Real-World Functional Programming

With examples in F# and C#

SAMPLE CHAPTER

Tomas Petricek
with Jon Skeet

Foreword by Mads Torgersen

MANNING

Real-World Functional Programming

by Tomas Petricek
with Jon Skeet

Chapter 12

brief contents

12 Sequence expressions and alternative workflows

This chapter covers

- Processing and generating sequences of values
- Working with F# sequence expressions
- Understanding monads and LINQ expressions
- Implementing F# computation expressions

Before we can start talking about sequence expressions, you must know what a *sequence* is. This is another F# term that comes from mathematics, where a sequence is an ordered list containing a possibly infinite number of elements. Don't worry if that all sounds a bit abstract; you're already familiar with the type that expresses the same idea in .NET: `IEnumerable<T>`.

The primary reason for having the `IEnumerable<T>` type in the .NET Framework is it gives us a unified way to work with collections of data such as arrays, dictionaries, mutable lists, and immutable F# lists. In F# we'll be talking about sequences, because this is a more general term. A sequence can represent a finite number of elements coming from a collection, but it can be also generated dynamically and retrieved on an on-demand basis. You'll learn that infinite sequences, which sound somewhat academic, can still be useful in real applications.

We'll begin by looking at ways to create and process sequences. The traditional functional technique is to use higher-order functions, but modern languages often provide an easier way. In C#, we can use iterators to generate a sequence and LINQ queries to process an existing one. The F# language unifies these two concepts into one and allows us to write most of the operations using *sequence expressions*.

The syntax used for writing sequence expressions in F# isn't a single-purpose language feature designed only for sequences. That is just one (very useful!) application of a more general construct called *computation expressions*. Computation expressions can be used for writing code that looks like ordinary F# but behaves differently. In the case of sequence expressions, a sequence of results is generated instead of just one value, but we'll look at other examples. We'll show you how to use computation expressions for logging, and how they can make option values easier to work with.

NOTE Computation expressions can be used for customizing the meaning of the code in many ways, but some limits exist. In particular, the code written using computation expressions has to be executed as compiled .NET code and we can customize only a few primitives inside it. It can't be used to manipulate the code and execute it in a different environment, in the way that LINQ to SQL does, for example. To do similar things in F#, we have to combine ideas from this chapter with a feature called *F# quotations,* which isn't discussed in this book. You'll find resources about quotations on the book's website.

We'll start by talking about sequences, and once you become familiar with sequence expressions, we'll look at computation expressions and how they relate to LINQ queries in C#. Let's take our first steps with sequences. Before we can start working with them, we need to know how to create them.

12.1 Generating sequences

There are several techniques for generating sequences, so let's look at our options. The direct way is to implement the `IEnumerator<T>` interface, providing a `Current` property and a `MoveNext` method, which moves the enumerator object to the next element. This forces us to explicitly create an object with mutable state, which obviously goes against the functional style. Normally we can apply techniques that hide the mutation and give us a more declarative way of expressing the generated sequence's contents. This is similar to using lazy values that we've seen in the previous chapter. Using mutable state explicitly (for example, to implement caching) doesn't look like a good functional style, but when we hide the mutation into a `Lazy<'T>` type, we'll get a perfectly reasonable functional code.

As usual in functional programming, we can use higher-order functions. The F# library supports quite a few of these for working with sequences, but we'll look at only one example. As we'll see later, both C# and F# give us a simpler way to generate sequences. In C#, we can use *iterators* and F# supports a general-purpose sequence-processing feature called *sequence expressions*.

12.1.1 Using higher-order functions

The functions used to work with sequences in F# are in the `Seq` module, and we'll examine one very general function called `Seq.unfold`. You can see it as an opposite to the `fold` function, which takes a collection and "folds" it into a single value. `unfold` takes a single value and "unfolds" it into a sequence. The following snippet shows how to generate a sequence containing numbers up to 10 formatted as strings:

```
> let nums = Seq.unfold (fun num ->
    if (num <= 10) then Some(string(num), num + 1) else None) 0
  ;;
val nums : seq<string> = seq ["0"; "1"; "2"; "3"; ...]
```

The `num` value represents the state used during the generation of the sequence. When the lambda function is called for the first time, the value of `num` is set to the initial value of the second parameter (zero in our example). The lambda function returns an option type containing a tuple. The value `None` marks the end of the sequence. When we return `Some`, we give it two different values in a tuple:

- A value that will be returned in the sequence (in our case, the number converted to a string).
- A value that is the new state to use when the lambda function is next called.

As you can see from the output, the type of the returned value is `seq<string>`. This is an F# type alias for the `IEnumerable<string>` type. It's a different way of writing the same type, in the same way that `float` is a C# alias for `System.Single`, so you can mix them freely. The output also shows the first few elements of the sequence, but since the sequence can be infinite, the F# Interactive shell doesn't attempt to print all of them.

The standard .NET library doesn't contain a similar method for C#. One of the few methods that generate sequences in C# is `Enumerable.Range` (from the `System.Linq` namespace), which returns an ascending sequence of numbers of the specified length (second argument) from the specified starting number (the first argument). We could implement a function like `Seq.unfold` in C# as well, but we'll see that similar results can be easily achieved using C# iterators, which we'll look at next.

12.1.2 Using iterators in C#

When iterators were first introduced in C# 2.0, the most common use for them was to simplify implementing the `IEnumerable<T>` interface for your own collections. The programming style used in C# has been evolving, and iterators are now used together with other functional constructs for a variety of data processing operations.

Iterators can be used for generating arbitrary sequences. We'll start with a simple example that generates a sequence of factorials that are less than 1 million, formatted as strings. Listing 12.1 shows the complete source code.

Listing 12.1 Generating factorials using iterators (C#)

```
static IEnumerable<string> Factorials() {
  int factorial = 1;                                    ❶
  for(int num = 0; factorial < 1000000; num++) {
```

```
      factorial = factorial * num;                                    ❷
      yield return String.Format("{0}! = {1}", num, factorial);
    }                                                  Returns next string
}
```

The C# compiler performs a rather sophisticated transformation on the iterator code to create a "hidden" type that implements the `IEnumerable<T>` interface. The interesting thing about listing 12.1 is how it works with the local state. We declare one local variable to store some mutable state ❶, and a second mutable variable is declared as part of the `for` loop. The algorithm is implemented inside a loop, which is executed every time we want to pull another value from the iterator. The loop body updates the local state of the iterator ❷ and yields the newly calculated value.

The code is very imperative, because it heavily relies on mutation, but from the outside iterators look almost like functional data types, because the mutable state is hidden. Let's look at the *sequence expression*, which is the general F# mechanism for generating, but also for processing, sequences.

12.1.3 Using F# sequence expressions

Iterators in C# are very comfortable, because they allow you to write complicated code (a type that implements the `IEnumerable<T>/IEnumerator<T>` interfaces) in an ordinary C# method. The developer-written code uses standard C# features such as loops, and the only change is that we can use one new kind of statement to do something nonstandard. This new statement is indicated with `yield return` (or `yield break` to terminate the sequence), and the nonstandard behavior is to return a value as the next element of a sequence. The sequence is then accessed on demand (end evaluated element-by-element) using the `MoveNext` method. Sequence expressions in F# are similar: they use a construct that's equivalent to `yield return`.

WRITING SEQUENCE EXPRESSIONS

In C#, we can use iterators automatically when implementing methods that return `IEnumerable<T>`, `IEnumerator<T>`, or their nongeneric equivalents. F# sequence expressions are marked explicitly using the `seq` identifier, and don't have to be used as the body of a method or function. As the name suggests, sequence expressions are a different type of expression, and we can use them anywhere in our code. Listing 12.2 shows how to create a simple sequence using this syntax.

Listing 12.2 Introducing sequence expression syntax (F# Interactive)

```
> let nums =
    seq { let n = 10          ❶
          yield n + 1         ❷
          printfn "second.."  ❸
          yield n + 2 };;
val nums : seq<int>           ❹
```

When writing sequence expressions, we enclose the whole F# expression that generates the sequence in a `seq` block ❶. The block is written using curly braces and the

`seq` identifier[1] at the beginning denotes that the compiler should interpret the body of the block as a sequence expression. There are other possible identifiers that specify other alternative workflows, as you'll see later. In the case of `seq`, the block turns the whole expression into a lazily generated sequence. You can see this by looking at the inferred type of the value ❹.

The body of the sequence expression can contain statements with a special meaning. Similarly to C#, there's a statement for returning a single element of the sequence. In F# this is written using the `yield` keyword ❷. The body can also contain other standard F# constructs, such as value bindings, and even calls that perform side effects ❸.

Similar to C#, the body of the sequence expression executes lazily. When we create the sequence value (in our previous example, the value `nums`), the body of the sequence expression isn't executed. This only happens when we access elements of the sequence, and each time we access an element, the sequence expression code only executes as far as the next `yield` statement. In C#, the most common way to access elements in an iterator is using a `foreach` loop. In the following F# example, we'll use the `List.ofSeq` function, which converts the sequence to an immutable F# list:

```
> nums |> List.ofSeq;;
second..
val it : int list = [11; 12]
```

The returned list contains both of the elements generated by the sequence. This means that the computation had to go through the whole expression, executing the `printfn` call on the way, which is why the output contains a line printed from the sequence expression. If we take only a single element from the sequence, the sequence expression will only evaluate until the first `yield` call, so the string won't be printed:

```
> nums |> Seq.take 1 |> List.ofSeq;;
val it : int list = [11]
```

We're using one of the sequence processing functions from the `Seq` module to take only a single element from the sequence. The `take` function returns a new sequence that takes the specified number of elements (one in the example) and then terminates. When we convert it to an F# list, we get a list containing only a single element, but the `printfn` function isn't called.

When you implement a sequence expression, you may reach a point where the body of the expression is too long. The natural thing to do in this case would be to split it into a couple of functions that generate parts of the sequence. If the sequence uses multiple data sources, we'd like to have the code that reads the data in separate functions. So far, so good—but then we're left with the problem of composing the sequences returned from different functions.

---

[1] You may be surprised that we're referring to `seq` as an identifier instead of a keyword, but you'll see later that it's an identifier (that we could even define ourselves) rather than a special keyword built into the F# language. The `seq` identifier also isn't defined automatically by the `seq<'a>` type. The name is the same, but `seq` identifier here is a different symbol defined by the F# library.

