DateTime option
let (|ValidDate|_|) (input:string) =
    match DateTime.TryParse(input) with
    | true, value -> Some value
    | false, _ -> None
```

Anytime that you see the banana clips (`|...|`), you are looking at an active pattern and if you see (`|...|_|`), then you are looking at a partial active pattern. Remember, the partial part of the name comes from the use of the wildcard in the pattern definition, implying that the active pattern only returns success for a subset of the possible input values.

Notice that we are returning the parsed value in the success case. We will be able to access that when we use the active pattern.

The logic in this active pattern could be written to use an *if* expression instead of a *match* expression:

```
open System

// string -> DateTime option
let (|ValidDate|_|) (input:string) =
    let success, value = DateTime.TryParse(input)
    if success then Some value else None
```

Either version is valid.

Now we can plug it into our parse function:

```
// string -> unit
let parse input =
    match input with
    | ValidDate dt -> printfn "%A" dt
    | _ -> printfn "'%s{input}' is not a valid date"

parse "2019-12-20" // 2019-12-20 00:00:00
parse "Hello" // 'Hello' is not a valid date
```

We can access the parsed date for the success case because we returned it from the partial active pattern.

There's more code but it is much more readable and we have functionality that we can re-use in other pattern matches. In this case, the partial active pattern returned the parsed value as part of the match. If you didn't care about the value being returned, you can return `Some ()` instead of `Some value` as shown in the following example:

```
open System

// string -> unit option
let (|IsValidDate|_|) (input:string) =
    let success, _ = DateTime.TryParse input
    if success then Some () else None

// string -> bool
let isValidDate input =
    match input with
    | IsValidDate -> true
    | _ -> false
```

Partial active patterns are used a lot in activities like validation, as we will discover in Chapter 8.

Parameterized Partial Active Patterns

Parameterized partial active patterns differ from basic partial active patterns by supplying additional input items.

We are going to investigate how an old interview favourite, FizzBuzz, can be implemented using a parameterized partial active pattern. Let's start with the canonical solution:

```
let calculate i =
    if i % 3 = 0 && i % 5 = 0 then "FizzBuzz"
    elif i % 3 = 0 then "Fizz"
    elif i % 5 = 0 then "Buzz"
    else i |> string

[1..15] |> List.map calculate
```

It works but can we do better with Pattern Matching? How about this?

```
let calculate i =
    match (i % 3, i % 5) with
    | (0, 0) -> "FizzBuzz"
    | (0, _) -> "Fizz"
    | (_, 0) -> "Buzz"
    | _ -> i |> string
```

Or this?

```
let calculate i =
    match (i % 3 = 0, i % 5 = 0) with
    | (true, true) -> "FizzBuzz"
    | (true, _) -> "Fizz"
    | (_, true) -> "Buzz"
    | _ -> i |> string
```

Neither of these is any more readable than the original. How about if we could use an active pattern to do something like this?

```
let calculate i =
    match i with
    | IsDivisibleBy 3 & IsDivisibleBy 5 -> "FizzBuzz"
    | IsDivisibleBy 3 -> "Fizz"
    | IsDivisibleBy 5 -> "Buzz"
    | _ -> i |> string
```

Note the single `&` to apply both parts of the pattern match in the calculate function. There is also a single `|` for the or case.

We can use a parameterized partial active pattern to do this:

```
// int -> int -> unit option
let (|IsDivisibleBy|_|) divisor n  =
    if n % divisor = 0 then Some () else None
```

The parameterized name comes from the fact that we can supply additional parameters. The value being tested must always be the last parameter. In this case, we include the divisor parameter as well as the number we are checking.

This is much nicer but what happens if we need to add `(7, Bazz)` or even more options into the mix? Look at the code now:

```
let calculate i =
    match i with
    | IsDivisibleBy 3 & IsDivisibleBy 5 & IsDivisibleBy 7 -> "FizzBuzzBazz"
    | IsDivisibleBy 3 & IsDivisibleBy 5 -> "FizzBuzz"
    | IsDivisibleBy 3 & IsDivisibleBy 7 -> "FizzBazz"
    | IsDivisibleBy 5 & IsDivisibleBy 7 -> "BuzzBazz"
    | IsDivisibleBy 3 -> "Fizz"
    | IsDivisibleBy 5 -> "Buzz"
    | IsDivisibleBy 7 -> "Bazz"
    | _ -> i |> string
```

Maybe not such a good idea? How confident would you be that you had covered all of the permutations?

An alternative suggested by Isaac Abraham would be to use a list as the input parameter of the active pattern:

```
let (|IsDivisibleBy|_|) divisors n  =
    if divisors |> List.forall (fun div -> n % div = 0)
    then Some ()
    else None

let calculate i =
    match i with
    | IsDivisibleBy [3;5;7] -> "FizzBuzzBazz"
    | IsDivisibleBy [3;5] -> "FizzBuzz"
    | IsDivisibleBy [3;7] -> "FizzBazz"
    | IsDivisibleBy [5;7] -> "BuzzBazz"
    | IsDivisibleBy [3] -> "Fizz"
    | IsDivisibleBy [5] -> "Buzz"
    | IsDivisibleBy [7] -> "Bazz"
    | _ -> i |> string
```

In many ways, this is much nicer but it still involves a lot of typing and is potentially prone to errors.

Maybe a different approach would yield a better result?

```
let calculate n =
    [(3, "Fizz"); (5, "Buzz"); (7, "Bazz")]
    |> List.map (fun (divisor, result) -> if n % divisor = 0 then result else "")
    |> List.reduce (+) // (+) is a shortcut for (fun acc v -> acc + v)
    |> fun input -> if input = "" then string n else input

[1..15] |> List.map calculate
```

The `List.reduce` function will concatenate all of the strings generated by the `List.map` function. The last line is to cater for numbers that are not divisible by any of the numbers in the initial list.

It is less readable than the original but it's much less prone to errors covering all of the possible permutations. You could even pass the mappings in as a parameter:

```
let calculate mapping n =
    mapping
    |> List.map (fun (divisor, result) -> if n % divisor = 0 then result else "")
    |> List.reduce (+)
    |> fun input -> if input = "" then string n else input

[1..15] |> List.map (calculate [(3, "Fizz"); (5, "Buzz")])
```

This is the type of function that F# developers write all the time.

Back to active patterns! Let's have a look at another favourite interview question: Leap Years. The basic F# function looks like this;

```
let isLeapYear year =
    year % 400 = 0 || (year % 4 = 0 && year % 100 <> 0)

[2000;2001;2020] |> List.map isLeapYear = [true;false;true]
```

But can parameterized partial active patterns help make it more readable? Let's try:

```
let (|IsDivisibleBy|_|) divisor n  =
    if n % divisor = 0 then Some () else None

let (|NotDivisibleBy|_|) divisor n  =
    if n % divisor <> 0 then Some () else None

let isLeapYear year =
    match year with
    | IsDivisibleBy 400 -> true
    | IsDivisibleBy 4 & NotDivisibleBy 100 -> true
    | _ -> false
```

There are special pattern matching operators for *and* (`&`) and *or* (`|`) for logic but not for *not*, so we have to write the negative case as well as the *IsDivisibleBy* active pattern. This doesn't apply to a `when` guard clause which uses the standard F# logic operators.

You may be wondering why we wrote two parameterized partial active patterns when it seems like a nice candidate for a multi-case active pattern:

```
let (|IsDivisibleBy|NotDivisibleBy|) divisor n =
    if n % divisor = 0 then IsDivisibleBy else NotDivisibleBy
```

Sadly, whilst this does compile, it will not work in situ due to the way that the F# compiler processes multi-case active patterns.

There's more code than our original version but you could easily argue that it is more readable. We could also do a similar thing to the original with helper functions rather than Active Patterns;

```
let isDivisibleBy divisor year =
    year % divisor = 0

let notDivisibleBy divisor year =
    not (year |> isDivisibleBy divisor)

let isLeapYear year =
    year |> isDivisibleBy 400 || (year |> isDivisibleBy 4 && year |> notDivisibleBy \
100)
```

You could also use a match expression with guard clauses:

```
let isLeapYear input =
    match input with
    | year when year |> isDivisibleBy 400 -> true
    | year when year |> isDivisibleBy 4 && year |> notDivisibleBy 100 -> true
    | _ -> false
```

This would allow us to remove the *notDivisibleBy* function and pipe the *isDivisible* result to the *not* function:

```
let isLeapYear input =
    match input with
    | year when year |> isDivisibleBy 400 -> true
    | year when year |> isDivisibleBy 4 && year |> isDivisibleBy 100 |> not -> true
    | _ -> false
```

All of these approaches are valid; it's down to preference which one you choose.

Multi-Case Active Patterns

Multi-Case Active Patterns are different from Partial Active Patterns in that they always return one of the possible values rather than an option type. The maximum number of choices supported is currently seven although that may increase in a future version of F#.

A simple example of a multi-case active pattern is to determine the colour of a playing card; it can either be red or black. Firstly, we can model a playing card:

```
type Rank = Ace|Two|Three|Four|Five|Six|Seven|Eight|Nine|Ten|Jack|Queen|King
type Suit = Hearts|Clubs|Diamonds|Spades
type Card = Rank * Suit
```

Rank and suit are discriminated unions with no data attached to any case.

The active pattern needs to take a *Card* as input, determine its suit, and return either Red or Black as output:

```
let (|Red|Black|) (card:Card) =
    match card with
    | (_, Diamonds) | (_, Hearts) -> Red
    | (_, Clubs) | (_, Spades) -> Black
```

This is not a partial active pattern, so there is no wildcard.

Only cases that are used as output should be included in the active pattern.

We can now use the newly created active pattern in a function that will describe the colour of a chosen card:

```
let describeColour card =
    match card with
    | Red -> "red"
    | Black -> "black"
    |> printfn "The card is %s"

describeColour (Two, Hearts) // The card is red
```

Single-Case Active Patterns

The last type of active pattern that we will look at is the single-case. The purpose of the single-case is to allow you to decompose an input in different ways. We will use a simple string for this.

In this example, we are going to create some rules for creating a password. The rules will be:

- At least 8 characters in length
- Must contain at least one number

Our first action is to create the single-case active patterns for our rules. We will put the rule logic in the number check but not the length check to see what difference that makes to the code that uses the active patterns:

```
open System

// string -> int
let (|CharacterCount|) (input:string) =
    input.Length

// string -> bool
let (|ContainsANumber|) (input:string) =
    input
    |> Seq.filter Char.IsDigit
    |> Seq.length > 0
```

A string in F# is also a sequence of characters.

We can use our new active patterns to help perform the password logic in a similar way to how we defined it in the requirements:

```
let (|IsValidPassword|) input =
    match input with
    | CharacterCount len when len < 8 -> (false, "Password must be at least 8 charac\
ters.")
    | ContainsANumber false -> (false, "Password must contain at least 1 digit.")
    | _ -> (true, "")
```

If we were doing this in production code, we would probably create a discriminated union as the return type rather than use a tuple. This would enable us to remove the need to supply an empty string in the output for false.

Finally, we can make use of the *IsValidPassword* active pattern in our code:

```
let setPassword input =
    match input with
    | IsValidPassword (true, _) as pwd -> Ok pwd
    | IsValidPassword (false, failureReason) -> Error $"Password not set: %s{failure\
Reason}"

let badPassword = setPassword "password"
let goodPassword = setPassword "passw0rd"
```

You may not use the single-case active pattern very much but when you do, it does make the code much more readable.

Now that we have seen the most commonly used active patterns, we are going to use them to help us solve a little problem.

Using Active Patterns in a Practical Example

I play, rather unsuccessfully, an online Football (the sport where participants kick a spherical ball with their feet rather than throw an egg-shaped object with their hands) Score Predictor.

The rules are simple;

- 300 points for predicting the correct score (e.g. 2-3 vs 2-3)
- 100 points for predicting the correct result (e.g. 2-3 vs 0-2)
- 15 points per home goal & 20 points per away goal using the lower of the predicted and actual scores

We have some sample predictions, actual scores, and expected calculated points that we can use to validate the code we write;

```
(0, 0) (0, 0) = 400 // 300 + 100 + (0 * 15) + (0 * 20)
(3, 2) (3, 2) = 485 // 300 + 100 + (3 * 15) + (2 * 20)
(5, 1) (4, 3) = 180 // 0 + 100 + (4 * 15) + (1 * 20)
(2, 1) (0, 7) = 20 // 0 + 0 + (0 * 15) + (1 * 20)
(2, 2) (3, 3) = 170 // 0 + 100 + (2 * 15) + (2 * 20)
```

Firstly we define a simple tuple type to represent a score:

```
type Score = int * int
```

To determine if we have a correct score, we need to check that the prediction and the actual score are the same. F# uses structural equality to compare tuples, so this is trivial to evaluate with a partial active pattern:

```
// Score * Score -> option<unit>
let (|CorrectScore|_|) (expected:Score, actual:Score) =
    if expected = actual then Some () else None
```

We don't care about the actual scores, just that they are the same, so we can return unit.

We also need to determine what the result of a Score is. This can only be one of three choices: Draw (Tie), Home win or Away win. We can easily represent this with a multi-case active pattern:

```
let (|Draw|HomeWin|AwayWin|) (score:Score) =
    match score with
    | (h, a) when h = a -> Draw
    | (h, a) when h > a -> HomeWin
    | _ -> AwayWin
```

Note that this active pattern returns one of the pattern choices rather than an option type. We can now create a new partial active pattern for determining if we have predicted the correct result:

```
let (|CorrectResult|_|) (expected:Score, actual:Score) =
    match (expected, actual) with
    | (Draw, Draw) -> Some ()
    | (HomeWin, HomeWin) -> Some ()
    | (AwayWin, AwayWin) -> Some ()
    | _ -> None
```

We have to provide the input data as a single structure, so we use a tuple of Scores.

Without the multi-case active pattern for the result of a score, we would have to write something like this:

```
let (|CorrectResult|_|) (expected:Score, actual:Score) =
    match (expected, actual) with
    | ((h, a), (h', a')) when h = a && h' = a' -> Some ()
    | ((h, a), (h', a')) when h > a && h' > a' -> Some ()
    | ((h, a), (h', a')) when h < a && h' < a' -> Some ()
    | _ -> None
```

I prefer the version using the multi-case active pattern.

Now we need to create a function to work out the points for the goals scored:

```
let goalsScore (expected:Score) (actual:Score) =
    let (h, a) = expected
    let (h', a') = actual
    let home = [ h; h' ] |> List.min
    let away = [ a; a' ] |> List.min
    (home * 15) + (away * 20)
```

There are a couple of helper functions for tuples called `fst` and `snd` that we can use to get the first and second parts of the tuple respectively. This can simplify the *goalScore* function:

```
let goalsScore (expected:Score) (actual:Score) =
    let home = [ fst expected; fst actual ] |> List.min
    let away = [ snd expected; snd actual ] |> List.min
    (home * 15) + (away * 20)
```

We now have all of the parts to create our function to calculate the total points for each game:

```
let calculatePoints (expected:Score) (actual:Score) =
    let pointsForCorrectScore =
        match (expected, actual) with
        | CorrectScore -> 300
        | _ -> 0
    let pointsForCorrectResult =
        match (expected, actual) with
        | CorrectResult -> 100
        | _ -> 0
    let pointsForGoals = goalsScore expected actual
    pointsForCorrectScore + pointsForCorrectResult + pointsForGoals
```

Note how the tuple in the match expression matches the input required by the parameterized active patterns.

There is a bit of duplication/similarity that we will resolve later but firstly we should use our test data to validate our code using FSI:

```
let assertnoScoreDrawCorrect =
    calculatePoints (0, 0) (0, 0) = 400
let assertHomeWinExactMatch =
    calculatePoints (3, 2) (3, 2) = 485
let assertHomeWin =
    calculatePoints (5, 1) (4, 3) = 180
let assertIncorrect =
    calculatePoints (2, 1) (0, 7) = 20
let assertDraw =
    calculatePoints (2, 2) (3, 3) = 170
```

We can simplify the *calculatePoints* function by combining the pattern matching for *CorrectScore* and *CorrectResult* into a new function:

```
let resultScore (expected:Score) (actual:Score) =
    match (expected, actual) with
    | CorrectScore -> 400
    | CorrectResult -> 100
    | _ -> 0
```

Note that we had to return 400 from *CorrectScore* in this function as we are no longer able to add the *CorrectResult* points later. This allows us to simplify the *calculatePoints* function:

