



**Computer Science
from the
Metal Up**

**Functional Programming
in Haskell and VB**

Richard Pawson

**Foreword by
Simon Peyton Jones**

Computer Science from the Metal Up:

Functional Programming

By Richard Pawson

v1.2.0

©Richard Pawson, 2019. The moral right of the author has been asserted.



This book is distributed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License: <https://creativecommons.org/licenses/by-nc-nd/4.0/>.

The author is willing, in principle, to grant permission for distribution of derivative versions, on a case by case basis, and may be contacted as rpawson@metalup.org in relation to permissions, or to report errors found in the book.

'Metal Up' is a registered trademark, number UK00003361893.

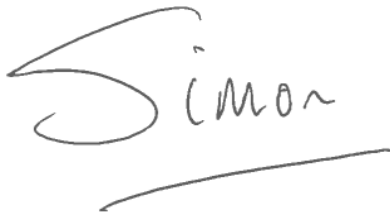
Foreword

When I first encountered functional programming, at university, I thought it could not possibly work. How can you write useful programs without variables, without assignment, without state? All the programs I had written up to that point utterly relied on all these things. Even a loop to add up the numbers between 1 and 10 relies on mutation for the running total.

And then the light dawned. Functional programming is not just another programming language; it is a radical and elegant attack on the entire enterprise of programming. It makes you think in a new way about programming, focusing on *immutable values*, and the functions that compute them, rather than on *state*, and the procedures that mutate it. I was completely captivated, and have since spent my entire professional life working out the consequences of this one idea. That led me to be part of the informal group that designed Haskell, which has since played a central part in my research. (My children insisted on calling our cat Haskell, saying that Haskell was really my first child.)

So I am delighted to see a book that introduces young people to the joys of functional programming. Computer science as a school subject is immature: we are still figuring out what to teach, in what order, and how to teach it. I believe that functional programming will have a part to play in this drama, although I do not yet know quite what it is. But the only way to find out is to try, and that's exactly what this book does, based on the author's classroom experience of using this material in practice.

Enjoy the adventure. But beware: once you have grown to love Haskell, it can be hard to go back to imperative programming!

A handwritten signature in black ink that reads "Simon". The signature is written in a cursive style with a long horizontal line underneath the name.

Simon Peyton Jones



Simon Peyton Jones is a Principal Researcher at Microsoft Research, in Cambridge. He was one of the designers of Haskell, and co-leads the GHC open-source project, the main compiler for Haskell. He has been the Chair of Computing At School (CAS) since its inception, and now also chairs the new National Centre for Computing Education (NCCE).

Book 1: Fundamentals of Functional Programming	1
<i>Chapter 1: A New Programming Paradigm</i>	2
<i>Chapter 2: Defining Functional Programming</i>	7
<i>Chapter 3: Using expressions rather than statements</i>	15
<i>Chapter 4: Returning multiple values from a function</i>	21
<i>Chapter 5: Handling conditions</i>	27
<i>Chapter 6: Using functional lists</i>	30
<i>Chapter 7: Replacing loops with recursion</i>	35
<i>Chapter 8: Case study – Merge Sort</i>	39
<i>Chapter 9: Introducing higher order functions</i>	42
<i>Chapter 10: Map, Filter, Reduce</i>	48
<i>Chapter 11: A more formal approach</i>	54
Book 2 – Delving a little deeper	58
<i>Chapter 12: Input/Output in Functional Programming</i>	59
<i>Chapter 13: The Haskell type system</i>	65
<i>Chapter 14: Folding Left vs. Folding Right</i>	71
<i>Chapter 15: Using LINQ in .NET to emulate Map, Filter, Reduce</i>	75
<i>Chapter 16: Ranges</i>	78
<i>Chapter 17: Programming exercises</i>	81
Appendices	83
<i>Appendix I: Installing Software</i>	84
<i>Appendix II: Further Reading</i>	85
<i>Appendix III: Versioning (of this book)</i>	86
<i>Appendix IV: References</i>	87

Book 1: Fundamentals of Functional Programming

Chapter 1: A New Programming Paradigm

Can you imagine a form of programming that looks like this?

- Your program consists of functions, but each function must be implemented as a single expression, not a sequence of statements to be executed in order.
- You cannot specify the ‘flow of control’, so there are no ‘branches’ as such – though functions can calculate results differently, depending upon the input values provided
- There are ‘variables’, which hold a value, but you can never *change* that value.
- There are no loops, as such, so the only way to ‘iterate’ is to use recursion.
- There are pre-defined data structures, such as lists, and user-defined data structures (similar to object classes), but if you want to add, remove, or change a value within an existing data structure, you must create a new data structure each time.

This might sound like some sort of extreme challenge undertaken by people with nothing better to do, like riding a bicycle backwards, ironing a shirt while parachuting, or writing a whole novel without using the letter ‘e’ (it has been done¹). But many Computer Scientists believe that the approach partially described above, is the future of programming.

Welcome to ‘Functional Programming’.

A bit like HTML? Not really...

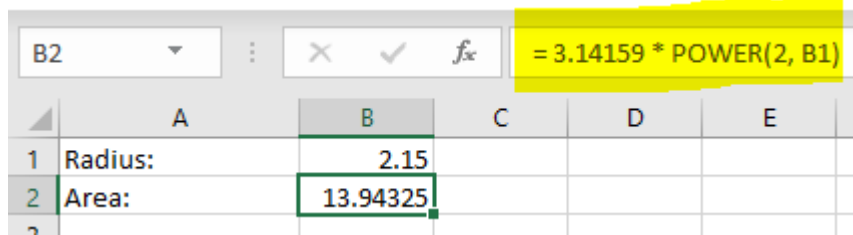
Perhaps you are thinking that this sounds a bit like writing HTML, which has no variables, loops, or branches, but which some people still describe as a form of programming. HTML is a form of ‘declarative’ programming, and so is Functional Programming. But there the resemblance ends – because every Computer Scientist knows that HTML is not a ‘Turing complete’ language - it can only be used for certain tasks, not to solve problems in general.

Functional Programming (‘FP’ from now on) *is* Turing complete, which means it can be used to solve any problem that can be solved by any other programming approach or language: in other words, any problem that is ‘computable’.

If you are seeking to relate FP to something more familiar, a better analogy is the spreadsheet (see panel).

Spreadsheets *are* a form of Functional Programming

Each cell in a spreadsheet may contain a literal value, such as 2.15 in the example below, or a formula, such as the highlighted example below:



The image shows a screenshot of an Excel spreadsheet. The formula bar at the top displays the formula `= 3.14159 * POWER(2, B1)` in a yellow highlight. The spreadsheet grid shows the following data:

	A	B	C	D	E
1	Radius:	2.15			
2	Area:	13.94325			

The expression may be made up of literal values (3.14159), arithmetic operators (*), references to other cells (B1), and/or calls to ready-made functions (POWER).

Spreadsheets are ‘declarative’: when writing a spreadsheet, you do not specify the *order* in which formulae are to be executed - in other words there is no sequencing. (We are excluding the use of the ‘macros’, which *would be* a form of sequencing). Moreover, spreadsheets, like all forms of FP, use ‘lazy evaluation’ – when you make a change to a cell, the spreadsheet application will evaluate only those cells that need to be re-evaluated.

Surprisingly, perhaps, spreadsheets *are* Turing complete: you can use a spreadsheet to write a Turing machine, and without resorting to using macros. Felienne Hermans, Associate Professor at the Leiden Institute of Advanced Computer Science, has implemented a Turing machine in Excel, and written a clear explanation, together with an example implementation, that you may download and run from here: <http://www.felienne.com/archives/2974>

However, even though spreadsheets can be proven to be Turing complete, few would consider a spreadsheet as a practical proposition for writing many of the kinds of programs that you could easily write in a mainstream programming language. In this book we shall be looking at what you might call ‘mainstream’ software development using a pure FP approach. However, we will return more than once to the analogy with the spreadsheet, and you will see that the assertion that spreadsheets are a form of FP, holds up very well.

A new programming paradigm

FP is definitely a ‘paradigm shift’ in programming. Computing technology and programming techniques evolve continuously, but a paradigm shift is a dramatic change, and hence relatively infrequent (see panel). When a change of this scale is proposed it often faces a hostile response from existing programmers, who have learned a previous paradigm, and who find it difficult to understand that there is any case for changing it, and/or that the proposed new paradigm could even work at any significant scale. Nonetheless, the world of programming has already undergone more than one paradigm shift.

What is meant by ‘paradigm shift’ ?

Historically, the word ‘paradigm’, just meant ‘pattern’. In the 1960s the word became more commonplace, and with a slightly different meaning, thanks to the book *The Structure of Scientific Revolutions* by Thomas Kuhn (pictured²). Kuhn argued that progress in science was largely incremental, interspersed with occasional radical changes – which he termed ‘paradigm shifts’ – as a result of which key aspects of a particular science come to be viewed in a very different way. Classic examples of this phenomenon include: the Copernican revolution that shifted from an earth-centric, to a helio-centric model of astronomy; the shift from Newtonian to quantum physics; or Kurt Gödel’s proof that in any formal system, such as mathematics, there would necessarily be some theorems that could not be proven either true or false.



Kuhn also argued that from a long-term historical perspective these changes might appear sudden, but paradigm shifts typically take decades, from when they are first proposed to when they become widely accepted by the scientific community. Specifically, he claimed that the average period was 30 years - a human generation. During that period there is often intense disagreement and argument within the community.

The transition from low-level languages – machine code and assembly language – to higher level languages such as Fortran, C, and Basic, is generally *not* considered a paradigm shift. The high-level languages made programmers more productive, but they didn’t change the fundamental way they thought about programming.

The first paradigm shift in programming, was the transition to ‘structured programming’ - as defined, for example, in Edsger Dijkstra’s famous 1968 paper *GOTO statements considered harmful*³. Today, it is likely that your first introduction to programming was using a structured programming language, so you might never have experienced the previous paradigm, where control was frequently transferred to another arbitrary line of code using a GOTO statement. Unless, that is, you have done any assembly language programming, in which case you have almost certainly used a Branch instruction (or similar), which is equivalent to a GOTO.

Object-oriented programming (OOP) was another paradigm shift: first proposed and implemented in the late 1960s, it did not become widely adopted until the late 1990s. Today, it is considered the ‘dominant paradigm’ of programming: the majority of high-level languages support OOP. Some programmers seek to develop their applications using a *pure* OOP approach, others adopt what may be described as ‘mixed paradigm’ programming, that combines OOP with traditional procedural (also known as ‘imperative’) programming.

Like OOP, FP was first explored many years ago. John McCarthy's revolutionary programming language, LISP, encapsulated some of the core ideas of what we now call FP in 1958 - and it became the favourite language for Artificial Intelligence research for several decades.

Today, FP has already become the dominant paradigm of programming within specific domains such as Computer Science research, scientific computing, and numerical analysis. Its adoption within commercial and consumer computing is still small. Many theorists and practitioners, however, believe that within a few years, FP will have become the dominant paradigm for programming – across the board.

FP and programming languages

Several programming languages have been designed from scratch specifically for FP. Examples include ML, OCaml, and Microsoft's F#. The best known and most widely adopted, today, is Haskell.

The functional programmer who never was

The programming language Haskell is named after Haskell Curry (1900 – 1982), but it was not invented by him. Curry was an American mathematician much of whose work was completed before the modern computing era. His ideas later prompted others to develop functional programming. The concept of 'currying' – a term used both in mathematics and in functional programming – is also named after him, and there is a separate programming language, derived from Haskell, called Curry. A third programming language – Brook – is a contraction of his middle name, Brooks. Not a bad achievement: having three programming languages named after you, despite not being a programmer yourself!⁴



In recent years several of the most popular programming languages that were designed originally for object-oriented, or even for procedural programming, have acquired features of FP. Such languages are sometimes now described as 'multi-paradigm languages'. In some of the mainstream languages – C#, VB, Python, Java, JavaScript, for example - the support for FP-like features is such that it is now *possible* to write programs in the pure FP paradigm.

Using these multi-paradigm programming languages, it is also possible to write some parts of your application in a pure-FP style, and others in conventional OOP or procedural manner. Some people argue that this is an advantage - allowing the benefits of FP to be realised in a wider range of contexts, where adopting pure FP throughout would not yet be practical. The counter argument, however, is that programming in a mixed paradigm runs the risk that you do not see the full benefit of adopting any single paradigm in a pure fashion.

This dilemma also extends into the best way to learn FP.

Learning FP via Haskell

One option is to learn FP through a pure FP language such as Haskell. This has several advantages:

- You will be *forced* to ‘do pure FP’, because the compiler enforces the rules.
- You get immediate access to language features designed specifically for FP, and which have yet to appear in many multi-paradigm languages.
- Haskell and other purpose-design FP languages are very succinct (or ‘terse’) - you will be surprised by how much functionality you can implement in a few lines – even just a few characters – of code.

However, this approach to learning FP also has some disadvantages:

- The unfamiliar syntax and semantics of a pure FP language may result in a slower start. Writing a simple example functions is straightforward, but writing a complete application is not.
- The tools for working with Haskell are less friendly for beginners, especially in regard to error messages, because, to date, they have mostly been used by experienced programmers.
- There is a risk that you will see FP primarily as a new language, rather than a new way of thinking about problems (a new paradigm, in other words).

Learning FP via a multi-paradigm language

An alternative approach is to learn FP via a suitable multi-paradigm language with which you are already familiar such as VB. The advantages of this approach are that:

- The principles and techniques of FP can be introduced progressively, even within existing non-FP programs.
- It is clearer that, while certain aspects of FP demand specific new capabilities within the programming language, others are merely a different choice of programming technique.

By being able to choose between two programming patterns at each point, the advantages (and, if any, disadvantages) of FP become more explicit.

The disadvantages are that:

- Adopting the FP techniques require more self-discipline, because the language will allow you to program in a non-FP approach also.
- Multi-paradigm languages do not typically achieve the same performance optimisations, and may therefore run more slowly or take more memory. (However, you might need to build quite a complex system before this difference would become noticeable).
- In learning FP as a series of small steps, you might fail to see how radical FP is, taken as a whole.

Learning FP two ways

This book takes both approaches. In the following chapters we will introduce the techniques of FP, first in VB, and then Haskell - switching to Haskell first as we get to more advanced ideas.

Before we get down to the coding, we need to establish a clearer definition of the distinction between FP and previous programming paradigms. And we also need to be clear on why these new principles constitute an advantage, as distinct from just defining an arbitrary new set of constraints.

Chapter 2: Defining Functional Programming

In FP, the fundamental building block of programming is the function. ‘But, surely,’ you say, ‘functions are used extensively in any well-structured example of procedural programming, and aren’t object ‘methods’ just functions, but encapsulated on the objects?’ Yes - to both points. In FP, however, the functions must be ‘proper’. In procedural programming, or OOP, they don’t have to be proper, and while some might be, many are not.

Proper functions

What, then, constitutes a ‘proper function’?

- A proper function must always be called with one or more arguments. By contrast, in VB you can define so-called ‘functions’ that require no arguments, and hence are not proper.
- A proper function always returns a result.

(In C# you may define ‘functions’ that are `void` - they do not return a value. VB, by contrast, distinguishes between a `Sub` (subroutine) which does not return a result, and a `Function` which may.)

Pure functions

As well as being ‘proper’, FP requires that functions be ‘pure’. This means that:

- The result returned, depends *solely* upon the values explicitly passed in as arguments. It cannot depend on any other value obtained from outside the function that might change, including: ‘global variables’, the time of day, values read from the keyboard, or from a file or database, or anything accessed via a network.
- The relationship between the result and the passed-in arguments is ‘deterministic’: the same input values must *always* produce the same result.
- The evaluation of the function must generate no ‘side-effects’ - we say that they must be ‘side-effect free’. We’ve already ruled out any use of ‘global variables’, or any variables defined outside the scope of the function, but additionally, a pure function may not modify any of the values passed in as parameters. Nor may it alter any other aspect of the system, such as writing to a file, or a database, sending a communication over a network, printing, or even writing to a screen
- Pure functions may call other functions in their implementation, but only if those functions are also be pure.

The third bullet point above might have been the biggest surprise to you: if pure functions cannot ask the user to enter a value, or even print to the screen, how can FP be a practical proposition for writing real applications? This is the central conundrum in FP, and we are going to defer it until Chapter 12. In the interim you’ll find you can explore the rudiments of FP, practically, without having to do explicit input/output.

Modifying parameter values in .NET

As well as other, more obvious, side effects, pure functions must not modify values that are passed into them as arguments. In VB, any simple values (such a Boolean, integer, or character) passed into a function via the declared parameters, are copies, so any changes made to those copies within the function would not be visible outside the function. But the language also allows you to pass a value ‘by reference’ – specifically to allow changes to be observed from outside, so this would not be allowed in FP. More subtly, in VB, if you pass in a ‘reference type’ (such as an `Array`, `List`, or an

instance of user-defined class such as `Student` or `Missile`), any change made to that object *will* be visible outside the function, even if you didn't explicitly pass it 'by reference'. In VB it is easy to write impure functions (ones that generate side effects) both deliberately, and by accident. For this reason and others, while it is possible to write pure FP in VB, it requires more self-discipline. One of several arguments in favour of purpose-designed FP languages such as Haskell, is that the compiler prevents the *accidental* creation of side-effects.

As an introduction, let's look at an existing VB program, which contains several named blocks of code that *might* be called 'functions'. But how many are proper functions, and, of those, how many are pure functions?

Exercise 1

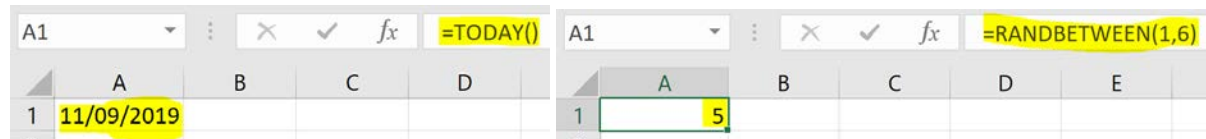
For this question you will be looking at the source code for a program called **Battleships** - an implementation of a familiar game originally played with paper and pencil. See Appendices – Installing Software for details. You may examine the code on paper, or copy and paste it into a new Console project in Visual Studio.

Your task is to complete the empty cells in the table below, by examining the code to determine which of the 'functions' listed are **proper functions** (they take arguments, return a result), and which of those that are proper are *also* **pure functions** (they do not depend on any value not passed in as a parameter; they do not make any changes to the parameter values; and they do not make any other changes to the system outside the function). Give reasons for your answers.

Labelled section of code	Proper function?	Pure function?	Reasons
GetMainMenuChoice			
GetRowColumn			
OpenFile			
CheckWin			
WouldFitWithinBoard			
ValidateBoatPosition			

Pure functions in a spreadsheet

Most of the ready-made functions that may be used in constructing a spreadsheet, such as SUM and AVERAGE, are pure functions. The few exceptions include RAND and related functions for generating random numbers, and TODAY, for using the current date, as the result returned by those functions will not necessarily be the same each time they are called with the same arguments.



Provided that the formulas you write into cells steer clear of these few exceptions, then your formulas may be considered pure functions, too. And if your whole spreadsheet follows this pattern (and many do) then your entire spreadsheet may be thought of as a *single* pure function running inside a spreadsheet application (e.g. Excel).

Functions as first-class objects

Another defining characteristic of FP is that functions are treated as ‘first-class objects’. This means that you can treat functions just like data, in the following senses:

- You can build a list of functions, just like you can build a list of numbers or strings.
- You can pass a function as an argument into another function. A simple example, that we shall explore later, is a Sort function that takes two arguments: the list of data items to be sorted, and a ‘comparison function’ that will be used to compare any two items in the list, to determine their relative order.
- A function may return, as its result, another function. For example, you might have a single function that could return one of several different possible functions for converting a percentage into an exam grade, according to the subject and/or the exam board passed in.