COMPOSING SEQUENCE EXPRESSIONS

The `yield return` keyword in C# only allows us to return a single element, so if we want to yield an entire sequence from a method implemented using iterators in C#, we'd have to loop over all elements of the sequence using `foreach` and yield them one by one. This would work, but it would be inefficient, especially if we had several sequences nested in this way. In functional programming, composability is a more important aspect, so F# allows us to compose sequences and yield the whole sequence from a sequence expression using a special language construct: `yield!` (usually pronounced *yield-bang*). Listing 12.3 demonstrates this, generating a sequence of cities in three different ways.

Listing 12.3 Composing sequences from different sources (F# Interactive)

```
> let capitals = [ "Paris"; "Prague" ];;
val capitals : string list

> let withNew(name) =
    seq { yield name
          yield "New " + name };;
val withNew : string -> seq<string>

> let allCities =
    seq { yield "Oslo"          (1)
          yield! capitals       (2)
          yield! withNew("York") };;   (3)
val allCities : seq<string>

> allCities |> List.ofSeq;;
val it : string list = ["Oslo"; "Paris"; "Prague"; "York"; "New York"]
```

Lists capital cities

Returns name and name with prefix

All data composed together

Listing 12.3 starts by creating two different data sources. The first one is an F# list that contains two capital cities. The type of the value is `list<string>`, but since F# lists implement the `seq<'a>` interface, we can use it as a sequence later in the code. The second data source is a function that generates a sequence containing two elements. The next piece of code shows how to join these two data sources into a single sequence. First, we use the `yield` statement to return a single value (1). Next, we use the `yield!` construct to return all the elements from the F# list (2). Finally, we call the function `withNew` (3) (which returns a sequence) and return all the elements from that sequence. This shows that you can mix both ways of yielding elements inside a single sequence expression.

Just like `yield`, the `yield!` construct also returns elements lazily. This means that when the code gets to the point where we call the `withNew` function, the function gets called, but it only returns an object representing the sequence. If we wrote some code in the function before the `seq` block it would be executed at this point, but the body of the `seq` block wouldn't start executing. That only happens after the `withNew` function returns, because we need to generate the next element. When the execution reaches the first `yield` construct, it will return the element and transfers the control back to the caller. The caller then performs other work and the execution of the sequence resumes when the caller requests another element.

We've focused on the syntax of sequence expressions, but they can sound quite awkward until you start using them. There are several patterns that are common when using sequence expressions; let's look at two of them.

12.2 Mastering sequence expressions

So far, we've seen how to return single elements from a sequence expression and also how to compose sequences in F#. We haven't yet examined the F# version of the previous factorial example using mutable state. Somewhat predictably, the F# code will be quite different.

12.2.1 Recursive sequence expressions

The primary control flow structure in functional programming is recursion. We've used it in many examples when writing ordinary functions, and it allows us to solve the same problems as imperative loops but without relying on mutable state. When we wanted to write a simple recursive function, we used the `let rec` keyword, allowing the function to call itself recursively.

The `yield!` construct for composing sequences also allows us to perform recursive calls inside sequence expressions, so we can use the same functional programming techniques when generating sequences. Listing 12.4 generates all factorials under 1 million just like the C# example in listing 12.1.

Listing 12.4 Generating factorials using sequence expressions (F# Interactive)

```
> let rec factorialsUtil(num, factorial) =                    ❶
    seq { if (factorial < 1000000) then
            yield sprintf "%d! = %d" num factorial             ❷
            let num = num + 1
            yield! factorialsUtil(num, factorial * num) };;    ❸
val factorialsUtil : int * int -> seq<string>

> let factorials = factorialsUtil(0, 1)                       ❹
val factorials : seq<string> =
  seq ["0! = 1"; "1! = 1"; "2! = 2"; "3! = 6"; "4! = 24 ...]
```

Listing 12.4 begins with a utility function that takes a number and its factorial as an argument ❶. When we want to compute the sequence of factorials later in the code, we call this function and give it the smallest number for which a factorial is defined to start the sequence ❹. This is zero, because by definition the factorial of zero is one.

The whole body of the function is a `seq` block, so the function returns a sequence. In the sequence expression, we first check whether the last factorial is smaller than 1 million, and if not, we end the sequence. The `else` branch of the `if` expression is missing, so it won't yield any additional numbers. If the condition is true, we first yield a single result ❷, which indicates the next factorial formatted as a string. Next, we increment the number and perform a recursive call ❸. This returns a sequence of factorials starting from the next number; we use `yield!` to compose it with the current sequence.

Note that converting this approach to C# is difficult, because C# doesn't have an equivalent of the `yield!` feature. We'd have to iterate over all the elements using a `foreach` loop, which could cause stack overflow. Even if it worked, it would be inefficient due to a large number of nested loops. In F#, there's an optimization for tail-recursive calls using `yield!`, similar to the usual function calls. This means that when a sequence expression ends with the `yield!` call and there's no subsequent code (like in the previous example), the call won't add any inefficiency even if we use several nested `yield!` calls.

This example shows that we can use standard functional patterns in sequence expressions. We used the `if` construct inside the sequence expression and recursion to loop in a functional style. F# allows us to use mutable state (using reference cells) and imperative loops such as `while` inside sequence expressions as well, but we don't need them very often. On the other hand, `for` loops are used quite frequently, as we'll see when we discuss sequence processing later in this chapter.

List and array expressions

So far, we've seen sequence expressions enclosed in curly braces and denoted by the `seq` identifier. This kind of expression generates a lazy sequence of type `seq<'a>`, which corresponds to the standard .NET `IEnumerable<T>` type. F# also provides support for creating immutable F# lists and .NET arrays in a simple way. Here's a snippet showing both collection types:

```
> let cities =
    [ yield "Oslo"
      yield! capitals ]
  ;;
val cities : string list =
  [ "Oslo";
    "London"; "Prague" ]
```

```
> let cities =
    [| yield "Barcelona"
       yield! capitals |]
  ;;
val cities : string array =
  [| "Barcelona";
     "London"; "Prague" |]
```

As you can see, we can also enclose the body of the sequence expression in square brackets just as we normally do to construct F# lists, and in square brackets followed by the vertical bar symbol (|) to construct arrays. F# treats the body as an ordinary sequence expression and converts the result to a list or an array, respectively.

When we use array or list expressions, the whole expression is evaluated eagerly, because we need to populate all the elements. Any side effects (such as printing to the console) will be executed immediately. Although sequences may be infinite, arrays and lists can't: evaluation would continue until you ran out of memory.

Take another look at listing 12.4, where we generated factorials up to a certain limit. What would happen if we removed that limit (by removing the `if` condition)? In ordinary F# we'd get an infinite loop, but what happens in a sequence expression?

The answer is we'd create an infinite sequence, which is a valid and useful functional construct.

12.2.2 Using infinite sequences

In the previous chapter, we briefly demonstrated how to implement a lazy list using lazy values. This data structure allowed us to create infinite data structures, such as a list of all integers starting from zero. This was possible because each evaluation of an element was delayed: the element's value was only calculated when we accessed it, and each time we only forced the calculation of a single element.

Sequences represented using `seq<'a>` are similar. The interface has a `MoveNext` method, which forces the next element to be evaluated. The sequence may be infinite, which means that the `MoveNext` method will be always able to calculate the next element and never returns `false` (which indicates the end of sequence). Infinite sequences may sound just like a curiosity, but we'll see that they can be quite valuable and give us a great way to separate different parts of an algorithm and make the code more readable.

In chapter 4, we talked about drawing charts. We used random colors to fill the individual parts, which didn't always give the best result. You can represent colors of the chart as an infinite sequence. In listing 12.5, we'll start by generating sequence of random colors, but we'll look at other options soon.

Listing 12.5 Generating an infinite sequence of random colors in C# and F#

```
// C# version using loops
IEnumerable<Color> RandomColors(){
  var rnd = new Random();
  while(true) {                                                    ❶
    int r = rnd.Next(256), g = rnd.Next(256), b = rnd.Next(256);
    yield return Color.FromArgb(r, g, b);                          ❷
  }
}

// F# version using recursion
let rnd = new Random()
let rec randomColors = seq {                                       ❸
  let r, g, b = rnd.Next(256), rnd.Next(256), rnd.Next(256)
  yield Color.FromArgb(r, g, b)                                    ❹
  yield! randomColors }        ❺
```

Both implementations contain an infinite loop that generates colors. In C#, the loop is achieved using `while(true)` ❶. The functional way to create infinite loops is to use recursion ❺. In the body of the infinite loop, we yield a single randomly generated color value. In F# we use the `yield` construct ❹ and in C# we use `yield return` ❷.

If you compile the F# version of the code, you'll get a warning on the line with the recursive call ❺. The warning says that the recursive reference will be checked at runtime. We saw this warning in chapter 8. It notifies us that we're referencing a value inside its own definition. In this case, the code is correct, because the recursive call will be performed later, after the sequence is fully initialized.

Listing 12.5 also uses a different indentation style when enclosing F# code in a `seq` block ❸. Instead of starting on a new line and indenting the whole body, we added the `seq` identifier and the opening curly brace to the end of the line. We'll use this option in some listings in the book, to make the code more compact. In practice, both of these options are valid and you can choose whichever you find more readable.

Now that we have an infinite sequence of colors, let's use it. Listing 12.6 demonstrates how infinite sequences allow a better separation of concerns. Only the F# code is shown here, but the C# implementation (which is very similar) is available at the book's website.

Listing 12.6 Drawing a chart using sequence of colors (F#)

```
open System.Drawing
open System.Windows.Forms

let dataSource = [ 490; 485; 450; 425; 365; 340; 290; 230; 130; 90; 70; ]
let coloredSequence = Seq.zip dataSource randomColors                    ❶
let coloredData = coloredSequence |> List.ofSeq          ❷

let frm = new Form(ClientSize = Size(500, 350))
frm.Paint.Add(fun e ->
  e.Graphics.FillRectangle(Brushes.White, 0, 0, 500, 350)
  coloredData |> Seq.iteri(fun i (num, clr) ->                ❸
    use br = new SolidBrush(clr)
    e.Graphics.FillRectangle(br, 0, i * 32, num, 28) )
  )
frm.Show()
```

Calculates location of bar using index

To provide a short but complete example, we've just defined some numeric data by hand. We use the `Seq.zip` function to combine it with the randomly generated colors ❶. This function takes two sequences and returns a single sequence of tuples: the first element of each tuple is from the first sequence, and the second element comes from the second sequence. In our case, this means that each tuple contains a number from the data source and a randomly generated color. The length of the returned sequence is the length of the shorter sequence from the two given sequences, so it will generate a random color for each of the numeric value and then stop. This means that we'll need only limited number of colors. We could generate, say, one hundred colors, but what if someone gave us 101 numbers? Infinite sequences give us an elegant way to solve the problem without worrying about the length.