```
let calculatePoints (expected:Score) (actual:Score) =
    let pointsForResult = resultScore expected actual
    let pointsForGoals = goalScore expected actual
    pointsForResult + pointsForGoals
```

As the *resultScore* and *goalScore* functions have the same signature, we can use a higher order function to remove the duplication:

```
let calculatePoints (expected:Score) (actual:Score) =
    [ resultScore; goalScore ]
    |> List.sumBy (fun f -> f expected actual)
```

Yes, we can put functions into a List. `List.sumBy` is equivalent to `List.map` followed by `List.sum`.

There are other types of active patterns that we haven't met in this chapter; I recommend that you investigate the F# Documentation to make yourself familiar with them.

Summary

Active patterns are very useful and powerful features but they are not always the best approach. When used well, they improve readability.

In the next chapter we will be taking some of the features we've met in this chapter to add validation to the code we used in chapter 6.

8 - Functional Validation

In this chapter, we are going to look at adding validation to the code that we created in Chapter 6. We will make use of active patterns and we will see how you can easily model domain errors without using exceptions. At the end of the chapter, we will have a quick look at one of the more interesting functional patterns which is perfect for the validation that we are doing.

Setting Up

Create a new folder for the code in this chapter.

Create a console application by typing the following into a terminal window:

```
dotnet new console -lang F#
```

Using the Explorer view, create a folder called *resources* in the code folder for this chapter and then create a new file called *customers.csv*. Copy the following data into the new file:

```
CustomerId|Email|Eligible|Registered|DateRegistered|Discount
John|john@test.com|1|1|2015-01-23|0.1
Mary|mary@test.com|1|1|2018-12-12|0.1
Richard|richard@nottest.com|0|1|2016-03-23|0.0
Sarah||0|0||
```

Replace the code in *Program.fs* with the following which is where we left the code at the end of Chapter 6:

```
open System
open System.IO

type Customer = {
    CustomerId : string
    Email : string
    IsEligible : string
    IsRegistered : string
    DateRegistered : string
    Discount : string
}
```

```
type DataReader = string -> Result<string seq,exn>

let readFile : DataReader =
    fun path ->
        try
            seq {
                use reader = new StreamReader(File.OpenRead(path))
                while not reader.EndOfStream do
                    reader.ReadLine()
            }
            |> Ok
        with
        | ex -> Error ex

let parseLine (line:string) : Customer option =
    match line.Split('|') with
    | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
        Some {
            CustomerId = customerId
            Email = email
            IsEligible = eligible
            IsRegistered = registered
            DateRegistered = dateRegistered
            Discount = discount
        }
    | _ -> None

let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id

let output data =
    data
    |> Seq.iter (fun x -> printfn "%A" x)

let import (dataReader:DataReader) path =
    match path |> dataReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex.Message
```

```
[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import readFile
    0
```

Run it using `dotnet run` in the terminal.

Solving the Problem

We are going to take each unvalidated customer record, validate it, and if it is valid, convert it to a validated customer.

The first thing we need to do is create a new record type that will store our validated data. Add this type below the *Customer* record type definition:

```
type ValidatedCustomer = {
    CustomerId : string
    Email : string option
    IsEligible : bool
    IsRegistered : bool
    DateRegistered : DateTime option
    Discount : decimal option
}
```

We need to add a new function to create a *ValidatedCustomer*:

```
// string -> string option -> bool -> bool -> DateTime option -> decimal option
// -> ValidatedCustomer
let create customerId email isEligible isRegistered dateRegistered discount =
    {
        CustomerId = customerId
        Email = email
        IsEligible = isEligible
        IsRegistered = isRegistered
        DateRegistered = dateRegistered
        Discount = discount
    }
```

Now we need to think about how we handle validation errors. The first thing that we need to do is to consider what types of errors we expect. The obvious ones are missing data and invalid data. Let's create the validation error type as a discriminated union:

```
type ValidationError =
    | MissingData of name: string
    | InvalidData of name: string * value: string
```

For missing data, we only need the name of the item but for invalid data, we want the name and the value that failed.

You will need to add an import declaration at the top of the file so that we can use regular expressions:

```
open System.Text.RegularExpressions
```

We need to create some partial active patterns to handle parsing string data:

```
let (|ParseRegex|_|) regex str =
    let m = Regex(regex).Match(str)
    if m.Success then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None

let (|IsValidEmail|_|) input =
    match input with
    | ParseRegex ".*?@(.*)" [ _ ] -> Some input
    | _ -> None

let (|IsEmptyString|_|) (input:string) =
    if input.Trim() = "" then Some () else None

let (|IsDecimal|_|) (input:string) =
    let success, value = Decimal.TryParse input
    if success then Some value else None

let (|IsBoolean|_|) (input:string) =
    match input with
    | "1" -> Some true
    | "0" -> Some false
    | _ -> None

let (|IsValidDate|_|) (input:string) =
    let (success, value) = input |> DateTime.TryParse
    if success then Some value else None
```

Now let's create validate functions using our active patterns and the new *ValidationError* discriminated union type:

```
// string -> Result<string, ValidationError>
let validateCustomerId customerId =
    if customerId <> "" then Ok customerId
    else Error (MissingData "CustomerId")

// string -> Result<string option, ValidationError>
let validateEmail email =
    if email <> "" then
        match email with
        | IsValidEmail _ -> Ok (Some email)
        | _ -> Error (InvalidData ("Email", email))
    else
        Ok None

// string -> Result<bool, ValidationError>
let validateIsEligible (isEligible:string) =
    match isEligible with
    | IsBoolean b -> Ok b
    | _ -> Error (InvalidData ("IsEligible", isEligible))

// string -> Result<bool, ValidationError>
let validateIsRegistered (isRegistered:string) =
    match isRegistered with
    | IsBoolean b -> Ok b
    | _ -> Error (InvalidData ("IsRegistered", isRegistered))

// string -> Result<DateTime option, ValidationError>
let validateDateRegistered (dateRegistered:string) =
    match dateRegistered with
    | IsEmptyString -> Ok None
    | IsValidDate dt -> Ok (Some dt)
    | _ -> Error (InvalidData ("DateRegistered", dateRegistered))

// string -> Result<decimal option, ValidationError>
let validateDiscount discount =
    match discount with
    | IsEmptyString -> Ok None
    | IsDecimal value -> Ok (Some value)
    | _ -> Error (InvalidData ("Discount", discount))
```

We now need to create a function that uses our new validation functions on the properties of the customer record and creates a *ValidatedCustomer* instance:

```
let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    let customerId = input.CustomerId |> validateCustomerId
    let email = input.Email |> validateEmail
    let isEligible = input.IsEligible |> validateIsEligible
    let isRegistered = input.IsRegistered |> validateIsRegistered
    let dateRegistered = input.DateRegistered |> validateDateRegistered
    let discount = input.Discount |> validateDiscount
    // This won't compile
    create customerId email isEligible isRegistered dateRegistered discount
```

Notice that we have added the expected return type to the new function. This is good practice with some functions since it forces us to fix any issues to make the code compile.

We can see that we have a problem; the `create` function isn't expecting *Result* types from the validation functions. With the skills and knowledge that we have gained so far, we can solve this.

Firstly, we create a couple of helper functions to extract *Error* and *Ok* data from each validation function:

```
// Result<'a, ValidationError> -> ValidationError list
let getError input =
    match input with
    | Ok _ -> []
    | Error ex -> [ ex ]

// Result<'a, ValidationError> -> 'a
let getValue input =
    match input with
    | Ok v -> v
    | _ -> failwith "Oops, you shouldn't have got here!"
```

Now we create a list of potential errors using a list comprehension, concatenate them using `List.concat`, and then check to see if the result has any errors. If there are no errors, we can safely call the create function:

```
let validate (input:Customer) : Result<ValidatedCustomer, ConversionError list> =
    let customerId = input.CustomerId |> validateCustomerId
    let email = input.Email |> validateEmail
    let isEligible = input.IsEligible |> validateIsEligible
    let isRegistered = input.IsRegistered |> validateIsRegistered
    let dateRegistered = input.DateRegistered |> validateDateRegistered
    let discount = input.Discount |> validateDiscount
    let errors =
        [
            customerId |> getError
            email |> getError
            isEligible |> getError
            isRegistered |> getError
            dateRegistered |> getError
            discount |> getError
        ]
        |> List.concat
    match errors with
    | [] -> Ok (create (customerId |> getValue) (email |> getValue) (isEligible |> g\
etValue) (isRegistered |> getValue) (dateRegistered |> getValue) (discount|> getValu\
e))
    | _ -> Error errors
```

Finally, we need to plug the validation into the pipeline:

```
// seq<string> -> seq<Result<ValidatedCustomer, ValidationError list>>
let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id
    |> Seq.map validate
```

If you run the code using 'dotnet run' in the terminal, you should get some validated customer data as output.

Add an extra row to the customers.csv file that will fail validation:

```
|||||
```

If you run the code again, the last item in the output to the terminal will be an error.

Where are we now?

This is the code we have ended up with:

```
open System
open System.IO
open System.Text.RegularExpressions

type Customer = {
    CustomerId : string
    Email : string
    IsEligible : string
    IsRegistered : string
    DateRegistered : string
    Discount : string
}

type ValidatedCustomer = {
    CustomerId : string
    Email : string option
    IsEligible : bool
    IsRegistered : bool
    DateRegistered : DateTime option
    Discount : decimal option
}

type ValidationError =
| MissingData of name: string
| InvalidData of name: string * value: string

type FileReader = string -> Result<string seq, exn>

let readFile : FileReader =
    fun path ->
        try
            seq {
                use reader = new StreamReader(File.OpenRead(path))
                while not reader.EndOfStream do
                    yield reader.ReadLine()
            }
            |> Ok
        with
```

```
        | ex -> Error ex

let parseLine (line:string) : Customer option =
    match line.Split('|') with
    | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
        Some {
            CustomerId = customerId
            Email = email
            IsEligible = eligible
            IsRegistered = registered
            DateRegistered = dateRegistered
            Discount = discount
        }
    | _ -> None

let (|ParseRegex|_|) regex str =
    let m = Regex(regex).Match(str)
    if m.Success then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None

let (|IsValidEmail|_|) input =
    match input with
    | ParseRegex ".*?@(.*)" [ _ ] -> Some input
    | _ -> None

let (|IsEmptyString|_|) (input:string) =
    if input.Trim() = "" then Some () else None

let (|IsDecimal|_|) (input:string) =
    let success, value = Decimal.TryParse input
    if success then Some value else None

let (|IsBoolean|_|) (input:string) =
    match input with
    | "1" -> Some true
    | "0" -> Some false
    | _ -> None

// string -> Result<string, ValidationError>
let validateCustomerId customerId =
    if customerId <> "" then Ok customerId
    else Error (MissingData "CustomerId")
```

```
// string -> Result<string option, ValidationError>
let validateEmail email =
    if email <> "" then
        match email with
        | IsValidEmail _ -> Ok (Some email)
        | _ -> Error (InvalidData ("Email", email))
    else
        Ok None

// string -> Result<bool, ValidationError>
let validateIsEligible (isEligible:string) =
    match isEligible with
    | IsBoolean b -> Ok b
    | _ -> Error (InvalidData ("IsEligible", isEligible))

// string -> Result<bool, ValidationError>
let validateIsRegistered (isRegistered:string) =
    match isRegistered with
    | IsBoolean b -> Ok b
    | _ -> Error (InvalidData ("IsRegistered", isRegistered))

// string -> Result<DateTime option, ValidationError>
let validateDateRegistered (dateRegistered:string) =
    match dateRegistered with
    | IsEmptyString -> Ok None
    | IsValidDate dt -> Ok (Some dt)
    | _ -> Error (InvalidData ("DateRegistered", dateRegistered))

// string -> Result<decimal option, ValidationError>
let validateDiscount discount =
    match discount with
    | IsEmptyString -> Ok None
    | IsDecimal value -> Ok (Some value)
    | _ -> Error (InvalidData ("Discount", discount))

let getError input =
    match input with
    | Ok _ -> []
    | Error ex -> [ ex ]

let getValue input =
    match input with
    | Ok v -> v
```

```
    | _ -> failwith "Oops, you shouldn't have got here!"

let create customerId email isEligible isRegistered dateRegistered discount =
    {
        CustomerId = customerId
        Email = email
        IsEligible = isEligible
        IsRegistered = isRegistered
        DateRegistered = dateRegistered
        Discount = discount
    }

let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    let customerId = input.CustomerId |> validateCustomerId
    let email = input.Email |> validateEmail
    let isEligible = input.IsEligible |> validateIsEligible
    let isRegistered = input.IsRegistered |> validateIsRegistered
    let dateRegistered = input.DateRegistered |> validateDateRegistered
    let discount = input.Discount |> validateDiscount
    let errors =
        [
            customerId |> getError;
            email |> getError;
            isEligible |> getError;
            isRegistered |> getError;
            dateRegistered |> getError;
            discount |> getError
        ]
        |> List.concat
    match errors with
    | [] -> Ok (create (customerId |> getValue) (email |> getValue) (isEligible |> g\
etValue)
                    (isRegistered |> getValue) (dateRegistered |> getValue) (discount|> getValue))
    | _ -> Error errors

let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id
    |> Seq.map validate

let output data =
```

```
    data
    |> Seq.iter (fun x -> printfn "%A" x)

let import (fileReader:FileReader) path =
    match path |> fileReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex

[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import readFile
    0
```

This code works well but there is a more idiomatic way of handling this type of problem in F#, and most functional programming languages called Applicatives. It works in a similar way to our version but more elegantly.

Functional Validation the F# Way

We are going to use the *Computation Expression* support for *Applicatives* that was introduced in F# 5. We are not going to deep dive into computation expressions in this chapter but will do so in Chapter 12. It is enough at this stage to know how they can be used.

> If you want to see an explanation of using *Applicatives* for validation using the idiomatic style prior to F# 5, have a look at the post on Functional Validation in F# Using Applicatives[21] that I wrote for the 2019 F# Advent Calendar[22]. It's well worth understanding how it works under the covers as it builds on many of the features we have encountered so far.

The first thing that we need to do is to convert each of our individual errors into *ValidationError* lists. We can use the built-in `Result.mapError` function to help us to do this:

[21] https://trustbit.tech/blog/2019/12/09/functional-validation-in-f-using-applicatives
[22] https://sergeytihon.com/2019/11/05/f-advent-calendar-in-english-2019/

```
let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    let customerId =
        input.CustomerId
        |> validateCustomerId
        |> Result.mapError (fun ex -> [ ex ])
    let email =
        input.Email
        |> validateEmail
        |> Result.mapError (fun ex -> [ ex ])
    let isEligible =
        input.IsEligible
        |> validateIsEligible
        |> Result.mapError (fun ex -> [ ex ])
    let isRegistered =
        input.IsRegistered
        |> validateIsRegistered
        |> Result.mapError (fun ex -> [ ex ])
    let dateRegistered =
        input.DateRegistered
        |> validateDateRegistered
        |> Result.mapError (fun ex -> [ ex ])
    let discount =
        input.Discount
        |> validateDiscount
        |> Result.mapError (fun ex -> [ ex ])
    // Compile problem
    create customerId email isEligible isRegistered dateRegistered discount
```

Rather than using `(fun ex -> [ ex ])` we could use another of the functions from the *List* module, `singleton`, which does the same thing:

```
let discount =
        input.Discount
        |> validateDiscount
        |> Result.mapError List.singleton
```

The other change that we could make is to decide that our individual validation helper functions are only used by this `validate` function, in which case, we could make each one return a list of *ValidationError*. This would simplify the `validate` function:

```
let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    let customerId = input.CustomerId |> validateCustomerId
    let email = input.Email |> validateEmail
    let isEligible = input.IsEligible |> validateIsEligible
    let isRegistered = input.IsRegistered |> validateIsRegistered
    let dateRegistered = input.DateRegistered |> validateDateRegistered
    let discount = input.Discount |> validateDiscount
    // Compile problem
    create customerId email isEligible isRegistered dateRegistered discount
```

Which style you decide to take is not really that important but you should try to stick to one.

Now back to fixing our compile problem.

We are going to use some code from the FsToolkit.ErrorHandling NuGet package. To install it in our project, we need to add the package via the Terminal:

```
dotnet add package FsToolkit.ErrorHandling
```

After the package has downloaded, add the following import declaration:

```
open FsToolkit.ErrorHandling.ValidationCE
```

To finish, we need to plug in the validation computation expression:

```
let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    validation {
        let! customerId =
            input.CustomerId
            |> validateCustomerId
            |> Result.mapError (fun ex -> [ ex ])
        and! email =
            input.Email
            |> validateEmail
            |> Result.mapError (fun ex -> [ ex ])
        and! isEligible =
            input.IsEligible
            |> validateIsEligible
            |> Result.mapError (fun ex -> [ ex ])
        and! isRegistered =
            input.IsRegistered
            |> validateIsRegistered
            |> Result.mapError (fun ex -> [ ex ])
```

```
        and! dateRegistered =
            input.DateRegistered
            |> validateDateRegistered
            |> Result.mapError (fun ex -> [ ex ])
        and! discount =
            input.Discount
            |> validateDiscount
            |> Result.mapError (fun ex -> [ ex ])
        return create customerId email isEligible isRegistered dateRegistered discou\
nt
    }
```

The key thing to notice is the `and` keyword that is new to F# in version 5.

The bang unwraps the value from the effect, in this case, the value from the *Ok* track of the validated item. Move your mouse over `customerId` and the tooltip will tell you it is a string. Remove the bang and the tooltip will inform you that *customerId* is a `Result<string, ValidationError list>`.

If we had used `let` rather than `and`, the code would have only returned the first error found and then would not have the other lines as it would be on the Error track. The style before `and` was introduced is referred to as *monadic*, as opposed to the *applicative* style we are now using where the use of `and` forces the code to run all of the validation, even if there are validation errors reported. The final function to create a *ValidatedCustomer* is only called if there are no errors. It ends up doing exactly what we did initially but much more elegantly!

Computation expressions are the only place in F# where the `return` keyword is required.

Don't worry if you don't fully understand this code as we will be covering computation expressions in detail in Chapter 12 and when we start with web development in Chapter 13.

Check that everything is working by running the code from the terminal:

```
dotnet run
```

Applicatives are very useful for a range of things, so it's an important pattern to know, even in the old, pre-computation expression style.

Summary

In this chapter, we have looked at how we can add validation by using active patterns and how easy it is to add additional functionality into the data processing pipeline. We also looked at a more elegant solution than our original one to the validation problem using applicatives with the computation expression support introduced in F# 5.

In the next chapter, we will look at improving the code from the first chapter by using more domain terminology and reducing the use of primitives.

9 - Single-Case Discriminated Union

In this chapter, we are going to improve the robustness and readability of our code by increasing our use of domain concepts and reducing our use of primitives. The primary approach adopted by the F# community for this task is the Single-Case Discriminated Union.

Setting Up

We are going to improve the code that we wrote in Chapter 1.

Create a new folder for the code in this chapter and open the new folder in VS Code.

Add a new file called *code.fsx*.

Solving the Problem

This is where we left the code from the first chapter of this book:

```
type RegisteredCustomer = {
        Id : string
}

type UnregisteredCustomer = {
    Id : string
}

type Customer =
    | Eligible of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Eligible _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

```
let john = Eligible { Id = "John" }
let mary = Eligible { Id = "Mary" }
let richard = Registered { Id = "Richard" }
let sarah = Guest { Id = "Sarah" }

// 'a -> 'a -> bool
let isEqualTo expected actual =
        actual = expected

let assertJohn = calculateTotal john 100.0M |> isEqualTo 90.0M
let assertMary = calculateTotal mary 99.0M |> isEqualTo 99.0M
let assertRichard = calculateTotal richard 100.0M |> isEqualTo 100.0M
let assertSarah = calculateTotal sarah 100.0M |> isEqualTo 100.0M
```

Copy the code into *code.fsx*.

It's nice but we still have some primitives where we should have domain concepts (Spend and Total). I would like the signature of the `calculateTotal` function to change from `Customer -> decimal -> decimal` to `Customer -> Spend -> Total`. The easiest way to achieve this is to use Type Abbreviations, which are simple aliases:

```
type Spend = decimal
type Total = decimal
```

To use the new types, we simply decorate the `calculateTotal` function input parameters and output with them:

```
// Customer -> Spend -> Total
let calculateTotal (customer:Customer) (spend:Spend) : Total =
    let discount =
        match customer with
        | Eligible _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

An alternative would be to create a type abbreviation for the function signature:

```
type CalculateTotal = Customer -> Spend -> Total

//Customer -> Spend -> Total
let calculateTotal : CalculateTotal =
    fun customer spend ->
        let discount =
            match customer with
            | Eligible _ when spend >= 100.0M -> spend * 0.1M
            | _ -> 0.0M
        spend - discount
```

Note the change in the way that input parameters are used with the type abbreviation.

Either approach gives us the signature we want. Both styles are useful to know, but for the rest of this chapter, we will use the first style without the type abbreviation for the function signature.

There is a potential problem with type abbreviations; if the underlying type matches, I can use anything for the input, not just the limited range of values expected for *Spend*. There is nothing stopping you from supplying either an invalid value or the wrong value to a parameter as shown in this example code with GPS coordinates:

```
type Latitude = decimal
type Longitude = decimal

type GpsCoordinate = { Latitude: Latitude; Longitude: Longitude }

// Latitude -90° to 90°
// Longitude -180° to 180°
let badGps : GpsCoordinate = { Latitude = 1000M; Longitude = -345M }

let latitude = 46M
let longitude = 15M

// Swap latitude and longitude
let badGps2 : GpsCoordinate = { Latitude = longitude; Longitude = latitude }
```

We can write tests to prevent this from happening but that is additional work. Instead, we want the type system to help us to write safer code and prevent this from happening. Thankfully, there are a number of ways that we can prevent these scenarios in F#. We will start with the Single-Case Discriminated Union.

Let's define one for *Spend* by replacing the existing code with:

```
type Spend = Spend of decimal
```

If you hover your mouse over the first spend you will see the following tooltip:

```
union Spend =
        | Spend of decimal
```

It is the convention to omit the bar (|) if this discriminated union was intended to only ever contain a single case, hence the name, single-case discriminated union. If a discriminated union currently only has one case but may have others in the future, you would generally include the bar.

You will notice that the `calculateTotal` function now has errors. We can fix that by deconstructing the *Spend* parameter value in the function using pattern matching:

```
let calculateTotal (customer:Customer) (spend:Spend) : Total =
    let (Spend value) = spend
    let discount =
        match customer with
        | Eligible _ when value >= 100.0M -> value * 0.1M
        | _ -> 0.0M
    value - discount
```

If the *Spend* type had been defined as `type Spend = xSpend of decimal`, the deconstructor would have been `xSpend value` rather than `Spend value`.

We need to change the asserts to use the new type constructor:

```
let assertJohn = calculateTotal john (Spend 100.0M) |> isEqualTo 90.0M
let assertMary = calculateTotal mary (Spend 99.0M) |> isEqualTo 99.0M
let assertRichard = calculateTotal richard (Spend 100.0M) |> isEqualTo 100.0M
let assertSarah = calculateTotal sarah (Spend 100.0M) |> isEqualTo 100.0M
```

The `calculateTotal` function is now safer but less readable. There is a simple fix for this, we can deconstruct the value in the function parameter directly:

```
// Customer -> decimal -> decimal
let calculateTotal customer (Spend spend) =
    let discount =
        match customer with
        | Eligible _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

If you replace all of your primitives and type abbreviations with single-case discriminated unions, you cannot supply the wrong parameter as the compiler will stop you. Of course, it doesn't stop the determined from abusing our code but it is another hurdle they must overcome.

The next improvement is to restrict the range of values that the *Spend* type can accept since very few domain values will be unbounded. We will restrict *Spend* to between 0.0M and 1000.0M. To support this, we are going to add a *ValidationError* type and prevent the direct use of the *Spend* constructor:

```
type ValidationError =
    | InputOutOfRange of string

type Spend = private Spend of decimal
    with
        member this.Value = this |> fun (Spend value) -> value
        static member Create input =
            if input >= 0.0M && input <= 1000.0M then
                Ok (Spend input)
            else
                Error (InputOutOfRange "You can only spend between 0 and 1000")
```

The use of the private accessor prevents code outside the containing module from directly accessing the type constructor. This means that to create an instance of Spend, we need to use the `Spend.Create` static member function defined on the type.

To extract the value, we use the `Value` member property defined on the instance. Although it is common to use `this` to describe the instance, it has no meaning in F#; `this` is just an identifier and we could have easily used `x` or `s`. If we hadn't used the identifier in the logic for extracting the value, we could have used the underscore _, although again it has no real meaning in this situation; it's just an identifier.

We need to make some changes to get the code to compile. Firstly we change the `calculateTotal` function to use the new type member functions:

```
// Customer -> 'a -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Eligible _ when spend.Value >= 100.0M -> spend.Value * 0.1M
        | _ -> 0.0M
    spend.Value - discount
```

Sadly, the compiler is unable to determine the type of the `spend` parameter using type inference, so we need to add the type abbreviation to it:

```
// Customer -> Spend -> decimal
let calculateTotal customer (spend:Spend) =
    let discount =
        match customer with
        | Eligible _ when spend.Value >= 100.0M -> spend.Value * 0.1M
        | _ -> 0.0M
    spend.Value - discount
```

Which type abbreviations you apply to function parameters depends on how much the code is likely to change and whether it's important to retain the same signature. Type inference works very well most of the time and you should learn to rely on it but sometimes, it makes sense to specify the types.

We also need to fix the asserts since we are now returning a `Result<Spend,ValidationError>` instead of a *Spend*:

```
let isEqualTo expected actual =
    expected = actual

let assertEqual customer spent expected =
    Spend.create spent
    |> Result.map (fun spend -> calculateTotal customer spend)
    |> isEqualTo (Ok expected)

let assertJohn = assertEqual john 100.0M 90.0M
let assertMary = assertEqual mary 99.0M 99.0M
let assertRichard = assertEqual richard 100.0M 100.0M
let assertSarah = assertEqual sarah 100.0M 100.0M
```

We have done some extra work to the Spend type but by doing so we have ensured that an instance of it can never be invalid.

As a piece of homework, think about how you would use the features we have covered in this chapter on the code from the last chapter.

Another change that we can make is to move the discount rate from the `calculateTotal` function to be with the *Customer* type definition. The primary reason for doing this would be for re-use:

```
type Customer =
    | Eligible of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
    with
        member this.Discount =
            match this with
            | Eligible _ -> 0.1M
            | _ -> 0.0M
```

This also allows us to simplify the `calculateTotal` function:

```
let calculateTotal customer (spend:Spend) =
    let discount =
            if spend.Value >= 100.0M then spend.Value * customer.Discount
            else 0.0M
    spend.Value - discount
```

Whilst this looks nice, it has broken the link between the *Customer* type and the *Spend* value. Remember, the rule is a 10% discount if an eligible customer spends 100.0 or more. Let's have another go:

```
type Customer =
    | Eligible of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
    with
        member this.CalculateDiscountPercentage(spend:Spend) =
            match this with
            | Eligible _ ->
                if spend.Value >= 100.0M then 0.1M else 0.0M
            | _ -> 0.0M
```

> **WARNING** If the compiler shows an error, you will need to move the Spend type to before the Customer type declaration in the file as it is currently defined after it.

We now need to modify the `calculateTotal` function to use our new member function:

```
let calculateTotal customer (spend:Spend) =
    let discount = spend.Value * customer.CalculateDiscountPercentage spend
    spend.Value - discount
```

To run this code, you will need to load all of the code back into FSI using `ALT+ENTER` as we have changed quite a lot of code. Your tests should still pass.

A final change would be to simplify the `calculateTotal` function:

```
let calculateTotal customer spend =
    spend.Value * (1.0M - customer.CalculateDiscountPercentage spend)
```

Having done all of this work, I'm going to suggest that we don't use this approach. We should try to maintain a clean separation between data and behaviour. This isn't a rule, just strong guidance. Thankfully, there are other approaches that we can use instead.

Using Modules

The last approach we will look at is to use a module with the same name as the type. This style is used throughout F#. Let's move our function to a new module:

```
type Customer =
    | Eligible of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer

module Customer =
        let calculateDiscountPercentage (spend:Spend) customer =
        match customer with
        | Eligible _ -> if spend.Value >= 100.0M then 0.1M else 0.0M
        | _ -> 0.0M
```

You may have noticed when dealing with other modules like *List* or *Option* that it is the convention to pass in the instance of the defining type as the last parameter.

We again need to modify our `calculateTotal` function:

```
let calculateTotal customer spend =
    let discountPercentage = customer |> Customer.calculateDiscountPercentage spend
    spend.Value * (1.0M - discountPercentage)
```

An alternative style would be to do the following, which some may find more readable:

```
let calculateTotal customer spend =
    customer
    |> Customer.calculateDiscountPercentage spend
    |> fun discountPercentage -> spend.Value * (1.0M - discountPercentage)
```

It might also be a good idea to add the `calculateTotal` function to the *Customer* module:

```
type Customer =
    | Eligible of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer

module Customer =

    let calculateDiscountPercentage spend customer =
        match customer with
        | Eligible _ -> if Spend.Value spend >= 100.0M then 0.1M else 0.0M
        | _ -> 0.0M

    let calculateTotal customer spend =
        customer
        |> calculateDiscountPercentage spend
        |> fun discountPercentage -> spend.Value * (1.0M - discountPercentage)
```

You can also move the functions for the *Spend* type to a module as well.

```
type Spend = private Spend of decimal

module Spend =

    let value input = input |> fun (Spend value) -> value

    let create input =
        if input >= 0.0M && input <= 1000.0M then
            Ok (Spend input)
        else
            Error (InputOutOfRange "You can only spend between 0 and 1000")
```

We need to make a few small changes to our code to support these changes:

```
module Customer =

    let calculateDiscountPercentage spend customer =
        match customer with
        | Eligible _ -> if Spend.value spend >= 100.0M then 0.1M else 0.0M
        | _ -> 0.0M

    let calculateTotal customer spend =
        customer
        |> calculateDiscountPercentage spend
        |> fun discountPercentage -> Spend.value spend * (1.0M - discountPercentage)
```

We also need to make one change to the assertion helper functions as we have moved the `calculateTotal` function to the Customer module:

```
let assertEqual customer spent expected =
    Spend.create spent
    |> Result.map (fun spend -> Customer.calculateTotal customer spend)
    |> isEqualTo (Ok expected)
```

If you run all of the code through FSI, your asserts will now pass again.

A Few Minor Improvements

There are a few minor improvements that we can implement quite easily. Firstly, we will replace the string used in the *Customer* record types with a single-case discriminated union called *CustomerId*. We will add type abbreviations for *Total* and *DiscountPercentage*. Finally, we need to fix the compile errors. This is what we end up with:

```
type CustomerId = CustomerId of string

type RegisteredCustomer = {
    Id : CustomerId
}

type UnregisteredCustomer = {
    Id : CustomerId
}

type ValidationError =
    | InputOutOfRange of string
```

```
type Spend = private Spend of decimal

module Spend =

    let value input = input |> fun (Spend value) -> value

    let create input =
        if input >= 0.0M && input <= 1000.0M then
            Ok (Spend input)
        else
            Error (InputOutOfRange "You can only spend between 0 and 1000")

type Total = decimal
type DiscountPercentage = decimal

type Customer =
    | Eligible of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer

module Customer =

    // Spend -> Customer -> DiscountPercentage
    let calculateDiscountPercentage spend customer : DiscountPercentage =
        match customer with
        | Eligible _ -> if Spend.value spend >= 100.0M then 0.1M else 0.0M
        | _ -> 0.0M

    // Customer -> Spend -> Total
    let calculateTotal customer spend : Total =
        customer
        |> calculateDiscountPercentage spend
        |> fun discountPercentage -> Spend.value spend * (1.0M - discountPercentage)

let john = Eligible { Id = CustomerId "John" }
let mary = Eligible { Id = CustomerId "Mary" }
let richard = Registered { Id = CustomerId "Richard" }
let sarah = Guest { Id = CustomerId "Sarah" }

let isEqualTo expected actual =
    expected = actual
```

```
let assertEqual customer spent expected =
    Spend.create spent
    |> Result.map (fun spend -> Customer.calculateTotal customer spend)
    |> isEqualTo (Ok expected)

let assertJohn = assertEqual john 100.0M 90.0M
let assertMary = assertEqual mary 99.0M 99.0M
let assertRichard = assertEqual richard 100.0M 100.0M
let assertSarah = assertEqual sarah 100.0M 100.0M
```

We now have quite a bit more code than we started with but it is much more robust and domain-centric which was our original goal for this chapter.