Back to the spreadsheet again ...

Like FP, a spreadsheet doesn’t make a hard distinction between functions and data. If, when writing a formula, you include a reference to the cell B3, you don’t have to specify (or even know) whether cell B3 contains a constant value, a value to be entered by the user, or another formula.

Benefits of FP

What is the advantage of adopting the principles described above?

Re-usability

It is often claimed that FP improves re-usability - and it does - but it should be remembered that improved re-use is one of the claims made for a high proportion of advances in programming technology and practice. There is truth in most of these claims. The principle contribution toward improved re-use from OOP, for example, was ‘polymorphism’ (not, as is often wrongly stated, ‘inheritance’). FP facilitates re-use in several ways:

- Pure functions may be re-used in different contexts without the need to understand possible unwanted interactions with your existing code.
- Functions in FP may be made more general-purpose, by passing in part of the algorithm as another function.

- FP languages such as Haskell have a rich ‘type system’ that again facilitates writing functions that are more general purpose. We will say more about this in Chapter 13, but a simple example in Haskell would be that you could write a function that could work with any type of number (integer, floating point, or double-precision, say), without having to write specific code for the separate types.

The following paragraphs address benefits that are either unique to FP, are at least not common to many other forms of programming.

Improved quality through improved testability

Side-effect free functions are easier to test - because the output value depends solely on the arguments passed in. FP therefore lends itself to writing ‘executable tests’, which may be automated. In the old days, quality (of code) was largely expressed in terms of the thoroughness of the testing. But tests are limited to the test scenarios that you can imagine. In recent years, with the threat of viruses and other forms of security breaches, Computer Scientists are focusing less on empirical testing of code and more on finding proofs that a given piece of code will execute correctly under all possible circumstances. To do this, it is necessary to be able to ‘reason about’ code, using mathematically rigorous techniques, and the mathematical roots of FP are key to this.

Improved quality through improved reasonability

If you can be sure that simple functions named Foo and function Bar are both correct, then, in an FP environment, calling Foo with the result of calling Bar, can be guaranteed correct too. Note that this is *not* true for other forms of programming: if either or both of Foo and Bar had side-effects, or depended on things other than the arguments passed in, then there could be unpredictable consequences from calling one with the result of another. FP applies this rigorous approach from the very bottom to the very top of the functional hierarchy: from proving that the addition of two integers is correct (and will not produce errors for any combination of arguments), right up to proving that an entire program is both correct (does what was intended) and cannot produce any unhandled error conditions.

Tony Hoare's 'billion dollar mistake'

If you have written much code in VB you will almost certainly have encountered, at some point, a `Null` reference exception. One estimate is that this specific form of error has cost industry more than a billion dollars, since it first appeared in the early 1960s.

Tony Hoare is often cited as the author of the Quicksort algorithm, but has made many big contributions to Computer Science during a career spanning more than half a century. In 2009, he gave a presentation entitled 'Null References: The Billion Dollar Mistake', which you may view here: <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>. In the video he very honestly admits that he was responsible for designing the possibility of null references, which has caused so much pain since. (The design was deliberate, and introduced with the best of intentions.)



Haskell does not permit a null reference exception to arise. If you have a function that might produce, say, an `Integer` result, or might not, you must return a special type called `Maybe Integer`. This forces any function that uses the result to cater for, explicitly, the case where the result is null. (C# 8, released as this book was being finalised, has introduced a similar capability, and it seems likely that VB will follow suit.)

Tony also discusses a vulnerability in a single function within the C programming language (this was not his work) as being the enabler for the earliest viruses, and, thereby, of the entire 'malware' ecosystem (though modern malware now exploits far more subtle vulnerabilities).

The move towards more provable and 'reason-able' programs, of which FP is a significant part, is driven partly by these, and other, historical issues.

Terseness

Using FP, and especially when using a pure FP language like Haskell, it is possible to write more functionality with the same amount of code, or the same functionality in less code. Terse FP code is not necessarily easier to write, or to read, *while you are still learning it* - though with experience it would become as familiar to you as any other form of coding.

Improved efficiency through parallelism

When functions are side-effect free, it does not matter when or where they are called. Say, for example, we have a function named `Qux` that is called with three arguments, but the value of *each* of those arguments is the result of another separate function call. To evaluate `Qux` the system must evaluate each of those other functions to get the values - but it does not matter in what *order* it evaluates those three because they each depend only on their own arguments and are side-effect free. The three function calls (whether they are to the same function, or different functions) could be executed on separate 'threads' - which, if your processor has multiple cores, means that they may be executed on separate processor cores. When the last of the three has returned its value, the original function, `Qux`, may be evaluated.

This parallelism is one of the main reasons why FP is now gaining popularity in 'big data' processing, where a single request may be delegated to multiple processor cores, or even many separate devices, running in parallel.

Improved efficiency through referential transparency

Consider the following three lines of VB code:

```
Dim a = 3
Dim b = 4
Dim c = a + b
```

At a casual glance you can see that evaluating just these three lines of code is *always* going to result in `c` having the value 7 at the end, because the addition function (+) is both deterministic and side-effect free. An 'optimising compiler' might even work out this specific case automatically and write the machine code to just load 7 directly into the memory location representing `c`. But in procedural coding, this kind of optimisation cannot be extended very far because programmer-defined (or even library) functions might have side-effects and/or obscure dependencies.

In pure FP, a function with specific values will *always* produce the same result, and with no side-effects. This means that if you have a `SquareRoot` function, then once the system has calculated the square root of 3, it may store (we say 'cache') that result. The next time the program encounters a call to the `SquareRoot` function *for 3*, the system can simply look up the result from the cache, without having to calculate it again. The principal that you may replace any function call in code with the result of calling that function is known as 'referential transparency'.

Pure FP languages such as Haskell take full advantage of this: this is another example of designed-in 'laziness' - doing the minimum of calculation necessary each time. (Multi-paradigm languages won't *typically* be as efficient in this way, even when used to write code in an FP style).

Your times table as an example of using 'referential transparency'

If you want to calculate 7×9 in your head, you can work it out the long way - by repeated addition. But if you have learned your times table, you can just retrieve the memorized result of 63. In effect, the times table converts the *function* for single digit denary multiplication into a data structure.

That works only because multiplication of two integers is a *pure* function: 7×9 is *always* 63, irrespective of the time of day, your location, or what other calculations you have just made, for example. And evaluating it generates no side-effects.

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Unfortunately, most so-called 'functions' used in procedural programming don't obey this rule, so it would not be safe to pre-evaluate, nor to memorise ('cache'), the result for a given set of input values. But in FP *all* functions may be treated like multiplication. A programming language optimised for FP may build a dictionary (equivalent to a times table) to store the results of function calls, keyed by the input value(s). This increases speed, though it might use more memory.

Disadvantages of FP

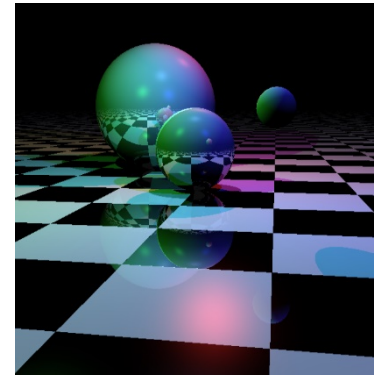
The disadvantages to FP are those that are associated with *any* paradigm shift that has yet to become the dominant paradigm: there is a steep learning curve involved for those transitioning from

another paradigm, and the supporting ‘eco-system’ (tools, books, training courses, examples, case studies) will be less rich.

It’s time for us to start coding. Initially we’ll be looking at some simple examples of procedural coding in VB, and seeing how the same things would be achieved using the principles of FP *in the same programming language*, before then doing the same thing in Haskell. In later chapters we’ll introduce more advanced concepts the other way around.

Case study on quality control - a ray tracing program

The screenshot (right) was created using 'ray tracing' - where individual light rays are traced, in three dimensions, from various light sources to a 'camera', reflected and diffused by the surfaces of various 'physical' objects modelled in the scene. The program, a modified version of a C# program written by Luke Hoban, has been used by the author for teaching vectors, since the code makes extensive use of ready-made `Vector3D` type. It is also a good example of OOP, as the application logic is almost entirely encapsulated on domain objects such as: `Camera`, `Sphere`, `Plane`, `LightSource`, and `Ray`.



Prior to exploring this program in a lesson, the author was making a series of small modifications to the code, just to standardise the coding style. He was confident that the small changes he was making would not affect the behaviour of the application. All went smoothly, until the following small change, where the public 'fields' on the `Camera` object were turned into 'properties'. The before (left) and after (right) versions of the `Camera` class are shown below, with the *only* changes made shown highlighted:

```
public class Camera
{
    public Vector3D Pos;
    public Vector3D Forward;
    public Vector3D Up;
    public Vector3D Right;

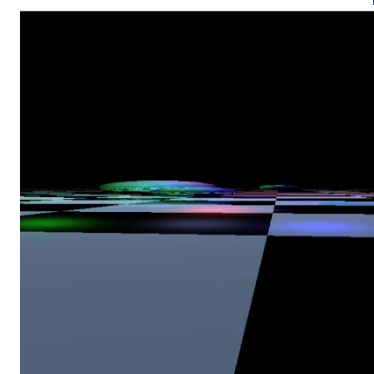
    public Camera(Vector3D pos, Vector3D lookAt)
    {
        Pos = pos;
        Forward = lookAt - pos;
        Forward.Normalize();
        Vector3D Down = new Vector3D(0, -1, 0);
        Right = Vector3D.CrossProduct(Forward, Down);
        Right.Normalize();
        Right *= 1.5;
        Up = Vector3D.CrossProduct(Forward, Right);
        Up.Normalize();
        Up *= 1.5;
    }
}
```

```
public class Camera
{
    public Vector3D Pos { get; private set; }
    public Vector3D Forward { get; private set; }
    public Vector3D Up { get; private set; }
    public Vector3D Right { get; private set; }

    public Camera(Vector3D pos, Vector3D lookAt)
    {
        Pos = pos;
        Forward = lookAt - pos;
        Forward.Normalize();
        Vector3D Down = new Vector3D(0, -1, 0);
        Right = Vector3D.CrossProduct(Forward, Down);
        Right.Normalize();
        Right *= 1.5;
        Up = Vector3D.CrossProduct(Forward, Right);
        Up.Normalize();
        Up *= 1.5;
    }
}
```

As expected, the modified code compiled and ran, but, to the author's astonishment, this small change, which he thought would have had no impact on the program's behaviour, generated the badly distorted image shown on the right.

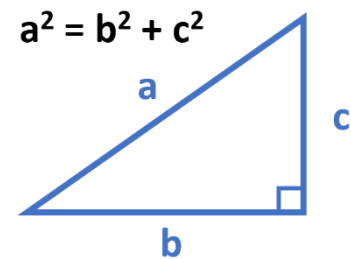
It took the author a long time to work out why the behaviour had changed. If you like a *difficult* technical programming challenge you may download the Ray Tracing (see Appendices – Installing Software), make the change to the `Camera` class and verify the result, and then try to figure out why it happened. But the author doesn't recommend it as an exercise!



The real point of the story is that had the whole system adopted the disciplines of FP, including side-effect free functions and immutable types, then this problem would not have arisen.

Chapter 3: Using expressions rather than statements

In procedural programming, the body of a function typically consists of a series of statements to be executed in a defined order. In FP a function returns the result of evaluating a single expression. Let's see how this works in practice using the example of a function that will calculate the hypotenuse of a right-angled triangle, given the length of the other two sides, using Pythagoras' theorem (right).



A procedural implementation of this function in VB might look like this:

```
Function Hypotenuse(SideB As Double, SideC As Double) As Double
    Dim bSq = SideB * SideB
    Dim cSq = SideC * SideC
    Dim sum = bSq + cSq
    Dim hyp = Math.Sqrt(sum)
    Return hyp
End Function
```

This is a simple function, and it is a straightforward exercise to re-write the body to return the result of evaluating a single expression – indeed you might even have chosen to write it this way in the first place:

```
Function Hypotenuse(SideB As Double, SideC As Double) As Double
    Return Math.Sqrt(SideB * SideB + SideC * SideC)
End Function
```

(Note that we have not changed the 'signature' of the function – meaning that it will still be called in the same way).

Our single expression may combine multiple 'operators' (such as +, *), following rules of operator precedence, and may embed calls to other functions, such as `Math.Sqrt` here. We could also have used another ready-made function to calculate the squares of the other two sides e.g. `Math.Pow(SideB,2)`.

If the expression is long enough, we might *choose* to format the code over more than one line, for example ...

```
Function Hypotenuse(SideB As Double, SideC As Double) As Double
    Return Math.Sqrt(
        SideB * SideB +
        SideC * SideC)
End Function
```

... but note that this is just a presentation change, not a coding change. The implementation still returns the result of a evaluating a single expression, that happens to be formatted over three lines - it is *not* a sequence of three sequential statements.

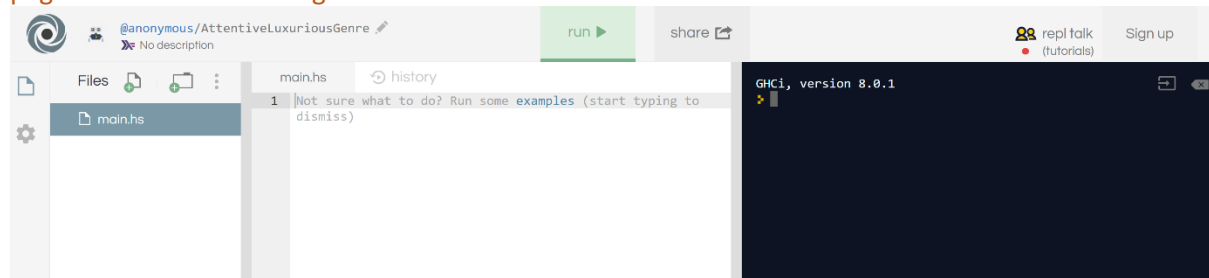
In purpose-designed FP languages such as Haskell, since every function returns the result of evaluating an expression, pure FP languages typically dispense with the `Return` keyword.

Introducing Haskell

We'll now take a first look at Haskell. We'll start by simply evaluating expressions that make use of standard mathematical operators or inbuilt functions. Then we'll use this learning to build an equivalent to the Hypotenuse function we've already looked at in VB, and then evaluate the function with some example arguments. The correct term for the latter is 'function application': you 'apply' a function to specific arguments.

Exercise 2

Visit <https://repl.it> and select the language **Haskell** from the drop-down list. You will be taken to a page that looks something like this:



In the example above AttentiveLuxuriousGenre is a randomly assigned project name. You don't have to 'sign up' but it is a good idea because you can then save work between sessions.

The right-hand pane (black background) functions like a console: you can just type in an expression and hit enter to evaluate it.

Try each of these examples and record the results returned alongside the expression.

```
3 + 4 * 5
(3 + 4) * 5
(3 + 20) * 5
20/7-4
```

Does it matter if you insert additional spaces between terms within the expression?

You will have noticed that, before each result, the Haskell compiler wrote the symbol `=>`. In this context, the symbol may be verbalised as 'evaluates to' or 'yields'.

Exercise 3

Now try these expressions, capturing the results.

```
True || True && False
(True || True) && False
not True
not True || True
not (True || True)
```

What logical functions do you think `||` and `&&` represent?

Why were the results for the two expressions (above) different?

We can also start using some existing functions that come as part of the standard Haskell library, but we first need to just say something about notation.

Infix, prefix, and postfix notation

Hopefully, you've already encountered the idea of 'prefix', 'infix' and 'postfix' notation (the last one is also known as 'Reverse Polish Notation' or RPN). Haskell makes use of the first two of these.

You've already seen the use of the familiar infix notation for mathematical operators, as in the example `4 * 5`. This is infix, because the operator symbol comes between the two operands. Infix notation makes sense only for operators or functions that take exactly two arguments.

In general, Haskell uses prefix notation: the function name precedes the arguments. This allows functions to take one, two, three, or more, arguments. VB code adopts the same approach. In our earlier code:

```
Math.Sqrt(SideB * SideB + SideC * SideC)
```

the `Math.Sqrt` function uses prefix notation, but the `*` and `+` operators use infix. The difference in Haskell is that the prefix notation does not have brackets around the argument(s). And if there is more than one argument, they are separated only by spaces, not by commas.

Exercise 4

Try these expressions that apply in-built functions to arguments. Record the result returned in each case.

```
min 6 5
max 7 86
max 6 5+2           (Make sure you understand why the result is what it is.)
max 3 (min 7 2)
succ 77
```

You might have been surprised by the result of the third expression. The reason is that Haskell evaluates expressions left-to-right. The `max` function takes two arguments, and the Haskell compiler takes the first thing that can constitute a valid argument in each case - 6 and 5 in that case. The result of `max 6 5` is evaluated (to 6) and then the expression evaluator continues with `+2`. If we want `5+2` to be the second argument for `max` then we need to put brackets around that sub-expression.

One of the hardest aspects of beginning Haskell is the obscurity of its error messages.

Exercise 5

Each of the following expressions will generate an error message. Record the message in each case. Don't worry about trying to understand the specific meaning of each message for now, just see if you can understand why an error has been generated.

```
MAX 3 7
max 3,7
max 3 7 2
max 3 max 7 2
```

How *should* we find the maximum of the three values?

Now let's define our own function to calculate the hypotenuse. First, we will define the function directly from within the console window, as before:

Enter the following function *definition* in the console, and hit Enter:

```
hypotenuse b c = sqrt (b*b + c*c)
```

The function definition starts with the function name, followed by a list of named values. The equals sign indicates that this function is evaluated using the expression that follows, which makes use of the named values.

Note that you may format a function definition over multiple lines, but if you do this then the body of the function should be indented by at least one space from the name:

```
hypotenuse b c =  
  sqrt (b*b + c*c)
```

'Variables' in Haskell

In our `hypotenuse` function above, `b` and `c` are known – in Haskell – as 'variables'. In VB the equivalent would be known as 'formal parameters' (or, in common parlance, just 'parameters'). Haskell calls them variables because their values will typically vary each time the function is called. This is the same usage of the term variable as in mathematics: in the mathematical statement

$$f(x) = x^2$$

`x` is described as 'the variable'.

But this is not quite the same meaning of 'variable' as you will be used to in procedural programming, where a variable is an identifier whose value may be explicitly assigned, and subsequently re-assigned, by the program code. In procedural programming, loops depend upon this ability to keep changing the value of at least one variable, such as a loop counter.

Haskell does *not* permit the value of a variable to be altered within a function, because there is no need to do so in FP. This will become clearer as we develop our examples in the next few chapters. For the time being remember that a 'variable' in FP refers to an identifier for a value passed as an argument into a function, and whose value *may* vary each time the function is called; it does *not* mean an identifier whose value may be altered explicitly by the program code.

Exercise 6

Having defined the `hypotenuse` function, we may now use it within expressions. In the `repl.it` console, the following expressions, recording the result in each case:

```
hypotenuse 3 4  
hypotenuse 5 12  
hypotenuse 6 6
```

Note: Haskell requires that identifiers for both functions and variables begin with a lower-case letter.

Coding style

In procedural or object-oriented programming, such as is *usually* practiced with VB, it is considered good practice to use long, explanatory names for variables. In Haskell, and other FP languages, identifiers for variables are commonly kept to single letters. In part this is because Haskell programmers favour succinct code. But it is also because the 'scope' of variables is limited to the single expression evaluated by the function, and it is rare to have functions with many separate arguments - so a programmer reading Haskell code does not have to mentally keep track of the meaning of variables over multiple statements.

We'll now start defining functions in code files, much as you will be used to doing in VB. For this we will be using the code editor pane in `repl.it` (the pane with the white-background to the left of the Console). You compile and run the code using the **Run** button. This will automatically save your source code file before it is compiled and run.