Before using the sequence, we convert it to a list ❷. We do this is because the sequence of random colors isn't pure and returns different colors each time we reevaluate it. This means that if we didn't convert it to a list before using it, we'd get different colors during each redraw of the form. Once we have the list of data with colors, we need to iterate over its elements and draw the bars. We're using `Seq.iteri` ❸, which calls the specified function for every element, passing it the index of the element in the sequence and the element itself. We immediately decompose the element using a tuple pattern into the numeric value (the width of the bar) and the generated color.

What makes this example interesting is that we can easily use an alternative way to generate colors. If we implemented it naïvely, the color would be computed in the drawing function ❸. This would make it relatively hard to change which colors are used. However, the solution in listing 12.6 completely separates the color-generation code from the drawing code, so we can change the way chart is drawn just by providing a different sequence of colors. Listing 12.7 shows an alternative coloring scheme.

Listing 12.7 Generating a sequence with color gradients (C# and F#)

```
// C# version using loops
IEnumerable<Color> GreenBlackColors() {
  while(true) {                                    ❶
    for(int g = 0; g < 255; g += 25)
      yield return Color.FromArgb(g / 2, g, g / 3);
  }
}

// F# version using loop and recursion
let rec greenBlackColors = seq {
  for g in 0 .. 25 .. 255 do                       ❷
    yield Color.FromArgb(g / 2, g, g / 3)
  yield! greenBlackColors }                        ❸
```

The code in listing 12.7 again contains an infinite loop, implemented either using a `while` loop ❶ or recursion ❸. In the body of the loop, we generate a color gradient containing 10 different colors. We're using a `for` loop to generate the green component of the color and calculating the blue and red components from that. This example also shows the F# syntax for generating a sequence of numbers with a specified step ❷. The value of `g` will start off as 0 and increment by 25 for each iteration until the value is larger than 250. Figure 12.1 shows the end result.

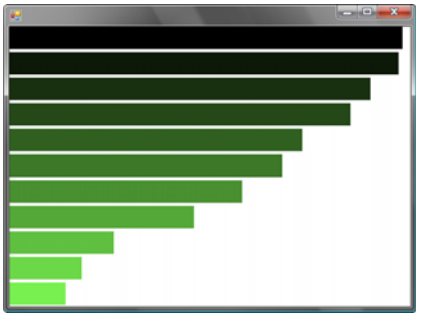

Figure 12.1 A chart painted using color gradient generated as a sequence of colors

As you can see, infinite sequences can be useful in real-world programming, because they give us a way to easily factor out part of the code that we may want to change later. Infinite sequences are also curious from the theoretical point of view. In Haskell, they're often used to express numerical calculations.

So far we've mostly examined *creating* sequences. Now we're going to have a look at some techniques for *processing* them.

Infinite lists in Haskell and sequence caching in F#

As we mentioned in chapter 11, Haskell uses lazy evaluation everywhere. We've seen that `Lazy<'a>` type in F# can simulate lazy evaluation for values when we need it, and sequences enable us to emulate some other Haskell constructs in the same way.

(continued)

Let's look at one slightly obscure example, just to get a feeling for what you can do. In Haskell, we can write the following code:

```
let nums = 1 : [ n + 1 | n <- nums ]
```

Once we translate it into F#, you'll understand what's going on. The standard functional lists in Haskell are lazy (because everything is) and the `:` operator corresponds to the F# `::` operator. The expression in square brackets returns all the numbers from the list incremented by 1. In F#, we could write the same thing using sequence expressions:

```
let rec nums =
  seq { yield 1
        for n in nums do yield n + 1 };;
```

The code constructs a sequence that starts with 1 and recursively takes all numbers from the sequence and increments them by 1. This means that the returned sequence will contain numbers 1, 2, 3, and so on. The F# version is horribly inefficient, because in each recursive call, it starts constructing a new sequence from the first element. To evaluate the sequence of length 3, we create one instance of `nums` of length 3, one of length 2, and one of length 1.

The idiomatic version of the code in F# would look differently. Just like when generating factorials in listing 12.4, we could implement a utility function that generates sequence from the given number. Then we'd recursively call it using the optimized `yield!` primitive. In Haskell, the situation is different because evaluated values are cached. This means that it doesn't have to recalculate the sequence from the beginning. We can get similar behavior in F# using a function `Seq.cache`:

```
let rec nums =
  seq { yield 1
        for n in nums do yield n + 1 } |> Seq.cache;;
```

The `Seq.cache` function returns a sequence that caches values that have already been calculated, so this version of the code performs a lot more sensibly. Accessing the 1000th element is about 100 times faster with the caching version than with the original. Combining caching and sequence expressions gives us some of the same expressive power as the more mathematically oriented Haskell. However, it's usually a better idea to look for an idiomatical F# solution, in this case using `yield!`.

12.3 Processing sequences

When processing sequences, we have a wide variety of options ranging from low-level techniques where we can control all the details, but that make it difficult to express more complicated but common processing patterns, to higher-level techniques that can't express everything we may want, but the rest can be expressed very elegantly.

In C#, the lowest level (aside from implementing the `IEnumerable<T>` interface directly) is to use iterator blocks and read the input sequence using either `foreach` or the enumerator object. At the higher level, we can use predefined (or our own) higher-order methods such as `Where` and `Select`, and if the processing involves only certain operations, we can use the C# 3.0 query syntax.

The most common approach for processing sequences in F# is similar to those for other collection types. We've seen that lists can be processed with functions like `List.filter` and `List.map`, and that similar functions are available for arrays in the `Array` module. It should come as no surprise that the same set of functions exists for sequences as well, in the `Seq` module. The F# language doesn't explicitly support any query syntax, but we'll see that sequence expressions to some point unify the ideas behind lower-level iterators and higher-level queries.

12.3.1 Transforming sequences with iterators

So far, we've only used iterators to generate a sequence from a single piece of data (if any). However, one common use of iterators is to transform one sequence into another in some fashion. As a simple example, here's a method that takes a sequence of numbers and returns a sequence of squares:

```
IEnumerable<int> Squares(IEnumerable<int> numbers) {
  foreach(int i in numbers)
    yield return i * i;
}
```

We're using the familiar `foreach` construct, but keep in mind that `foreach`, which contains the `yield return` statement, has a different meaning. It doesn't run the loop eagerly and instead evaluates it on demand. The `foreach` statement allows us to write code that generates some elements for every iteration of the loop, which corresponds to pulling a single element from the input sequence and pushing zero or more elements to the output sequence (in the previous case, we always generate exactly one element). We'd use the very same approach if we wanted to implement generic `Where` and `Select` methods from LINQ to Objects.

As a more complicated example, let's implement a `Zip` method with the same behavior as the `Seq.zip` function in F#. We'll give it two sequences and it will return a single sequence containing elements from the given sequences joined in tuples. This method is available in the .NET 4.0 libraries, but we'll look at it, because it shows an interesting problem. We can't use `foreach` to simultaneously take elements from two source sequences. As you can see in listing 12.8, the only option we have is to use the `IEnumerable<T>` and `IEnumerator<T>` interfaces directly.

Listing 12.8 Implementing the `Zip` method (C#)

```
public static IEnumerable<Tuple<T1, T2>> Zip<T1, T2>
    (IEnumerable<T1> first, IEnumerable<T2> second) {
  using(var firstEn = first.GetEnumerator())
  using(var secondEn = second.GetEnumerator()) {
```

❶

```
    while (firstEn.MoveNext() && secondEn.MoveNext()) {                ❷
      yield return Tuple.Create(firstEn.Current, secondEn.Current);    ❸
    }
  }
}
```

Looking at the signature of the method, we can see that it takes two sequences as arguments. The method is generic, with each input sequence having a separate type parameter. We're using a generic C# tuple, so the returned sequence contains elements of type `Tuple<T1, T2>`. In the implementation, we first ask each sequence for an enumerator we can use to traverse the elements ❶. We repeatedly call the `MoveNext` method on each enumerator to get the next element from both of the sequences ❷. If neither sequence has ended, we yield a tuple containing the current element of each enumerator ❸.

This example shows that sometimes, processing methods need to use the `IEnumerator<T>` interface explicitly. The `foreach` loop gives us a way to pull elements from a single source one by one, but once we need to pull elements from multiple sources in an interleaving order, we're in trouble. If we wanted to implement `Seq.zip` in F#, we'd have to use the same technique. We could use either a `while` loop inside a sequence expression or a recursive sequence expression. Most of the processing functions we'll need are already available in the .NET and F# libraries so we'll use these where we can, either explicitly or by using C#'s query expression syntax.

12.3.2 Filtering and projection

The two most frequently used sequence processing operators are filtering and projection. We used both of them in chapter 6 with functional lists in F# and the generic .NET `List<T>` type in C#. The `Where` and `Select` extension methods from LINQ libraries already work with sequences, and in F# we can use two functions from the `Seq` module (namely `Seq.map` and `Seq.filter`) to achieve the same results.

USING HIGHER-ORDER FUNCTIONS

Working with the `Seq` module in F# is the same as with `List`, and we've already seen how to use LINQ extension methods in C#. There's one notable difference between working with lists and sequences: sequences are lazy. The processing code isn't executed until we take elements from the returned sequence, and even then it only does as much work as it needs to in order to return results as they're used. Let's demonstrate this using a simple code snippet:

```
var nums1 =
   nums.Where(n => n%3 == 0)
       .Select(n => n * n)
```

```
let nums1 =
   nums |> Seq.filter (fun n -> n%3=0)
        |> Seq.map (fun n -> n * n)
```

When we run this code, it won't process any elements; it only creates an object that represents the sequence and that can be used for accessing the elements. This also means that the `nums` value can be an infinite sequence of numbers. If we only access the first 10 elements from the sequence, the code will work correctly, because both filtering and projection process data lazily.

You're probably already familiar with using higher-order processing functions after our extensive discussion in chapter 6, and we've provided many examples throughout the book. In this chapter, we'll instead look at other ways to express alternative workflows.