Using a Record Type

You can also use a record type to serve the same purpose as the single-case discriminated union:

```
type Spend = private { Spend : decimal }

module Spend =

    let value input = input.Spend

    let create input =
        if input >= 0.0M && input <= 1000.0M then
            Ok { Spend = input }
        else
            Error (InputOutOfRange "You can only spend between 0 and 1000")
```

Summary

We have covered quite a lot in this chapter but reducing our usage of raw primitives and adding more domain-centric types will improve the quality of our code as well as making more readable.

In this post we have learned about single-case discriminated unions. They allow us to restrict the range of values that a type can accept compared to raw primitives. We have also seen that we can extend types by adding helper functions and properties to them and the use of modules to do the same.

In the next chapter, we will look at object programming in F#.

10 - Object Programming

In this chapter, we are going to see how we can utilise some of the object programming features that F# offers. F# is a functional-first language but sometimes it's beneficial to use objects, particularly when interacting with code written in other, less functional .NET languages or when you want to encapsulate some internal data structures and/or mutable state.

F# can do most of the things that C#/VB.Net can do with objects but we are going to concentrate on the core object programming features; class types, interfaces, encapsulation, and equality.

Setting Up

Create a new folder for the chapter code and open it in VS Code.

Add three new files, *FizzBuzz.fsx*, *RecentlyUsedList.fsx* and *Coordinate.fsx*.

Class Types

We will start in *FizzBuzz.fsx* where we will be implementing FizzBuzz using object programming.

We are going to create our first class type and use the code from the `fizzBuzz` function we created in Chapter 7:

```
type FizzBuzz() =
    member _.Calculate(value) =
        [(3, "Fizz");(5, "Buzz")]
        |> List.map (fun (v, s) -> if value % v = 0 then s else "")
        |> List.reduce (+)
        |> fun s -> if s = "" then string value else s
```

Points of interest:

- The parentheses after the type name are required. They can contain tupled arguments as we will see later in the chapter.
- The member keyword defines the accessible members of the type.
- The _ is just a placeholder - it can be anything. It is the convention to use one of _ or `this`.

Now that we have created our class type, we need to instantiate it to use it:

```
let fizzBuzz = FizzBuzz()
```

Unlike C#, we don't use *new* when creating an instance of a class type in F# except when it implements `IDisposible<'T>` and then we would write `use` instead of `let` to create a scope block.

Now we can call the `Calculate` member function on the instance:

```
let fifteen = fizzBuzz.Calculate(15) // FizzBuzz
```

Whilst the parentheses for the parameter is optional, it's worth including them as the constructor arguments are a tuple and it helps me to see that I'm calling an object member. You can use the forward pipe should you wish:

```
let fifteen = 15 |> fizzBuzz.Calculate // FizzBuzz
```

Let's create a function that will use our new *FizzBuzz* object type:

```
// int list -> string list
let doFizzBuzz range =
    let fizzBuzz = FizzBuzz()
    range
    |> List.map (fun n -> fizzBuzz.Calculate(n))

let output = doFizzBuzz [1..15]
```

The code can be simplified to:

```
// int list -> string list
let doFizzBuzz range =
    let fizzBuzz = FizzBuzz()
    range
    |> List.map fizzBuzz.Calculate

let output = doFizzBuzz [1..15]
```

At the moment, we can only use [(3, "Fizz");(5, "Buzz")] as the mapping but it is easy to pass the mapping in through the default constructor:

```
type FizzBuzz(mapping) =
    member _.Calculate(value) =
        mapping
        |> List.map (fun (v, s) -> if value % v = 0 then s else "")
        |> List.reduce (+)
        |> fun s -> if s = "" then string value else s
```

Notice that we don't need to assign the constructor argument with a let binding to use it.

Now we need to pass the mapping in as the argument to the constructor in the doFizzBuzz function:

```
let doFizzBuzz mapping range =
    let fizzBuzz = FizzBuzz(mapping)
    range
    |> List.map fizzBuzz.Calculate

let output = doFizzBuzz [(3, "Fizz");(5, "Buzz")] [1..15]
```

We can move the function code in the object from the member into the body of the class type as a new inner function:

```
type FizzBuzz(mapping) =
    let calculate n =
        mapping
        |> List.map (fun (v, s) -> if n % v = 0 then s else "")
        |> List.reduce (+)
        |> fun s -> if s = "" then string value else s

    member _.Calculate(value) = calculate value
```

You cannot access the new `calculate` function from outside the class type, only through the `Calculate` member. You don't have to do this but I find that it makes the code easier to read, especially as the number of class members increases.

Interfaces

Interfaces are very important in object programming as they define a contract that an implementation must handle. Let's create an interface to use in our fizzbuzz example:

```
// The 'I' prefix to the name is not required but is used by convention in .NET
type IFizzBuzz =
    abstract member Calculate : int -> string
```

Abstract Classes

To convert IFizzBuzz into an abstract class, decorate it with the [<AbstractClass>] attribute.

Now we need to implement the interface in our *FizzBuzz* class type:

```
type FizzBuzz(mapping) =
    let calculate n =
        mapping
        |> List.map (fun (v, s) -> if n % v = 0 then s else "")
        |> List.reduce (+)
        |> fun s -> if s <> "" then s else string n

    interface IFizzBuzz with
        member _.Calculate(value) = calculate value
```

Nice and easy but you will see that we have a problem; the compiler has highlighted the `fizzBuzz.Calculate` method call in our `doFizzBuzz` function. This is because the `fizzBuzz` instance doesn't have a `Calculate` member, the interface *IFizzBuzz* does:

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz([(3, "Fizz");(5, "Buzz")])
    [1..15]
    |> List.map (fun n -> fizzBuzz.Calculate(n)) // Problem
```

This is a problem because F# does not support implicit casting, so we have to upcast the instance to the *IFizzBuzz* type ourselves:

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz([(3, "Fizz");(5, "Buzz")]) :> IFizzBuzz //Upcast
    [1..15]
    |> List.map (fun n -> fizzBuzz.Calculate(n)) // Fixed
```

An alternative would be to upcast as you use the interface function:

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz([(3, "Fizz");(5, "Buzz")])
    [1..15]
    |> List.map (fun n -> (fizzBuzz :> IFizzBuzz).Calculate(n))
```

If you have come from a language like C# which supports implicit casting, it may seem odd to have to explicitly cast in F# but it gives you the safety of being certain about what your code is doing as you are not relying on compiler magic.

The code above is designed to show how to construct class types and use interfaces. If you find yourself constructing interfaces with one function, ask yourself if you really, really need the extra code and complexity or whether a simple function is enough.

There is another feature that interfaces offer that we haven't covered so far: Object Expressions.

Object Expressions

Create a new script file called *expression.fsx* and add the following import declaration:

```
open System
```

Now we are going to create a new interface:

```
type ILogger =
    abstract member Info : string -> unit
    abstract member Error : string -> unit
```

We would normally create a class type:

```
type Logger() =
    interface ILogger with
        member _.Info(msg) = printfn "Info: %s" msg
        member _.Error(msg) = printfn "Error: %s" msg
```

To use this, we need to create an instance and cast it to an ILogger.

Instead, we are going to create an object expression which simplifies the usage:

```
let logger = {
    new ILogger with
        member _.Info(msg) = printfn "Info: %s" msg
        member _.Error(msg) = printfn "Error: %s" msg
}
```

We have actually created an anonymous type, so we are restricted in what we can do with it. Let's create a class type and pass the ILogger in as a constructor argument:

```
type MyClass(logger:ILogger) =
    let mutable count = 0

    member _.DoSomething input =
        logger.Info $"Processing {input} at {DateTime.UtcNow.ToString()}"
        count <- count + 1
        ()

    member _.Count = count
```

Note that we can create members of the class type that are not associated with an interface.

We can now use the class type with the object expression logger:

```
let myClass = MyClass(logger)
[1..10] |> List.iter myClass.DoSomething
printfn "%i" myClass.Count
```

Run all of these blocks of code in FSI.

Create a function that takes in an *ILogger* as a parameter:

```
let doSomethingElse (logger:ILogger) input =
    logger.Info $"Processing {input} at {DateTime.UtcNow.ToString()}"
    ()
```

It's now very easy to use the *ILogger*:

```
let logger = {
    new ILogger with
        member _.Info(msg) = printfn "Info: %s" msg
        member _.Error(msg) = printfn "Error: %s" msg
}

let doSomethingElse (logger:ILogger) input =
    logger.Info $"Processing {input} at {DateTime.UtcNow.ToString()}"
    ()

doSomethingElse logger "MyData"
```

This is a really useful feature if you have one-off services that you need to run, and for testing.

Next, we move on to a more complex example of using interfaces, a recently used list.

Encapsulation

We are going to create a recently used list as a class type. We will encapsulate a mutable collection within the class type and provide an interface for how we can interact with it. The *RecentlyUsedList* is an ordered list with the most recent item first but it is also a set, so each value can only appear once in the list.

```
type RecentlyUsedList() =
    let items = ResizeArray<string>()

    let add item =
        items.Remove item |> ignore
        items.Add item

    let get index =
        if index >= 0 && index < items.Count
        then Some items[items.Count - index - 1]
        else None

    member _.IsEmpty = items.Count = 0 // bool
    member _.Size = items.Count // int
    member _.Clear() = items.Clear() // unit -> unit
    member _.Add(item) = add item // string -> unit
    member _.TryGet(index) = get index // int -> string option
```

The `ResizeArray<'T>` is the F# synonym for the standard .NET mutable `List<'T>`. Encapsulation ensures that you cannot access it directly, only via the members in the public interface.

By looking at the signatures, we can see that `IsEmpty` and `Size` are read-only properties and that `Clear`, `Add`, and `TryGet` are functions.

Let's test our code in FSI. Run each of the following lines separately:

```
let mrul = RecentlyUsedList()

mrul.Add "Test"

mrul.IsEmpty = false // Should return true

mrul.Add "Test2"
mrul.Add "Test3"
mrul.Add "Test"

mrul.TryGet(0) = Some "Test" // Should return true
```

Let's create an interface for this class type and we'll add a maximum size (capacity) to it at the same time:

```
type IRecentlyUsedList =
    abstract member IsEmpty : bool
    abstract member Size : int
    abstract member Capacity : int
    abstract member Clear : unit -> unit
    abstract member Add : string -> unit
    abstract member TryGet : int -> string option
```

Add the capacity as a constructor argument and add the *IRecentlyUsedList* interface:

```
type RecentlyUsedList(capacity:int) =
    let items = ResizeArray<string>(capacity)

    let add item =
        items.Remove item |> ignore
        if items.Count = items.Capacity then items.RemoveAt 0
        items.Add item

    let get index =
        if index >= 0 && index < items.Count
        then Some items.[items.Count - index - 1]
        else None
```

```
    interface IRecentlyUsedList with
        member _.IsEmpty = items.Count = 0
        member _.Size = items.Count
        member _.Capacity = items.Capacity
        member _.Clear() = items.Clear()
        member _.Add(item) = add item
        member _.TryGet(index) = get index
```

Let's test our recently used list with a capacity of five:

```
let mrul = RecentlyUsedList(5) :> IRecentlyUsedList

mrul.Capacity // Should be 5

mrul.Add "Test"
mrul.Size // Should be 1
mrul.Capacity // Should be 5

mrul.Add "Test2"
mrul.Add "Test3"
mrul.Add "Test4"
mrul.Add "Test"
mrul.Add "Test6"
mrul.Add "Test7"
mrul.Add "Test"

mrul.Size // Should be 5
mrul.Capacity // Should be 5
mrul.TryGet(0) = Some "Test" // Should return true
mrul.TryGet(4) = Some "Test3" // Should return true
```

Run the code in FSI.

Encapsulation inside class types works really nicely, even for mutable data because you control access via members only.

Equality

Most of the types in F# support structural equality but class types do not. Instead, they rely on reference equality like most things in .NET. Structural equality is when two things contain the same data. Reference equality is when two things point to the same underlying instance.

Let's create a simple class type to store GPS coordinates in *Coordinate.fsx*:

```
type Coordinate(latitude: float, longitude: float) =
    member _.Latitude = latitude
    member _.Longitude = longitude
```

To test equality, we can write some simple checks to run in FSI:

```
let c1 = Coordinate(25.0, 11.98)
let c2 = Coordinate(25.0, 11.98)
let c3 = c1
c1 = c2 // false
c1 = c3 // true - reference the same instance
```

To support something that works like structural equality, we need to override the `GetHashCode` and `Equals` functions, implement `IEquatable<'T>`. If we are going to use it in other .NET languages, we need to handle the equality operator using `op_Equality` and apply the `AllowNullLiteral` attribute:

```
open System

[<AllowNullLiteral>]
type GpsCoordinate(latitude: float, longitude: float) =
    let equals (other: GpsCoordinate) =
        if isNull other then
            false
        else
            latitude = other.Latitude
            && longitude = other.Longitude

    member _.Latitude = latitude
    member _.Longitude = longitude

    override this.GetHashCode() =
        hash (this.Latitude, this.Longitude)

    override _.Equals(obj) =
        match obj with
        | :? GpsCoordinate as other -> equals other
        | _ -> false

    interface IEquatable<GpsCoordinate> with
        member _.Equals(other: GpsCoordinate) =
            equals other
```

```
    static member op_Equality(this: GpsCoordinate, other: GpsCoordinate) =
        this.Equals(other)
```

We have used two built-in functions: `hash` and `isNull`. We make use of pattern matching in the Equals function by using a match expression to see if the object passed in can be cast as a *GpsCoordinate* and if it can, it is checked for equality.

If we test this, we get the expected equality:

```
let c1 = GpsCoordinate(25.0, 11.98)
let c2 = GpsCoordinate(25.0, 11.98)
c1 = c2 // true
```

Summary

In this chapter, we have had an introduction to object programming in F#. In particular, we have looked at scoping/visibility, encapsulation, interfaces/casting, and equality. The more you interact with the rest of the .NET ecosystem, the more you will potentially need to use object programming.

We have only scratched the surface of what is possible with F# object programming. If you want to find out more about this topic (plus many other useful things), I highly recommend that you read Stylish F#[23] by Kit Eason[24].

In the next chapter we will discover how F# handles recursion.

[23] https://www.apress.com/us/book/9781484239995
[24] https://twitter.com/kitlovesfsharp

11 - Recursion

In this chapter, we are going to look at recursive functions, that is functions that call themselves in a loop. We will start with a naive implementation and then make it more efficient using an accumulator. As a special treat, I will show you a way of writing FizzBuzz and an elegant quicksort algorithm using this technique.

Setting Up

Create a new folder for the code in this chapter.

All of the code in this chapter can be run using FSI from .fsx files that you create.

Solving The Problem

We are going to start with a naive implementation of the factorial function (!):

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

To create a recursive function, we use the *rec* keyword and we would create a function like this:

```
// int -> int
let rec fact n =
    match n with
    | 1 -> 1
    | n -> n * fact (n-1)
```

You'll notice that we have two cases: a base case, in this example where n equals 1, at which point the recursion ends, and a general case of greater than 1 which is recursive. If we were to write out what happens as we run the recursion, it would look like this:

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

This is a problem because you can't perform any calculations until you've completed all of the iterations but you also need to store all of the parts as well. This means that the larger n gets, the more memory you need to perform the calculation and it can also lead to stack overflows. We can solve this problem with **Tail Call Optimisation**.

Tail Call Optimisation

There are a few possible approaches available but we are going to use an accumulator. The accumulator is passed around the recursive function on each iteration:

```
let fact n =
    let rec loop n acc =
        match n with
        | 1 -> acc
        | _ -> loop (n-1) (acc * n)
    loop n 1
```

We leave the public interface of the function intact but create an enclosed function to do the recursion. We also need to add a line at the end of the function to start the recursion and return the result. You'll notice that we've added an additional parameter `acc` which holds the accumulated value. As we are multiplying, we need to initialise the accumulator to 1. If the accumulator used addition, we would set it to 0 initially.

Again we have two cases to cover; a base case when n = 1 that returns the accumulated value and anything else where the recursion continues with the input value being decremented by 1 and the accumulator being multiplied by the input value. If we write out what happens when we run this function as we did the previous version in pseudocode, we see this:

```
fact 5 = loop 5 1
       = loop 4 5
       = loop 3 20
       = loop 2 60
       = loop 1 120
       = 120
```

This is much simpler than the previous example, requires very little memory, and is efficient.

Now that we've learnt about tail call optimisation using an accumulator, let's look at a harder example, the Fibonacci Sequence.

Expanding the Accumulator

The Fibonacci Sequence is a simple list of numbers where each value is the sum of the previous two:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

Let's start with the naive example to calculate the nth item in the sequence:

```
let rec fib (n:int64) =
    match n with
    | 0L -> 0L
    | 1L -> 1L
    | s -> fib (s-1L) + fib (s-2L)
```

If you want to see just how inefficient this is, try running `fib 50L`. It will take almost a minute on a fast machine! Let's have a go at writing a more efficient version that uses tail call optimisation:

```
let fib (n:int64) =
    let rec loop n (a,b) =
        match n with
        | 0L -> a
        | 1L -> b
        | n -> loop (n-1L) (b, a+b)
    loop n (0L,1L)
```

Let's write out what happens (ignoring the type annotation):

```
fib 5L = loop 5L (0L, 1L)
       = loop 4L (1L, 1L+0L)
       = loop 3L (1L, 1L+1L)
       = loop 2L (2L, 1L+2L)
       = loop 1L (3L, 2L+3L)
       = 3L
```

The 5th item in the sequence is indeed 3 as the list starts at index 0. Try running `fib 50L`; It should return almost instantaneously.

Next, we'll now continue on our journey to find as many functional ways of solving FizzBuzz as possible. :)

Using Recursion to Solve FizzBuzz

We start with a list of mappings that we are going to recurse over:

```
let mapping = [ (3, "Fizz"); (5, "Buzz") ]
```

Our fizzbuzz function using tail call optimisation and has an accumulator that will use string concatenation and an initial value of an empty string (`""`).