Exercise 7

Copy the code that you used above to define the `hypotenuse` function into the `main.hs` code file in the code editor.

Then, on a separate line, add a `main` function to call the `hypotenuse` function with specific values and print the result to the console:

```
main = print (hypotenuse 3 4)
```

Run the program and record the result.

Does it matter which order the functions are declared in the file? (try swapping them)

Tip: Do you need a main function in Haskell?

If you define multiple functions in a program file, then the Haskell implementation on `repl.it` will look for one called `main` to run, and if it fails to find one you will get an error message like this (the second part of which is unhelpful in this case):

- Variable not in scope: main
- Perhaps you meant 'min' (imported from Prelude)

But what if you have no real need of a `main` function - or are not yet ready to think about what it should do - and you want to compile and test your code in the console? Many Haskell programmers using `repl.it` choose to add this 'default' `main` function:

```
main = print "OK"
```

If your code has no compile errors then, when you hit **Run** you will see the message **OK** on the console, and you will know that you can then start invoking your function(s) directly from the console.

Static typing

VB is a 'statically typed' language: you will be used to specifying the types (e.g. `Integer`, `String` or `Boolean`) for all variables, formal parameters, and the function return type. You might be aware that VB also supports 'type inference', as in this example:

```
Dim a = Math.Sqrt(3)
```

Exercise 8

In the code above, the VB compiler will infer the type of the variable `a` as being a `Double`. But why has it inferred that type, rather than, say, `Float` or `Decimal`?

Note, however, that even when the type is *inferred*, it is still *static*. In the example above, the variable `a` *permanently* has the type `Double`, just as if you had declared it explicitly, like this:


```
Dim a As Double = Math.Sqrt(3)
```

You will not be able to subsequently assign the variable `a` to a `String`, for example. This contrasts with 'dynamically typed' languages, such as Python and JavaScript, where the type of a variable is determined only at run-time, and may even change within the program.

Haskell, like VB, is a statically typed language. For every function, the type of the value returned, and the type of each of its arguments, is determined at compile time. This is possible either because the programmer has explicitly specified those types, or because the Haskell compiler has been able to infer the types from the code (as it has done for our hypotenuse function, so far). If the compiler is not able to infer the type, because there is ambiguity, then the programmer must specify the type explicitly. You may define the type signature for *any* function, even if there would be no ambiguity. In the code below, we have added an explicit type signature (highlighted) for our hypotenuse function:

```
hypotenuse :: Float -> Float -> Float
hypotenuse b c = sqrt (b*b + c*c)
```

Experienced Haskell programmers typically define the type signature explicitly for every function they write. By convention, the type signature is written on the line above the function definition. It begins with the name of the function followed by `::` and then a list of types. We will explain the significance of the arrow (`->`) later in the book. For now, the important thing to remember is that the *last* entry defines the type of the value that will be *returned* by the function - the others define the type of the arguments (in this case corresponding to the variables `b` and `c` respectively) that must be passed in. In our example here, the types are all `Float` i.e. single-precision fractional numbers.

Just as a variable has a type, every function in Haskell has a type. In our example, the type of `b` is `Float`, and the type of `hypotenuse` is `Float -> Float -> Float`. Remember that in FP, functions are first-class objects: and it is important when, say, passing a function as an argument to another function (which we will come to in Chapter 9), or when adding a function to list, say, that the function is of the correct type.

The type system of Haskell is one of its strongest features. For those interested, we will learn more about this in Chapter 13, but it is not necessary to understand this type system to make initial progress with learning the language.

Chapter 4: Returning multiple values from a function

FP was adopted by the mathematicians and scientists, long before it was adopted for any business applications, perhaps because mathematicians are very familiar with the concept and terminology of functions. It would be wrong to think that FP is used only for mathematically oriented applications. However, we will stick with the mathematical theme for the moment, and write a function that will calculate the roots of a quadratic equation using the formula shown on the right.

$$ax^2 + bx + c = 0.$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Quadratic equations have two roots (hence the +/- in the formula), so our function must return two values. In VB we *might* choose to implement this function by passing in arguments 'by reference'. The code below shows a possible function definition for `QuadraticRoots` using this approach, and a `Main` function from which it is called. Note that we have restricted the values for `a`, `b`, and `c` to integers just for simplicity, but the roots are of type `Double`:

```
Sub Main()
    Dim root1 As Double = 0
    Dim root2 As Double = 0
    QuadraticRoots(5, 11, -12, root1, root2)
    Console.WriteLine(root1)
    Console.WriteLine(root2)
    Console.ReadKey()
End Sub

Sub QuadraticRoots(a As Integer, b As Integer, c As Integer, ByRef root1
As Double, ByRef root2 As Double)

    Dim partRoot = Math.Sqrt(b * b - 4 * a * c)
    root1 = (-b + partRoot) / (2 * a)
    root2 = (-b - partRoot) / (2 * a)
End Sub
```

Exercise 9

Run the code above and then record the values for the two roots that it returns below

Why are the `root1` and `root2` parameters of the function being passed 'by reference' here?

Why has `partRoot` been defined as an intermediate variable here?

Now apart from the fact that we are back to using sequenced statements, which we will address shortly, we now have another pattern in our code that would violate the rules of FP: the body of the function is (deliberately, here) changing the values being passed in. In other words, the function has 'side-effects'.

If we are going to avoid side-effects, we need to return the values *as the result* of the function, not by treating them as 'pseudo arguments'.

In VB, if we need to return multiple values from a function then we could simply return them as two elements of an array or a list. This is fine if the two (or more) values have the same type (as they do in this case); if they have different types then we could put them both/all in an array or list of type `Object`, though this would mean that we would lose the type information. Another option would be to define our own specific data type - as a class, or as a Structure with properties of the correct type for each value. This might seem a bit like using a sledgehammer to crack a nut, however.

Tuples

A better option is to use a 'tuple'. A tuple is a very simple data structure that typically holds a small number (in VB the maximum is seven) of related 'items', which may be of different types, while preserving the type information. Tuples are especially useful for FP because they are 'immutable': having created the tuple its contents cannot be changed, which also helps towards the goal of eliminating side effects.

Traditionally, tuples in VB used syntax like this:

```
Dim foo As Tuple(Of String, Integer, Char) = Tuple.Create("MyName", 41, 'A');
```

In 2019 Microsoft released C# 8.0 with many new FP features including a far more elegant syntax for tuples, very similar to that of Haskell and other purpose-built FP languages. If you use an earlier version of C#, or you use VB, you may add this capability by installing the **System.ValueTuple** NuGet package (see Appendix I: Installing SoftwareAppendix I: Installing Software). *Do this before attempting the next exercise.*

The tuple syntax now looks like this:

```
Dim foo As (String, Integer, Char) = ("MyName", 41, 'A');
```

Having installed the **System.ValueTuple** NuGet package, modify your code from the previous exercise to use tuples, by making the highlighted changes shown below. Note that in this case the tuple has two elements, and both are double-precision floating point numbers.

```
Sub Main()
    Dim result As (Double, Double) = QuadraticRoots(5, 11, -12)
    Console.WriteLine(result.Item1)
    Console.WriteLine(result.Item2)
    Console.ReadKey()
End Sub

Function QuadraticRoots(ByVal a As Integer, ByVal b As Integer, ByVal c As Integer) As (Double, Double)

    Dim partRoot = Math.Sqrt(b * b - 4 * a * c)
    Dim root1 = (-b + partRoot) / (2 * a)
    Dim root2 = (-b - partRoot) / (2 * a)
    Return (root1, root2)
End Function
```

In VB, the items in a tuple are labelled `Item1`, `Item2` etc and each has the type it was created with. (This is one of the rare cases in the field of Computer Science, where a list starts the numbering from 1 instead of 0!)

Now, we *could* turn this code back into a single expression like this perhaps:

```
Return ((-b + Math.Sqrt(b * b - 4 * a * c)) / (2 * a),
        (-b - Math.Sqrt(b * b - 4 * a * c)) / (2 * a))
```

Note that to improve readability, we have formatted the code over two lines, but it is still a single expression. However, there is another problem here: duplication of code: the sub-expression `Math.Sqrt(b * b - 4 * a * c) / (2 * a)` appears twice.

Hopefully, by now, you have already encountered the 'DRY' principle (if not, see panel).

The DRY principle: Don't Repeat Yourself

Repetition – using the same fragment of code in more than one place – is a 'code smell' (that's the term professional programmers use!). As well as being wasteful (though this is not typically a big issue in modern computers, which typically have plenty of memory), duplicated code runs the risk of the two versions becoming inconsistent, especially when code is modified subsequently. The DRY principle can be applied at many different levels within a program, and much of the skill of programming is about learning to spot more subtle violations of this principle and learning patterns for eliminating the duplication.

One way around this duplication would be to declare `PartRoot` as another function.

Make the changes shown below. The `Main` function does not change, because we have not changed the signature of the `QuadraticRoots` function.

```
Function QuadraticRoots(ByVal a As Integer, ByVal b As Integer, ByVal c As Integer) As (Double, Double)
    Return ((-b + PartRoot(a, b, c)) / (2 * a),
            (-b - PartRoot(a, b, c)) / (2 * a))
End Function

Function PartRoot(a As Integer, b As Integer, c As Integer) As Double
    Return Math.Sqrt(b * b - 4 * a * c)
End Function
```

Our `QuadraticRoots` function is still a proper function and side-effect free, *provided that* `PartRoot` is also a proper function and side-effect free, which in turn depends upon `Math.Sqrt` being the same (which it is). If we were using a purpose-design FP programming language, such as Haskell, that rule would be automatically enforced by the compiler. Because we are working here in a multi-paradigm language, VB, we need to take care to check that we are adhering to this rule.

Tuples in Haskell

Turning to Haskell, we could take the same approach of decomposing the problem into two single-expression functions:

```
partRoot a b c = sqrt(b * b - 4 * a * c)

quadraticRoots a b c = ((-b +partRoot a b c)/(2 * a),
                       (-b -partRoot a b c)/(2 * a))

main = print (quadraticRoots 5 11 (-12))
```

Some things to notice:

- As with the VB code, we have formatted the definition of the `quadraticRoots` function across two lines to make it easier to read. By default, it would be formatted over a single line.
- Creating a Tuple in Haskell is very simple: `(2.5, 3.7)` would define a two-item tuple containing two fractional values. See how this is used within the `quadraticRoots` function.
- When applying the `quadraticRoots` function (in `main`) we had to pass the third, negative, argument in brackets i.e. `(-12)`. Otherwise, Haskell would interpret the `-` and `12` as two separate arguments, which would not fit the requirements of the function. This will seem strange but remember that in FP, functions can be passed as arguments, and `'-'` is a function in its own right. We'll look at how to use this capability to our advantage, later.

Exercise 10

Paste the Haskell code below into the `main.hs` file on repl.it – replacing any previous code entirely. Run the program and paste a screen-snippet showing just the output produced in the console.

Does the order in which the functions have been declared matter? (Try swapping them).

Notice that the result is also shown formatted as a tuple (the `print` function can, fortunately, handle a tuple).

We should also add the type signatures, shown below in the simplest form, restricted to using the type `Float`.

```
main = print (quadraticRoots 5 11 (-12))

partRoot :: Float -> Float -> Float -> Float
partRoot a b c = sqrt(b * b - 4 * a * c)

quadraticRoots :: Float -> Float -> Float -> (Float, Float)
quadraticRoots a b c = ((-b +partRoot a b c)/(2 * a),
                       (-b -partRoot a b c)/(2 * a))
```

The type signature for `partRoot` specifies that the three arguments must all be of type `Float`, and that the function also returns a `Float` type as a result.

In the type signature for `quadraticRoots`, there are also three parameters of type `Float`, but the return type is shown as `(Float, Float)` – indicating that the function returns a tuple containing exactly two items, each of type `Float`.

Exercise 11

Run the program, then call the `quadraticRoots` function directly from the console by typing in the following expressions into the console, recording the results in each case:

```
quadraticRoots 1 3 2
quadraticRoots 1 1.5 (-59.5)
quadraticRoots 3 7 9
```

Can you explain what has happened in the third case? (Try solving the quadratic equation manually, or using a calculator).

What do the letters in the returned tuple stand for? (Search the web, including 'Haskell' in your search terms).

Exercise 12

In the `quadraticRoots` function, the `partRoot` function is called twice, with the same three arguments – `a b c` – in each case. This is the case for both the VB version and the Haskell version, and they will produce the same results. But the Haskell compiler will do something clever here, capitalising on a principle of FP that was mentioned earlier in the book. Can you recall what this that principle was called and what it means?

Using 'let' in Haskell

The problem with creating the `partRoot` function, as shown above, is that it feels somewhat *artificial*. It has been written solely for use within the `quadraticRoots` function, and is unlikely to be re-used in any other context. If `quadraticRoots` formed part of a library of mathematical functions for, say, solving polynomial equations, then the `partRoot` function would just be unwanted clutter.

To avoid this, Haskell offers another way to avoid duplicating the part root expression, using the `let ... in` pattern as shown below. (In this example, the `partRoot` function has been deleted because it is no longer needed):

```
quadraticRoots:: Float -> Float -> Float -> (Float, Float)
quadraticRoots a b c = let p = sqrt(b * b - 4 * a * c)
                        in ((-b + p)/(2 * a), (-b - p)/(2 * a))
```

In this pattern, the 'let clause' - `let p = sqrt(b * b - 4 * a * c)` - defines a value `p` (standing for 'part root') that is going to be needed, typically more than once, to evaluate the main expression that defines the function, which follows the `in` keyword: `((-b + p)/(2 * a), (-b - p)/(2 * a))`. The key difference between this pattern and our previous solution (that defined a standalone `partRoot` function) is that `p`, and the expression that defines `p`, are visible only *inside* the `quadraticRoots` function.

Again, the function definition, including the `let...in` structure, could have been written on a single line, but breaking it across two lines and using indentation helps readability. (Indentation is required when you format a function over multiple lines, though it needn't be as much as shown here).

Exercise 13

Delete the `partRoot` function and make the change to `quadraticRoots` shown above. Then evaluate `quadraticRoots` with the arguments `36` and `77`. Record the result.

You might well be thinking that `let...in` structure looks rather like defining statements to be executed in sequence (in other words: procedural programming) – as we did in the first VB example in this Chapter. But this is not the case - the whole implementation of our modified `quadraticRoots` function is still a *single* expression.

Chapter 5: Handling conditions

In our two examples so far, converting sequential statements into a single expression has been straightforward. But how is this going to work for ‘selection’, for example using `If` or `Select`. .Case statements? Consider the following function, written using the procedural approach, to grade an exam score:

```
Function GradeScore(percentage As Integer) As String
    Dim grade As String = ""
    If percentage > 50 Then
        grade = "Pass"
    Else
        grade = "Fail"
    End If
    Return grade
End Function
```

Using the conditional function (ternary operator) in .NET

To perform selection in FP we need to use a conditional *function*. Such a thing does exist in VB, where it is also known as the ‘ternary (conditional) operator’, because it takes *three* arguments:

- the condition (as an expression)
- value (or expression to be evaluated) to be returned if the condition evaluates to true
- value (or expression to be evaluated) to be returned if the condition evaluates to false

The following code is functionally identical to the previous version, is more succinct, and, more importantly, now implements the whole function as a single expression:

```
Function GradeScore(percentage As Integer) As String
    Return If(percentage > 50, "Pass", "Fail")
End Function
```

The key difference between the ternary conditional operator and a regular `If statement`, is that the ternary conditional operator *is a function* and, hence, it returns a value. A regular `If` statement, by contrast, does not return any value – it just changes the flow of control.

You can nest ternary conditional operators within other ternary operators, just as you can nest `If` statements within other `If` statements.

Exercise 14

In the code above, replace just the code `"Pass"` with

```
If(percentage > 80, "Distinction", "Pass")
```

Your code is now handling a nested condition, allowing for three possible outcomes. Capture your modified version of the function.

Spreadsheets have an IF function

If the idea of ‘If as a function’ sounds vaguely familiar, perhaps you have used the IF function on a spreadsheet. This, too, has three arguments: a condition, and two expressions, which may be as simple as references to two other cells, one yields the value if the condition evaluates to true, the other if the condition evaluates to false.

The screenshot shows a spreadsheet with the following data:

	A	B	C	D	E
1	7	Big	Small		
2	Big				
3					

The formula bar for cell A2 shows the formula: `=IF(A1>5, B1, C1)`

Selection in Haskell

Haskell offers more than one way to implement selection. The first way is, perhaps surprisingly, with `if ... then ... else` keywords, as shown below:

```
gradeScore :: Int -> String
gradeScore p = if p > 50 then "Pass" else "Fail"

main = print "OK"
```

Note that this is *not* equivalent to the *If statement* in VB, but it *is* equivalent to the conditional function or ‘ternary operator’: it returns a result.

Exercise 15

Enter the code above into `main.hs` and run the program, then invoke the function directly from the console with the following expressions recording the results:

```
gradeScore 51
gradeScore 50
```

Now modify the Haskell implementation, nesting the `if then else` clauses so that it results in `Distinction`, `Pass` or `Fail` the same way as your most recent VB version. Paste in the complete code for your modified `gradeScore` function, and a screenshot showing it being called with a score of 90.

Using guards in Haskell

The other approach to conditional logic in Haskell is illustrated below:

```
gradeScore :: Int -> String
gradeScore p
  | p > 80 = "Distinction"
  | p > 50 = "Pass"
  | otherwise = "Fail"
```

Here we see three uses of the symbol `|` (which may be verbalised as “vertical bar” or “pipe”) followed by a condition. This pattern is known as a ‘guard’ - so we can say that our `gradeScore` function now has three guards. The first guard can be verbalised as ‘If `p` is greater than 80 then the overall function will return “Distinction”’.

In the example above each guard has been placed on a new line, for readability - in which case each one *must* be indented from the definition of `gradeScore` by at least one space - but it is possible to format the whole function on one line if preferred.

You will probably agree that the syntax using guards is more elegant than using nested `if` then `else` clauses. Note that Haskell will return the specified expression *for the first guard where the condition evaluates to true*, and will not then evaluate any further guards in that function: another example of designed-in laziness.

Exercise 16

Try out the new version (above) and confirm that it works correctly. What happens if we call this function, from the console, using the percentage score of 64.5? Why? How could we accommodate this possibility?

Instead of using the `otherwise` keyword as the final guard, we could have specified another condition e.g.:

```
| p < 50 "Fail"
```

What is the risk when writing a final condition instead of `otherwise`? What is the specific error that we have made in using the condition given above as our last guard, and what error would this result in?

We can thus use the guard pattern to perform the equivalent to a `select-case` statement in VB.

Chapter 6: Using functional lists

In procedural programming you will already have encountered the idea of a List – a data structure that can hold multiple elements, typically of the same type, for example:

```
Dim list1 = New List(Of Integer)() From {7, 2, 3, 99, 4}
Dim list2 = New List(Of String)() From {"Tomasina", "Ricardo", "Harriet"}
```

Functional lists have a head and a tail

FP supports the idea of lists, but they have a different structure. A ‘functional list’, consists of exactly two elements: a ‘head’ and a ‘tail’. In a list of integers, the head will be an integer. The tail, however, will be another *list of integers*. Which means that the tail will have its own head, and its own tail, and so on.

This is best understood by going straight to Haskell; after that we will see how the same pattern may be adopted in VB.

Functional lists in Haskell

In Haskell we may *set up* a list in a familiar way, for example:

```
colours = ["red", "yellow", "green"]
```

Exercise 17

In the Haskell *console* window on repl.it type in the line that defined the list of three colours above. Then evaluate each of the following expressions, which use ready-made functions that work on lists. In each case record what is returned.

```
length colours
head colours
length (tail colours)
tail colours
head (tail colours)
tail (tail colours)
tail (tail (tail colours)) - what do you think the returned result means?
tail (head colours) – this result might surprise you.
```

The last expression evaluated to the result that it did because, in Haskell, strings are just lists of characters: `head colours` returned just the first colour (as a string), so calling the `tail` function then returned the string minus its head (character).