USING QUERIES AND SEQUENCE EXPRESSIONS

In C# 3.0, we can write operations with data that involve projection and filtering using the new query expression syntax. Query expressions support many other operators, but we'll stick to only projection and filtering in order to demonstrate functional techniques and F# features.

Although F# doesn't have specific query expression support, we can easily write queries that project and filter data using sequence expressions. This is due to the way that sequence expressions can be used anywhere in F#, rather than just as the implementation of a function returning a sequence. Listing 12.9 shows how we can implement our earlier example using a query in C# and a sequence expression in F#.

Listing 12.9 Filtering and projecting sequences in C# and F#

C#

```
var nums1 =
    from n in nums
    where n%3 == 0
    select n * n;
```

F#

```
let nums1 = seq {
    for n in nums do
        if (n%3 = 0) then
            yield n * n }
```

In C#, query expressions and iterators are quite different, but sequence expressions in F# show how they're conceptually related. Each part of the query expression has an equivalent construct in F#, but it's always more general: the `from` clause is replaced by a simple `for` loop, the `where` clause is replaced by an `if` condition, and the `select` clause corresponds to the `yield` statement with the projection expressed as a normal calculation.

C# query expression syntax supports several other operators that aren't easily expressible using F# sequence expressions. This means that the C# version is more powerful, but the F# implementation is more uniform.

Note how both C# query expressions and F# sequence expressions work internally. A C# query expression is translated in a well-defined way into a sequence of calls such as `Where`, `Select`, `SelectMany`, `Join`, and `GroupBy` using lambda expressions. These are typically extension methods but they don't have to be—the compiler doesn't care what the query expression *means*, only that the translated code is valid. This "data source agnosticism" is essential for data processing technologies such as LINQ to Objects and LINQ to SQL, but we'll use it shortly to show how the query syntax can be used for working with other kinds of values.

On the other hand, sequence expressions can be used to express more complicated and general-purpose constructs. We could duplicate the `yield` construct to return two elements for a single item from the data source. This would be easy enough to achieve in C# using iterators, but it's not possible to express the transformation "inline" using the query syntax.

Additional query operators in LINQ

Query expression syntax in C# 3.0 is tailored for retrieving and formatting data from various data sources, so it includes operations beyond projection and filtering. These operators are mostly present for this single purpose, and there's no special syntax for them in F#. All these standard operators are available as regular higher-order functions operating on sequences. For instance, take ordering data:

```
var q =
    from c in customers
    orderby c.Name
    select c;
```

```
let q =
    customers
    |> Seq.sortBy (fun c -> c.City)
```

The function that we give as the first argument to the `Seq.sortBy` operator specifies which property of the processed element should be used when comparing two elements. In the C# query syntax, this corresponds to the expression following the `orderby` clause. The C# compiler transforms this expression into a call to a standard `OrderBy` method using a lambda function. Another operation that is available only as a higher-order function in F# is grouping:

```
var q =
    from c in customers
    group c by c.City;
```

```
let q =
    customers
    |> Seq.groupBy (fun c -> c.City)
```

To group a sequence we need to specify a function that returns the key that identifies the group in which the element belongs. Again, C# has special syntax for this, but in the F# snippet we're using a standard lambda function.

In these examples, both versions of the code look reasonable. However, when we need to write F# code that mixes projection and filtering together with some operations that can only be written using higher-order functions, the equivalent C# query expression will be easier to understand.

The implementation of sequence expressions in the F# compiler is optimized, but without these optimizations, it would work similarly to the C# query expressions. An arbitrary sequence expression could be translated into a sequence of standard function calls. Similar to C#, we can provide our own implementation of these functions, so it's wise to look deeper and understand how the translation works in F#. The F# language uses a smaller number of operations and heavily relies on a single operation called flattening projection.

12.3.3 Flattening projections

A *flattening projection* allows us to generate a sequence of elements for each element from the source collection and merges all the returned sequences. As we'll soon see, it's an essential operation that can be used to define other processing operations including projection and filtering. The unique thing about flattening projection is that it lets us generate multiple output elements for each input element.

NOTE In LINQ libraries, this operation is called `SelectMany`. In query expressions it's represented by having more than one `from` clause. The name reflects the fact that it's similar to the `Select` operation with the exception that we can return many elements for each item in the source. The F# library's equivalent function is `Seq.collect`. Here, the name suggests the implementation—it's like calling the `Seq.map` function to generate a sequence of sequences and then calling `Seq.concat` to concatenate them.

We'll start off by looking at an example where this is needed, which means that we couldn't write the example just using higher-order functions from the previous section. We'll start by looking at the implementation that uses F# sequence expressions, then we'll gradually change the code to use flattening projection.

FLATTENING PROJECTIONS IN SEQUENCE EXPRESSIONS

Suppose we have a list of tuples, each of which contains a city's name and the country it's in, and we have a list of cities selected by a user. We can represent sample data for this scenario like this:

```
let cities = [ ("New York", "USA"); ("London", "UK");
               ("Cambridge", "UK"); ("Cambridge", "USA") ]
let entered = [ "London"; "Cambridge" ]
```

Now suppose we want to find the countries of the selected cities. We could iterate over the entered cities and find each city in the `cities` list to get the country. You can probably already see the problem with this approach: there is a city named Cambridge in both the United Kingdom and the United States, so we need to be able to return multiple records for a single city. You can see how to write this using two nested `for` loops in a sequence expression in listing 12.10.

Listing 12.10 Joining collections using sequence expressions (F# Interactive)

```
> seq { for name in entered do                    (1)
          for (n, c) in cities do                 (2)
            if (n = name) then
              yield sprintf "%s (%s)" n c };;     (3)
val it : seq<string> =
  seq [ "London (UK)"; "Cambridge (UK)"; "Cambridge (USA)" ]
```

Both countries returned for Cambridge

The outer `for` loop iterates over the entered names (1), and the nested loop iterates over the list of known cities (2). This means that inside the body of the nested loop, we'll get a chance to compare whether the name of each entered city is equal to a name of each of the known cities. The code that's nested in these two loops (3) uses the `yield` statement to produce a single item if the names are the same. If the names aren't the same, it doesn't yield any elements.

In database terminology, this operation could be explained as a join. We're joining the list of entered names with the list containing information about cities using the name of the city as the key. Writing the code using sequence expressions is quite easy, and it's the preferred way for encoding joins in F#.

We mentioned that any sequence expression can be encoded using the flattening projection operation, so let's see how we can rewrite the previous example using `Seq.collect` explicitly. You wouldn't do this in practice, but it will be invaluable when we explore defining our own alternative workflows similar to sequence expressions.

USING FLATTENING PROJECTIONS DIRECTLY

First, let's see what the flattening projection looks like. As usual, the initial step in understanding how the function works is to examine its type signature. Figure 12.2 compares the signatures of `Seq.map` (ordinary projection) and `Seq.collect` (flattening projection).

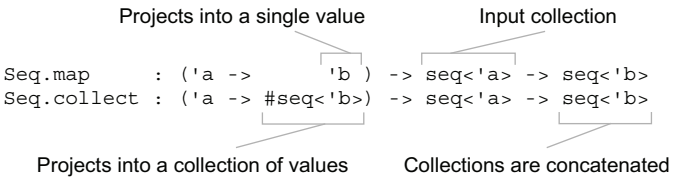

Figure 12.2 Projection returns a single element for each input element while flattening projection can return any collection of elements.

As a reminder, the # symbol in the part of the type signature describing the projection function passed to `collect` means that the return type of the function doesn't have to be exactly the `seq<'b>` type. We talked about types declared using the # symbol in the previous chapter—the actual type used in place of the `#seq<'b>` can be any type implementing the `seq<'b>` interface. This means that we can return a sequence, also an F# list, an array, or even our own collection type.

Now, let's see how we can rewrite the previous example using the `Seq.collect` function. The general rule is that we can replace each use of the `for` loop inside a sequence expression with a single call to `Seq.collect`. This is exactly how the F# compiler compiled sequence expressions in early versions. Since our example has two nested loops, we'll do the transformation in two steps. In listing 12.11, we start by replacing the outer loop.

Listing 12.11 Replacing the outer loop with flattening projection (F# Interactive)

```
> entered |> Seq.collect (fun name ->                    ❶
    seq { for (n, c) in cities do                         ❷
          if (n = name) then
            yield sprintf "%s (%s)" n c });;
val it : seq<string> =
  seq [ "London (UK)"; "Cambridge (UK)"; "Cambridge (USA)" ]
```

We replaced the outer loop with a flattening projection, so listing 12.11 calls `Seq.collect` and gives it a list of cities entered by the user as input ❶. The lambda function we provide takes the name of a single city and iterates over the collection of all known cities to find the country or countries containing that city ❷. The searching is implemented using a sequence expression from listing 12.10 with the outer loop deleted. The lambda function we use returns a sequence with information about cities with the

specified name, and the `Seq.collect` function concatenates all of these to return a single sequence with results.

Now we have a combination of function call and a sequence expression, so let's see how we can replace the inner `for` loop to complete the translation. We could implement the nested part with `Seq.filter` and `Seq.map`, or, even better, with `Seq.choose`, which lets us combine the two operations into one. We're showing what the compiler would do and it would naïvely follow the rule to replace every `for` loop with a flattening projection. Listing 12.12 shows the same processing code again, but using only `Seq.collect` calls.

Listing 12.12 Replacing both loops with flattening projection (F# Interactive)

```
> entered |> Seq.collect (fun name ->
    cities |> Seq.collect (fun (n, c) ->          (1)
      if (n = name) then
        [ sprintf "%s (%s)" n c ]                 (2)
      else [] ));;
val it : seq<string> =
  seq [ "London (UK)"; "Cambridge (UK)"; "Cambridge (USA)" ]
```

The outer call is the same as in listing 12.11, but inside the lambda function we now perform another call to `Seq.collect` (1). The nested call iterates over all the cities and for each city returns either an empty list if the name of the city doesn't match the entered name or a list containing a single element when the name matches. As you can see, we've replaced the use of `yield` with code that returns a list containing a single element. If the code contained multiple yields, we'd return a longer list. It's also worth noting that we had to add an `else` clause that returns an empty list; inside sequence expressions this is implicit.

Even though the `Seq.collect` function is sometimes useful when writing sequence-processing code using higher-order functions, its real importance is that it can be used to translate arbitrary sequence expression into function calls. As we'll see shortly, sequence expressions are one specific example of a more general F# construct and the flattening projection is the primitive operation that defines how sequence expressions work. We'll also see that the translation we demonstrated in this section works in a similar way for other computations that we can define for our own values.