```
let fizzBuzz initialMapping n =
    let rec loop mapping acc =
        match mapping with
        | [] -> if acc = "" then string n else acc
        | head::tail ->
            let value =
                head |> (fun (div, msg) -> if n % div = 0 then msg else "")
            loop tail (acc + value)
    loop initialMapping ""
```

The pattern match is asking:

1. If there are no mappings left, then the loop completes and we return either the original input as a string if the accumulator is an empty string specifying no matches or the accumulator.
2. If there are rules left, continue the recursive loop with the tail as the mapping and add the result of the logic to the accumulator.

Finally, we can run the function and print out the results to the terminal:

```
[ 1 .. 105 ]
|> List.map (fizzBuzz mapping)
|> List.iter (printfn "%s")
```

This is quite a nice extensible approach to the FizzBuzz problem as adding `(7, "Bazz")` is trivial.

```
let mapping = [ (3, "Fizz"); (5, "Buzz"); (7, "Bazz") ]
```

Having produced a nice solution to the FizzBuzz problem using recursion, can we use the `List.fold` function to solve it? Of course we can!

```
let fizzBuzz n =
    [ (3, "Fizz"); (5, "Buzz") ]
    |> List.fold (fun acc (div, msg) ->
        if n % div = 0 then acc + msg else acc) ""
    |> fun s -> if s = "" then string n else s

[1..105]
|> List.iter (fizzBuzz >> printfn "%s")
```

We can modify the code to do all of the mapping in the fold function rather than passing the value on to another function:

```
let fizzBuzz n =
    [ (3, "Fizz"); (5, "Buzz") ]
    |> List.fold (fun acc (div, msg) ->
        match (if n % div = 0 then msg else "") with
        | "" -> acc
        | s -> if acc = string n then s else acc + s) (string n)
```

Whilst recursion is the primary mechanism for solving these types of problems in pure functional languages like Haskell, F# has some other approaches that may prove to be simpler to implement and potentially more performant.

Let's have a look at how we can solve another popular algorithm, Quicksort.

Quicksort using recursion

Quicksort[25] is a nice algorithm to create in F# because of the availability of some very useful collection functions in the List module:

[25] https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm

```
let rec qsort input =
    match input with
    | [] -> []
    | head::tail ->
        let smaller, larger = List.partition (fun n -> head >= n) tail
        List.concat [qsort smaller; [head]; qsort larger]

[5;9;5;2;7;9;1;1;3;5] |> qsort |> printfn "%A"
```

The `List.partition` function splits a list into two based on a predicate function, in this case, items smaller than or equal to the head value and those larger than it. `List.concat` converts a deeply nested sequence of lists into a single list.

Recursion with Hierarchical Data

Recursion can be used to create and deconstruct hierarchical data structures. We are going to tackle Episode 2.1 of the Trustbit Transport Tycoon challenge:

https://github.com/trustbit/exercises/blob/master/transport-tycoon_21.md

We are given a map and some CSV data detailing the distances between connected locations:

```
A,B,km
Cogburg,Copperhold,1047
Leverstorm,Irondale,673
Cogburg,Steamdrift,1269
Copperhold,Irondale,345
Copperhold,Leverstorm,569
Leverstorm,Gizbourne,866
Rustport,Cogburg,1421
Rustport,Steamdrift,1947
Rustport,Gizbourne,1220
Irondale,Gizbourne,526
Cogburg,Irondale,1034
Rustport,Irondale,1302
```

We will assume that the return journeys are the same distance.

Create a new file called *shortest-distance.fsx*.

Create a new folder called *resources*, add a file called *data.csv*, and add the data above to it.

Open the *shortest-distance.fsx* file.

Add the following declaration to the top of the file:

```
open System.IO
```

We can define a simple generic hierarchical discriminated union like this:

```
type Tree<'T> =
    | Branch of 'T * Tree<'T> seq
    | Leaf of 'T
```

Add the *Tree* type to the file.

We are going to tackle this problem in three stages:

- Load the data.
- Generate possible routes from start to finish.
- Find the shortest from the possible routes.

Let's get started!

Load the Data

We need to define a record type to hold the data that we load from the CSV file:

```
type Connection = { Start:string; Finish:string; Distance:int }
```

We next need to write a function that takes a file path and returns the loaded data:

```
// string -> Map<string, Connection list>
let loadData path =
    path
    |> File.ReadLines
    |> Seq.skip 1
    |> fun rows -> [
        for row in rows do
            match row.Split(",") with
            | [|start;finish;distance|] ->
                { Start = start; Finish = finish; Distance = int distance }
                { Start = finish; Finish = start; Distance = int distance }
            | _ -> failwith "Row is badly formed"
    ]
    |> List.groupBy (fun cn -> cn.Start)
    |> Map.ofList
```

Instead of returning a *Connection list*, we are returning *Map<string,Connection list>*. A Map is a read-only dictionary that better suits our needs since the question we will ask of this data is which routes are there from a location?

We need to call this new function with a start and finish location, so we will add that code:

```
let run start finish =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "data.csv")
    |> loadData
    |> printfn "%A"

let result = run "Cogburg" "Leverstorm"
```

Run the code in FSI. At the moment, you will see the data from the *Map*.

Find Possible Routes

We are going to define a record type that defines a waypoint on our route. It will include data on where we are, how we got there, and how far we have travelled to get here. Not only does this save time when it comes to calculating the shortest route but it will be very helpful in the next stage. Let's define our *Waypoint* type:

```
type Waypoint = { Location:string; Route:string list; TotalDistance:int }
```

We need to create a function that determines where we can go to next that we haven't already been to:

```
// Connection list -> Waypoint -> Waypoint list
let getUnvisited connections current =
    connections
    |> List.filter (fun cn -> current.Route |> List.exists (fun loc -> loc = cn.Fini\
sh) |> not)
    |> List.map (fun cn -> {
        Location = cn.Finish
        Route = cn.Start :: current.Route
        TotalDistance = cn.Distance + current.TotalDistance })
```

We pass in the Map of *Connections* and the current *Waypoint*. We can filter out those possible next locations that we have already been to on that route. Finally, we create a list of *Waypoints* with updated accumulated data.

Now we are ready to generate our Tree structure that defines all of the possible routes between the start and the finish:

```
// string -> string -> Map<string, Connection list> -> Tree<Waypoint>
let findPossibleRoutes start finish (routeMap:Map<string, Connection list>) =
    let rec loop current =
        let nextRoutes = getUnvisited routeMap[current.Location] current
        if nextRoutes |> List.isEmpty |> not && current.Location <> finish then
            Branch (current, seq { for next in nextRoutes do loop next })
        else
            Leaf current
    loop { Location = start; Route = []; TotalDistance = 0 }
```

We have two cases where a branch would not continue: when there are no locations that we can go to that we haven't already been and if the location is the finish location.

We need to add the `findPossibleRoutes` function to our `run` function:

```
let run start finish =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "data.csv")
    |> loadData
    |> findPossibleRoutes start finish
    |> printfn "%A"

let result = run "Cogburg" "Leverstorm"
```

Execute the code in FSI and you will get a tree structure built with Waypoints.

Now we need to convert our tree structure to a list of Waypoints, so that we can find the shortest route:

```
// Tree<Waypoint> -> List<Waypoint>
let rec treeToList tree =
    match tree with
    | Leaf x -> [x]
    | Branch (_, xs) -> List.collect treeToList (xs |> Seq.toList)
```

We are only interested in the leaves because they should be the only place where the finish location exists in the output.

We need to add this function to the end of the `findPossibleRoutes` function. We also need to filter out any Waypoints that do not point at the finish.

```
// string -> string -> Map<string, Connection list> -> Waypoint list
let findPossibleRoutes start finish (routeMap:Map<string, Connection list>) =
    let rec loop current =
        let nextRoutes = getUnvisited routeMap[current.Location] current
        if nextRoutes |> List.isEmpty |> not && current.Location <> finish then
            Branch (current, seq { for next in nextRoutes do loop next })
        else
            Leaf current
    loop { Location = start; Route = []; TotalDistance = 0 }
    |> treeToList
    |> List.filter (fun wp -> wp.Location = finish)
```

Execute the code in FSI and you will see that we are getting closer to completing this task.

Select Shortest Route

Our final task is to determine the shortest route from the output:

```
let selectShortestRoute routes =
    routes
    |> List.minBy (fun wp -> wp.TotalDistance)
    |> fun wp -> wp.Location :: wp.Route |> List.rev, wp.TotalDistance
```

We should add the `selectShortestRoute` function to the `run` function:

```
let run start finish =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "data.csv")
    |> loadData
    |> findPossibleRoutes start finish
    |> selectShortestRoute
    |> printfn "%A"

let result = run "Cogburg" "Leverstorm"
```

Execute the code in FSI and you will see something like this:

```
> let result = run "Cogburg" "Leverstorm";;
(["Cogburg"; "Copperhold"; "Leverstorm"], 1616)
```

This was quite a long task but notice that we broke it down into smaller sections to make it easier to resolve.

Finished Code

```
open System.IO

type Tree<'T> =
    | Branch of 'T * Tree<'T> seq
    | Leaf of 'T

type Waypoint = { Location:string; Route:string list; TotalDistance:int }

type Connection = { Start:string; Finish:string; Distance:int }

// string -> Map<string, Connection list>
let loadData path =
    path
    |> File.ReadLines
    |> Seq.skip 1
    |> fun rows -> [
        for row in rows do
            match row.Split(",") with
            | [|start;finish;distance|] ->
                { Start = start; Finish = finish; Distance = int distance }
                { Start = finish; Finish = start; Distance = int distance }
            | _ -> failwith "Row is badly formed"
    ]
    |> List.groupBy (fun cn -> cn.Start)
    |> Map.ofList

// Map<string, Connection list> -> Waypoint -> Waypoint list
let getUnvisited connections current =
    connections
    |> List.filter (fun cn -> current.Route |> List.exists (fun loc -> loc = cn.Fini\
sh) |> not)
    |> List.map (fun cn -> {
        Location = cn.Finish
        Route = cn.Start :: current.Route
        TotalDistance = cn.Distance + current.TotalDistance })

// Tree<Waypoint> -> List<Waypoint>
let rec treeToList tree =
    match tree with
    | Leaf x -> [x]
    | Branch (_, xs) -> List.collect treeToList (xs |> Seq.toList)

// string -> string -> Map<string, Connection list> -> Waypoint list
```

```
let findPossibleRoutes start finish (routeMap:Map<string, Connection list>) =
    let rec loop current =
        let nextRoutes = getUnvisited routeMap[current.Location] current
        if nextRoutes |> List.isEmpty |> not && current.Location <> finish then
            Branch (current, seq { for next in nextRoutes do loop next })
        else
            Leaf current
    loop { Location = start; Route = []; TotalDistance = 0 }
    |> treeToList
    |> List.filter (fun wp -> wp.Location = finish)

// Waypoint list -> (string list * int)
let selectShortestRoute routes =
    routes
    |> List.minBy (fun wp -> wp.TotalDistance)
    |> fun wp -> wp.Location :: wp.Route |> List.rev, wp.TotalDistance

// string -> string -> unit
let run start finish =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "data.csv")
    |> loadData
    |> findPossibleRoutes start finish
    |> selectShortestRoute
    |> printfn "%A"

let result = run "Cogburg" "Leverstorm"
```

Other Uses Of Recursion

Recursion is great for handling hierarchical data like the file system and XML, converting between flat data and hierarchies, and for event loops or loops where there is no defined finish.

As always, Scott Wlaschin's excellent site[26] has many posts on the topic.

Summary

In this chapter, we have looked at recursion in F#. In particular, we have looked at the basics and how to use accumulators with tail call optimisation to make recursion more efficient. Finally, we used recursion to help us solve a more complex problem involving hierarchical data.

In the next chapter, we will look at a feature we met in Chapter 8 on functional validation: Computation Expressions.

[26] https://fsharpforfunandprofit.com/posts/recursive-types-and-folds-3b/#series-toc

12 - Computation Expressions

In this chapter, we are going to look at computation expressions including a first look at asynchronous code in F#. Computation expressions are syntactic sugar for simplifying code when working with effects like Option, Result, and Async. Async is the asynchronous support in F# and is similar to async/await in C#, except that it is lazily evaluated. We now also have *Task* support in the F# core. We will use this in the next chapter when we start working with web development.

We will learn how to create our own simple computation expression, how to use it, and use a more complex example where we combine two effects, Async and Result, together.

We are only going to scratch the surface of what is possible with computation expressions. Once you have finished this chapter, it might be worth going back to Chapter 8 and having another look at the functional validation example which used a custom computation expression.

Setting Up

Create a new folder in VS Code, open a new Terminal to create a new console app using:

```
dotnet new console -lang F#
```

Introduction

Add a new file above *Program.fs* called *OptionDemo.fs*. In the new file, create the following code:

```
namespace ComputationExpression

module OptionDemo =

    let multiply x y = // int -> int -> int
        x * y

    let divide x y = // int -> int -> int option
        if y = 0 then None
        else Some (x / y)

    // The formula is: f x y = ((x / y) * x) / y
    let calculate x y =
```

```
        divide x y
        |> fun v -> multiply v x // compiler error
        |> fun t -> divide t y
```

We have two simple functions, `multiply` and `divide`, one of which returns an *Option*, and we are using them to compose a function called `calculate`. Don't worry about what the functions are doing, we are only interested in the effect, in this case, *Option*.

Because the order of the input parameters is not ideal for forward-piping, we have used anonymous functions to get access to the data passed down.

This function does not compile due to the divide function returning an *Option*. Let's expand out the function using the match expression to make the function compile:

```
let calculate x y =
    divide x y
    |> fun result ->
        match result with
        | Some v -> multiply v x |> Some
        | None -> None
    |> fun result ->
        match result with
        | Some t -> divide t y
        | None -> None
```

This function is much larger with all of the pattern matching but it does make it explicitly clear what the function does and shows that there is no early return.

Think about what happens in the cases where `y = 0` and `y = 1`. In the first case, the `multiply` and the second `divide` function will not be called because the code is on the None track.

The expanded match expressions follow patterns that are well known to us, which means that we can use `Option.map` and `Option.bind` to simplify the function:

```
let calculate x y =
    divide x y
    |> Option.map (fun v -> multiply v x)
    |> Option.bind (fun t -> divide t y)
```

We use `map` when the function we are applying it to does not generate the effect, in this case, *Option*, when executed and `bind` when the function does apply the effect as `divide` does.

This is much nicer but this is a chapter about computation expressions and how they can simplify coding with effects like *Option*, so let's move on.

Whilst there are some computation expressions built into the core F# language, there isn't one for *Option*, so we are going to create a basic computation expression ourselves.

Create the following code between the namespace and the module declaration in *OptionDemo.fs*:

```
[<AutoOpen>]
module Option =

    type OptionBuilder() =
        // Supports let!
        member _.Bind(x, f) = Option.bind f x
        // Supports return
        member _.Return(x) = Some x
        // Supports return!
        member _.ReturnFrom(x) = x

    // Computation Expression for Option
    // Usage will be option {...}
    let option = OptionBuilder()
```

The `[<AutoOpen>]` attribute means that when the namespace is referenced, we automatically get access to the types and functions in the module.

This is a very simple computation expression but is enough for our needs. At this stage, it's not important to fully understand how this works but you can see that we define a class type with some required member functions and then create an instance of that type for use in our code.

Comment out the `calculate` function and create the following that uses our new option computation expression:

```
// let calculate x y =
//     divide x y
//     |> Option.map (fun v -> multiply v x)
//     |> Option.bind (fun t -> divide t y)

let calculate x y =
    option {
        let! v = divide x y
        let t = multiply v x
        let! r = divide t y
        return r
    }
```

The code does exactly the same as the commented-out version but it hides the *None* track from the user, allowing you to concentrate on the happy path.

The bang (`!`) unwraps the effect using the `Bind` function, in this case, the *Option*, for the happy path. If you look at the *OptionBuilder* code, `return` does the opposite and applies the effect to the data.

If you hover your mouse over the first value, it tells you that it is an `int` value because the bang (`!`) has unwrapped the *Option* for us. Delete the bang and hover over the first value again. You'll

see that it is now an `option<int>`, so it hasn't been unwrapped. You will also get an error from the compiler. Put the bang back.

The nice thing about this function is that we don't need to know about `Option.bind` or `Option.map`; It's nearly all hidden away from us, so we can concentrate on providing the functionality.

The bindings with the `let!` get processed by the `Bind` function in the computation expression. Let bindings that would have used `Option.map` are automatically handled by the underlying computation expression code. The `return` matches the `Return` function in the computation expression. If we want to use the `ReturnFrom` function, we do the following:

```
let calculate x y =
    option {
        let! v = divide x y
        let t = multiply v x
        return! divide t y
    }
```

Notice the bang (`!`) on the return. If you look at the *OptionBuilder* code, you will notice that this does nothing to the output which is correct because the `divide` function already applies the effect.

To test the code, replace the code in *Program.fs* with the following:

```
open ComputationExpression.OptionDemo

[<EntryPoint>]
let main argv =
    calculate 8 0 |> printfn "calculate 8 0 = %A" // None
    calculate 8 2 |> printfn "calculate 8 2 = %A" // Some 16
    0
```

Leave the `0` at the end to signify the code has been completed successfully.

Now type `dotnet run` in the Terminal. You should get None and Some 16 as the output in the terminal window.

The Result Computation Expression

One of the great things about computation expressions is that they are a generic solution to the effect problem.

Create a new file *ResultDemo.fs* above *Program.fs*.

Rather than create our own computation expression for *Result*, we will use an existing one from the *FsToolkit.ErrorHandling* NuGet package.

Use the Terminal to add a NuGet package that contains the result computation expression that we want to use:

```
dotnet add package FsToolkit.ErrorHandling
```

Add the following code to *ResultDemo.fs*:

```
namespace ComputationExpression

module ResultDemo =

    open FsToolkit.ErrorHandling

    type Customer = {
        Id : int
        IsVip : bool
        Credit : decimal
    }

    // Customer -> Result<(Customer * decimal),exn>
    let getPurchases customer =
        try
            // Imagine this function is fetching data from a Database
            let purchases =
                if customer.Id % 2 = 0 then (customer, 120M) else (customer, 80M)
            Ok purchases
        with
        | ex -> Error ex

    // Customer * decimal -> Customer
    let tryPromoteToVip purchases =
        let customer, amount = purchases
        if amount > 100M then { customer with IsVip = true }
        else customer

    // Customer -> Result<Customer,exn>
    let increaseCreditIfVip customer =
        try
            // Imagine this function could cause an exception
            let increase = if customer.IsVip then 100M else 50M
            Ok { customer with Credit = customer.Credit + increase }
        with
        | ex -> Error ex

    // Customer -> Result<Customer,exn>
    let upgradeCustomer customer =
```

```
        customer
        |> getPurchases
        |> Result.map tryPromoteToVip
        |> Result.bind increaseCreditIfVip
```

Now let's see what turning the `upgradeCustomer` function into a computation expression looks like by replacing it with the following:

```
let upgradeCustomer customer =
    result {
        let! purchases = getPurchases customer
        let promoted = tryPromoteToVip purchases
        return! increaseCreditIfVip promoted
    }
```

Notice that the bang `!` is only applied to those functions that apply the effect, in this case, Result.

I find this style easier to read but some people do prefer the previous version.

Introduction to Async

Using the Explorer view, create a folder called *resources* in the code folder for this chapter and then create a new file called *customers.csv*. Copy the following data into the new file:

```
CustomerId|Email|Eligible|Registered|DateRegistered|Discount
John|john@test.com|1|1|2015-01-23|0.1
Mary|mary@test.com|1|1|2018-12-12|0.1
Richard|richard@nottest.com|0|1|2016-03-23|0.0
Sarah||0|0||
```

Create a new file *AsyncDemo.fs* above *Program.fs* and add the following code:

```
namespace ComputationExpression

module AsyncDemo =

    open System.IO

    type FileResult = {
        Name: string
        Length: int
    }
```

```
    let getFileInformation path =
        async {
            let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
            let fileName = Path.GetFileName(path)
            return { Name = fileName; Length = bytes.Length }
        }
```

The `Async.AwaitTask` function call is required because `File.ReadAllBytesAsync` returns a *Task* because it is a .NET function rather than an F# one. It is very similar in style to Async/Await in C# but Async in F# is lazily evaluated and Task is not.

Let's test our new code. Add the following import declaration to Program.fs:

```
open ComputationExpression.AsyncDemo
```

Replace the code in the main function with the following code:

```
Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
|> getFileInformation
|> Async.RunSynchronously
|> printfn "%A"
0
```

We have to use `Async.RunSynchronously` to force the *Async* code to run. You should only do this at the entry point of the application.

Run the code by typing the following in the terminal window:

```
dotnet run
```

Compound Computation Expressions

So far we have seen how to use a single effect but what happens if we have to use two (or more) like *Async* and *Result*? Thankfully, such a thing is possible and quite commonplace in this style of programming. We are going to use the `asyncResult` computation expression from the *FsToolkit.ErrorHandling* NuGet package we installed earlier.

The original code for this example comes from FsToolkit.ErrorHandling[27].

Create a new file called *AsyncResultDemo.fs* above *Program.fs* and add the following code:

[27] https://demystifyfp.gitbook.io/fstoolkit-errorhandling/asyncresult/ce

```
namespace ComputationExpression

module AsyncResultDemo =

    open System
    open FsToolkit.ErrorHandling

    type AuthError =
        | UserBannedOrSuspended

    type TokenError =
        | BadThingHappened of string

    type LoginError =
        | InvalidUser
        | InvalidPwd
        | Unauthorized of AuthError
        | TokenErr of TokenError

    type AuthToken = AuthToken of Guid

    type UserStatus =
        | Active
        | Suspended
        | Banned

    type User = {
        Name : string
        Password : string
        Status : UserStatus
    }
```

Add some constants below the type definitions, marked with the [<Literal>] attribute:

```
[<Literal>]
let ValidPassword = "password"
[<Literal>]
let ValidUser = "isvalid"
[<Literal>]
let SuspendedUser = "issuspended"
[<Literal>]
let BannedUser = "isbanned"
[<Literal>]
let BadLuckUser = "hasbadluck"
[<Literal>]
let AuthErrorMessage = "Earth's core stopped spinning"
```

They could also be written like this:

```
let [<Literal>] ValidPassword = "password"
let [<Literal>] ValidUser = "isvalid"
let [<Literal>] SuspendedUser = "issuspended"
let [<Literal>] BannedUser = "isbanned"
let [<Literal>] BadLuckUser = "hasbadluck"
let [<Literal>] AuthErrorMessage = "Earth's core stopped spinning"
```

Now we add the core functions, some of which are asynchronous, some return a Result, and one returns both. Don't worry about the how the functions are implemented internally, concentrate only on the inputs and outputs to the functions:

```
// string -> Async<User option>
let tryGetUser username =
    async {
        let user = { Name = username; Password = ValidPassword; Status = Active }
        return
            match username with
            | ValidUser -> Some user
            | SuspendedUser -> Some { user with Status = Suspended }
            | BannedUser -> Some { user with Status = Banned }
            | BadLuckUser -> Some user
            | _ -> None
    }

// string -> User -> bool
let isPwdValid password user =
    password = user.Password
```

```
// User -> Async<Result<unit, AuthError>>
let authorize user =
    async {
        return
            match user.Status with
            | Active -> Ok ()
            | _ -> UserBannedOrSuspended |> Error
    }

// User -> Result<AuthToken, TokenError>
let createAuthToken user =
    try
                if user.Name = BadLuckUser then failwith AuthErrorMessage
                else Guid.NewGuid() |> AuthToken |> Ok
    with
    | ex -> ex.Message |> BadThingHappened |> Error
```

The final part is to add the main `login` function that uses the previous functions and the asyncResult computation expression. The function does four things - tries to get the user from the datastore, checks the password is valid, checks the authorization and creates a token to return:

```
let login username password : Async<Result<AuthToken, LoginError>> =
    asyncResult {
        let! user = username |> tryGetUser |> AsyncResult.requireSome InvalidUser
        do! user |> isPwdValid password |> Result.requireTrue InvalidPwd
        do! user |> authorize |> AsyncResult.mapError Unauthorized
        return! user |> createAuthToken |> Result.mapError TokenErr
    }
```

The return type from the function is *Async<Result<AuthToken, LoginError>>*, so `asyncResult` is *Async* wrapping *Result*. This is very common in Line of Business (LOB) applications written in F#.

Notice the use of functions from the referenced library which make the code nice and succinct. The `do!` supports functions that return *unit* and is telling the computation expression to ignore the returned value unless it is an error. The `Result.mapError` functions are used to convert the specific errors from the `authorize` and `createAuthToken` functions to the *LoginError* type used by the `login` function.

Only those functions that are asynchronous need to use helper functions from the *AsyncResult* module for handling error cases, the others can use those from the *Result* module.

To test the code, copy the following under the existing code in *AsyncResultDemo.fs* or create a new file called *AsyncResultDemoTests.fs* between *AsyncResultDemo.fs* and *Program.fs*.

```
module AsyncResultDemoTests =

    open AsyncResultDemo

    let [<Literal>] BadPassword = "notpassword"
    let [<Literal>] NotValidUser = "notvalid"

    let isOk (input:Result<_,_>) : bool =
        match input with
        | Ok _ -> true
        | _ -> false

    let matchError (error:LoginError) (input:Result<_,LoginError>) =
        match input with
        | Error ex -> ex = error
        | _ -> false

    let runWithValidPassword (username:string) =
        login username ValidPassword |> Async.RunSynchronously

    let success =
        let result = runWithValidPassword ValidUser
        result |> isOk

    let badPassword =
        let result = login ValidUser BadPassword |> Async.RunSynchronously
        result |> matchError InvalidPwd

    let invalidUser =
        runWithValidPassword NotValidUser
        |> matchError InvalidUser

    let isSuspended =
        runWithValidPassword SuspendedUser
        |> matchError (UserBannedOrSuspended |> Unauthorized)

    let isBanned =
        let result = runWithValidPassword BannedUser
        result |> matchError (UserBannedOrSuspended |> Unauthorized)

    let hasBadLuck =
        let result = runWithValidPassword BadLuckUser
        result |> matchError (AuthErrorMessage |> BadThingHappened |> TokenErr)
```

Replace the code in the *Program.fs* `main` function with the following:

```
printfn "Success: %b" success
printfn "BadPassword: %b" badPassword
printfn "InvalidUser: %b" invalidUser
printfn "IsSuspended: %b" isSuspended
printfn "IsBanned: %b" isBanned
printfn "HasBadLuck: %b" hasBadLuck
0
```

Run the program from the Terminal using `dotnet run`. You should see successful asserts.

Debugging Code

The observant amongst you may have noticed that the ionide extension supports debugging. It's something that F# developers don't do very often as pure functions and F# Interactive (FSI) are more convenient. You can add a breakpoint as you would in other IDEs by clicking in the border to the left of the code.

If you put a breakpoint in the `createAuthToken` function on the if expression and debug the code by pressing the green button, you will see that you only hit the breakpoint twice, on the `success` and `hasbadluck` cases. Once the running code is on the *Error* track, it won't run any of the remaining code on the *Ok* track.

Further Reading

Computation Expressions are very useful tools to have at your disposal and are well worth investigating further. We have only looked at one usage which is dealing with effects but they can also be used for creating Domain-Specific Languages (DSL). Examples of this that you can investigate are **Saturn**[28] and **Farmer**[29].

It is important that you learn more about how F# handles asynchronous code with *Async* and how it interacts with the *Task* type from .Net Core. You can find out more at the following link:

https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/asynchronous-and-concurrent-programming/async

[28] https://saturnframework.org/
[29] https://compositionalit.github.io/farmer/

Summary

In this chapter, we have used computation expressions in F#. They can be a little confusing to understand but provide a generic approach to dealing with effects, simplify our code, and tend to deliver extremely readable code.

In the next chapter, we are going to start to put some of the skills we have developed throughout this book into practical use by looking at how to create APIs and websites with the Giraffe library.

13 - Introduction to Web Programming with Giraffe

In this chapter, we will start investigating web programming in F#.

We are going to use Giraffe[30]. Giraffe is "an F# micro web framework for building rich web applications" and was created by Dustin Moris Gorski[31]. It is a thin functional wrapper around ASP.NET Core. Both APIs and web pages are supported by Giraffe and we will cover both in the last three chapters of this book.

Giraffe comes with excellent documentation[32]. I highly recommend that you spend some time looking through it to see what features Giraffe offers.

If you want something more opinionated or want to use F# everywhere including to create JavaScript, have a look at Saturn[33] and the Safe Stack[34].

Getting Started

Create a new folder called *GiraffeExample* and open it in VS Code.

Using the Terminal in VS Code, type in the following command to create an empty ASP.NET Core web app project:

```
dotnet new web -lang F#
```

Open *Program.fs*. If you've done any ASP.NET Core development before, it will look very familiar, even though it is in F#:

[30] https://github.com/giraffe-fsharp/Giraffe
[31] https://twitter.com/dustinmoris
[32] https://github.com/giraffe-fsharp/Giraffe/blob/master/DOCUMENTATION.md
[33] https://saturnframework.org/
[34] https://safe-stack.github.io/

```
open System
open Microsoft.AspNetCore.Builder
open Microsoft.Extensions.Hosting

[<EntryPoint>]
let main args =
    let builder = WebApplication.CreateBuilder(args)
    let app = builder.Build()

    app.MapGet("/", Func<string>(fun () -> "Hello World!")) |> ignore

    app.Run()

    0 // Exit code
```

Run the code using the Terminal:

```
dotnet run
```

You will see something like this:

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7274
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5264
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\Code\GiraffeExample\
```

Use `CTRL+CLICK` on one of the URLs. Your browser should display:

```
Hello World!
```

To shut the running website down, press `CTRL+C` in the Terminal window where the code is running. You should be back to the command prompt.

Using Giraffe

Firstly, add the following NuGet packages to your project from the Terminal:

```
dotnet add package Giraffe
dotnet add package Giraffe.ViewEngine
```

Replace the code in *Program.fs* with the following:

```
open System
open Microsoft.AspNetCore.Builder
open Microsoft.Extensions.Hosting
open Microsoft.Extensions.DependencyInjection
open Microsoft.AspNetCore.Http
open Giraffe
open Giraffe.EndpointRouting
open Giraffe.ViewEngine

let endpoints =
    [
        GET [
            route "/" (text "Hello World from Giraffe")
        ]
    ]

let notFoundHandler =
    "Not Found"
    |> text
    |> RequestErrors.notFound

let configureApp (appBuilder : IApplicationBuilder) =
    appBuilder
        .UseRouting()
        .UseGiraffe(endpoints)
        .UseGiraffe(notFoundHandler)

let configureServices (services : IServiceCollection) =
    services
        .AddRouting()
        .AddGiraffe()
    |> ignore

[<EntryPoint>]
let main args =
    let builder = WebApplication.CreateBuilder(args)
    configureServices builder.Services
```

```
let app = builder.Build()

if app.Environment.IsDevelopment() then
    app.UseDeveloperExceptionPage() |> ignore

configureApp app
app.Run()
0
```

There is a lot more code but anyone who has configured ASP.NET Core applications will know that there is a lot that you will probably need to configure that isn't in the minimal template. The version for Giraffe looks similar to the pre-minimal template code with a few exceptions like the new endpoint routing and the `notFoundHandler`.

Endpoints

Endpoint routing was actually introduced in Giraffe 5.x and is designed to integrate with the ASP.NET Core endpoint routing APIs. You can read more about Giraffe endpoint routing at https://github.com/giraffe-fsharp/Giraffe/blob/master/DOCUMENTATION.md#endpoint-routing.