Functional lists in .NET

We are now going to make use of a library, purpose-written for this book, that provides a data structure equivalent to Haskell’s list for use with VB. The library can be installed as a NuGet package into any VB project where you want to use functional lists. If you are familiar with installing NuGet packages, just search (in the **NuGet Package Manager** window) for **MetalUp.FunctionalLibrary** and install it (see Appendices – Installing Software).

Here is an example of code making use of a few of the capabilities from the **MetalUp.FunctionalLibrary** to emulate the behaviour of the Haskell code above.

```
Imports MetalUp.FunctionalLibrary.FList

Module Module1

    Sub Main()
        Dim colours = NewFList("red", "yellow", "green")
        Console.WriteLine(Length(colours))
        Console.WriteLine(Head(colours))
        Console.WriteLine(Length(Tail(colours)))
        Console.WriteLine(Tail(colours))
        Console.WriteLine(Head(Tail(colours)))
        Console.WriteLine(Tail(Tail(colours)))
        Console.WriteLine(Tail(Tail(Tail(colours))))
        Console.WriteLine(Tail(Head(colours)))
        Console.ReadKey()
    End Sub

End Module
```

(Note: You will have used a `Imports` statement before, but you might not have seen it used in exactly the form used above, specifically with the highlighted code. The reason for this form is that the new functions being used in the code: `NewFList`, `Length`, `Head` and `Tail`, are all defined on the class `FList`. By specifying the `Imports` as we have, the new functions can be invoked directly, instead of having to write the type and function name each time i.e. : `FList.NewFList`, `FList.Length`, `FList.Head` and `FList.Tail`.

Exercise 18

Install the package into a new Console project, then add the code above and run the program. Record what is printed to the console

You will see that the **MetalUp.FunctionalLibrary** has given us the ability to emulate the behaviour of 'functional lists' in VB, but in a form that is consistent with VB syntax. Many of the functions e.g. `Head()` and `Tail()` can even work with strings, *as though they were functional lists* (even though this is not how they are implemented in VB).

Why did Haskell (and other pure FP languages also) decide to adopt this different structure for the lists?

One reason is that this structure works very well with 'recursive' functions that operate on lists. We will look at this in the next chapter.

The other reason is that this head:tail structure is convenient when working with 'immutable types' - meaning that you can never change an existing data value or data structure, you must create a new value or data structure each time. This, you might recall from Chapter 1, is one of the definitional principles of FP, because to change existing values or data structures would be to create side-effects.

Working with immutable types

VB has some standard immutable types - one of them is `String`. When you first programmed in VB you might have been caught out by this occasionally. Look at the code below:

```
Sub Main()  
    Dim name = "richard"  
    name.ToUpper()  
    Console.WriteLine(name)  
    Console.ReadKey()  
End Sub
```

Exercise 19

Without running the code, write down what you think will be output on the Console?

Now run the code in a new Console program. If your answer to the previous question was right: well done! If not, don't worry: it is a very easy mistake. But can you figure out why you got it wrong?

In VB all types are objects, and many of them are 'mutable' - meaning that it is possible to invoke a method and thereby change the instance itself. That's what the programmer of the code snippet above probably thought was going to happen. But in VB a string is an 'immutable' type: you can never change an existing string, but there are many functions that will take in an existing string and return a *new* one, based on the input string, but changed in some specified way. The original (input) string remains unmodified.

Exercise 20

So what should the VB programmer have written in order to get the name to upper case?

In FP, *all* types are immutable: you cannot change a data value, or the contents of any data structure. But you will often have functions that take in an existing data value or structure, and return a brand new one that is a copy of the input, but with specified differences. When you want to add an element to a list, remove an element, or just update an existing value within the list, you always end up creating a new list, even if you don't realise that this is happening. Let's look at this in Haskell first.

Adding to a functional list

In Haskell, the two simplest ways to add to a list are:

- 'Prepending' - adding a single item to the start of the input list. The syntax for this is the `newItem : existingList`
- 'Appending' - adding a new *list* (of multiple items, or a list containing a single item) to the end of the input list. The syntax for this is `inputList ++ listOfItemsToAdd`

Exercise 21

Type the following in the Console on **Repl.It**, and record the result returned in each case.

```
colours = ["red", "yellow", "green"]
```

Now evaluate these expressions directly, and record the result returned in each case:

```
"black" : colours
colours ++ ["blue", "violet"]
```

These two expressions produce errors. Why?

```
colours ++ "white"
["brown", "white"] : colours
```

How could we just add the colour white to the end of the original list?

Now enter just:

```
colours
```

Were you surprised by what it evaluated to? Why has it not changed?

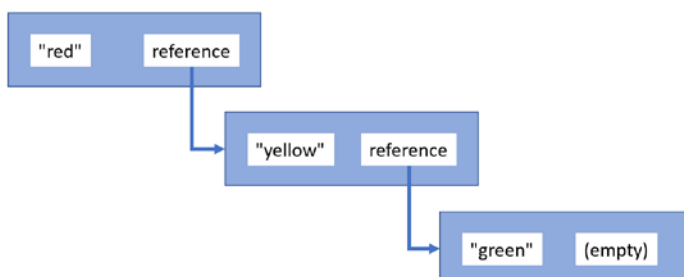
There are many functions for manipulating lists. Two more include `take` and `drop`. They each take two arguments: an integer value followed by a list name. Experiment with `take` and `drop` and describe what they do. (You might find it helpful to start with a longer list of colours).

The **MetalUp.FunctionalLibrary** contains functions for manipulating existing lists by prepending, appending, taking, and dropping, as shown below. Note that all of them use the standard prefix notation rather than dedicated operators as Haskell does.

```
Console.WriteLine(Prepend("black", colours))
Console.WriteLine(Append(colours, NewFList("blue", "violet")))
Console.WriteLine(Take(2, colours))
Console.WriteLine(Drop(2, colours))
```

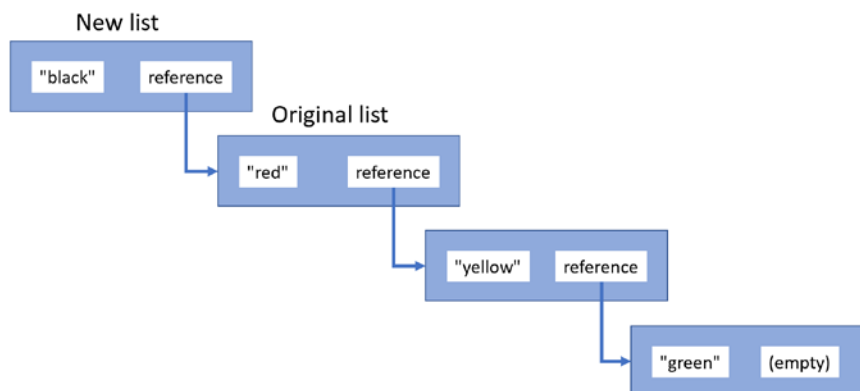
Why the Head:Tail structure is an advantage when working with immutable lists

Copying a whole list each time you want to add/remove or change a single element might seem very wasteful both in terms of memory and processing effort. But this is not always the case. A functional list consists of just two elements: the head, and a 'reference' (or 'pointer') to the tail of the list, which is another list. When we get to the end of the list, the tail will be an empty list. Our short list of the colours 'red', 'yellow', and 'green', looks like this:



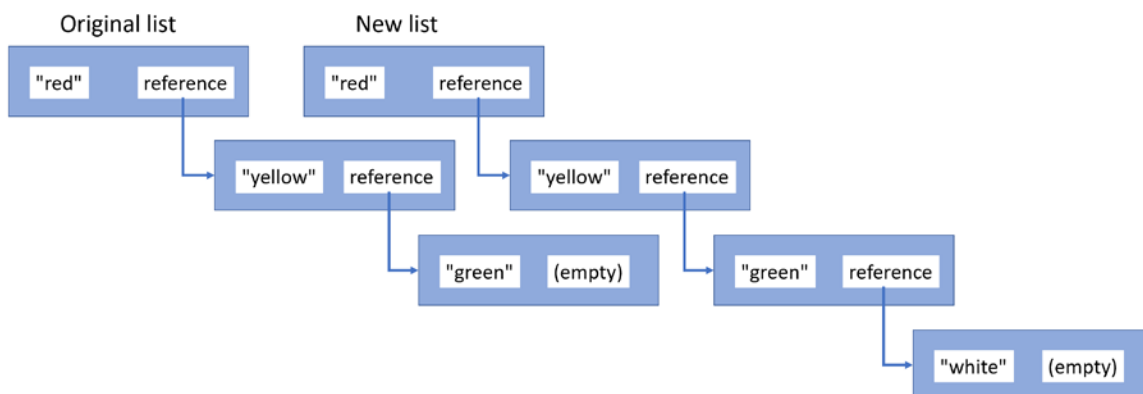
In other words, under the covers, a list of three elements consists of three separate list entities, each with just a head element and a reference to the tail list.

Crucially, each of these lists is *immutable*. When you create a new list by prepending 'black' to it, you are not actually copying any of the elements: you are merely creating a new list with a head of 'black' and a tail-reference pointing to the existing list:



Anything referencing the original list (with 'red' as its head) does not see any change.

What about *appending*? You might now have realised why, in Haskell, you may only append a *list* to a list. You might think that you could simply update the reference on the last list (with 'green' as its head) to point to the new list being appended. However, this would be breaking the rules, because it could impact other functions that are using the original list. So, for appends, the system does have to copy the whole existing list first:



For this reason, if you are building up a list one element at a time, it is more efficient to build it from the back end, prepending new elements. That said, unless you are working with significant amounts of data, you are not going to notice any different: the Haskell compiler generates highly optimised code.

A further advantage of the head:tail structure of a functional list, is that it facilitates the task of processing the elements in a list, recursively, because your function to process a list can just process the head and then call itself with the tail, as we shall see in the next chapter.

Chapter 7: Replacing loops with recursion

We have already encountered the idea that although Haskell and other pure FP languages use the term ‘variable’, it doesn’t have quite the same meaning as in procedural programming, because you can never assign an existing variable to a new value. So far this has not presented any significant a real problem, but that’s because we haven’t yet tried to do any kind of looping.

In procedural programming we make extensive use of various kinds of loops. Let’s say we need a function that will return the sum of the squares of all the values from 1 up to the input number, so, for example, calling the function with the value 3 will return $1^2 + 2^2 + 3^2$, or 14. One implementation might look like this:

```
Function SumOfSquaresTo(n As Integer) As Integer
    Dim result As Integer = 0
    For i As Integer = 1 To n
        result += i * i
    Next
    Return result
End Function
```

Loops, whether of the ‘for-next’ kind (used where we know in advance how many iterations we require), or the ‘while’ kind (where we just keep going until a condition is met), all depend on having one or more variables that will potentially change in their value as the loop iterates.

Since FP does not permit variables to change in value, it follows that you can’t have procedural style loops. But since FP is ‘Turing complete’ (meaning that it can implement any ‘computable’ problem) how can we implement a function to generate the sum of squares from 1 to n? The answer lies in recursion.

Anything you can do with a loop, you can do using recursion instead

Hopefully you have already encountered the idea of recursion in procedural programming. A recursive procedure or function is one that ‘calls itself’ - typically with a smaller version of the same problem. Each call to itself stores the state of the calculation on the ‘call stack’ until some end-condition is reached, and which point it starts ‘popping’ the intermediate results back off the stack, combining them to form the result.

You have probably applied recursion to problems that *appear* to have an inherently recursive structure to them, but it is important to understand that any problem that may be solved using recursion may also be solved using loops, *and vice versa*.

Thinking about our original problem recursively, the result can be expressed, in general terms, as follows:

$$n * n + \text{SumOfSquaresTo}(n - 1)$$

Were we just to make our function return this expression, though, we would hit a problem?

Exercise 22

What have we forgotten to do? And what would be the consequence if we were to run that code?

So, staying with VB here’s a recursive implementation that will work correctly. (Notice that we are using the ‘ternary conditional operator’ (the ‘if function’) that was introduced in Chapter 5, so our

whole function is implemented as a single expression. Notice that the function also uses a ‘ternary conditional operator’ (or ‘If function’), that was introduced in Chapter 5, to specify the ‘end condition’ for the recursion - so that we have kept our implementation to a single expression.

```
Function SumOfSquaresTo(ByVal n As Integer) As Integer
    Return If(n <= 1, n, n * n + SumOfSquaresTo(n - 1))
End Function
```

Each time the function is called, recursively, the parameter `n` (which would be called a ‘variable’ in Haskell) has a smaller value. Importantly, though, we are never re-assigning the value of `n`. In fact, the call stack (until it is eventually popped) will have multiple, separate, versions of a variable called ‘`n`’, each one specific to the context (or ‘scope’) of one call of the function. The compiler does not ‘think’ of these various separate variables called ‘`n`’ as being related to each other at all. When the value of `n` gets down to 1, the exit condition will apply, and the system will progressively pop the ‘frames’ from stack, building up the final result.

If you wish, you can verify that the code above works, or you can move straight onto implementing the function in Haskell. Here we will make use of guards to implement the ‘end condition’ (though we could have used Haskell `if...then` syntax instead).

```
sumOfSquaresTo :: Int -> Int
sumOfSquaresTo n
  | n <= 1 = n
  | otherwise = n * n + sumOfSquaresTo(n- 1)

main = print ("OK")
```

Exercise 23

Try the code above in repl.it and record the results when you evaluate these expressions:

```
sumOfSquaresTo 3
sumOfSquaresTo 100
sumOfSquaresTo 1
sumOfSquaresTo 0
```

Recurring over lists

Many of the loops that you write in procedural coding involve working through an array or list of items (we sometimes say ‘iterating over a list’ or even ‘enumerating’) - whether that be a list of numbers, strings, or user-defined objects such as a list of Students. Others will involve iterating over the characters in a string, though, as you probably know, in many procedural languages a string is treated, ‘under the covers’, as just a special case of an array, or list, of characters.

Now *functional lists* have an inherently recursive *structure*, which means that they fit very well with recursive *functions*. Let’s write a function, this time starting in Haskell, that will find the sum of the squares of a list of numbers. We’ll start with the type signature:

```
sumSquares :: [Float] -> Float
```

Here, the square brackets around the first type specify that the function will take a list, where each member is a `Float`, and it will return a single `Float`. Thinking recursively, the sum will be the value of the *head* of the list, plus the sum of the *tail* of this list, something like this:

```
sumSquares l = (head l) * (head l) + sumSquares (tail l)
```

(Note: the brackets around `head l` are not strictly needed, here, but have been added for readability). However, if we were to define the function using the code above and then calling it with, for example:

```
sumSquares [3,5,1,18]
```

then the system will throw an error because eventually the function tries to obtain the head of an empty list. We have forgotten to specify an exit condition. Remember that, in each recursion, the size of the list being summed gets shorter, so we need to watch for the condition where the function is being passed a list with just a single element. We can use the `length` function to check this. Here's a new version of the function, again using guards, with an exit condition when the list gets down to 1 element.

```
sumSquares :: [Float] -> Float
sumSquares l
  | length l == 1 = (head l) * (head l)
  | otherwise = (head l) * (head l) + sumSquares (tail l)
```

Notice that we have some repeated code, though. Previously we have managed to eliminate the duplication with a `let...in` construct (see Chapter 4). Unfortunately, the re-usable expression is available only for use within the single expression that follows the `in` keyword. If we want a sub-expression to be re-usable within multiple guards, we need to use a `where` clause, which comes after all the guards. We can now write this:

```
sumSquares :: [Float] -> Float
sumSquares l
  | length l == 1 = headSq
  | otherwise = headSq + sumSquares (tail l)
  where headSq = (head l) * (head l)
```

Note that in the latter example especially, indentation is very important - if you don't align the multiple `where` clauses correctly you might get an obscure syntax error.

Many newcomers to Haskell understandably find the similarity between `let` and `where` confusing, but here's the distinction:

- With the `let` keyword the re-usable sub-expression is defined at the start, and is only available for use in the single expression that follows the `in` keyword. `let` does not work with guards, which, by definition, contain multiple expressions.
- `where` is for use with guards. It is specified at the end of all the guards, but is available for use within any of the guard expressions.

Exercise 24

Enter the function into repl.it and then use the `sumSquare` function to calculate the sum of the list of these numbers: `8,19` and `2`, recording the result.

Now call the function passing in an empty list, recording the result.

To prevent the error, we can add another guard. Paste a snippet showing your modified code and the new result.

Chapter 8: Case study – Merge Sort

Let's now combine several of the ideas introduced in previous chapters, in order to write a more powerful function, to sort a list. Hopefully you have already studied sorting and are aware that there are many possible algorithms. Any sorting algorithm may be implemented using FP, just as it can be using procedural programming. We will pick the Merge Sort both because it is one of the more sophisticated algorithms, and because the FP solution is particularly elegant.

Rather than try to solve the whole problem in one go, let's start with the ability to merge two *already sorted* functional lists - as we know we are going to need this. (This technique, of solving one part of the problem to begin with, is a good practice in programming).

```
merge :: [String] -> [String] -> [String]
merge a b
  | null a = b
  | null b = a
  | head a <= head b = head a : merge (tail a) b
  | otherwise = head b : merge a (tail b)
```

The first two guards cover the case where either of the input lists is empty. (This could occur at the outset, but is more likely to occur when calling the function recursively on successively smaller lists):

- If the first list is empty, then the result of the merge is just the second list.
- (reverse of the case above)

The second two guards then cover the 'normal' case where neither of the lists is empty. They compare the heads of the two (remaining) lists:

- If the head of the first list should be before the head of the second list, then create a new list using that as the head, merging the *remaining* parts of the two lists to form the tail.
- (reverse of the case above)

Exercise 25

Enter the function above into repl.it, and use it to merge two lists of at least three strings. Capture a screenshot demonstrating this.

Before moving on let's just explore an alternative way to write this merge function, using another powerful capability of Haskell: 'pattern matching'.

```
merge :: [String] -> [String] -> [String]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
  | x <= y = x:(merge xs (y:ys))
  | otherwise = y:(merge (x:xs) ys)
```

Notice that this time we have written three separate versions of the merge function, but where the conditions are handled by the different ways of writing the arguments. Each of the three conforms to the same type signature, so that doesn't need to be repeated. And each deals with a separate case:

- The 'edge' case where the second argument is an empty list

- The 'edge' case where the first argument is an empty list

We could have continued to use the parameter names `a` and `b`, but where pattern matching is used, the use of `xs` (and `ys`, `zs` if we need more variables) is a widely-adopted convention; the reason becomes clearer when we look at the third implementation of `merge`:

- The 'normal' case where neither list is empty.

In this case, the two input arguments have become `x:xs` and `y:ys` (surrounded by brackets to avoid ambiguity). Looking at the first argument, Haskell will 'match' the format to the structure of the list, so it will put the head of the list into `x`, and the tail of the list into `xs` - and similarly for the second argument into `y` and `ys`.

This allows us to write the third version of `merge` very succinctly. You might think that the gain is pretty small compared to the cost of learning a new technique and, in this simple example, we'd have to agree. But in more complex examples the gain becomes more significant.

Before moving on, validate that the 'pattern matching' implementation produces the same result as the previous implementation.

Now let's pick off another part of the merge-sort problem: breaking a list into two 'halves'. (For a list with an odd number of members, one 'half' will be one item longer than the other, it doesn't matter which is the longer, provided that the two 'halves' make up the whole list exactly). We can make use of the `take` and `drop` functions that we used in Chapter 6.