We mentioned earlier that we could use projection and filtering to implement the nested loop (2), but as you can see, `for` loops in sequence expressions are expressive enough to implement the projection, filtering, and joins we've seen in this section. Now, let's examine the same operation in C#.

USING FLATTENING PROJECTIONS IN C#

The LINQ operator analogous to the `collect` function is called `SelectMany`. Differences exist between the two versions, because LINQ has different requirements. While F# sequence expressions can be expressed using just the `collect` function, LINQ queries use many other operators, so they need different ways for sequencing operations.

Let's again start by looking at the usual syntax and then examine how it's translated to the explicit syntax using extension methods. We'll use the same data as in the

previous F# example. The list of cities with the information about the country contains instances of a class `CityInfo` with two properties, and the list of entered names contains only strings. Listing 12.13 shows a LINQ query that we can write to find countries of the entered cities.

Listing 12.13 Searching for country of entered cities using a query (C#)

```
var q =
  from e in entered                                    ❶
  from known in cities                                 ❷
  where known.City == e
  select string.Format("{0} ({1})", known.City, known.Country);   ❸
```

The query expresses exactly the same idea as we did in the previous implementations. It iterates over both of the data sources (❶ and ❷), which gives us a cross join of the two collections and then yields only records where the name entered by the user corresponds to the city name in the "known city" list; finally it formats the output ❸.

In C# query expression syntax, we can also use the `join` clause, which directly specifies keys from both of the data sources (in our case, this would be the value `e` and the `known.City` value). This is slightly different: `join` clauses can be more efficient, but multiple `from` clauses are more flexible. In particular, the second sequence we generate can depend on which item of the first sequence we're currently looking at.

As we said earlier, query expressions are translated into normal member invocations. Any `from` clause in a query expression after the first one is translated into a call to `SelectMany`. Listing 12.14 shows the translation as it's performed by the C# compiler.

Listing 12.14 Query translated to explicit operator calls (C#)

```
var q = entered
  .SelectMany(
    e => cities,                                       ❶
    (e, known) => new { e, known })                    ❷
  .Where(tmp => tmp.known.City == tmp.e)
  .Select(tmp => String.Format("{0} ({1})",
    tmp.known.City, tmp.known.Country));
```

Filters, formats output

Unlike in F#, where the `if` condition was nested inside the two `for` loops (flattening projections), the operations in C# are composed in a sequence without nesting. The processing starts with the `SelectMany` operator that implements the join; the filtering and projection are performed using `Where` and `Select` at the end of the sequence.

The first lambda function ❶ specifies a collection that we generate for every single item from the source list. This parameter corresponds to the function provided as an argument to the F# `collect` function. In the query, we return all the known cities, so the operation performs only joining, without any filtering or further processing. The second parameter ❷ specifies how to build a result based on an element from the original sequence and an element from the newly generated sequence returned by the function. In our example, we build an anonymous type that contains both items so we can use them in later query operators.

In F#, all the processing is done inside the filtering projection, so we return only the final result. In C# most of the processing is done later, so we need to return both elements combined into one value (using an anonymous type), so that they can be accessed later. In general, the first `from` clause specifies the primary source of the query, and if we add more `from` clauses, they're joined with the original source using the `SelectMany` operator. Any further operators such as `where` and `select` are appended to the end and work with the joined data source. This is different from the F# translation, because in F# both filtering and projection are nested in the innermost call to `Seq.collect`.

Understanding how the translation works isn't that important, but we'll need to know a little bit about the translation in the next section. We'll see that F# sequence expressions represent a more general idea that can be also partly expressed using LINQ queries. The flattening projection we've just been looking at plays a key role.

12.4 Introducing alternative workflows

Computation expressions is an F# feature that has been partly inspired by Haskell *monads*. Monads have an unfortunate reputation for being brain-bustingly difficult—but don't worry. We'll look at implementing an interesting set of techniques that let us work with `Option<T>` values nicely in C#. We'll see how to do a similar thing in F# and also how to write simple logger with a nice syntax in F#.

We could do all of this without even mentioning the word monad. Since the book is about functional programming in a more general sense, we want to give you more than an overview of all F# features. We'll occasionally explain some of the underlying terminology, which can be helpful if you want to look at other programming languages. You can always skip the part that sounds complicated and move on to the next example. You may be surprised to know we've already explained monads in this chapter. In fact, you've probably used them before even picking up this book: LINQ is based on monads too.

In section 6.7 we looked at the `bind` function for option values, and you learned that a similar operator makes sense for lists as well. Its name in the standard F# libraries is `List.collect`, so you won't be surprised to hear that `Seq.collect` is also a form of *bind* operator, but this time working with sequences. In this chapter, we've seen that this operation is important in LINQ queries and F# sequence expressions. Again, here are the type signatures of the three operations:

```
Option.bind  : ('a -> option<'b>) -> option<'a> -> option<'b>
List.collect : ('a -> list<'b>)   -> list<'a>   -> list<'b>
Seq.collect  : ('a -> #seq<'b>)   -> seq<'a>    -> seq<'b>
```

The function provided as the argument specifies what to do with each value (of type `'a`) contained in the value given as the second argument. For lists and sequences, that means the function will be called for each element of the input sequence. For option values, the function will be executed at most once, only when the second argument is `Some` value. Just a reminder: the option value can be viewed as a list of either zero or one elements.

You may already know that you can create your own implementation of LINQ query operators and use them to work with your own collection types. Nothing limits us to using the query syntax only for working with collections.

12.4.1 Customizing query expressions

In principle, we can use queries to work with any type that supports the *bind* operation. This is the standard name used in functional programming for functions with type signatures of the form shown in the previous section. Technically speaking, we need to implement methods that are used by the C# compiler when translating the query expression into standard calls. We'll implement these for the `Option<T>` type in section 12.6. The type doesn't implement `IEnumerable<T>`, so the standard query operators can't be used.

Let's first consider what the meaning of a query applied to option types would be. Listing 12.15 shows two queries. The one on the left one works with lists and the one on the right works with option types. We're using two simple functions to provide input: the `ReadIntList` function reads a list of integers (of type `List<int>`) and `TryReadInt` returns an option value (of type `Option<int>`).

Listing 12.15 Using queries with lists and option values (C#)

```
var list =
    from n in ReadIntList()
    from m in ReadIntList()
    select n * m;
```

```
var option =
    from n in TryReadInt()
    from m in TryReadInt()
    select n * m;
```

The queries are the same with the exception that they work with different types of data, so they use different query operator implementations. Both read two different inputs and return multiples of the entered integers. Table 12.1 gives examples of inputs to show what the results would be.

Table 12.1 Results produced by queries working with lists and option values for different possible inputs

Type of values	Input #1	Input #2	Output
Lists	`[2; 3]`	`[10; 100]`	`[20; 200; 30; 300]`
Options	`Some(2)`	`Some(10)`	`Some(20)`
Options	`Some(3)`	`None`	`None`
Options	`None`	Not required	`None`

For lists, the query performs a cross join operation (you can imagine two nested `for` loops as in the F# sequence expression). It produces a single sequence consisting of a single entry for each combination of input values. For option values there are three possibilities.

- When the first input is a value, we need to read the second one. Then, the following two cases can occur depending on the second input:
  - If the second input is also a value, the result is again `Some` value containing the result of the multiplication.
  - If the second input is `None` we don't have values to multiply, so the query returns `None`.
- When the first input is `None`, we know the result without needing the second input. The whole query is executed lazily, so we don't have to read the second input: the `TryReadInt` function will be called only once.

As you can see, query expressions give us a convenient way of working with option values. Listing 12.15 is definitely easier to write (and read) than the equivalent code we saw in chapter 6, where we used higher-order functions explicitly. We'll see how to implement all the necessary query operators later in this chapter, but let's first look at similar syntax in F#.

12.4.2 Customizing the F# language

So far, we've talked about sequence expressions, which were denoted using the `seq` identifier preceding the block of code enclosed in curly braces. However, F# allows us to create our own identifiers that give a special meaning to a block of code. In general, this feature is called *computation expressions* and sequence expressions are a single special case that's implemented in the F# core and optimized by the compiler.

We've seen that computation expressions can contain standard language constructs such as `for` loops, but also additional constructs like `yield`. The identifier that precedes the block gives the meaning to these constructs in the same way query operators (such as `Select` and `Where` extension methods) specify what a LINQ query does. This means that we can create a customized computation expression for working with option values. We could work with option values using the `for` construct, but F# gives us a nicer way to customize the expression. You can see these alternative approaches in listing 12.16. The first version uses syntax similar to sequence expressions; the second is a more natural way of writing the same thing.

Listing 12.16 Computation expressions for working with option values (F#)

```
// Value binding using customized 'for' primitive
option {
  for n in tryReadInt() do
    for m in tryReadInt() do
      yield n * m
}

// Value binding using special 'let!' primitive
option {
  let! n = tryReadInt()
  let! m = tryReadInt()
  return n * m
}
```

The behavior of all custom primitives that occur inside the computation expression (such as `for`, `yield`, and `let!`) is determined by the `option` identifier that defines what kind of computation expression we're writing. Now you can see that a sequence expression is just a special case that's defined by the `seq` identifier. We'll see how to define identifiers in section 12.5, but first let's look at the two examples in listing 12.16.

The first version closely resembles the LINQ query in listing 12.15. Each `for` loop can be executed at most once. When the option value contains a value, it will be bound to the symbol `n` or `m`, respectively, and the body of the loop will execute. Developers have an expectation that loops work with collections and not option values, so the constructs `for` and `yield` are usually only used with sequences. When we create a computation expression that works with other types of values, we'll use the later syntax. The second version uses two more computation expression primitives. The first one is `let!` , which represents a customized value binding.

In both versions, the type of values `n` and `m` is `int`. The customized value binding unwraps the actual value from the value of type `option<int>`. It may fail to assign the value to the symbol when the value returned from `TryReadInt` is `None`. In that case, the whole computation expression will immediately return `None` without executing the rest of the code. The second nonstandard primitive in the expression is `return`. It specifies how to construct an option value from the value. In listing 12.16, we give it an `int` value and it constructs the result, which has a type `option<int>`.

The concepts we've just seen can be regarded as a functional design pattern. We can use F# computation expressions without understanding all the details of the pattern. If you want to learn how to define your own computation expressions, it's useful to learn about the background concepts and terminology. The sidebar "Computation expressions and monads" discusses the pattern in more detail and explains how it relates to Haskell monads.