The endpoint routing allows us to make use of existing HttpHandlers and new custom ones that we are going to write.

404 - Not Found Handler

We have added a handler for when a route cannot be found, which is represented by an HTTP response status code of 404. If you run the web app and add a route like `/api` to your request, you will get a response of 404 and a payload of `Not Found`.

We should run the new code to verify that it works. In the Terminal, type the following to run the project:

```
dotnet run
```

Use `CTRL+CLICK` as before and you will see the following in your browser:

```
Hello World from Giraffe
```

The message is generated by the `text` HttpHandler that Giraffe supplies. In addition, it will return a 200 status code.

Shut the running website down by pressing `CTRL+C` in the Terminal window where the code is running. You should be back to the command prompt.

Creating an API Route

We are going to add a new route '/api' to our list of endpoints and respond with some JSON using another built-in Giraffe HttpHandler, *json*:

```
let endpoints =
    [
        GET [
            route "/" (text "Hello World from Giraffe")
            route "/api" (json {| Response = "Hello world!!" |})
        ]
    ]
```

The payload for the JSON response on the `/api` route is an anonymous record, designated by the `{|..|}` structure. You define the structure and data of the anonymous record in place rather than defining a type and creating an instance.

Now run the code and try the following Url in the browser. You should see a JSON response:

```
{"response":"Hello world!!"}
```

Creating a Custom HttpHandler

There are lots of built-in HttpHandlers but it is simple to create your own because they are functions with the signature `HttpFunc -> HttpContext -> HttpFuncResult`.

Let's add a new route in our endpoints for the API that takes a string value in the request querystring that will be handled by a custom handler that we haven't written yet:

```
let endpoints =
    [
        GET [
            route "/" (text "Hello World from Giraffe")
            route "/api" (json {| Response = "Hello world!!" |})
            routef "/api/%s" sayHelloNameHandler
        ]
    ]
```

Note that the querystring item is automatically bound to the handler function if it has a matching input parameter of the same type in the handler, so the `sayHelloNameHandler` needs to take a string input parameter. Create the custom handler above the `endpoints` binding:

```
let sayHelloNameHandler (name:string) : HttpHandler =
    fun (next:HttpFunc) (ctx:HttpContext) ->
        task {
            let msg = $"Hello {name}, how are you?"
            return! json {| Response = msg |} next ctx
        }
```

This function could be rewritten as the following but it is convention to use the style above:

```
let sayHelloNameHandler (name:string) (next:HttpFunc) (ctx:HttpContext) : HttpHandle\
r =
    task {
        let msg = $"Hello {name}, how are you?"
        return! json {| Response = msg |} next ctx
    }
```

The `task { ... }` is a computation Expression of type `System.Threading.Tasks.Task<'T>` and is the equivalent to *async/await* in C#. F# has another asynchronous feature called *async* but we won't use that as Giraffe has chosen to work directly with *Task* instead.

You will see a lot more of the *task* computation expression in the next chapter as we expand the API side of our website.

Now run the code and then try the `api/{yourname}` route in the browser. If you replace the placeholder `{yourname}` with your name, you should see a JSON response asking how you are.

Giraffe has some built-in extensions for *HttpContext* which can simplify some handlers:

```
let sayHelloNameHandler (name:string) : HttpHandler =
    fun (next:HttpFunc) (ctx:HttpContext) ->
        {| Response = $"Hello {name}, how are you?" |}
        |> ctx.WriteJsonAsync
```

The `WriteJsonAsync` function serializes the anonymous record to JSON and writes the output to the body of the HTTP response, sets the HTTP *Content-Type* header to `application/json`, and correctly sets the *Content-Length* header. The JSON serializer can be configured in the ASP.NET Core startup code by registering a custom class of type `Json.ISerializer`

Not only is Giraffe excellent as an API, but it is also equally suited to server-side rendered web pages using the Giraffe View Engine.

Creating a View

The Giraffe View Engine is a DSL that generates HTML. This is what a simple page looks like:

```
let indexView =
    html [] [
        head [] [
            title [] [ str "Giraffe Example" ]
        ]
        body [] [
            h1 [] [ str "I |> F#" ]
            p [ _class "some-css-class"; _id "someId" ] [
                str "Hello World from the Giraffe View Engine"
            ]
        ]
    ]
```

Each element of html has the following structure:

```
// element [attributes] [sub-elements/data]
html [] []
```

Each element has two supporting lists: The first is for attributes and the second for sub-elements or data. You may wonder why we need a DSL to generate HTML but it makes sense as it helps prevent badly formed structures. All of the features you need like master pages, partial views and model binding are included. We will have a deeper look at views in the final chapter of this book.

Add the `indexView` code above the `webApp` handler and replace the root (`"/"`) route with the following using the Giraffe `htmlView` handler function:

```
route "/" (htmlView indexView)
```

Run the web app and you will see the text from the indexView in your browser.

Adding Subroutes

One last thing to do and that is to simplify the routing as we have quite a lot of duplicate code:

```
let endpoints =
    [
        GET [
            route "/" (htmlView indexView)
            route "/api" (json {| Response = "Hello world!!" |})
            routef "/api/%s" sayHelloNameHandler
        ]
    ]
```

We can extract routes to another HttpHandler like we do with the '/api' route here:

```
// Endpoint list
let apiRoutes =
    [
        route "" (json {| Response = "Hello world!!" |})
        routef "/%s" sayHelloNameHandler
    ]

// Endpoint list
let endpoints =
    [
        GET [
            route "/" (htmlView indexView)
            subRoute "/api" apiRoutes
        ]
    ]
```

Note the change from 'route' to 'subRoute' if we extract the '/api' route to its own handler.

Run the app and try the routes out in your browser or by using a simple HTTP client like Postman or Insomnia.

Likely that the `apiRoutes` endpoint handler will need to deal with other HTTP verbs like POST, etc. We can modify the code to reflect this possibility:

```
let apiRoutes =
    [
        GET [
            route "" (json {| Response = "Hello world!!" |})
            routef "/%s" sayHelloNameHandler
        ]
    ]

let endpoints =
    [
        GET [
            route "/" (htmlView indexView)
        ]
        subRoute "/api" apiRoutes
    ]
```

Finally, run the app again and try the routes out in your browser or by using a simple HTTP client.

Reviewing the Code

The finished code for this chapter is:

```
open System
open Microsoft.AspNetCore.Builder
open Microsoft.Extensions.Hosting
open Microsoft.Extensions.DependencyInjection
open Microsoft.AspNetCore.Http
open Giraffe
open Giraffe.EndpointRouting
open Giraffe.ViewEngine

let indexView =
    html [] [
        head [] [
            title [] [ str "Giraffe Example" ]
        ]
        body [] [
            h1 [] [ str "HTML Generated by F#" ]
            p [ _class "some-css-class"; _id "someId" ] [
                str "Hello World from the Giraffe View Engine"
            ]
        ]
```

```
    ]

let sayHelloNameHandler (name:string) : HttpHandler =
    fun (next:HttpFunc) (ctx:HttpContext) ->
        {| Response = $"Hello {name}, how are you?" |}
        |> ctx.WriteJsonAsync

let apiRoutes =
    [
        GET [
            route "" (json {| Response = "Hello world!!" |})
            routef "/%s" sayHelloNameHandler
        ]
    ]

let endpoints =
    [
        GET [
            route "/" (htmlView indexView)
        ]
        subRoute "/api" apiRoutes
    ]

let notFoundHandler =
    "Not Found"
    |> text
    |> RequestErrors.notFound

let configureApp (appBuilder : IApplicationBuilder) =
    appBuilder
        .UseRouting()
        .UseGiraffe(endpoints)
        .UseGiraffe(notFoundHandler)

let configureServices (services : IServiceCollection) =
    services
        .AddRouting()
        .AddGiraffe()
    |> ignore

[<EntryPoint>]
let main args =
    let builder = WebApplication.CreateBuilder(args)
```

```
configureServices builder.Services

let app = builder.Build()

if app.Environment.IsDevelopment() then
    app.UseDeveloperExceptionPage() |> ignore

configureApp app
app.Run()
0
```

Summary

We have only scratched the surface of what is possible with Giraffe. In this chapter we had an introduction to using Giraffe to build an API and how to use the Giraffe View Engine to create HTML pages. **We also learnt about the importance of HttpHandlers in Giraffe routing and we're introduced to a number of the built-in ones Giraffe offers and we have seen that we can create our own handlers**.

In the next chapter, we will expand the API side of the application.

14 - Creating an API with Giraffe

In this chapter, we'll be creating a simple API with Giraffe. We are going to add new functionality to the project we created in the last chapter.

Getting Started

You will need a tool to run HTTP calls (GET, POST, PUT, and DELETE). I use Postman[35] but any tool including those available in VS Code will work.

Open the code from the last chapter in VS Code.

Our Task

We are going to create a simple API that we can view, create, update and delete Todo items.

Sample Data

Rather than work against a real data store, we are going to create a simple store with a dictionary and use that in our handlers.

Create a new file above *Program.fs* called *TodoStore.fs* and add the following code to it:

```
module GiraffeExample.TodoStore

open System
open System.Collections.Concurrent

type TodoId = Guid

type NewTodo = {
    Description: string
}

type Todo = {
    Id: TodoId
```

[35] https://www.postman.com/

```
    Description: string
    Created: DateTime
    IsCompleted: bool
}

type TodoStore() =
    let data = ConcurrentDictionary<TodoId, Todo>()

    let get id =
        let (success, value) = data.TryGetValue(id)
        if success then Some value else None

    member _.Create(todo) = data.TryAdd(todo.Id, todo)
    member _.Update(todo) = data.TryUpdate(todo.Id, todo, data[todo.Id])
    member _.Delete(id) = data.TryRemove id
    member _.Get(id) = get id
    member _.GetAll() = data.Values |> Seq.toArray
```

TodoStore is a simple class type that wraps a concurrent dictionary that we can use to test our API out without needing to connect to a database. This means that it will not persist between runs. You could pass the initial state through the TodoStore constructor if you need data.

Add a reference to the module import declarations in *Program.fs*:

```
open GiraffeExample.TodoStore
```

To be able to use the *TodoStore*, we need to make a change to the `configureServices` function in *Program.fs*:

```
let configureServices (services : IServiceCollection) =
    services
        .AddRouting()
        .AddGiraffe()
        .AddSingleton<TodoStore>(TodoStore())
    |> ignore
```

If you're thinking that this looks like dependency injection, you would be correct; we are using the one provided by ASP.NET Core. We add the TodoStore as a singleton as we only need one instance to exist.

Routes

We saw in the last chapter that Giraffe uses individual route handlers, so we need to think about how to add our new routes. The routes we need to add are:

```
GET     /api/todo       // Get a list of todos
GET     /api/todo/id    // Get one todo
POST    /api/todo       // Create a todo
PUT     /api/todo/id    // Update a todo
DELETE  /api/todo/id    // Delete a todo
```

Let's create a new handler with the correct HTTP verbs just above the `endpoints` value:

```
let apiTodoRoutes =
    [
        GET [
            routef "/%O" viewTodoHandler
            route "" viewTodosHandler
        ]
        POST [
            route "" createTodoHandler
        ]
        PUT [
            routef "/%O" updateTodoHandler
        ]
        DELETE [
            routef "/%O" deleteTodoHandler
        ]
    ]
```

We will create the missing handlers after we have plugged the new handler into our `endpoints` value as a subroute:

```
let endpoints =
    [
        GET [
            route "/" (htmlView indexView)
        ]
        subRoute "/api/todo" apiTodoRoutes
        subRoute "/api" apiRoutes
    ]
```

There are lots of ways of arranging the routes. It's up to you to find an efficient approach. I like this style with all of the route strings in one place. This makes them easy to change.

Next, we have to implement the new handlers.

Handlers

Rather than create the new handlers in *Program.fs*, we are going to create them in a new file. We will also move the `apiTodoRoutes` handler there as well.

Create a new file between *Program.fs* and *TodoStore.fs* called *Todos.fs*. Copy the following code into the new file:

```
module GiraffeExample.Todos

open System
open System.Collections.Generic
open Microsoft.AspNetCore.Http
open Giraffe
open Giraffe.EndpointRouting
open GiraffeExample.TodoStore
```

Move (Cut & Paste) the `apiTodoRoutes` handler function to the new *Todos.fs* file.

You will need to fix the error in the `endpoints` binding by adding an open declaration to *Project.fs*:

```
open GiraffeExample
```

Add the subroute:

```
subRoute "api/todo" Todos.apiTodoRoutes
```

Now we can concentrate on adding the handlers for our new routes to a new Handlers module above `apiTodoRoutes` in *Todos.fs*. We'll start with the two GET requests:

```
module Handlers =

    let viewTodosHandler =
        fun (next : HttpFunc) (ctx : HttpContext) ->
            let store = ctx.GetService<TodoStore>()
            store.GetAll()
            |> ctx.WriteJsonAsync

    let viewTodoHandler (id:Guid) =
        fun (next : HttpFunc) (ctx : HttpContext) ->
            task {
                let store = ctx.GetService<TodoStore>()
```

```
            return!
                (match store.Get(id) with
                | Some todo -> json todo
                | None -> RequestErrors.NOT_FOUND "Not Found") next ctx
        }
```

We are using the *HttpContext* instance (`ctx`) to gain access to the *TodoStore* instance we set up earlier using service location, a simple form of dependency injection.

Let's add the handlers for POST and PUT in the Handlers module:

```
let createTodoHandler =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        task {
            let! newTodo = ctx.BindJsonAsync<NewTodo>()
            let store = ctx.GetService<TodoStore>()
            let created =
                {
                    Id = Guid.NewGuid()
                    Description = newTodo.Description
                    Created = DateTime.UtcNow
                    IsCompleted = false
                }
                |> store.Create
            return! json created next ctx
        }

let updateTodoHandler (id:Guid) =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        task {
            let! todo = ctx.BindJsonAsync<Todo>()
            let store = ctx.GetService<TodoStore>()
            return!
                (match store.Update(todo) with
                | true -> json true
                | false -> RequestErrors.GONE "Gone") next ctx
        }
```

The most interesting thing here is that we use a built-in Giraffe function to gain strongly-typed access to the request body passed into the handler via the *HttpContext* (`ctx`).

Finally, we handle the Delete route:

```
let deleteTodoHandler (id:Guid) =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        task {
            let store = ctx.GetService<TodoStore>()
            return!
                (match store.Get(id) with
                | Some existing ->
                    let deleted = store.Delete(KeyValuePair<TodoId, Todo>(id, existi\
ng))
                    json deleted
                | None -> RequestErrors.GONE "Gone") next ctx
        }
```

Finally, we need to fix the errors in the `apiTodoRoutes` handler by adding the module name to the handler calls:

```
let apiTodoRoutes =
    [
        GET [
            routef "/%O" Handlers.viewTodoHandler
            route "" Handlers.viewTodosHandler
        ]
        POST [
            route "" Handlers.createTodoHandler
        ]
        PUT [
            routef "/%O" Handlers.updateTodoHandler
        ]
        DELETE [
            routef "/%O" Handlers.deleteTodoHandler
        ]
    ]
```

One last thing that we probably should do is to remove the `Handler` suffix from the handler names since they are in a module called *Handlers*. You could also argue that the *Todo* in the names is redundant as well. I'll leave these as tasks for the reader!

We should now be able to use the new *Todo* API.

Using the API

Run the app and use a tool like Postman to work with the API.

To get a list of all Todos, we call `GET /api/todo`. This should return an empty JSON array because we don't have any items in our store currently.

We create a *Todo* by calling `POST /api/todo` with a JSON request body like this:

```
{ "Description": "Finish blog post" }
```

You will receive a response of true or false. If you now call the list again, you will receive a JSON response containing the newly created item:

```
[
    {
        "key": "ff5a1d35-4573-463d-b9fa-6402202ab411",
        "value": {
            "id": "ff5a1d35-4573-463d-b9fa-6402202ab411",
            "description": "Finish blog post",
            "created": "2021-03-12T13:47:39.3564455Z",
            "isCompleted": false
        }
    }
]
```

I'll leave the other routes for you to investigate.

Summary

We have only scratched the surface of what is possible with Giraffe for creating APIs such as content negotiation and model validation. I highly recommend reading the Giraffe documentation[36] to get a fuller picture of what is possible.

In the final chapter of the book, we will dive deeper into HTML views with the Giraffe View Engine.