We will need to divide the length of the list by 2. The ordinary division operator (`/`) doesn't work on an integer (which the `length` function will return) so we need to use the ready-made `div` function instead. Here's a function to find the left 'half' of a list:

```
lefthalf :: [String] -> [String]
lefthalf a = take (div(length a) 2) a
```

Exercise 26

Try out this function on a couple of lists of strings, one with an even number of members, and one with an odd number. Capture the results.

However, this is a situation where the use of the 'prefix notation' makes a simple expression harder to read. So Haskell offers another piece of 'syntactic sugar' - the 'backtick' or ```. (You need to find this on your keyboard - it is *not* an apostrophe or single-quotation mark). If a function takes exactly two arguments, you can surround the function name with two backticks and then use it in 'infix notation' (as would be the case when using `/` for dividing fractional numbers). Here's the same function re-written, and you'll probably agree that it is easier to read:

```
lefthalf :: [String] -> [String]
lefthalf a = take (length a `div` 2) a
```

Exercise 27

Using the infix notation again, write a function that will find the *right*-half of a list. Test this and show that your `lefthalf` and `righthalf` functions will, between them, cover all the members of a list with an odd number of members. Include your code for the `righthalf` function.

Now that we've picked off two core parts of the problem, we can write the whole merge sort function.

```
mergesort :: [String] -> [String]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge (mergesort (lefthalf xs)) (mergesort (righthalf xs))
```

Here we are using pattern matching again.

- The first version handles an empty list – the sorted version of which is ... another empty list! (You can construct a new empty list, when needed, just by using [])
- The second version of the `mergesort` function matches the case where the input list contains exactly one element (labelled `x` for convention only).
- The third version picks up the main case where the list has more than one element. We could have matched this `x:xs`, but as we don't need to use the head and tail separately in the *immediate* implementation, there is no point, so we use just `xs`, again by convention.

Exercise 28

Whichever way you have been working so far, copy the four functions created in this chapter into a single file (not forgetting that you will need the default `main` function) and then invoke `mergesort` from the console on a list of at least seven unsorted strings, to show that it works. Capture both your complete code and the results of calling the function in the console.

Chapter 9: Introducing higher order functions

Suppose we were given a list of numbers and we needed to calculate the square of each number in that list – returning the results as another list. Using the type `Float` for flexibility, the type signature in Haskell will look like this.

```
square :: [Float] -> [Float]
```

When writing a recursive function in a procedural programming language, many programmers leave the handling of edge conditions to last - and can all-too-easily forget them. In Haskell, pattern matching is such an elegant way to handle the edge conditions that it is easiest to write them first. So, for an empty list or a list containing exactly one element we can implement the function thus:

```
square [] = []  
square [x] = [x*x]
```

For the rest we must use recursion, squaring the head and then prepending it to the squared version of the tail. (We will use pattern matching again to extract the head and tail into `x` and `xs`.)

```
square (x:xs) = x*x : square xs
```

Exercise 29

Put the elements of code together, and test the function with a list of at least 3 different numbers. Paste a screen-snippet that shows it working.

We could use a similar pattern to write a function that would double each number in the list, or to find the square root of each number. However, it would be nicer if we could somehow generalise the capability, and this is where FP comes to the fore, because, as stated in Chapter 1, ‘functions are treated as first-class objects’. This means that we can pass a *function* as an argument into another function. A function that *takes in* another function as one of its arguments, and/or *returns* a function as its result, is known as a ‘higher-order function’.

Higher order functions in mathematics

In mathematics, a ‘higher order function’ is a function that is applied to another function rather than to simple values. One of the clearest examples of this is ‘differentiation’ in calculus. For example, we might define an ordinary function, as:

$$y(x) = x^2 - 5x - 6$$

We can then apply the differentiation function, d/dx to the *function* y (this is *not* the same thing as applying it to a specific value that has been returned by y). In this example the result returned from the ‘differentiation’ is itself a function, derived from the function passed as an argument, but not the same. So:

$$d/dx (y) = 2x - 5$$

The meaning of higher order functions in FP deliberately matches the meaning in mathematics, although FP is not just applicable to problems that are obviously mathematical in nature.

Let’s go straight to an implementation of this in Haskell defining two functions:

- A simple `double` function that simply returns the input value multiplied by 2.
- An `apply` function that takes in a function (passed in as the variable `f`) that is to be applied to each member of a specified list, causing the `apply` function to create and return another list.

```
double :: Float -> Float
double x = 2*x

apply :: (Float -> Float) -> [Float] -> [Float]
apply f [] = []
apply f [x] = [f x]
apply f (x:xs) = f x : apply f xs
```

Exercise 30

Enter the code above into a file in [repl.it](#) (not forgetting to add a default `main` function) and run the code. Then in the console invoke the `apply` function with the following expressions, recording the result (note that `sqrt` is a ready-made function in Haskell):

```
apply double [3,4,5]
apply sqrt [3,4,5]
apply (*7) [3,4,5]
```

In the first case we ‘applied’ the function `double`, which we defined in code, to each member of the list; in the second case we applied the in-built `sqrt` function; and in the third case we defined a function *on-the-fly* (we would normally say ‘in-line’ or even ‘anonymously’), to multiply each member of the list by 7. Notice that in the last case we had to use the brackets to show the limits of the function.

Looking again at the code for `apply`, we can see that:

- The three implementations of the `apply` define the second parameter as `f` (though we could have chosen any valid parameter name), to accept the function to be applied to the list members.
- In the type signature for `apply`, we can see that the second argument has a type definition of `(Float -> Float)`. This means that this second argument is itself a function that takes a (single) `Float` as an argument and returns a (single) `Float` as its result. Notice that this requirement is fulfilled by the type signature of the `double` function that we defined, and by the `sqrt`, and by the ‘anonymous’ multiply-by-seven function – `(*7)` – that we used subsequently.
- The passed-in function, `f`, gets applied to a single value in the third version of `apply`, specifically as highlighted here:
`apply f (x:xs) = f x : apply f xs`

Generalising the Merge Sort function

Let’s now look at a very useful example. In the previous chapter we created a `mergesort` function, which sorts a list of strings into alphabetical order. However, passing a function as a parameter would allow us to ‘generalise’ `mergesort`: such that it could sort a list in reverse alphabetical order, or by the length of the strings, or by the number of vowels each contains ...

The code below highlights the changes that must be made to the `mergesort` function from the previous chapter. We have defined a second parameter `p` (short for ‘precedes’) as a *function* that determines whether a given string should precede another (i.e. be placed in front of it in the sorted list). We can see from the modified type signature for `merge` that it requires, as its second argument, a function that takes two `Strings` as arguments and returns a `Bool` – where `true` indicates that the first string should precede the second string in the sorted list, and `false` indicates the reverse.

```

lefthalf :: [String] -> [String]
lefthalf a = take (length a `div` 2) a

righthalf :: [String] -> [String]
righthalf a = drop (length a `div` 2) a

merge :: [String] -> [String] -> (String -> String -> Bool) -> [String]
merge xs [] p = xs
merge [] ys p = ys
merge (x:xs) (y:ys) p
  | p x y = x:(merge xs (y:ys) p)
  | otherwise = y:(merge (x:xs) ys p)

mergesort :: [String] -> (String -> String -> Bool) -> [String]
mergesort [] p = []
mergesort [x] p = [x]
mergesort xs p = merge (mergesort (lefthalf xs) p) (mergesort (righthalf
xs) p) p

```

Notice that we have similarly modified the type signature of the merge function, called from mergesort when the latter has two sorted sub-lists that need to be merged into one, and it is within merge that p is finally applied in this line:

```
| p x y = x:(merge xs (y:ys) p)
```

which may be read as ‘if x precedes y (according to the definition of p) then put x at the front of the list, followed by a merge of the remainder of the two lists’.

Notice also that because both merge and mergeSort are called recursively, we must pass p on into each call.

Exercise 31

Paste the complete new code into a file in repl.it, adding a default main function. Then in the console use the function to sort a list of names two different ways using these calls, noting the (highlighted) difference between them.

```
mergesort ["Thomasina", "Richa", "Harriet", "Thomas", "Richard", "Harry"] (>)
mergesort ["Thomasina", "Richa", "Harriet", "Thomas", "Richard", "Harry"] (<)
```

Then define a function:

```
byLength :: String -> String -> Bool
byLength a b = length a < length b
```

Now call mergesort again (on the original list), but this time passing byLength instead of (>) as the function i.e.:

```
mergesort ["Thomasina", "Richa", "Harriet", "Thomas", "Richard", "Harry"]
byLength
```

Capture the result.

We can generalise our mergesort function further, to sort lists of any type: integers, floats, even user-defined types (which are discussed in Chapter 13).

Exercise 32

In the type signatures for each of the functions that make up our mergesort capability above (but not including the signature of byLength function), change every use of the word String to use just the letter a instead, for example:

```
lefthalf :: [a] -> [a]
```

This example may be read as ‘left half takes a single parameter of a list of *any* type, and returns a list of that same type.

Run the program, and in the console, call mergesort, passing a list of integers as the first argument, and (<) or (>) as the second. Paste a screen snippet showing your results.

Show that the function still works for a list of strings as before, sorted alphabetically, and using byLength.

What happens if you call mergeSort with a list of *numbers* and try to use byLength as the function?

What happens if you try to change the type signature of `byLength` to use any type (`a`) rather than `String`?

Although we have now implemented a sophisticated higher-order sorting function from scratch, it won't surprise you to learn that Haskell has several ready-made sorting functions. But the latter are built on the same principles.

VB also supports the passing of functions as arguments. Here's an implementation of the general-purpose `MergeSort` function, with its supporting functions. It uses the **MetalUp.FunctionalLibrary**, both for the `FList<T>` type and various ready-made functions such as `Take` and `Drop` that mimic the behaviour of the similarly-named functions in Haskell. The highlights are to draw your attention to certain new coding constructs that will be explained underneath:

```
Public Function MergeSort(Of T)(list As FList(Of T), p As Func(Of T, T, Boolean)) As FList(Of T)
    Return If(Length(list) < 2, list, Merge(MergeSort(LeftHalf(list), p), MergeSort(RightHalf(list), p), p))
End Function

Public Function Merge(Of T)(a As FList(Of T), b As FList(Of T), p As Func(Of T, T, Boolean)) As FList(Of T)
    Return If(IsEmpty(a),
        b,
        If(IsEmpty(b),
            a,
            If(p(Head(a), Head(b)),
                NewFList(Head(a), Merge(Tail(a), b, p)),
                NewFList(Head(b), Merge(a, Tail(b), p))))))
End Function

Public Function LeftHalf(Of T)(list As FList(Of T)) As FList(Of T)
    Return Drop(Length(list) / 2, list)
End Function

Public Function RightHalf(Of T)(list As FList(Of T)) As FList(Of T)
    Return Take(Length(list) / 2, list)
End Function
```

Our VB code is starting to look a lot like Haskell. As in the Haskell version, the `MergeSort` and `Merge` functions now define in a function parameter (highlighted above) named `p` (for 'precedes'), clearly indicated by the word `Func`. (`Func` is a keyword in VB, not a name we have chosen here). This `Func` type definition also specifies, in a manner somewhat like the type signature of Haskell, that the function must take in two parameters of type `T`, and return a `Boolean`. `T` can be any type, but it must be the same as the type specified for the `FList` parameters.

We can then write implementations of the function to determine precedence that we have coded elsewhere. For example:

```
Private Function Alphabetical(ByVal s1 As String, ByVal s2 As String) As Boolean
    Return String.Compare(s2, s1) > 0
End Function

Private Function ByLength(ByVal s1 As String, ByVal s2 As String) As Boolean
    Return s2.Length > s1.Length
End Function
```

And then use these when calling the functions, for example in a Main function:

```
Sub Main()
    Dim list = NewFList("Flag", "Nest", "Cup", "Burg", "Yacht", "Next")
    Dim sorted = MergeSort(list, AddressOf Alphabetical)
    'VB requires AddressOf when passing a function, rather than a value or variable
    Console.WriteLine(sorted)
    Console.ReadKey()
End Sub
```

Exercise 33

In a VB console program, into which you have already installed the **MetalUp.FunctionalLibrary**, define the modified VB MergeSort and associated functions, plus the Main function above. Run the program and capture the result.

Change the Main function only, such that the call to MergeSort now uses the ByLength function to determine the ordering. Capture the result.

Chapter 10: Map, Filter, Reduce

In the previous chapter we saw how to write higher-order functions, which take another function as one or more of its arguments. Now we are going to use three powerful ready-made higher-order functions that all work on lists, and which are referred to *generically* as ‘map, filter, reduce’.

Generic name	Generic description	Haskell function
Map	Creates a list of new items where each item is derived from the corresponding item in the input list.	map
Filter	Produces a new list containing only those items from the input that match specified criteria.	filter
Reduce	Applies a function cumulatively to each member of the list, resulting in a single value.	fold...

However, these same capabilities are implemented in different languages with different function names. In Haskell, as tabulated below, the function names for Map and Filter are `map` and `filter`; the Reduce capability is referred to as ‘folding’ and there are several functions that implement it, with names all beginning with `fold`.

Does this remind of SQL?

SQL’s `where` clause acts somewhat like Filter, and `select` somewhat like Map. SQL does not have a single operator that performs Reduce, but several of the in-built SQL functions, such as `Sum`, `Average`, and `Count`, are examples of reducing multiple rows to a single value.

Does this mean that SQL is a functional programming language? It is important to understand that SQL is not a ‘Turing complete’ programming language, and therefore cannot be used to solve *all* computable problems - it is targeted at the specific task of interfacing with a database. However, SQL is a ‘declarative’ programming language, and pure FP languages are also declarative in their nature, so it is not surprising that there are some similarities.

Map

The generic ‘Map’ capability creates a list of new items where each item is derived from the corresponding item in the input list. The new item might be a sub-part extracted from the original, or it might be a new value altogether, in some way based on, or associated with the original.

You might be thinking that the description for the Map capability above sounds like that of the `apply` function that we wrote in the last chapter. In fact, they are identical: you just wrote your own version of the Map capability - but at least you now understand how it works internally. So, as before, we can invoke Haskell’s `map` by passing in an in-built Haskell function, or one that we have defined ourselves. We can also define a function inside another function definition, for ‘private’ use only, by means of Haskell’s `where` keyword. These are all illustrated in the exercises below:

Exercise 34

In the **repl.it** *console* enter the following and record the results:

```
map (* 2) [7,1,64,9]
```

```
square a = a * a
map square [7,1,64,9]
```

```
inverses xs = map inv xs where inv x = 1 / x
inverses [7,1,64,9]
```

Following the pattern used in the third example (`inverses`), write a function named `areas` that takes in a list of numbers where each is the circle of a radius, and returns a list of the corresponding areas, calculated as πr^2 . (In Haskell, `pi` will return the value of π). Apply this function to the list `7,1,64,9` (as above), and capture a screenshot showing both your function and the result of applying it.

Filter

'Filter', when applied to a list, produces a new list containing only those items from the original that match specified criteria. The criteria are specified in the form of a function that, given a single item, returns `True` or `False` to indicate whether that item should be included or not.

Here are some simple examples. The second one uses the in-built `mod` function (modulus) and, since this takes two arguments, we have again used backticks to use this function in infix to make the code easier to read:

Exercise 35

In the **repl.it** *console* enter the following and record the results:

```
filter (>5) [7,1,64,9]
```

```
iseven n = n `mod` 2 == 0
filter iseven [7,2,33,4,5,6,8]
```

Now write your own function that will detect *odd* numbers and filter the same list as shown above using this new function, to return only the odd numbers from the original list. Capture a screenshot showing your code and the result of the application.

Lambda: an anonymous function defined in line

So far, the functions that we have passed in as an argument to a higher-order function have been 'named functions' - either ready-made functions (or operators like `+`, which are simple two-argument functions), or user-defined functions such as `iseven`, above.

Another technique, however, is to use a 'lambda', which is a way of declaring a function 'in-line' - i.e. just where you need it - and 'anonymously' - i.e. without giving it a name. We tend to do this if the function serves only one purpose and we have no need to re-use the same function elsewhere. (There are some other situations where lambdas have an advantage, which you will learn if you study FP in more depth).

For example, suppose we wanted to filter a list of numbers to find only those numbers that were divisible by 3. We could define a function `isDivisibleBy3`, but that might seem overkill. So instead we can write:

```
filter (\a -> a `mod` 3 == 0) [9,20,45,111]
```

The highlighted code is the lambda - contained within brackets to avoid ambiguity. This lambda may be verbalised as:

'given any value `a`, return (or 'yield') the result of evaluating `a `mod` 3 == 0`' (the latter expression returns true if `a` is divisible by 3). This is like an unnamed function that takes in a single argument `a` and returns a result calculated from `a`.

Exercise 36

In the **repl.it** console evaluate the expression and capture the result.

```
filter (\a -> a `mod` 3 == 0) [9,20,45,111]
```

Now modify the lambda so that the filter returns only those members of the list that are greater than 10 *and* less than 100. Capture a screenshot that shows your code and the result of it being evaluated on the list shown above.

Lambdas are especially useful for filtering, as above, but can be used to define the function argument for any higher-order function.

VB also supports lambdas. The equivalent to the Haskell lambda shown above would be:

```
Function(a) a Mod 2
```

'The VB syntax for a lambda is like an 'anonymous' function i.e. a Function that has no function name and is defined in-line, where it is needed.'

We will see how these are used in real VB examples, shortly.

Reduce (or Fold)

The generic Reduce capability applies a function cumulatively to each member of the list, resulting in a single value. *Simple* examples of how you might use this capability include:

- Calculating the sum of items in a list
- Finding the product of items in a list (my multiplying them together)
- Finding the largest or smallest item in a list

Using Haskell's foldr

As shown in the table at the start of this chapter, Haskell's term for the generic Reduce capability is fold, but there are several different variants, starting with `foldl` and `foldr`, verbalised as 'fold left' and 'fold right'. In this chapter we are going to use just `foldr`, which is enough to understand the principle of 'folding'. (For those that want to go a little deeper, Chapter 14 explores the difference between `foldl` and `foldr`. And for those that want to understand *all* the variants ... there are several books on Haskell programming recommended in Appendices – Further Reading.)

`foldr` (like all the variants) takes three arguments, in this order:

- The function to be applied, cumulatively, to the members of the list.

- The starting value to which the function will be applied, when called on the first member of the list. We should avoid the trap of saying that this value will be ‘modified’ with each call, because FP never modifies a value. Instead, each time the function is called on a member of the list, the starting value will be replaced by the result from the call on the previous member.
- The list to be ‘reduced’.

Exercise 37

We’ll start with by applying the addition and multiplication functions to a list of numbers. Evaluate these two expressions in the **repl.it console** and capture the results.

```
foldr (+) 0 [1,2,4,8,16]
```

```
foldr (*) 1 [1,2,4,8,16]
```

What happens if you change the starting value from 0 to 1 in the *first* expression above? Why?

What happens if you change the starting value from 1 to 0 in the *second* expression above? Why?

Now try passing the max function into the foldr (with a starting value of 0). It should return 16 from the list above.

We need to test that this really is finding the maximum, and not just returning the last item! So, change the values in the list to 17, 23, 4, 8, 16 and capture a screenshot showing your application of this function to a new list of values, with the result.

And run this same test example, but this time passing in the min function also. You might be surprised by the result! What is the reason for it? How could you fix it?

Haskell provides ready-made sum and product functions that do the work of our first two examples above. But it is important to understand that these ready-made functions are, themselves, built on the generic fold functions. In fact, most ready-made Haskell functions are written in Haskell themselves, and built out of a small set of very powerful abstract capabilities.

Using Map, Filter, Reduce together

It is possible, indeed commonplace, to use more than one of these generic higher-order functions together.