Computation expressions and monads

As we mentioned earlier, computation expressions in F# are an implementation of an idea called monads that has proven useful in Haskell. Monad refers to a term from mathematics, but F# uses a different name that better reflects how the idea is used in the F# language.

When defining a computation expression (or monad), we always work with a generic type such as `M<'a>`. This is often called a *monadic type*, and it specifies the meaning of the computation. This type can augment the usual meaning of the code we write. For example, the `option<'a>`, which we've just seen, augments the code with the possibility of returning an undefined value (`None`). Sequences also form a monad. The type `seq<'a>` augments the code with the ability to work with multiple values.

Each computation expression (or monad) is implemented using two functions–`bind` and `return`. `bind` allows us to create and compose computations that work with values of the monadic type. In listing 12.16, the `bind` operation was used whenever we used the `let!` primitive. `return` is used to construct a value of the monadic type.

(continued)

It's worth noting that sequence expressions are also an instance of a monad. For sequences, the bind operation is `Seq.collect`, even though in sequence expressions we don't use the `let!` syntax and instead use the more comfortable `for` loop syntax. Listing 12.16 shows that these two are closely related. The `return` operation for sequences is creating a sequence with a single element. Inside sequence expressions, this can be written using a more natural `yield` primitive.

In the next section, we'll look at the simplest possible custom computation. We'll implement it in both C# and F# to explain what the monadic type is and how the `bind` and `return` operations look.

12.5 First steps in custom computations

The example in this section doesn't have any real-world benefit, but it demonstrates the core concepts. The first task in designing a custom computation is to consider the type that represents the values produced by the computation.

12.5.1 Declaring the computation type

The type of the computation (the monadic type in Haskell terminology) in this example will be called `ValueWrapper<T>`, and it will simply store the value of the generic type parameter `T`. It won't augment the type with any additional functionality. This means that the computation will work with standard values, but we'll be able to write the code using query expressions in C# and computation expressions in F#.

Listing 12.17 shows the type declaration in both C# and F#. In C#, we'll create a simple class, and in F# we'll use a simple discriminated union with only a single case.

Listing 12.17 Value of the computation in C# and F#

```
// C# class declaration
class ValueWrapper<T> {
  public ValueWrapper(T value) {
    this.Value = value;
  }
  public T Value { get; private set; }      (1)
}

// F# discriminated union type
type ValueWrapper<'a> =
  | Value of 'a                             (2)
```

The C# class is a simple immutable type that stores the value of type `T` (1). The use of a discriminated union with a single case (2) in F# is also interesting. It allows us to create a named type that is easy to use. As we'll see shortly, we can access the value using pattern matching (using the `Value` discriminator). Pattern matching with this type can never fail because there's only a single case. This lets us use it directly inside

value bindings, which will prove useful when we implement the computation algorithm. First let's examine the kinds of computation we'll be able to write with this new type.

12.5.2 Writing the computations

C# query expressions and F# computation expressions allow us to use functions that behave in a nonstandard way (by returning some monadic value) as if they returned an ordinary value. The computation type we're using in this section is `ValueWrapper<T>`, so primitive functions will return values of type `ValueWrapper<T>` instead of only `T`.

These functions can be implemented either using another query or computation expression, or directly by creating the value of the computation type. Some computation expressions can encapsulate complicated logic, so it may be difficult to create the value directly. In that case, we'd typically write a small number of primitives that return the computation type and use these primitives to implement everything else. However, constructing a value of type `ValueWrapper<T>` is not difficult. The following code shows how to implement a method in C# that reads a number from the console and wraps it inside this computation type:

```
ValueWrapper<int> ReadInt() {
  int num = Int32.Parse(Console.ReadLine());
  return new ValueWrapper<int>(num);
}
```

The method reads a number from the console and wraps it inside the `ValueWrapper<T>` type. The F# version is equally simple, so we won't discuss it here. The important point is that these primitive functions are the only place where we need to know anything about the underlying structure of the type. For the rest of the computation, we'll need to know only that the type supports all the primitives (most importantly `bind` and `return`) needed to write a query or computation expression.

Once we define the value identifier that denotes a computation expression in F# (section 12.5.3) and implement the necessary extension methods in C# (section 12.5.4), we'll be able to work with values of the type easily. Note that the type we're working with doesn't implement the `IEnumerable<T>` interface. The query syntax and computation expression notation works independently from sequences. We'll define the meaning of the code by implementing a couple of methods for working with the `ValueWrapper<T>` type. Listing 12.18 shows a snippet that reads two integers using the primitive and performs a calculation with them.

Listing 12.18 Calculating with computation values in C# and F#

C#

```
var v =
   from n in ReadInt()          (1)
   from m in ReadInt()
   let add = n + m
   let sub = n - m
   select add * sub;            (3)
```

F#

```
value {
   let! n = readInt()           (2)
   let! m = readInt()
   let add = n + m
   let sub = n - m
   return add * sub }           (4)
```

In C# we're using the `from` clause to unwrap the value ❶. In F#, the same thing is achieved using the customized value binding ❷.

Once the calculation is done, we again wrap the value inside the computation type. In C#, we're using a `select` clause ❸, and in F# we're using the `return` primitive ❹.

As you can see, the structure of the code in C# and F# is quite similar. The code doesn't have any real-world use, but it will help us understand how nonstandard computations work. The only interesting thing is that it allows us to write the code in C# as a single expression using the `let` clause, which creates a local variable. This clause behaves very much like the F# `let` binding, so the whole code is a single expression.

In the following discussion, we'll focus more on the F# version, because it will make it simpler to explain how things work. The query expression syntax in C# is tailored to writing queries, so it's harder to use for other types of computations. We'll get back to C# once we've implemented the F# computation expression.

You can see that listing 12.18 is using only two primitives. The `bind` primitive is used when we call the computation primitives ❷, and the `return` primitive is used to wrap the result in the `ValueWrapper<int>` type. The next question you probably have is how the F# compiler uses these two primitives to interpret the computation expression and how can we implement them.

12.5.3 Implementing a computation builder in F#

The identifier that precedes the computation expression block is an instance of a class that implements the required operations as instance members. Numerous operations are available: we don't have to support them all. The most basic operations are implemented using the `Bind` and `Return` members. When the F# compiler sees a computation expression such as the one in listing 12.18, it translates it to F# code that uses these members. The F# example is translated to the following:

```
value.Bind(ReadInt(), fun n ->
  value.Bind(ReadInt(), fun m ->
    let add = n + m
    let sub = n - m
    value.Return(n * m) ))
```

Whenever we use the `let!` primitive in the computation, it's translated to a call to the `Bind` member. This is because the `readInt` function returns a value of type `ValueWrapper<int>`, but when we assign it to a symbol, `n`, using the customized value binding, the type of the value will be `int`. The purpose of the `Bind` member is to unwrap the value from the computation type and call the function that represents the rest of the computation with this value as an argument.

You can compare the behavior of the `let!` primitive with the standard value binding written using `let`. If we wrote `let n = readInt()` in listing 12.18, the type of `n` would be `ValueWrapper<int>` and we'd have to unwrap it ourselves to get the integer. In this case, we could use the `Value` property, but there are computations where the value is hidden and the only way to access it is via the `Bind` member.

The fact that the rest of the computation is transformed into a function gives the computation a lot of flexibility. The `Bind` member could call the function immediately, or it could return a result without calling the function. For example, when we're working with option values and the first argument to the `Bind` member is the `None` value, we know what the overall result will be (`None`) regardless of the function. In this case, the `bind` operation can't call the given function, because the option value doesn't carry an actual value to use as an argument. In other cases, the `bind` operation could effectively remember the function (by storing it as part of the result) and execute it later. We'll look at an example of this in the next chapter.

Our example also shows that multiple `let!` constructs are translated into nested calls to the `Bind` member. This is because the function given as the last argument to this member is a continuation, meaning that it represents the rest of the computation. The example ends with a call to the `Return` member, which is created when we use the `return` construct.

Understanding the type signatures of bind and return

The types of the two operations that we need to implement for various computation expressions will always have the same structure. The only thing that will vary in the following signature is the generic type `M`:

```
Bind   : M<'T> * ('T -> M<'R>) -> M<'R>
Return : 'T -> M<'T>
```

In our previous example, the type `M<'T>` is the `ValueWrapper<'T>` type. In general, the `bind` operation needs to know how to get the value from the computation type in order to call the specified function. When the computation type carries additional information, the `bind` operation also needs to combine the additional information carried by the first argument (of type `M<'T>`) with the information extracted from the result of the function call (of type `M<'R>`) and return them as part of the overall result. The `return` operation is much simpler, because it constructs an instance of the monadic type from the primitive value.

In the previous example, we used an identifier `value` to construct the computation. The identifier is an ordinary F# value and it is an instance of object with specific members. The object is called a *computation builder* in F#. Listing 12.19 shows a simple builder implementation with the two required members. We also need to create an instance called `value` to be used in the translation.

Listing 12.19 Implementing computation builder for values (F#)

```
type ValueWrapperBuilder() =
  member x.Bind(Value(v), f) = f(v)        (1)
  member x.Return(v) = Value(v)            (2)

let value = new ValueWrapperBuilder()
```

Creates instance of builder

The `Bind` member ❶ first needs to unwrap the actual value from the `ValueWrapper<'T>` type. This is done in the parameter list of the member, using the `Value` discriminator of the discriminated union as a pattern. The actual value will be assigned to the symbol `v`. Once we have the value, we can invoke the rest of the computation `f`. The computation type doesn't carry any additional information, so we can return the result of this call as the result of the whole computation. The `Return` member is trivial, because it wraps the value inside the computation type.

Using the `value` declared in this listing, we can now run the computation expression from listing 12.18. F# also lets us use computation expressions to implement the `readInt` function as well. We need to wrap the result in an instance of `ValueWrapper<int>` type, which can be done using the `return` primitive:

```
> let readInt() = value {
    let n = Int32.Parse(Console.ReadLine())
    return n };;
val readInt : unit -> ValueWrapper<int>
```

This function doesn't need the `bind` operation, because it doesn't use any values of type `ValueWrapper<'T>`. The whole function is enclosed in the computation expression block, which causes the return type of the function to be `ValueWrapper<int>` instead of just `int`. If we didn't know anything about the `ValueWrapper<'T>` type, the only way to use the function would be to call it using the `let!` primitive from another computation expression. The important point is that computation expressions give us a way to build more complicated values from simpler values by composing them. The monadic value is then a bit like a black box that we can compose, but if we want to look inside, we need some special knowledge about the monadic type. In the case of `ValueWrapper<'T>`, we need to know the structure of the discriminated union.