[36] https://github.com/giraffe-fsharp/Giraffe/blob/master/DOCUMENTATION.md

15 - Creating Web Pages with Giraffe

In the last chapter, we created a simple API for managing a Todo list. In this post, we are going to expand our journey into HTML views with the Giraffe View Engine.

If you haven't already done so, read the previous two chapters on Giraffe.

Getting Started

We are going to continue to make changes to the project we updated in the last chapter.

We are going to create a simple HTML view of a *Todo list* and populate it with dummy data from the server.

Rather than rely on my HTML/CSS skills, we are going to start with a pre-built sample: The *ToDo list* example from w3schools[37].

Configuration

Add a new folder to the project called *wwwroot*, two sub-folders called *css* and *js*, and add a file to each folder: *main.js* and *main.css*. Copy the CSS and JavaScript from Appendix 2 into the relevant files. The *wwwroot* folder will be automatically wired to the project when you perform the next step.

We have to tell the app to serve static files if requested. We need to add the `UseStaticFiles()` extension method to the `configureApp` function in *Program.fs*:

```
let configureApp (appBuilder : IApplicationBuilder) =
    appBuilder
        .UseRouting()
        .UseStaticFiles()
        .UseGiraffe(endpoints)
        .UseGiraffe(notFoundHandler)
```

Have a look here[38] if you need to configure a different folder than *wwwroot*.

[37] https://www.w3schools.com/howto/howto_js_todolist.asp
[38] https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files?view=aspnetcore-6.0

Adding a Master Page

We saw in Chapter 13 how we can create HTML using the Giraffe View Engine DSL.

We are going to create a master page and pass the title and content into it.

Create a new file called *Shared.fs* above the other created files.

Let's start with a basic HTML page:

```
<html>
    <head>
        <title>My Title</title>
        <link rel="stylesheet" href="css/main.css" />
    </head>
    <body />
</html>
```

If we convert the HTML to Giraffe View Engine format, we would get:

```
// string -> XmlNode list -> XmlNode
let masterPage msg content =
    html [] [
        head [] [
            title [] [ str msg ]
            link [ _rel "stylesheet"; _href "css/main.css" ]
        ]
        body [] content
    ]
```

Most tags have two lists, one for styling and one for content. Some tags, like *input*, only take the style list.

Add the following code to *Shared.fs*:

```
module GiraffeExample.Shared

open Giraffe.ViewEngine

let masterPage msg content =
    html [] [
        head [] [
            title [] [ str msg ]
            link [ _rel "stylesheet"; _href "css/main.css" ]
        ]
        body [] content
    ]
```

To update the existing view with our new master page, we need to add an open declaration to *Program.fs*:

```
open GiraffeExample
```

We then need to change `indexView` to use the new master page:

```
let indexView =
    [
        h1 [] [ str "I |> F#" ]
        p [ _class "some-css-class"; _id "someId" ] [
            str "Hello World from the Giraffe View Engine"
        ]
    ]
    |> Shared.masterPage "Giraffe View Engine Example"
```

This approach is very different to most view engines which rely on search and replace but are primarily still HTML. The primary advantage of the Giraffe View Engine approach is that you get full type safety when creating and generating views and can use the full power of the F# language.

Run the website to prove that you haven't broken anything.

Creating the Todo List View

The HTML below from the w3schools source has to be converted into Giraffe View Engine format:

```
<div id="myDIV" class="header">
  <h2>My To Do List</h2>
  <input type="text" id="myInput" placeholder="Title...">
  <span onclick="newElement()" class="addBtn">Add</span>
</div>

<ul id="myUL">
  <li>Hit the gym</li>
  <li class="checked">Pay bills</li>
  <li>Meet George</li>
  <li>Buy eggs</li>
  <li>Read a book</li>
  <li>Organise office</li>
</ul>
```

Create a new module in *Todos.fs* called `Views` and add a new function to generate our `todoView` binding:

```
module Views =

    open Giraffe.ViewEngine

    let todoView =
        [
            div [ _id "myDIV"; _class "header" ] [
                h2 [] [ str "My To Do List" ]
                input [ _type "text"; _id "myInput"; _placeholder "Title..." ]
                span [ _class "addBtn"; _onclick "newElement()" ] [ str "Add" ]
            ]
            ul [ _id "myUL" ] [
                li [] [ str "Hit the gym" ]
                li [ _class "checked" ] [ str "Pay bills" ]
                li [] [ str "Meet George" ]
                li [] [ str "Buy eggs" ]
                li [] [ str "Read a book" ]
                li [ _class "checked" ] [ str "Organise office" ]
            ]
            script [ _src "js/main.js"; _type "text/javascript" ] []
        ]
        |> Shared.masterPage "My ToDo App"
```

We only need to tell the endpoints router about our new view. Change the root '/' route in webApp to:

```
let endpoints =
    [
        GET [
            route "/" (htmlView Todos.Views.todoView)
        ]
        subRoute "/api/todo" apiTodoRoutes
        subRoute "/api" apiRoutes
    ]
```

If you now run the app using dotnet run in the Terminal. Follow the HTTPS link in the Terminal and you will see the *Todo* app running in the browser.

Loading Data on Startup

Rather than hard code the list of *Todos* into the view, we can load it in on the fly by creating a list of items. Create the following above the *Views* module in *Todos.fs*:

```
module Data =

    let todoList = [
        { Id = Guid.NewGuid(); Description = "Hit the gym"; Created = DateTime.UtcNo\
w; IsCompleted = false }
        { Id = Guid.NewGuid(); Description = "Pay bills"; Created = DateTime.UtcNow;\
 IsCompleted = true }
        { Id = Guid.NewGuid(); Description = "Meet George"; Created = DateTime.UtcNo\
w; IsCompleted = false }
        { Id = Guid.NewGuid(); Description = "Buy eggs"; Created = DateTime.UtcNow; \
IsCompleted = false }
        { Id = Guid.NewGuid(); Description = "Read a book"; Created = DateTime.UtcNo\
w; IsCompleted = true }
        { Id = Guid.NewGuid(); Description = "Read Essential F#"; Created = DateTime\
.UtcNow; IsCompleted = false }
    ]
```

There's a lot of duplicate code, so let's create a helper function to simplify things:

```
module Data =

    let private create description isCompleted =
        {
            Id = Guid.NewGuid()
            Description = description
            Created = DateTime.UtcNow
            IsCompleted = isCompleted
        }

    let todoList =
        [
            ("Hit the gym", false)
            ("Pay bills", true)
            ("Meet George", false)
            ("Buy eggs", false)
            ("Read a book", true)
            ("Read Essential F#", false)
        ]
        |> List.map (fun (todo, isCompleted) -> create todo isCompleted)
```

We need to create a partial view, which is simple helper function, to style each item in the *Todo list*. You should write this function above the `todoView` code in the *Views* module:

```
// Todo -> XmlNode
let private showListItem (todo:Todo) =
    let style = if todo.IsCompleted then [ _class "checked" ] else []
    li style [ str todo.Description ]
```

Now we can use the `showListItem` partial view function in the `todoView` and we can pass the *Todo* items in as an input parameter:

```
let todoView items =
    [
        div [ _id "myDIV"; _class "header" ] [
            h2 [] [ str "My ToDo List" ]
            input [ _type "text"; _id "myInput"; _placeholder "Title..." ]
            span [ _class "addBtn"; _onclick "newElement()" ] [ str "Add" ]
        ]
        ul [ _id "myUL" ] [
            for todo in items do
                showListItem todo
```

```
        ]
        script [ _src "js/main.js"; _type "text/javascript" ] []
    ]
    |> masterPage "My ToDo App"
```

We are generating the list of *Todos* in a list comprehension.

We need to change the root route as we are now passing the list of *Todos* into the `todoView` function:

```
let endpoints =
    [
        GET [
            route "/" (htmlView (Todos.Views.todoView Todos.Data.todoList)
        ]
        subRoute "/api/todo" apiTodoRoutes
        subRoute "/api" apiRoutes
    ]
```

Run the app to see the *Todo* items displayed on the HTML page.

Summary

We have only scratched the surface of what is possible with Giraffe and the Giraffe View Engine for creating web pages and APIs. I highly recommend reading the Giraffe View Engine documentation[39] to find out about some of the other features available.

If you like the Giraffe View Engine but don't like writing JavaScript, you should investigate the SAFE Stack[40] where **everything** is written in F#. The SAFE Stack is also great for creating Single Page Apps with React frontends, all in F#.

That concludes our journey into F# and Giraffe. I hope that I have given you enough of a taste of what F# can offer to encourage you to want to take it further.

[39] https://github.com/giraffe-fsharp/Giraffe.ViewEngine
[40] https://safe-stack.github.io/

Summary

Congratulations on reaching the end of this book. I hope that it has been a fun journey and that I have conveyed some of the joy that I get from working every day on F# codebases. Even if you don't get the chance to work in F#, many of the ideas and practices in this book will still positively impact the way that you think about and write code every day.

Programming in F# is different from working in C#. It's not just about the language features or the paradigm but it's the way that you are encouraged to explore different solutions to problems with minimal cost.

My F# ethos is:

> **First make it run, then make it pretty, and finally, if you need to, make it faster.**
>
> (I think I first heard this from the late Joe Armstrong, one of the creators of Erlang).

We have covered a lot in just under 200 pages! During the course of reading this book, you have been introduced to the essentials of F#:

- F# Interactive (FSI)
- Algebraic Type System
  - Records
  - Discriminated Unions
  - Tuples
- Pattern Matching
  - Active Patterns
  - Guard Clauses
- Let bindings
- Immutability
- Functions
  - Pure
  - Higher-Order
  - Currying
  - Partial Application
  - Pipelining
  - Signatures
  - Composition
- Unit

- Effects
  - Option
  - Result
  - Async
  - Task
- Computation Expressions
- Collections
  - Lists
  - Sequences
  - Arrays
  - Sets
- Recursion
- Object Programming
  - Class types
  - Interfaces
  - Object Expressions
- Exceptions
- Unit Tests
- Namespaces and Modules

You've also started the journey towards writing APIs and websites in Giraffe.

F# Software Foundation

Thank you for purchasing this book. All of the author's royalties will be going to the F# Software Foundation to support their efforts in promoting the F# language and community to the world.

If you haven't already joined, you should join the F# Software Foundation[41]; It's free. By joining, you will be able to access the dedicated Slack channel where there are excellent tracks for beginners and beyond.

Resources

There are plenty of other high-quality resources available for F#. These are some that I heartily recommend:

[41] https://foundation.fsharp.org/join

Books

- Stylish F# 6[42]
- Domain Modelling Made Functional[43]

Websites

- F# Docs[44]
- F# Software Foundation[45]
- F# For Fun and Profit[46]
- F# Weekly[47]
- Compositional-It Blog[48]

Courses

- F# From the Ground Up[49]

And Finally

What makes F# such a wonderful language to use is not just about the individual features but how they work together; It's the epitome of Aristotle's oft-quoted words:

> The whole is greater than the sum of its parts.

[42] https://link.springer.com/book/10.1007/978-1-4842-7205-3
[43] https://www.pragprog.com/titles/swdddf/domain-modeling-made-functional/
[44] https://fsharp.github.io/fsharp-core-docs
[45] https://fsharp.org/
[46] https://fsharpforfunandprofit.com/
[47] https://sergeytihon.com/category/f-weekly/
[48] https://www.compositional-it.com/news-blog/
[49] https://www.udemy.com/course/fsharp-from-the-ground-up/

Appendix 1

Creating Solutions and Projects in VS Code

We are going to create a new .NET Solution containing an F# console project and an XUnit test project using the dotnet CLI.

Open VS Code in a new folder.

Open a new Terminal window. The shortcut for this is CTRL+SHIFT+'.

Create a new file in the folder. It doesn't matter what it is called, so use 'setup.txt'.

Copy the following script into the file:

```
dotnet new sln -o MySolution
cd MySolution
mkdir src
dotnet new console -lang F# -o src/MyProject
dotnet sln add src/MyProject/MyProject.fsproj
mkdir tests
dotnet new xunit -lang F# -o tests/MyProjectTests
dotnet sln add tests/MyProjectTests/MyProjectTests.fsproj
cd tests/MyProjectTests
dotnet add reference ../../src/MyProject/MyProject.fsproj
dotnet add package FsUnit
dotnet add package FsUnit.XUnit
dotnet build
dotnet test
```

This script will create a new solution called MySolution, two folders called src and tests, a console app called MyProject in the src folder, and a test project called MyProjectTests in the tests folder. Change the names to suit. In VS Code, CTRL+F2 will allow you to edit all of the instances of the selected word at the same time.

Select all of the script text.

To run the script in the Terminal, you can do either of the following:

- Choose the Terminal menu item and then select 'Run Selected Text'.
- Press CTRL+SHIFT+P to open the Command Palette and then type 'TRSTAT'. Select the '**T**erminal: **R**un **S**elected **T**ext in **A**ctive **T**erminal' item.

The script will now execute in the Terminal.

You can delete the file with the script in if you want as you no longer need it.

Appendix 2

CSS and JavaScript for Chapter 15

CSS for Chapter 15

```
/* Include the padding and border in an element's total width and height */
* {
  box-sizing: border-box;
}

/* Remove margins and padding from the list */
ul {
  margin: 0;
  padding: 0;
}

/* Style the list items */
ul li {
  cursor: pointer;
  position: relative;
  padding: 12px 8px 12px 40px;
  background: #eee;
  font-size: 18px;
  transition: 0.2s;

  /* make the list items unselectable */
  -webkit-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
  user-select: none;
}

/* Set all odd list items to a different color (zebra-stripes) */
ul li:nth-child(odd) {
  background: #f9f9f9;
}
```

```
/* Darker background-color on hover */
ul li:hover {
  background: #ddd;
}

/* When clicked on, add a background color and strike out text */
ul li.checked {
  background: #888;
  color: #fff;
  text-decoration: line-through;
}

/* Add a "checked" mark when clicked on */
ul li.checked::before {
  content: '';
  position: absolute;
  border-color: #fff;
  border-style: solid;
  border-width: 0 2px 2px 0;
  top: 10px;
  left: 16px;
  transform: rotate(45deg);
  height: 15px;
  width: 7px;
}

/* Style the close button */
.close {
  position: absolute;
  right: 0;
  top: 0;
  padding: 12px 16px 12px 16px;
}

.close:hover {
  background-color: #f44336;
  color: white;
}

/* Style the header */
.header {
  background-color: #f44336;
  padding: 30px 40px;
```

```
  color: white;
  text-align: center;
}

/* Clear floats after the header */
.header:after {
  content: "";
  display: table;
  clear: both;
}

/* Style the input */
input {
  margin: 0;
  border: none;
  border-radius: 0;
  width: 75%;
  padding: 10px;
  float: left;
  font-size: 16px;
}

/* Style the "Add" button */
.addBtn {
  padding: 10px;
  width: 25%;
  background: #d9d9d9;
  color: #555;
  float: left;
  text-align: center;
  font-size: 16px;
  cursor: pointer;
  transition: 0.3s;
  border-radius: 0;
}

.addBtn:hover {
  background-color: #bbb;
}
```

JavaScript for Chapter 15

```
// Create a "close" button and append it to each list item
var myNodelist = document.getElementsByTagName("LI");
var i;
for (i = 0; i < myNodelist.length; i++) {
  var span = document.createElement("SPAN");
  var txt = document.createTextNode("\u00D7");
  span.className = "close";
  span.appendChild(txt);
  myNodelist[i].appendChild(span);
}

// Click on a close button to hide the current list item
var close = document.getElementsByClassName("close");
var i;
for (i = 0; i < close.length; i++) {
  close[i].onclick = function() {
    var div = this.parentElement;
    div.style.display = "none";
  }
}

// Add a "checked" symbol when clicking on a list item
var list = document.querySelector('ul');
list.addEventListener('click', function(ev) {
  if (ev.target.tagName === 'LI') {
    ev.target.classList.toggle('checked');
  }
}, false);

// Create a new list item when clicking on the "Add" button
function newElement() {
  var li = document.createElement("li");
  var inputValue = document.getElementById("myInput").value;
  var t = document.createTextNode(inputValue);
  li.appendChild(t);
  if (inputValue === '') {
    alert("You must write something!");
  } else {
    document.getElementById("myUL").appendChild(li);
  }
  document.getElementById("myInput").value = "";

  var span = document.createElement("SPAN");
```

```
  var txt = document.createTextNode("\u00D7");
  span.className = "close";
  span.appendChild(txt);
  li.appendChild(span);

  for (i = 0; i < close.length; i++) {
    close[i].onclick = function() {
      var div = this.parentElement;
      div.style.display = "none";
    }
  }
}
```