Exercise 38

Using all three of the higher-order functions introduced in this chapter – map, filter & foldr – (though not necessarily in that order), define a single expression that will find the sum of the squares of all the *positive* values in the input list: 3.2, -6, 1.5, 9.0, -7, 4.4. Capture a screenshot showing your code and the result.

Map, Filter, Reduce in .NET

The **MetalUp.FunctionalLibrary** contains ready-made functions Map, Filter, and FoldR, that mimic the Haskell functions of the same name, for use in VB code. They work with the FLIST type, which we have already explored.

As with Haskell, the function passed in as an argument to these higher order functions may be a named function, or a lambda. The following code uses these library functions to mimic the various Haskell examples from this chapter.

As always, when using the **MetalUp.FunctionalLibrary** you will need these Imports statements at the top of your code file:


```
Imports MetalUp.FunctionalLibrary
Imports MetalUp.FunctionalLibrary.FList
```

Map

The highlighting is to draw attention to the most important parts of the code.

```
Sub Main()
    Console.WriteLine(Map(Function(a) a * 2, NewFList(7, 1, 64, 9)))
    Console.WriteLine(Map(AddressOf Square, NewFList(7, 1, 64, 9)))
    Console.WriteLine(Inverses(NewFList(7.0, 1.0, 64.0, 9.0)))
    Console.ReadKey()
End Sub

Function Square(ByVal a As Integer) As Integer
    Return a * a
End Function

Function Inverses(ByVal xs As FList(Of Double)) As FList(Of Double)
    Return Map(AddressOf Inv, xs)
End Function

Function Inv(ByVal x As Double) As Double
    Return 1 / x
End Function
```

Filter

```

Sub Main()
    Console.WriteLine(Filter(Function(x) x > 5, NewFList(7, 1, 64, 9)))
    Console.WriteLine(Filter(AddressOf IsEven, NewFList(7, 2, 33, 4, 5, 6,
8)))

    Console.WriteLine(Filter(AddressOf IsOdd, NewFList(7, 2, 33, 4, 5, 6,
8)))

    Console.WriteLine(Filter(Function(x) x Mod 3 = 0, NewFList(9, 20, 45,
111)))

    Console.WriteLine(Filter(Function(x) x > 10 AndAlso x < 100,
NewFList(9, 20, 45, 111)))

    Console.ReadKey()
End Sub

Function IsEven(ByVal n As Integer) As Boolean
    Return n Mod 2 = 0
End Function

Function IsOdd(ByVal n As Integer) As Boolean
    Return n Mod 2 = 1
End Function

```

Reduce (FoldR)

```

'Sum of members:
Console.WriteLine(FoldR(Function(x, y) x + y, 0, NewFList(1, 2, 4, 8, 6)))
'For FoldR, the lambda (or a named function), takes two arguments:
'the first, x, is a value from the list,
'the second, y, is the accumulating value.
'Here, both those types are the same (int), but they needn't be.

'Product of members
Console.WriteLine(FoldR(Function(x, y) x * y, 1, NewFList(1, 2, 4, 8, 6)))

'Maximum
Console.WriteLine(FoldR(Function(x, y) If(x > y, x, y), 0, NewFList(17, 3,
4, 8, 16)))
'Note the use of the ternary operator in the lambda, to return whichever
Is the
'greater - the New value Or the accumulator.

'Minimum
Console.WriteLine(FoldR(Function(x, y) If(x < y, x, y), Integer.MaxValue,
NewFList(17, 23, 4, 8, 16)))
'Integer.MaxValue Is an effective starting value for minimum

```

Chapter 11: A more formal approach

In this Chapter we will take a more formal look at functions from a mathematical perspective, introducing some terminology, and some new ideas.

Domain and co-domain

A function generates a resulting value from its input arguments. If we consider a function that takes a single argument and returns a single value, we may represent it formally in this way:

f: A → B

which may be verbalised as ‘the function **f** takes in an argument of type **A** and returns a result of type **B**’. We also say that for this function, **A** is the ‘domain’ of the function, and **B** is the ‘co-domain’.

Both the domain and the co-domain are ‘sets’ of possible values, with a defined type in each case. The type of the domain may be the same as the type of the co-domain, or it may be different. In *most cases*, though, even where the type of the domain and co-domain are the same, the *sets of possible values* will not necessarily be the same. Think of a function that returns the double of an integer argument: the domain and co-domain have the same type (integer) but the returned values will be from the set of even numbers, whereas the argument may be an odd or even number.

The types of the domain and co-domain do not have to be simple single-value types: they could be data structures such as lists, tuples, or user-defined types such as Student, or Missile, each instance of which has multiple properties.

Functions with multiple arguments

Surprisingly perhaps, the formal position is that all proper functions take a single argument. Yet, throughout this book we have been writing functions that *appear* to take more than one argument, starting with the hypotenuse function that we looked at back in Chapter 3. How do we resolve this conundrum? There are two ways of thinking about it.

The first is to consider the argument (in the case of hypotenuse) as a single *pair* of numbers. So the domain is the set of all possible pairs of real numbers (and the co-domain is the set of real numbers).

The second approach is to consider that a function which (like hypotenuse) *appears* to have two arguments, is made up of two functions, each of which takes one argument. One function takes in a single argument (one of the two sides in our example) and returns a second function, not immediately visible to the programmer, which then takes the other argument and returns the result. This concept, or translating a function with multiple arguments into the successive application of single-argument functions is known as ‘currying’. (As mentioned in Chapter 1, currying takes its name from the mathematician that formalised the principle: Haskell Curry, after whom the Haskell programming language is also named.)

Not for the first time, perhaps, you might be feeling a bit like the victim of some sleight-of-hand conjuring trick! But the interesting thing is that you can see this working, in Haskell, through something called ‘partial application’.

Exercise 39

In the console pane of repl.it define the hypotenuse function:

```
hypotenuse b c = sqrt (b*b + c*c)
```

Then call it with a single argument:

```
hypotenuse 3
```

Unsurprisingly perhaps, you will get an error message (partly obscure as usual!) because this expression doesn't evaluate to a simple printable (this is what `Showable` means) value. Capture the part of the error message that gives you the best indication of what is wrong.

Now enter:

```
foo = hypotenuse 3
```

Here we are evaluating `hypotenuse 3` and assigning the result to an identifier, `foo`. You will find that the compiler accepts this, but still returns nothing.

Now enter:

```
foo 4
```

```
foo 7
```

What does the console return in each case?

Partial application of functions

What's happening in the above example is that `hypotenuse 3` is returning a new (invisible) function that takes a single value argument (the second of the two sides). We say that we have 'partially applied' or done a 'partial application of' `hypotenuse`. We've assigned that new function to the identifier `foo` and then applied `foo` with an argument such as 4. Invoking `foo 4` is really the same as writing:

```
(hypotenuse 3) 4
```

You may like to try that last expression in **repl.it** and confirm that it produces the same result. And, as you saw, we can re-use `foo` multiple times.

To be fair, if we really wanted a function that could find the hypotenuse of any right-angled triangle where one of the sides is always 3, we could have defined formally as follows:

```
foo c = hypotenuse 3 c
```

So defining `foo = hypotenuse 3` isn't saving us a great deal of code *in this case*. But it is important to understand the principle, and that functional programming has a very rigorous and consistent set of underpinning principles - and that these can add a lot of value in more complex cases.

You might also like to look back at Exercise 34, and realise that when we passed in `* 2` into the `map` function:

```
map (* 2) [7,1,64,9]
```

this was an example of partial application – we had 'partially applied' the multiplication function by specifying just one of the arguments, the 'multiplier', which would then be applied to each of the members of the list

Type signatures

Back in Chapter 3 we explicitly defined the type signature of our hypotenuse function as:

```
hypotenuse :: Float -> Float -> Float
hypotenuse b c = sqrt (b*b + c*c)
```

We saw that the first two Floats defined the argument types and the final one defined the return type. But how is this consistent with what we have just been learning?

The Haskell syntax for the type signature above is really a more compact and convenient way (professional programmers call this ‘syntactic sugar’) of writing this:

```
hypotenuse :: Float -> (Float -> Float)
hypotenuse b c = sqrt (b*b + c*c)
```

and we saw in Chapter 9 that when we see something of the form (Float -> Float) in a signature it is defining another function. So the syntax of the full type signature above may be verbalised as follows:

‘hypotenuse takes a single argument of type Float, and returns a function, which itself takes in a single argument of type Float and returns a single Float.’

Composition of functions

Back in Chapter 3 when we wrote this VB code:

```
Function Hypotenuse(SideB As Double, SideC As Double) As Double
    Return Math.Sqrt(SideB * SideB + SideC * SideC)
End Function
```

We were ‘in-lining’ the evaluation of the expression SideB * SideB within the application of the Sqrt function, and, hopefully, that idea was not new to you. We can do the same thing in Haskell.

Exercise 40

In the console pane of repl.it, evaluate this expression, using two ready-made functions, and record the result:

```
negate (sqrt 9)
```

What happens if you miss out the brackets? Can you recall why that is?

The error, in the second case, is because negate takes a single argument, and Haskell attempts to use sqrt as that argument. The brackets tell Haskell to evaluate sqrt 9 first.

But we may also *compose* the two functions into one using a dot.

Exercise 41

Record the result of evaluating these two lines:

```
bar = negate.sqrt
bar 9
```

It is important to understand that the dot here, in the context of an FP language, means ‘composed with’ or ‘applied to the result from’. It should not be confused with ‘multiply’ (as is sometimes used

in mathematical notation) nor with the 'dot syntax' used in object-oriented programming (OOP). For this reason, in some books and papers a much larger dot, or even an open circle, is used to show function combination, for example:

$\text{bar} = \text{negate} \bullet \text{sqrt}$

$\text{bar} = \text{negate} \circ \text{sqrt}$

We have used a regular full-stop in this book, because that is also the Haskell syntax.

Book 2 – Delving a little deeper

Chapter 12: Input/Output in Functional Programming

Thus far we have deliberately swept the problem of input/output in FP under the carpet.

Simon Peyton-Jones, the lead developer of the Glasgow Haskell Compiler (GHC) – the most popular Haskell compiler (used by **repl.it**, for example) – encapsulates the problem in this conundrum:

*“A functional program defines a pure function, with no side-effects.”
“The whole point of running a program is have some side-effect.”⁶*

In other words, there’s no point in writing your program from pure functions if you can’t read any data into the program, or output the results to a screen, printer, or to data storage, for example. How does FP address this conundrum?

We’ll start with a simple example, returning to the function to calculate a hypotenuse that we wrote in Chapter 3, but now incorporated into a simple interactive console application. We’ll start by looking at the VB version:

```
Function Hypotenuse(ByVal SideB As Double, ByVal SideC As Double) As Double
    Return Math.Sqrt(SideB * SideB + SideC * SideC)
End Function

Sub Main()
    Console.Write("Enter side B: ")
    Dim b = Console.ReadLine()
    Console.Write("Enter side C: ")
    Dim c = Console.ReadLine()
    Console.Write("Side A is: ")
    Console.WriteLine(Hypotenuse(Convert.ToDouble(b), Convert.ToDouble(c)))
    Console.ReadKey()
End Sub
```

Exercise 42

Copy the code above into a new Console program and run it. Capture a screenshot showing the program being used to calculate the hypotenuse for a right-angled triangle with sides of length 3 and 4.

Here is the equivalent program in Haskell, re-using the version of hypotenuse we wrote earlier, but with a new main function added.


```

hypotenuse :: Float -> Float -> Float
hypotenuse b c = sqrt (b*b + c*c)

main :: IO ()
main = do putStr "Enter side B: "
         b <- getLine
         putStr "Enter side C: "
         c <- getLine
         putStr "Side A is: "
         putStrLn (show (hypotenuse (read b :: Float) (read c :: Float)))

```

And here is a screenshot of the Haskell program being run:

```

*Main> main
Enter side B:3
Enter side C:4
Side A is: 5.0
*Main> |

```

(For this screenshot the program was run in the **WinGHCi** environment. Unfortunately, at the time of writing, there is a bug in the **repl.it** Haskell system that means that the code above will not execute correctly. If you want to be able to run this code, or try your own examples of interactive Haskell programs, you can find details of how to install WinGHCi in Appendices – Installing Software. However, there are no exercises requiring you to write or run Haskell in this chapter.)

Looking at the Haskell `main` function you can probably guess that `putStr` ('put string') is broadly equivalent to VB's `Console.WriteLine`, and `getLine` to `Readline`. The `<-` symbol is new, here, it means 'bind to' and may be thought of as a special form of assignment because we are getting the value from an external source. More importantly, you can see that the overall structure of `main` in the Haskell version is like that of the VB version.

You might even be thinking that the body of `main` looks *remarkably* like procedural programming! That's because Haskell's `do` construct (highlighted above) defines a sequence of expressions to be evaluated in order. 'Wait,' you say, 'I thought sequences of statements weren't permitted in FP?'

Actually, they are *permitted*, but using `do` in pure functions is unnecessary, and considered bad practice, because all pure functions can be implemented as a single expression, as we have seen. However, sequencing is often *necessary* in functions concerned with input/output.

Previously we have not explicitly entered the type signature for the `main` function because Haskell will infer it automatically, but we have done so in the example above to make it visible, and to explain the significance. As you can see from the type signature:

```
main :: IO ()
```

`main` is of type `IO` – which stands for Input/Output. The `()` is just a qualifier on the type `IO` (formally, we say that the type `IO` is 'parameterised') - in this example, the `()` defines an empty tuple, meaning that `main` itself has no need to return any value.

‘Actions’ are functions that deal with the real world

Functions of type IO, whether ready-made or user-defined, are known as ‘actions’ in Haskell. Actions should *not* be thought of as poorly written functions: in Haskell, actions are defined with just as much rigour as pure functions.

Some Computer Scientists even argue that actions are still ‘pure’ functions. They make the point that actions will only produce side effects when they are *executed*, not when they are *evaluated*: an action doesn’t *directly* produce any side effects, they say, under the covers it returns a ‘task’ or ‘delegate’ that the external system can then execute when it is needed (a bit like sending a person who is authorised to do the job when asked). However, to properly understand such arguments you need to know Haskell to a much deeper level than is covered in this book.

For *practical* purposes it is more helpful to think of functions as being *pure*, and actions being *impure* - meaning that actions may generate side effects (not within the program, but in the ‘real world’ outside it) and/or they may depend upon external variables such as input from a keyboard or a data stream.

Actions may call functions, but not *vice versa*

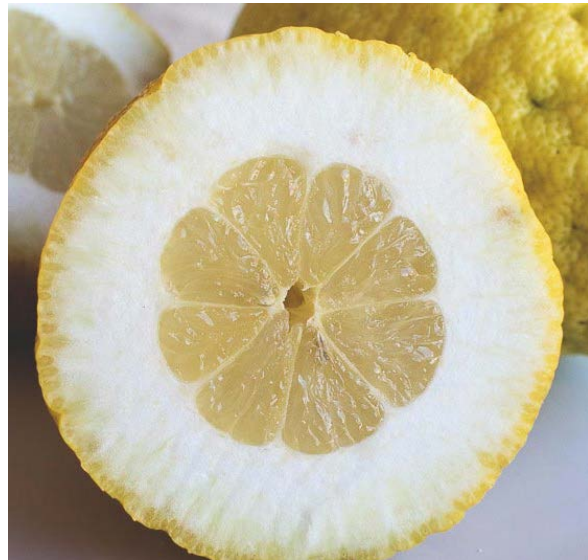
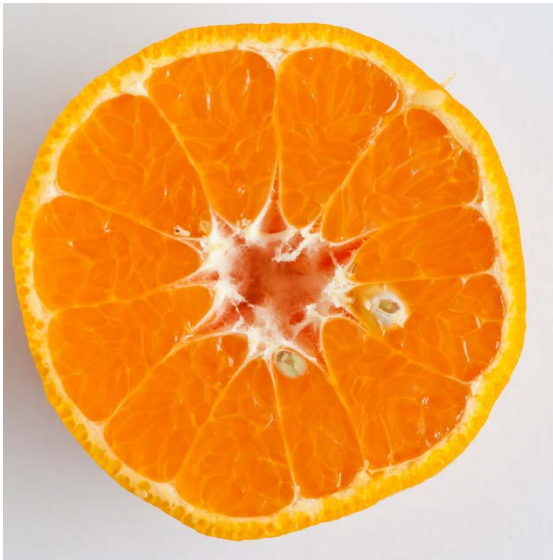
Actions may invoke other actions, and they may invoke (pure) functions. In the example above `main` (action) is calling `hypotenuse` (pure function). But a pure function may *not* invoke an action - doing so would make the original (pure) function *into* an (impure) action. This is an interesting and very powerful idea.

In the VB version of the program, there is nothing to stop you from inserting a `Console.Write` or `Console.Read` into the implementation of the `Hypotenuse` function. But if, in the Haskell version, you were to try using the equivalent capabilities (such as `putStr`, `getLine`) within `hypotenuse` you would get an error because your function would not then match the specified type signature (`Float -> Float -> Float`). You *could* change the type signature of `hypotenuse` to be of type IO, but you would thereby be specifying an *action* not a pure function. We could almost say that IO is a kind of virus that can ‘infect’ other functions that touch it - although not in the sense of malware.

A complex application, written in any programming approach, might entail millions of lines of code, and thousands of functions. In an FP system, you should think of the actions as being at the *edges* of the system, mediating between the pure core functionality and the ‘real world’ (including the hardware of the system). Good design in FP means, amongst other things, keeping the actions to a minimum, concerned only with the input/output, and keeping the whole of your domain logic in pure functions.

Oranges and Lemons?

An application written using FP may be envisaged as a core of pure functions that implement all the ‘domain logic’, surrounded by a layer of Input/Output (IO) functions – called ‘actions’ in Haskell – that may have side effects, and/or depend upon external inputs. Good FP application design aims to keep this outer ‘skin’ as thin as possible, like the juicy mandarin orange shown below (left). The thick-skinned variety of lemon on the right represents a poorer design: either the programmers have let domain logic creep into their IO actions, or functions intended to implement pure domain logic have called IO functions, thereby automatically turning those functions into IO actions, too.



FP makes it more obvious why this design principle is desirable, and FP languages like Haskell help you to implement it, but the principle can and should be applied to all forms of programming: including applications using just procedural, or OOP, code. The broader version of the principle is known as the ‘separation of concerns’ (see panel).

A generic input/output layer

You might just have wondered, when doing the exercises in previous chapters, how the examples that we have been writing in Haskell may be considered pure, when we have seen them produce results on the screen - which is a form of side effect. The answer, we can now see, is that the functions, such as `hypotenuse`, are pure, but we have been executing them within an *environment* – such as **repl.it** or the **WinGHCi** console – that provides a ready-made, generic, input output layer. This generic I/O layer is implemented as a set of actions (impure functions) that call our pure functions, but which are invisible to us as the programmer.

A 'pure' spreadsheet runs inside an 'impure' spreadsheet application

Returning to our running analogy of spreadsheets, if we steer clear of macros and the few impure ready-made functions (see panel on 'Pure functions in a Spreadsheet' in Chapter 2), then the spreadsheet that we create may be considered a pure function. However, any specific spreadsheet, such as the Financial Sample shown below (courtesy of Microsoft), does not *in itself* provide the functionality to input data or view the processed results. That input/output functionality is provided by the spreadsheet *application* – Excel in this case – which has many impure functions (equivalent to Haskell's actions). (The spreadsheet application also acts as the interpreter of our formulas).