Writing a function like `readInt` in C# using query syntax isn't possible, because queries need to have some input for the initial `from` clause. There is a `let` clause inside the query syntax, which roughly corresponds to a `let` binding in a computation expression, but a query can't start with it. Nevertheless, as we've seen in listing 12.15, there are many useful things that we can write using queries, so let's look at adding query operators for our `ValueWrapper<'T>` type.

12.5.4 Implementing query operators in C#

We've seen how the C# queries are translated to method calls in listing 12.14 when we were talking about sequences and when we analyzed the `SelectMany` operation. We'll support only queries that end with the `select` clause and ignore cases that are useful only for collections such as grouping. This means that we'll need to implement the `Select` extension method.

We said earlier that the second and subsequent `from` clauses are translated into a call to the `SelectMany` method. When writing computations using queries, we use the `from` clause in a similar way to the F# `let!` construct to represent a nonstandard value binding, so we'll use it quite often. This means that we'll need to implement the `SelectMany` operation for our `ValueWrapper<'T>` type as well.

You already know that the `SelectMany` method corresponds to the `bind` function, but it's slightly more complicated because it takes an additional function that we'll need to run before returning the result. The `Select` method is simpler, but we'll talk about that after looking at the code. Listing 12.20 shows the implementation of both of the primitives.

Listing 12.20 Implementing query operators (C#)

```
static class ValueWrapperExtensions {
  public static ValueWrapper<R> Select<T, R>
      (this ValueWrapper<T> source,
       Func<T, R> selector) {
    return new ValueWrapper<R>(selector(source.Value));
  }
  public static ValueWrapper<R> SelectMany<T, V, R>
      (this ValueWrapper<T> source,
       Func<T, ValueWrapper<V>> valueSelector,
       Func<T, V, R> resultSelector) {
    var newVal = valueSelector(source.Value);                    ❶
    var resVal = resultSelector(source.Value, newVal.Value);     ❷
    return new ValueWrapper<R>(resVal);                          ❸
  }
}
```

Projects value using given function

Both methods are implemented as extension methods. This means that C# will be able to find them when working with values of type `ValueWrapper<T>` using the standard dot notation, which is used during the translation from the query syntax. The `Select` operator implements projection using the given function, so it only needs to access the wrapped value, run the given function, then wrap the result again.

The `SelectMany` operator is confusing at first, but it's useful to look at the types of the parameters. They tell us what arguments we can pass to what functions. The implementation starts off like the F# `Bind` member by calling the function given by the second argument after unwrapping the first argument ❶. We also need to combine the value from the source with the value returned by the first function. To obtain the result, we call the second function ❷, giving it both of the values. Finally, we wrap the result into the computation type ❸ and return from the method.

After implementing the operators, the query expression in listing 12.18 will compile and run. The computation type that we created in this section doesn't augment the computation with any additional aspects. The very fact that it was so simple makes it a good template for the standard operations. We can implement more sophisticated monadic types by starting with this template and seeing where we need to change it. We'll put this idea into practice now by implementing similar operators for the option type.

12.6 Implementing computation expressions for options

We used option values as an example in section 12.4 when we introduced the idea of creating nonstandard computations using LINQ queries and F# computation expressions. The code we wrote worked with option values as if they were standard values, with a customized value binding to read the actual value. Now that we've seen how

computation expressions are translated, we know that our `Bind` member will receive a value and a lambda function. With our option type computation expression, we only want to execute the lambda expression if the value is `Some(x)` instead of `None`. In the latter case, we can return `None` immediately.

To run the earlier examples, we'll need to implement LINQ query operators in C# and the `option` computation builder in F#. Again we'll start with the F# version. Listing 12.21 shows an F# object type with two members. We've already implemented the `Option.bind` function in chapter 6, but we'll reimplement it here to remind you what a typical `bind` operation does.

Listing 12.21 Computation builder for option type (F#)

```
type OptionBuilder() =
  member x.Bind(opt, f) =
    match opt with                    ❶
    | Some(value) -> f(value)         ❷
    | _           -> None             ❸
  member x.Return(v) = Some(v)        <- Wraps actual value

let option = new OptionBuilder()
```

The `Bind` member starts by extracting the value from the option given as the first argument. This is similar to the `Bind` we implemented earlier for the `ValueWrapper<'T>` type. Again we're using pattern matching ❶, but in this case, the value may be missing so we're using the `match` construct. If the value is defined, we call the specified function ❷. This means that we bind a value to the symbol declared using `let!` and run the rest of the computation. If the value is undefined, we return `None` as the result of the whole computation expression ❸.

The `Return` member takes a value as an argument and has to return a value of the computation type. In our example, the type of the computation is `option<'a>`, so we wrap the actual value inside the `Some` discriminator.

To write the corresponding code in C# using the query syntax, we'll need to implement `Select` and `SelectMany` methods for the `Option<T>` type we defined in chapter 5. Listing 12.22 implements the two additional extension methods so that we can use options in query expressions. This time we'll use the extension methods we wrote in chapter 6 to make the code simpler.

Listing 12.22 Query operators for option type (C#)

```
static class OptionExtensions {
  public static Option<R> Select<S, R>
      (this Option<S> source, Func<S, R> selector) {
    return source.Map(selector);                          ❶
  }
  public static Option<R> SelectMany<S, V, R>
      (this Option<S> source,
       Func<S, Option<V>> valueSelector,
       Func<S, V, R> resultSelector) {
    return source.Bind(sourceValue =>                     ❷
```

```
        valueSelector(sourceValue).Map(resultValue =>          ❸
          resultSelector(sourceValue, resultValue)));
    }
}
```

The `Select` method should apply the given function to the value carried by the given option value if it contains an actual value. Then it should again wrap the result into an option type. In F# the function is called `Option.map`, and we used an analogous name (`Map`) for the C# method. If we'd looked at LINQ first, we'd probably have called the method `Select` from the beginning, but the simplest solution is to add a new method that calls `Map` ❶.

`SelectMany` is more complicated. It's similar to the `bind` operation, but in addition it needs to use the extra function specified as the third argument to format the result of the operation. We wrote the C# version of the `bind` operation in chapter 6, so we can use the `Bind` extension method in the implementation ❷. To call the formatting function `resultSelector`, we need two arguments: the original value carried by the option and the value produced by the binding function (named `selector`). We can do this by adding a call to `Map` at the end of the processing, but we need to place this call inside the lambda function given to the `Bind` method ❸. This is because we also need to access the original value from the source. Inside the lambda function, the original value is in scope (the variable named `sourceValue`), so we can use it together with the new value, which is assigned to the variable `resultValue`.

This implementation is a bit tricky, but it shows that many things in functional programming can be composed from what we already have. If you tried to implement this on your own, you'd see that the types are invaluable helpers here. You might start just by using the `Bind` method, but then you'd see that the types don't match. You'd see what types are incompatible and if you looked at what functions are available, you'd discover what needs to be added in order to get the correct types. At the risk of repeating ourselves: the types in functional programming are far more important and tell you much more about the correctness of your program.

Using the new extension methods, we can run the examples from section 12.3. In F#, we didn't provide an implementation for the `yield` and `for` primitives, so only the version using `return` and `let!` will work. This is intentional, because the first set of primitives is more suitable for computations that work with sequences of one form or another. We still need to implement the `TryReadInt` method (and the similar F# function). These are really simple, because they need to read a line from the console, attempt to parse it, and return `Some` when the string is a number or `None` otherwise.

The identity and maybe monads

The two examples that we've just seen are well known in the Haskell world. The first one is the *identity monad*, because the monadic type is the same as the actual type of the value, just wrapped inside a named type. The second example is the *maybe monad*, because `Maybe` is the Haskell type name that corresponds to the `option<'a>` type in F#.

(continued)

The first example was mostly a toy example to demonstrate what we need to do when implementing computations, but the second one can be useful when you're writing code that's composed from a number of operations, each of which can fail. When you analyze the two examples, you can see how important the *monadic type* is. Once you understand the type, you know what makes the computation nonstandard.

So far the examples have been somewhat abstract. Our next section is a lot more concrete; we'll add automatic logging into our code.

12.7 Augmenting computations with logging

Logging can be usually implemented using global mutable state. However, what if we wanted to avoid using global mutable state and keep the program purely functional? One option we'd have is to pass the state of the logger as an additional argument to every function we call. Implementing that would be quite difficult. (And imagine if we decided to add another parameter to the state!)

To solve this problem, we can create a custom computation type that enables logging and hides the state of the logger inside the computation type. This is similar to a technique that Haskell uses to embed working with state (such as filesystems) in a purely functional language without any side effects. The example that we'll implement relies on the fact that we can surround any piece of standard F# code with the computation expression block. As such, it's not feasible to use C# for this example. We'll start off by designing the computation type (monadic type) we need to allow simple logging.

12.7.1 Creating the logging computation

The computation will produce a value and will allow us to write messages to a local logging buffer. This means that the result of the computation will be a value and a list of strings for the messages. Again we'll use a discriminated union with a single discriminator to represent the type:

```
type Logging<'T> =
  | Log of 'T * list<string>
```

This type is quite similar to the `ValueWrapper<'a>` example we discussed earlier, but with the addition of an F# list of the messages written to the log. Now that we have the type, we can implement the computation builder. As usual, we'll need to implement the `Bind` and `Return` members. We'll also implement a new member called `Zero`, which enables us to write computations that don't return any value. We'll see how that's used later.

The implementation of the builder is shown in listing 12.23. The most interesting is the `Bind` member, which needs to concatenate the log messages from the original value and the value generated by the rest of the computation (which is the function given as an argument to the `Bind` member).