	A	B	C	D	E	F
1	Segment	Country	Product	Discount	Units	Manufac
2	Government	Canada	Carretera	None	1618.5	\$ 3.00
3	Government	Germany	Carretera	None	1321	\$ 3.00
4	Midmarket	France	Carretera	None	2178	\$ 3.00
5	Midmarket	Germany	Carretera	None	888	\$ 3.00
6	Midmarket	Mexico	Carretera	None	2470	\$ 3.00
7	Government	Germany	Carretera	None	1513	\$ 3.00
8	Midmarket	Germany	Montana	None	921	\$ 5.00
9	Channel Partners	Canada	Montana	None	2518	\$ 5.00

Separation of concerns

'Separation of concerns' is a design principle that should be applied to all forms of software development, not just FP. In writing a board game program, for example, you should aim to separate code that is responsible for the user interface (UI) from code that is responsible for the game logic (also known as 'domain logic'). (Note: this might not be feasible for a fast action game, however, where the code may need to be structured for optimum speed rather than flexibility.)

Where UI and domain logic can be separated, the UI will, necessarily, call code in the game logic project - we say that the UI is 'dependent upon' the game logic. But the reverse is not true: the game logic should know nothing about the user interface.

A particularly effective way to enforce this - in .NET - is to keep the code for these 'concerns' in separate projects, compiling to two separate 'assemblies' (.dlls), but within a single solution. In Visual Studio you can then, within the UI project, 'add a reference' to the game logic project; but Visual Studio would not then allow you to create a reference from the game logic back to the UI, because that would create a 'circular dependency'.

One huge advantage of adopting this principle, is that it is easier to create an alternative user-interface - for example a graphical user interface (GUI) or a web-based user interface - to replace, or to sit alongside a console UI.

To pick another example, if your game also stores the state of the game, or user profiles and their scores, in files or a database, you should try to keep the code concerned with handling storage separate from both the UI and the game logic. That way you could swap from file-based storage, to database storage, or 'cloud-based' storage, without having to change the game logic or UI code.

If you were choosing to do this in FP, then the tricky part, in FP is ensuring that your pure domain functions never deal directly with the database or storage functions - both are invoked by actions in the UI layer.

Chapter 13: The Haskell type system

Most languages define a set of ready-made types, such as integer, string, and Boolean.

Haskell's set of such types is especially rich, because those types are organised as a 'hierarchy of abstraction', illustrated in the on the right. (The term 'Prelude' in the caption, refers to the standard library of types and functions that comes with the Haskell compiler.)

Understanding all the types shown in this diagram is beyond the scope of this book: there is a list of recommended further resources at the end of this book if you want to proceed further with Haskell. For now, it is worth noting a couple of things:

- The diagram shows both ready-made types (such as `Float`, `Double`, `Int`), and 'classes', which are all in bold format, such as **`Fractional`**, **`Num`**, and **`Show`**. The meaning of 'class' in Haskell, is different to the meaning in VB, where a class can be considered as a template for 'instances' of a type. In Haskell, 'class' refers to an *abstract type*, which is closer, conceptually, to an 'abstract class' in OOP.
- In the diagram, we can see that the `Float`, and `Double` types both have the class **`Floating`**; that **`Floating`** inherits from **`Fractional`**; and **`Fractional`** inherits from **`Num`** (number).

The power of this idea is that if, for example, you wrote a simple function takes one or more numerical values as arguments, you are not forced to specify whether these numerical values must be of type `Float` or `Double`, for example, nor to write separate ('overloaded') versions of the function for each.

We'll explore this by writing a simple mathematical function to calculate the average of two numbers, initially without specifying the type signature, for example:

```
average a + b = (a + b) / 2
```

If we were to then type: (in the **repl.it** console):

```
:t average
```

(which may be verbalised as 'What is the *type* of `average`?') we will get back the type signature that Haskell has *inferred* from the implementation.

```
average :: Fractional a => a -> a -> a
```

This may be verbalised as follows:

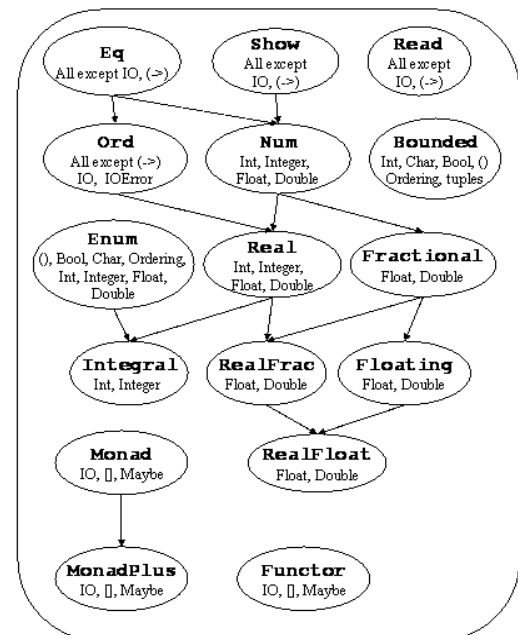


Figure 10-1: The Haskell Prelude hierarchy. Source: <https://www.haskell.org/onlinereport/basic.html>

‘average takes in two values of type `a` and returns a value of type `a`, where `a` can be any type of the class **Fractional**.’ (Note that `a` is lower-case here, because it is a *variable* representing a type, not a type as such).

In other words, if you pass in two `Floats`, you will get back a `Float`; if you pass in two `Doubles` you will get back a `Double`.

Why did the compiler infer the class **Fractional**, and not the class **Num**, which would cope with integers as well? Well, partly, that’s because it knows that the division by 2 may result in a fractional number (where the `a + b` yields an odd number). The return type must be fractional to cope with all cases. Also, although not obvious from the type signature, input arguments that are integers may *always* be ‘coerced’ (automatically converted) into a fractional type: `3` can be safely cast as `3.0`.

This is a powerful capability, which allows functions to be made more general-purpose than they would otherwise be.

Exercise 43

In the **repl.it** console, use `:t` (as above) and record the type signature of each of the following ready-made functions,

```
sqrt
min
mod
```

With reference to Figure 13.1, which of the three functions above can accept the largest number of different ready-made types as input values?

What’s the difference between `Int` and `Integer`?

You might have noticed that the diagram showing the Haskell type hierarchy included (within the **Real** class) both `Int` and `Integer`. What’s the difference between them?

`Int` is a representation of an integer, up to a pre-determined maximum size, which may vary between implementations of Haskell; typically the representation is either 32-bit or 64-bit (the Haskell specification states only that it must be at least 30 bits). This is equivalent to the pattern used in most programming languages: if your functions are likely to generate integer results larger than the maximum, you must guard against ‘overflow’.

`Integer`, by contrast has no pre-defined limit and will never result in an overflow. It is *theoretically* possible that you could get to an Out of Memory error on your computer, but think about how big the numbers would have to be!

You could write a Haskell program to show this by, for example, recursively multiplying by two, and leave it running. But be warned: you will have left school before your computer runs out of memory. In fact, the sun will probably have burnt out – not to mention your computer – before it finishes!

Scientists estimate the number of atoms in the universe at around 10^{80} , give or take a couple of powers of 10. Use a pocket calculator to work out how many bits this would require in binary format?

Approximately how many *bits* of main memory does your computer have (a byte being 8 bits)? Call it *n*. Using a mathematical rule-of-thumb, if you divide *n* by 3.32, that will tell you how many denary *digits* it would take to represent that number.

The downside to using Integer is that even basic arithmetic operations will execute more slowly – Int is the better type to use for most purposes.

User-defined types

Object-oriented languages such as VB allow the programmer to define their own types (known in VB as ‘classes’), such as Student, Missile, or SalesOrder.

Haskell also supports the idea of user defined types. For example, if we want a Student to have properties to represent the first name, last name and age, we can define and use that type in Haskell as shown here:

```
data Student = Student String String Int
ma = Student "Monica" "Ainslie" 17
cb = Student "Charlie" "Beckworth" 16
```

We can also give the properties names, which is much more like we are used to in VB:

```
data Student = Student {firstName :: String, lastName :: String, age ::
Int}
ma = Student {firstName = "Monica", lastName="Ainslie", age=17}
cb = Student {firstName = "Charlie", lastName="Beckworth", age=16}
```

Exercise 44

Enter the second example into **repl.it**, Run, and in the console enter the following expressions, capturing the result:

```
firstName ma
age cb
```

Notice that the syntax for reading the `firstName` property of a student is no different from the syntax for invoking a function named `firstName`, passing it the student (e.g. `ma`) as an argument. That’s because the properties really are functions. Strange as it may seem, in Haskell a data type can be treated as a function, just as a function can be treated as a data type.

FP vs. OOP?

Even if you accept the idea, suggested at the start of this book, that FP is going to become the next ‘dominant paradigm’ in programming, across the board, where does this leave OOP, the current dominant paradigm?

- Is FP *incompatible* with the principles of OOP?
- Does FP *extend* the idea of OOP?
- Does FP suggest that the principles of OOP are *wrong*, or just supersede them?

Unsurprisingly, the truth isn't as simple as a 'yes/no' answer to any of those questions above, but they are all valid questions. Let's explore them by looking back at the key ideas that make up OOP.

Encapsulation

In OOP, an object typically encapsulates properties with methods, in other words data with functionality, or state with behaviour.

Haskell does not support the OOP idea of encapsulated methods, but this is much less of a difference than you might imagine. For example, we can define a function that returns the full name of a Student, derived from the first and last names plus a space, and use it as follows:

```
> fullName s = firstName s ++ " " ++ ( lastName s)
> fullName cb
=> "Charlie Beckworth"
```

Is it obvious, here, whether the `fullName` function is encapsulated on the Student or not? The important thing is that you can invoke the function (that works only on Students) in the same way that you can read a property of the student.

Even in VB there's little difference between writing `FullName` as 'method', encapsulated on the class:

```
Class Student
    Public Property FirstName As String
    Public Property LastName As String
    Public Property Age As Integer

    Public Function FullName() As String
        Return FirstName + " " + LastName
    End Function
End Class
```

And as a free-standing function:

```
Class Student
    Public Property FirstName As String
    Public Property LastName As String
    Public Property Age As Integer
End Class

Module MyFunctions
    Public Function FullName(s As Student) As String
        Return s.FirstName + " " + s.LastName
    End Function
End Module
```

In fact, the code above is the recommended approach if you are seeking to do FP in VB.

'Ah!' you might be saying, 'but that is only because you've made everything public!' This is true. One benefit of encapsulation in OOP is that certain properties, and/or methods may be marked `Private` - accessible only within the object itself. Haskell does not have this concept of public/private. However, one of the *main* reasons for marking members as `private` in OOP is to prevent code outside an object from modifying the object in unintended ways. Remember that in Haskell all data types are, by design, *immutable* - you cannot modify an existing value, you may only make a new one, potentially copied from another one but with specified differences. So we are not worried about unintentional modification of properties.

Inheritance

Another feature of OOP is inheritance: you can choose to make both `Student` and `Teacher` inherit from a common 'super-class', `Person`, which potentially defines properties and/or methods that are common to all the sub-classes (such as the name properties, and methods for sending a communication to that person) and thereby reduces duplication.

It is important to understand that there are really two forms of inheritance in OOP, and most OOP languages support both:

- **Implementation inheritance.** This is where the superclass provides the complete *implementation* of a method to be inherited. A sub-class need only specify its own implementation if it needs to 'override' the implementation provided by the super-class, perhaps to provide a more specialised behaviour.
- **Abstract inheritance.** Here, the superclass defines the 'signature' of a method, but no implementation, thereby requiring any sub-class to provide its own implementation. In VB abstract inheritance may be specified either through the use of 'abstract classes' or through the use of 'interfaces'.

Some school textbooks argue, or imply, that *implementation inheritance* is the most powerful concept in OOP, because it eliminates duplication of code. This is very misleading. There are other ways, equally simple, to ensure that the code need not be duplicated. Most experienced professional OOP developers make only very limited use of implementation inheritance, but consider *abstract* inheritance to be far the more important of the two.

It should be no surprise, then, to learn that Haskell supports abstract inheritance but not implementation inheritance (at least, not in general: there are a few small exceptions). Haskell's approach to abstract inheritance involves classes. A class in Haskell defines one or more functions that must exist for a type to 'belong' to that class of types. (So, again, a 'class' in Haskell is more like an 'abstract class' or an 'interface' in VB).

Polymorphism

Far more important than inheritance, in OOP, is the concept of 'polymorphism' which may be defined as follows:

Where objects of different types define a method with the same signature (name, arguments, & return type), then, even if the implementations of the method are different, it is possible to invoke the method on any objects of those types without having to know the specific type of each object.

Returning to our previous example, having polymorphism means that you can have a single list containing both `Teachers` and `Students`, and provided each has a method `SendEmail`, taking the

same argument types and returning the same type, you can send an email to all of the members of this list without having to know the specific type of each member, and without having to know that, for example, the email might be sent to Teachers via one mail system, and to Students via another.

In VB polymorphism relies on abstract inheritance: in the example above, the list would be of type `Person`, and the abstract class (or interface) `Person` would define the abstract method `SendEmail` that `Teacher`, `Student` and any other sub-type must implement. (Dynamically typed languages such as Python and JavaScript support polymorphism but via a different mechanism).

Haskell's class mechanism allows abstract inheritance, and therefore polymorphism. And the lack of general-purpose implementation inheritance is no shortcoming: if it happens that the implementation of `SendEmail` can be the same for both `Teacher` and `Student`, then you just write a single implementation of the `SendEmail` function that takes a `Person` as its argument.

Warning!

It should be clear from the above that Haskell supports all the mechanisms of OOP that yielded the advantages of that paradigm. It is *possible*, then, to design a program using OOP design principles and then implement it in Haskell, or another pure FP language with equivalent capabilities.

The warning is that this would not be a good approach, any more than it was a good idea, 30 years ago, to design a program according to procedural design principles and then implement it in a pure OOP language such as Smalltalk.

A paradigm shift in programming, as introduced in Chapter 1, is not just a fundamentally new syntax for code, or even just a new way to design a program: it is a new way to *understand* the problem domain in the first place. The best way to improve your understanding of FP is not to try to translate existing code or design into a new language, but to learn how to use the language of functions to see a problem in a new way.

The danger of understanding a new technology in terms of the old one

A cartoon from an American newspaper from the 1940s, when television was still a novelty, showed two men chatting over the garden fence. 'Marvellous invention, this television,' says one, 'If you shut your eyes, it's just like listening to the radio!'

Chapter 14: Folding Left vs. Folding Right

In Chapter 10 we stated that Haskell's term for the generic idea of 'Reduce' was 'fold' and that this was implemented by two functions `foldl` ('fold left') and `foldr` ('fold right'), but we used only the latter. Here we will explore the difference. (There are two further variants, `foldl'` and `foldr'`, but these are not covered here: those wanting further explanation are referred to Appendices – Further Reading.)

Exercise 45

Evaluate these expressions in the `repl.it console` and capture the results.

```
foldl (+) 0 [1,2,4,8,16]
foldr (+) 0 [1,2,4,8,16]
foldl (*) 1 [1,2,4,8,16]
foldr (*) 1 [1,2,4,8,16]
```

Now we'll try subtraction and division. Capture the results.

```
foldl (-) 0 [1,2,4,8,16]
foldr (-) 0 [1,2,4,8,16]
foldl (/) 1 [1,2,4,8,16]
foldr (/) 1 [1,2,4,8,16]
```

You might initially be surprised at the results. See if you can work out on paper how the expression evaluator has got to these results, before going on to read the explanation below.

For the `+` and `*` functions you would have found that the expressions using `foldl` and `foldr` produce the same result, but for the `-` and `/` functions, the `foldl` and `foldr` versions produce *different* results. Before looking at the individual results, why should there be a difference for the second two functions and not for the first two?

The difference between `foldl` and `foldr` *may be thought of* as whether the function is starting from the left of the list, or starting from the right of the list, *and* whether the accumulating value is the first or the second argument in each operation. (Put another way: whether the application of the function is 'left-associated' or 'right-associated').

This is easiest to understand by *expanding* the evaluation of the expression, using just the operator being applied to the values in the list, for example `(-)`, together with brackets.

Exercise 46

The following two expressions, using addition, achieve the same results as the `foldl` and `foldr` examples above. Referring to the description of the difference between `foldl` and `foldr`, above, can you work out which is equivalent to which?

```
1 + ( 2 + (4 + ( 8 + (16 + 0))))
((((0 + 1) + 2) + 4) + 8) + 16
```

Now confirm your answer by evaluating these two expressions, using subtraction this time, in the `repl.it` console and capturing the results.

```
1 - (2 - (4 - ( 8 - (16 - 0))))
((((0 - 1) - 2) - 4) - 8) - 16
```

Looking the expanded evaluations, we can now see why there is no difference between the results of `foldl` and `foldr` when the function is *addition*, because addition, in mathematical terms, is ‘associative’:

```
(1 + 2) - 3 = 6
1 + (2 + 3) = 6.
```

But subtraction is not associative:

```
1 - (2 - 3) = 2
(1 - 2) - 3 = -4.
```

This visualisation also helps to make it clearer why `foldr` is *usually* going to be more efficient than `foldl`: because it aligns better with the head:tail structure of a functional list. This is clearer still if were to write our own implementation, which we will call `myReduce`.

```
myReduce :: (Int -> Int -> Int) -> Int -> [Int] -> Int
myReduce f s [] = s
myReduce f s [x] = f x s
myReduce f s (x:xs) = myReduce f (f x s) xs
```

Notice the ordering of the two arguments `x` and `s`, as highlighted above. Compile the code above and then test it with the following arguments:

Exercise 47

```
myReduce (+) 0 [1,2,4,8,16]
myReduce (-) 0 [1,2,4,8,16]
```

Are these results equivalent to using `foldl` or `foldr` ?

Looking at the code for the `myReduce` function, there are three pattern matching versions. In plain English we can describe what each of the three versions is doing as follows:

`myReduce f s [] = s` If the list is empty, return the starting value, `s`.

`myReduce f s [x] = f x s` If the list contains a single element `x` (i.e. has a head, but an empty tail), then apply the function `f` to that head value, with the starting value, `s`, as the *second* argument and return the result.

`myReduce f s (x:xs) = myReduce f (f x s) xs` Where the list has a head and a non-empty tail, calculate the 'next' starting value by applying `f` to the starting value `s` and the head of the list `x`, then call `myReduce` *recursively* on the tail of the list (third argument), passing the *newly calculated* starting value as the second argument, and the same function `f` as the first argument.

Exercise 48

Trace through, *by hand*, the recursive function calls, showing the exact values of the arguments that it will be called with each time, starting from:

```
myReduce (-) 0 [1,2,4,8,16]
myReduce
myReduce
```

etc. (hint: you should end up with a total of six calls to `myReduce` including the original one).

An equivalent implementation of `foldl`, which we shall call `myReduceL`, looks like this:

```
myReduceL :: (Int -> Int -> Int) -> Int -> [Int] -> Int
myReduceL f s [] = s
myReduceL f s [x] = f s x
myReduceL f s (x:xs) = myReduceL f (f s (last xs)) (init (x:xs))
```

Notice that there are two important differences (highlighted, above) between this and the `myReduce` function that we looked at previously:

- The ordering of arguments `s` and `x`, passed into the function `f` has been reversed. For an associative function `f`, such as `+` or `*`, this makes no difference, but for a non-associative function such as `-` or `/`, this makes a difference.
- In the recursive version (last line), instead of working on the head and tail of the list, is now working on the `last` element of the list (actually the last element of the tail, but that's the same thing) and on the `init` (short for initial part of) the list.

A reminder that `last` is an in-built function to find the last element of a list; `init` returns all the list *except* for the last element. An important point is that these two functions are both more expensive to execute than `head` or `tail` - because both must recurse over the elements of a list.

Exercise 49

Trace through, *by hand*, the recursive function calls, showing the exact values of the arguments that it will be called with each time, starting from:

```
myReduceL (-) 0 [1,2,4,8,16]
myReduceL
myReduceL
```

etc. (hint: you should end up with a total of six calls to `reducer` including the original one).