Listing 12.23 Computation builder that adds logging support (F#)

```
type LoggingBuilder() =
  member x.Bind(Log(value, logs1), f) =                    ❶
    let (Log(newValue, logs2)) = f(value)                  ❷
    Log(newValue, logs1 @ logs2)                           ❸
  member x.Return(value) =
    Log(value, [])                                         <-- Augments value with empty log
  member x.Zero() =
    Log((), [])                                            <-- Returns no value with empty log

let log = new LoggingBuilder()
```

As with our other examples, the most difficult part is implementing the `Bind` member. Our logging type follows all the normal steps, including a third one that was missing for both the `option` and `ValueWrapper` types:

1. We need to unwrap the value. Since we're using a single case discriminated union, we can use pattern matching in the argument list of the member ❶.
2. We need to call the rest of the computation if we have a value to do that. In listing 12.23, we always have the value, so we can run the given function ❷. We don't immediately return the result; instead we decompose it to get the new value and the log messages produced during the execution.
3. We've collected two buffers of log messages, so we need to wrap the new value and augment it with the new logger state. To create that new state, we concatenate the original message list with the new list that was generated when we called the rest of the computation ❸. This is written using a list concatenation operator (`@`).

The `Return` and `Zero` members are simple. `Return` needs to wrap the actual value into the `Logging<'T>` type, and the `Zero` represents a computation that doesn't carry any value (meaning that it returns a unit). In both cases, we're creating a new computation value, so the primitives return an empty logging buffer. All the log messages will be produced in other ways and appended in the `Bind` member. If you look at the code we have so far, there's no way we could create a nonempty log! This means that we'll need to create one additional primitive to create a computation value containing a log message. We can write it as a simple function:

```
> let logMessage(s) =
    Log((), [s])
val logMessage : string -> Logging<unit>
```

The function creates a computation value that contains a `unit` as the value. More importantly, it also contains a message in the logging buffer, so if we combine it with another computation using `Bind`, we get a computation that writes something to the log. Now we can finally write code that uses the newly created logging computation.

12.7.2 Creating the logging computation

Listing 12.24 begins by implementing two helper functions for reading from and writing to the console. Both will also write a message to the log, so they'll be enclosed in the `log` computation block. We then use these two functions in a third function, to

show how we can compose nonstandard computations. In our previous examples, we used the `let!` primitive, but listing 12.24 introduces `do!` as well.

Listing 12.24 Logging using computation expressions (F# Interactive)

```
> let write(s) = log {
    do! logMessage("writing: " + s)
    Console.Write(s) }
val write : string -> Logging<unit>

> let read() = log {
    do! logMessage("reading")
    return Console.ReadLine() }
val read : unit -> Logging<string>

> let testIt() = log {
    do! logMessage("starting")
    do! write("Enter name: ")
    let! name = read()
    return "Hello " + name + "!" }
val testIt : unit -> Logging<string>

> let res = testIt();;
Enter name: Tomas

> let (Log(msg, logs)) = res;;
val msg : string = "Hello Tomas!"
val logs : string list = ["starting"; "writing: Enter name:"; "reading"]
```

1 Writes string to console and to log

2 Calls primitive logging function

3 Calls another computation expression

4 Uses customized value binding

If you run the code in the listing, it waits for a console input. This doesn't always work perfectly in the F# Interactive add-in in the Visual Studio, so you may want to run the code in the standalone console version of the shell. We use the new `do!` primitive in several places to call functions that return `Logging<unit>`. In this case, we want to write a nonstandard binding that executes the `Bind` member, because we want to concatenate logging messages. We can ignore the actual value, because it's `unit`. That's the exact behavior of the `do!` primitive. In fact, when we write `do! f()`, it's shorthand for writing `let! () = f()`, which uses the customized binding and ignores the returned `unit` value.

When implementing the computation builder, we added a member called `Zero`. This is used behind the scenes in listing 12.24. When we write a computation that doesn't return anything 1, the F# compiler automatically uses the result of `Zero` as the overall result. We'll see how this member is used when we discuss how the compiler translates the code into method calls.

If you look at the type signatures in the listing, you can see that the result type of all the functions is the computation type (`Logging<'T>`), which is the same as the result type of the `logMessage` function we implemented earlier. This demonstrates that we have two ways of writing functions of a nonstandard computation type. We can build the computation type directly (as we did in the `logMessage` function) or use the computation expression. The first case is useful mostly for writing primitives; the second approach is useful for composing code from these primitives or other functions.

You can see the composable nature of computation expressions by looking at the `testIt` function. It first uses the `do!` construct to call a primitive function implemented

directly ❷. Writing to the screen (and to the log) is implemented using a computation expression, but we call it in exactly the same way ❸. We're calling a function that returns a value and writes to the log, so we're using the customized binding with the `let!` keyword ❹.

In practice it isn't necessary to understand how the compiler translates the computation expression into method calls, but if you're curious, listing 12.25 shows the translation of the code from the previous listing, including the use of `Zero` member and the translations of `do!` primitive.

Listing 12.25 Translated version of the logging example (F#)

```
let write(s) =
  log.Bind(logMessage("writing: " + s), fun () ->
    Console.Write(s)
    log.Zero())

let read() =
  log.Bind(logMessage("reading"), fun () ->
    log.Return(Console.ReadLine()))

let testIt() =
  log.Bind(logMessage("starting"), fun () ->
    log.Bind(write("Enter name: "), fun () ->
      log.Bind(read(), fun name ->
        log.Return("Hello " + name + "!"))))
```

❶ **Automatically uses zero as result**

❷ **Translates multiple bindings into nested calls**

The `Zero` primitive is used only in the `write` function ❶ because this is the only place where we aren't returning any result from the function. In the other two functions, the innermost call is to the `Return` member, which takes a simple value as an argument and wraps it into a `LoggingValue<'T>` type that doesn't contain any log messages.

As you can see, when translating the computation expression, each use of `do!` or `let!` is replaced with a call to the `Bind` member ❷. If you recall our earlier discussion about sequence expressions, you can see the similarity now. In sequence expressions, every `for` loop was translated into a call to `Seq.collect`. We could take this analogy even further, because the `Return` primitive corresponds to creating a sequence containing a single element and the `Zero` primitive for sequence expressions would return an empty sequence.

There's one other interesting point that we want to highlight. If you look at the original code in listing 12.24, you can see that it looks just like ordinary F# code with a couple of added `!` symbols, which makes it easy to wrap an ordinary F# code into a computation expression.

12.7.3 Refactoring using computation expressions

In the previous chapter, we saw ways of refactoring functional programs. The last topic was laziness, which changes the way code executes without affecting the outcome of the program. In one sense, adding laziness can be also viewed as a refactoring technique. Computation expressions are similar in that they augment the code with an additional aspect without changing its core meaning.

TIP There's a close relationship between computation expressions and laziness. It's possible to create a computation expression that turns code into a lazily evaluated version, with a computation type of `Lazy<'T>`. You can try implementing the computation on your own: the only difficult part is writing the `Bind` member. We won't talk about this anymore here, but you can find additional information on the book's website.

The interesting thing is how easy it is to turn standard F# code into code that has nonstandard behavior. We have to enclose the code in a computation expression block and add calls to the primitives provided for the computation expression, such as the `logMessage` function we just implemented. When the code we're implementing is split between several functions, we have to change the calls to these functions from a usual call or usual value bindings into customized value bindings using either `let!` or `do!` primitives. When writing code that uses computation expressions in F#, the typical approach is to start with the standard version of the code, which is easier to write and test, then refactor it into an advanced version using computation expressions.

12.8 Summary

In the first part of the chapter, we talked about .NET sequences, as represented by the `IEnumerable<T>` type, also known as `seq<'a>` in F#. We started by looking at techniques for generating sequences, including higher-order functions, iterators, and F# sequence expressions. We saw that sequences are lazy, which allows us to create infinite sequences. We looked at a real-world example using an infinite sequence of colors to separate the code to draw of a chart from the code that generates the colors used in the chart.

Next we discussed how to process sequences. We wrote the same code using higher-order functions, the corresponding LINQ extension methods, C# query expressions, and F# sequence expressions. This helped us to understand how queries and sequence expressions work. One most important operation is the *bind* operation, which occurs in sequences as the `collect` function in F# and the `SelectMany` method in LINQ.

The same conceptual operation is available for many other types, and we saw how to create F# computation expressions that look like sequence expressions but work with other types. We provided two practical examples, implementing computation expressions for working with option types and storing log messages during execution. The same idea can be implemented in C# to some extent, with query expressions used in the place of computation expressions. The F# language features are more general, while C# query expressions are tailored to queries.

Perhaps the most difficult thing about using computation expressions is to identify when it's beneficial to design and implement them. In the next chapter, we'll look at one of the most important uses of F# computation expressions. It allows us to execute I/O operations without blocking the caller thread. This is particularly important when performing slow I/O such as reading data from the internet. Later we'll see how F# enables us to interactively process and visualize data, which is becoming an important task in the today's increasingly connected world.

PROGRAMMING

Real-World Functional Programming

Tomas Petricek with Jon Skeet Foreword by Mads Torgersen

Functional programming languages are good at expressing complex ideas in a succinct, declarative way. Functional concepts such as "immutability" and "function values" make it easier to reason about code—as well as helping with concurrency. The new F# language, LINQ, certain new features of C#, and numerous .NET libraries now bring the power of functional programming to .NET coders.

This book teaches the ideas and techniques of functional programming applied to real-world problems. You'll see how the functional way of thinking changes the game for .NET developers. Then, you'll tackle common issues using a functional approach. The book will also teach you the basics of the F# language and extend your C# skills into the functional domain. No prior experience with functional programming or F# is required.

What's Inside

- Thinking the functional way
- Blending OO and functional programming
- Effective F# code

Microsoft C# MVP **Tomas Petricek** is one of the leaders of the F# community. He was part of the Microsoft Research team for F# and is interested in distributed and reactive programming using F#. Microsoft C# MVP **Jon Skeet** is a veteran C# and Java developer, prolific "Stack Overflow" contributor, and author of *C# in Depth*.

For online access to the authors, and a free ebook for owners of this book, go to manning.com/Real-WorldFunctionalProgramming

"You will never look at your code in the same way again!"
—From the Foreword by Mads Torgersen, C# PM, Microsoft

"A truly functional book!"
—Andrew Siemer, .NET Architect

".NET needs more functional programmers...this book shows you how to become one."
—Stuart Caborn, Lead Consultant Thoughtworks

"Warning: this book has a very high Wow! factor. It made my head hurt...in a good way!"
—Mark Seemann Developer/Architect, Safewhere

"I recommend it to all software craftspeople, not just .NET developers."
—Paul King, Director, ASERT

ISBN 13: 978-1-933988-92-4
ISBN 10: 1-933988-92-4
54999
9 781933 988924

MANNING

$49.99 / Can $62.99 [INCLUDING eBOOK]