FoldL and FoldR in the MetalUp.FunctionalLibrary

The code below shows the examples used earlier in VB using `FoldL` and `FoldR` from the **MetalUp.FunctionalLibrary**:

```
Sub Main()  
    Dim list = NewFList(1, 2, 4, 8, 16)  
    Console.WriteLine(FoldL(Function(x, y) x + y, 0, list))  
    Console.WriteLine(FoldR(Function(x, y) x + y, 0, list))  
    Console.WriteLine(FoldL(Function(x, y) x * y, 1, list))  
    Console.WriteLine(FoldR(Function(x, y) x * y, 1, list))  
    Console.WriteLine(FoldL(Function(x, y) x - y, 0, list))  
    Console.WriteLine(FoldR(Function(x, y) x - y, 0, list))  
    Console.WriteLine(FoldL(Function(x, y) x / y, 1, list))  
    Console.WriteLine(FoldR(Function(x, y) x / y, 1, list))  
    Console.ReadKey()  
End Sub
```

Exercise 50

Run the code above in a console program with the **MetalUp.FunctionalLibrary** installed. Capture the output and compare with the results from Exercise 10.3. Which result was different? Can you figure out why before reading on?

The reason for the difference is that VB has inferred the type of the elements in the FList as integer, and the result given is correct for integer division. It is possible to define the type as a Float explicitly, if that is what you want. Or you can simply change all the numbers to include a decimal point.

Exercise 51

Change the numbers in the previous example to floating point by adding `.0` on the end of each one - this includes having starting values of `0.0` and `1.0` in the calls to `FoldL` or `FoldR`. Run the code again and capture the output. Are the results now the same as in exercise 10.3?

Chapter 15: Using LINQ in .NET to emulate Map, Filter, Reduce

In Chapter 10 we used ready-made functions **MetalUp.FunctionalLibrary** to perform the Map, Filter and Reduce/Fold operations on the `FList` type from that library.

However, if you are writing a program in VB, it is useful to know that your language *natively* supports the concepts of Map, Filter and Reduce/Fold, for use on *ordinary* lists, arrays, and other familiar VB data structures (i.e. *without* the requirement to use a list with the head:tail structure).

These functions are provided by the powerful inbuilt capability called LINQ, which stands for ‘Language INtegrated Queries’, and may be thought of as a query language, somewhat like SQL but fully integrated within the VB language and able to operate on objects in memory (as well as on databases, incidentally, via the Microsoft ‘Entity Framework’).

LINQ *is* a form of FP - in fact the development of LINQ was what drove the introduction of first-class functions into .NET.

Most professional programmers writing VB, today, make extensive use of LINQ, whether or not they are deliberately attempting to use FP techniques.

The following table shows the specific LINQ functions that implement the generic ideas of Map, Filter, and Reduce:

Generic name	LINQ function	Description
Map	Select	Creates a list of new items where each is derived from the corresponding item in the original.
Filter	Where	Produces a new list containing only those items from the original that match specified criteria.
Reduce	Aggregate	Applies a function cumulatively to each item in the list, returning a single ‘aggregated’ value. There are also some more specific versions of Aggregate , such as Sum .

We will look at these in a slightly different order than previously: `Where` (filter), `Select` (map), then `Aggregate` (reduce). And we will apply them to conventional VB collections.

Note that for *each* of the examples in the rest of this chapter you will need to make the LINQ functions available by adding this statement at the top of the file:

```
Imports System.Linq
```

In LINQ, the Filter function is provided by ‘where’

The following code shows the use of LINQ's Where function to filter a list down to those values that pass a condition specified in a function, using the same example as we used to illustrate Haskell's filter function earlier:

```
Sub Main()  
    Dim list = New List(Of Integer) From {7, 1, 64, 9}  
    Dim newList = list.Where(Function(x) x > 5)  
  
    For Each item In newList  
        Console.WriteLine(item & " ")  
    Next  
    Console.ReadKey()  
End Sub
```

The Where function is applied to a List, using the familiar 'dot syntax', and then requires a function to be passed as a parameter. In the examples above the function is defined in line (as we learned in Chapter 10, this is also known as a 'lambda'), although we could have defined a function in another part of the code and re-used that function. Here the lambda may be read in English as:

Given any value x, return the result of evaluating 'x > 5'

We could have named x anything, but it will be of type integer because the Where function will have picked this up from the type of the specific List (or array) that it has been applied to.

Exercise 52

Run the code above in a Console application and paste a screen-snippet showing the result.

Now write your own example, defining an *array* (not List) of type string, containing the names Amy, John, Chloe, Pat, Max, Alexander, and then using Where to filter it down to just those names that are four or more characters in length. Paste in your code and a screen-snippet showing the result.

In LINQ the Map function is provided by 'select'

The following code shows the use of LINQ's Select function to create a list of objects that are derived in some way from the original list. Again, we are using the same example as we used to illustrate Haskell's map function earlier:

```
Sub Main()  
    Dim list = New List(Of Integer) From {7, 1, 64, 9}  
    Dim newList = list.Select(Function(x) x * x)  
  
    For Each item In newList  
        Console.WriteLine(item & " ")  
    Next  
    Console.ReadKey()  
End Sub
```

Exercise 53

Run the code above and paste in a screen-snippet to show the results.

Now create a new example that works with the list of first names that you used in the previous exercise, this time using `Select` to return a list of names formatted as all upper case. Paste in your code and a screen snippet showing the result.

Finally, show that you can 'chain' LINQ functions, using the dot syntax, by adding the `Where` clause that you wrote in the previous exercise *onto the end of* the `Select` clause from this one. Paste in only the line of code you changed, plus a screen snippet of the result.

Does it make a difference *in this specific example* if you invoked the `Where` before the `Select`?

In LINQ, the `Reduce` function is provided by 'aggregate'

The following code shows the use of LINQ's `Aggregate` function to reduce a list of objects to a single value by iteratively applying a function to each object in the list, along with a single 'running' value that may be changed each time. Again, we are using the same example as we used to illustrate Haskell's `reduce` function earlier:

```
Sub Main()  
    Dim list = New List(Of Integer) From {1, 2, 4, 8, 16}  
    Dim reduced = list.Aggregate(Function(x, s) s + x)  
    Console.WriteLine(reduced)  
    Console.ReadKey()  
End Sub
```

Exercise 54

Run the code above and paste a screen-snippet showing the result.

Then modify the code such that it produces the *sum of squares* of the original values. Paste in your single modified line of code and a screen-snippet showing the result.

Chapter 16: Ranges

Haskell includes some useful functions for defining ranges of numbers.

Exercise 55

Evaluate these expressions in the **repl.it** console and capture the results.

```
enumFromTo 1 10
```

```
enumFromTo (-5) 5
```

```
enumFromThenTo 3 5 10
```

Use the same pattern that provide a range starting from 10 and counting *down* to 0 in steps of 2.

What is the type of the returned value?

Will the functions work with fractional numbers?

What about characters (the character 'a' is defined in Haskell as 'a')?

In fact, these functions will work with values of any type that has the class Enum (standing for 'enumerable') - which you can see in the type hierarchy diagram at the beginning of Chapter 13.

There is an even simpler syntax for specifying ranges - 'syntactic sugar' (see panel) - to achieve the same thing.

Syntactic sugar

'Syntactic sugar' is where a language provides an alternative, simpler, syntax for writing the same functionality. VB has many examples of syntactic sugar: a simple example is the option to write `a += 1` instead of `a = a + 1`

Exercise 56

Evaluate these expressions in the **repl.it** console and capture the results.

```
[1..10]
```

```
[3,5..10]
```

Now because these ranges are functional lists, you can pass a range into any function that takes a list of the same type, including `head`, `tail`, `map`, `filter`, `fold`...

Exercise 57

Using a range as shown above, write an expression that will return a list of the squares of the numbers 1 to 10. Capture a screenshot showing your expression and the result.

Infinite ranges

And now for the big surprise: ranges in Haskell can be infinite! The following are examples of ranges that go on to infinity (remember that in Haskell, while the type `Int` might be limited to 32 bits, say, `Integer` is limited only by the memory of your machine). The following expressions define infinite ranges:

```
[1..]
```

```
[3,5]
```

Don't type these expressions straight into the **repl.it** console because the compiler will attempt to turn them into a printable list, and the console will just hang, until you hit the stop button (or the **repl.it** server decides to do it for you.)

The expressions above are the syntactic sugar for these underlying functions:

```
enumFrom 1
```

```
enumFromThen 3 5
```

But what use are infinite ranges if they hang the system when you try to use them? Well, surprisingly, you can define an infinite range, apply one or more functions to that range, and *then* ask for the first few results. This *looks*, when you first see it, as though it should not work: you might think the system would hang performing the transformation on the infinite range.

Exercise 58

Evaluate this expression and capture the result:

```
take 7 (map (*2) [1..])
```

Write an expression that starts with an infinite range (from 1) *filters* the range to just perfect squares (those where the square root is an integer) and then returns just the first 10 of those perfect squares. You may declare this function:

```
isInt x = x == fromInteger(round x)
```

and then use `isInt` in your own expression.

Capture a screenshot that shows your code and the result of evaluating it.

The reason this works, when at first glance it looks as though it shouldn't is down to deliberate 'lazy evaluation', again. In other words, Haskell effectively works backwards: it will only enumerate through the infinite range until it has the results it needs. In the first example, the enumeration will

involve the first 7 values. In the second example, however, it will enumerate the first 100 numbers in the range, in order to find the first 10 perfect squares.

Both finite and infinite ranges can be used as an alternative to writing a recursive function when you need to do some iteration. So you shouldn't be surprised to learn that the internal implementation of a range is itself a recursive function.

Using ranges in the MetalUp.FunctionalLibrary

Largely for demonstration purposes, the **MetalUp.FunctionalLibrary** provides functions equivalent to those of Haskell for generating ranges, including infinite ranges, although they work only with integer values, as shown below:

```
Dim a = EnumFromTo(-5, 5)
Dim b = EnumFromThenTo(1, 5, 100)
Dim c = EnumFrom(1)
Dim d = EnumFromThen(0, -1)
```

Now these functions *do* perform a degree of lazy evaluation. The returned list is a sub-type of `FList(Of Integer)` called `Range`. In the example above `a` will have the value `-5` as its head, but the tail of `a` will not be generated until you ask for it. Even when you do ask for it (via the `Tail()` function, or another function that makes use of it) the tail will just be generated as a new `Range`, with a head of `-4` and no tail ... until you ask for that.

If you are interested, you can see how this is implemented by looking at the source code for the **MetalUp.FunctionalLibrary**, which you can find here:

<https://github.com/MetalUp/FunctionalLibrary>

(You'll also find that the library has more functions than are mentioned in this book.)

If you used the `Take()` function to extract the first `n` values from an *infinite* range, this would still work fine. The problem is that there are a lot of circumstances where it won't work: for example trying to apply the `Map` or `Filter` functions to an *infinite* range. In such cases the system will attempt to work through to the end of the infinite range, before proceeding - probably hanging until you get to a `StackOverflowException`.

This point nicely illustrates the fact that while it is now possible to write programs in a pure FP style in VB or another multi-paradigm language, a purpose-designed FP language such as Haskell offers some far more powerful techniques and capabilities. Some of the latter will undoubtedly work their way into future releases of the VB language, but not all.

Chapter 17: Programming exercises

Prime Numbers

Exercise 59

Your task is to write a function that, given an integer, will return `True` if the integer is a prime number and `False` otherwise. You may write the function in Haskell, or in VB - but using a pure FP approach (it is recommended that you make use of the **MetalUp.FunctionalLibrary** in this case, not forgetting the usual `Import` statements needed at the top).

A prime number is a natural number (a positive integer), *greater than 1*, that cannot be formed by multiplying two smaller natural numbers. Another way to express it is that a prime number has the factors only of 1 and itself: 7 is prime because its only factors are 1 and 7; 15 is not prime because it has factors 1,3,5, and 15. The first few primes are thus: 2, 3, 5, 7, 11, 13...

You can tackle the problem from this description, or you may ask your teacher to give you one or more hints on how to approach it. When you think you have a solution, test it thoroughly.

Roman Numeral conversion

Exercise 60

Your task is to write a function that, given a natural number expressed in denary, will return the equivalent value in Roman numerals. It should work for all values in the range 1 to 2100, so, for example, 1066 should return MLXVI, and 2019 return MMXIX.

There are multiple algorithms for determining the Roman numeral, but the recommended approach is as follows:

1. From the input number, deduct the value of the *largest* symbol that is less than or equal to that number (so for 1066 deduct 1000). Add the corresponding symbol to a result string (that starts as an empty string).
2. Repeat step 1 using the remaining denary value as the new input value. So, in the example above, the next step is to look for the largest symbol whose value can be deducted from the number 66.
3. It is strongly recommended that, as well as the symbols M (1000), D (500), C (100), L (50), X (10), V (5), I (1), you also handle the so-called 'subtractive' elements - IV (4), IX (9), XL (40), XC (90), CD (400), CM (900) as single symbols in their own right, but having two characters each.

You may write the function in Haskell, or in VB - but using a pure FP approach (it is recommended that in this case you make use of the **MetalUp.FunctionalLibrary**, not forgetting to include the standard required Imports statements).

You can tackle the problem from this description, or you may ask your teacher to give you one or more further hints on how to approach it. Either way, *test your function thoroughly* when you think it is done.

Appendices

Appendix I: Installing Software

Battleships

The code for Battleships may be downloaded from: <https://github.com/MetalUp/Battleships>. Select the **Download ZIP** from the **Clone or Download** button, and then unzip the file to a suitable location. The unzipped folder contains two separate Visual Studio solutions, for the C# and VB versions of the code.

Ray Tracing

The code for the RayTracer program may be downloaded from: <https://github.com/MetalUp/RayTracing>. Select the **Download ZIP** from the **Clone or Download** button, and then unzip the file to a suitable location. The solution contains three projects. The core domain classes, including Camera, will be found in the **RayTracerModel** project.

Repl.it for Haskell

To run Haskell just enter repl.it into your browser's URL field. You can then just click the **+ new repl** button and select Haskell from the list of supported languages, and click **Create repl**. It is not necessary to *sign up* to **repl.it**, but it is recommended as this allows you to save and share work.

The WinGHCi Haskell environment

WinGHCi may be downloaded and installed from: <https://www.haskell.org/platform/>

Installing NuGet packages

If you have not previously installed a NuGet package into a .NET project within Visual Studio, there is an explanation here: <https://docs.microsoft.com/en-us/nuget/quickstart/install-and-use-a-package-in-visual-studio>. (This explanation suggests that Visual Studio 2019 is a pre-requisite, but this is not the case: all versions of Visual Studio have provided the NuGet Package Manager for many years.)

MetalUp.FunctionalLibrary NuGet package

The **MetalUp.FunctionalLibrary** package, used throughout this book, is published on the NuGet Public Gallery. In the NuGet Package Manager window search for 'MetalUp.FunctionalLibrary'..

If you wish to view the source code for the library (written in C#) you may download it from here: <https://github.com/MetalUp/FunctionalLibrary>.

System.ValueTuple NuGet package

Search, in the NuGet Package Manager window for 'value.tuple'. Note that it is not necessary to install this package if you are using C# version 8.0+, as they are a standard feature of the language.

Appendix II: Further Reading

The following publications are recommended for those who want to learn functional programming in Haskell, or in C#. to a much deeper level:

Haskell

Learn You Haskell for a Great Good! by Miran Lipovača. This is, as it says on the cover, 'A Beginner's Guide', and if you can cope with the rather chatty style, contains good, clear explanations. As well as being available in hardcopy form, it may be read freely online at:

<http://learnyouahaskell.com/chapters>

Programming in Haskell by Graham Hutton. A well-written book, aimed at university graduates, but full of practical examples and exercises.

Real World Haskell, by Bryan O'Sullivan, John Goerzen, and Don Stewart. Although 10 years old this is nonetheless still a useful, thorough, explanation of Haskell.

The Haskell Cheatsheet: <https://cheatsheet.codeslower.com/> A useful and succinct reference on Haskell syntax.

A History of Haskell by Simon Peyton Jones, presentation at *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, June 2007. This is an inspiring presentation, even if you get completely lost, technically, from near the beginning. The HOPL conference has been held only three times in the history of computing, in 1978, 1993, and 2007 (though there is another one scheduled for 2020). The participants in this conference include many of our greatest Computer Scientists, and it is wonderful to listen, first hand, to their personal accounts of the origins and development of specific programming languages. Video may be viewed at:

<https://www.microsoft.com/en-us/research/publication/a-history-of-haskell-being-lazy-with-class/>

C#

Real-World Functional Programming: With Examples in F# and C# by Tomas Petricek and John Skeet, Manning Publications, 2010. Written more for the professional programmer, this is a very good introduction to FP in both C# and F#. John Skeet, a software engineer at Google, is known throughout the C# world as the user of StackOverflow.com with the highest number of reputation points.

Appendix III: Versioning (of this book)

This book adopts ‘semantic versioning’ with the following meanings:

- A third-level version change (e.g. 1.0.n) indicates *minor* edits or corrections to text, layout, or formatting.
- A second level version change (e.g. 1.n.0) indicates correction to consequential error, in text or code.
- A first level version change (e.g. n.0.0) indicates new material and/or or re-structuring.

V1.0.0

Released 17th October 2019

V1.1.0

Released 18th October 2019

V1.2.0

Released 21st October 2019

Appendix IV: References & Acknowledgements

¹ For example, *Gadsby*, by Ernest Vincent Wright. See [https://en.wikipedia.org/wiki/Gadsby_\(novel\)](https://en.wikipedia.org/wiki/Gadsby_(novel))

² Picture credit, and more information: https://en.wikipedia.org/wiki/Thomas_Kuhn

³ [Edsger Dijkstra](#) (March 1968). "Go To Statement Considered Harmful" (PDF). *Communications of the ACM*. **11** (3): 147–148. doi:10.1145/362929.362947

⁴ https://en.wikipedia.org/wiki/Haskell_Curry

⁵ *The New Hacker's Dictionary*, Eric S. Raymond, Third Edition 1999, The MIT Press

⁶ 'Lazy functional programming for real - Tackling the awkward squad' presentation by Simon Peyton-Jones, <https://www.microsoft.com/en-us/research/uploads/prod/2016/07/Marktoberdorf.ppt>

Acknowledgements

The author wishes to express his gratitude to the following:

Simon Peyton Jones, who provided useful feedback on the overall approach of the book, and generously agreed to write the foreword.

Stefano Cascarini, colleague of the author at Naked Objects Group Ltd, who introduced the author to the idea of functional programming more than 10 years ago.

Graham Hutton, Professor of Computer Science at University of Nottingham, and author of *Programming in Haskell*, who provided hints and references in relation to several questions.

Ian Head of Head-e Design, who generously donated a lot of time to produce the custom image and overall design for the book cover.

Mark Sayers, John Stout, and others who have given useful feedback on drafts, or early versions.

'Functional programming is not just another programming language; it is a radical and elegant attack on the entire enterprise of programming. It makes you think in a new way about programming...' says Simon Peyton Jones in the foreword, 'So I am delighted to see a book that introduces young people to the joys of functional programming.'

This is the first book on Functional Programming written specifically for A-Level Computer Science students, focusing on the principles of Functional Programming, and the benefits of adopting them.

Using practical exercises throughout, the student is introduced to the Haskell language, purpose-designed for Functional Programming, but also learns how many of the same patterns can be implemented in VB, and the advantages and disadvantages of both options. As well as providing a deeper insight into the real nature of Functional Programming, this two-language approach makes it easier for students to apply what they have learned within their own A-level programming projects.



Richard Pawson worked in the computing industry for 40 years before teaching A-level Computer Science at Stowe School. He has a BSc in Engineering, a PhD in Computer Science, and a PGC in Intellectual Property Law. He now splits his time between managing a large open source product, writing books, and guest teaching in various schools.

