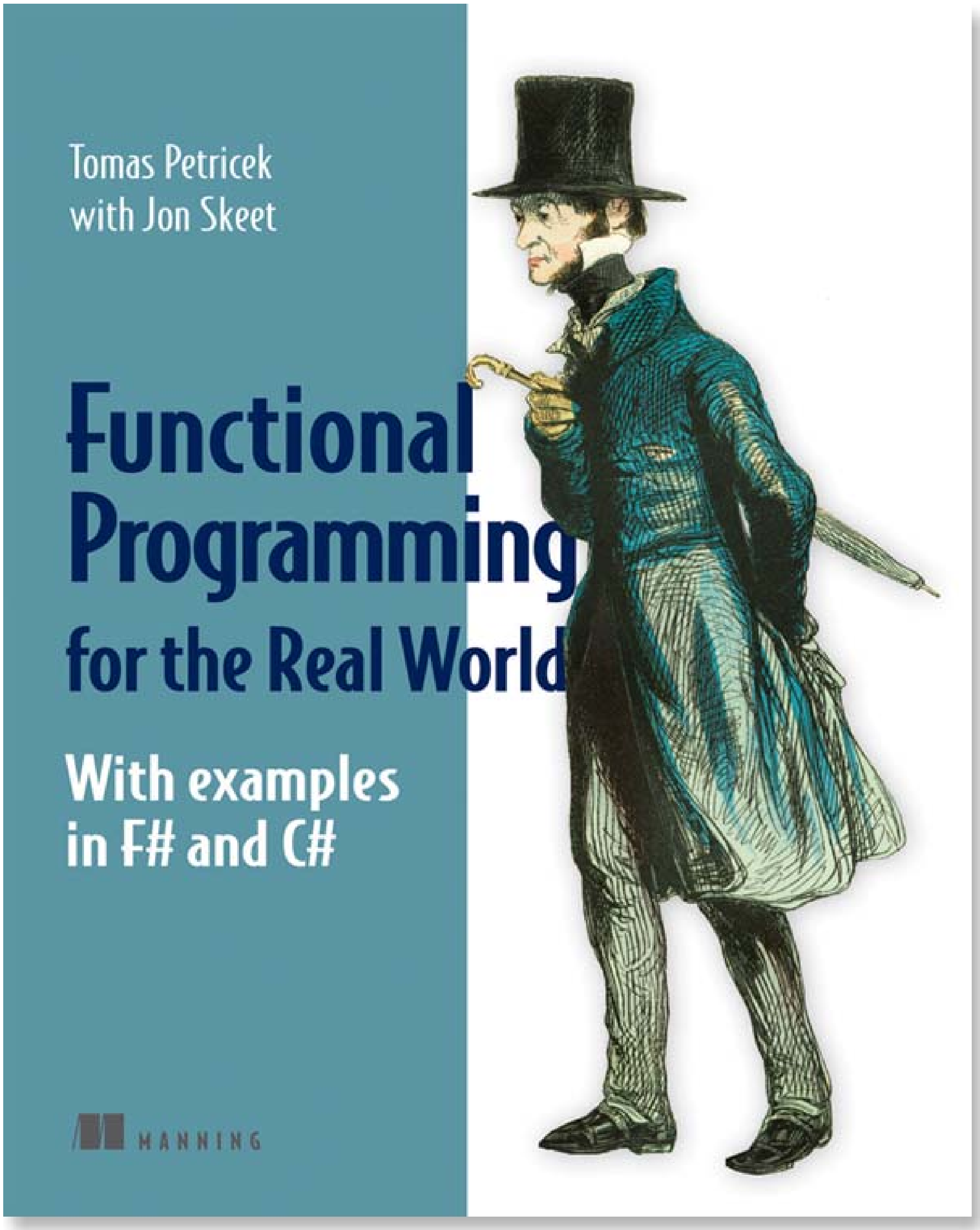

Tomas Petricek
with Jon Skeet
Functional Programming for the Real World
With examples in F# and C#
MANNING

Table of Contents

1
Thinking differently about problems

Functional programming is a paradigm that originated from ideas older than the first computers. The first functional programming language celebrated its 50th birthday in 2008. Functional languages are very succinct and expressible, yet everything is achieved using a minimal number of concepts. Despite their elegance, functional languages have largely been ignored by mainstream developers–until now.

Today we are facing new challenges and trends that open the door to functional languages. There has never been a better time to learn them. We need to write programs that process large sets of data and scale to a large number of processors or computers. We want to write programs that can be easily tested. We want to be able to express our logic in a declarative way which expresses *results* without explicitly specifying execution details–making the code easier to understand and reason about. All of these trends are embodied in functional programming, and we'll look at each of them later in this chapter.

As a result, many mainstream languages now include some functional features. In the .NET world, generics in C# 2.0 were heavily influenced by functional languages, anonymous methods in C# 2.0 and lambda expressions in C# 3.0 are examples of the most fundamental concept in functional programming and the whole of LINQ is rooted in a declarative, functional approach.

While the conventional languages are playing catch-up, truly functional languages have been receiving more attention too. The most significant example of this is probably F#, which is will be an official, fully supported Visual Studio language as of Visual Studio 2010. This evolution of functional languages on .NET is largely possible thanks to the common language runtime (CLR), which makes it possible to mix multiple languages when developing a single .NET application and also to access rich .NET libraries from new languages like F#. This makes it much easier to learn these new languages, as all of the platform knowledge that you've accumulated during your career can still be used in the new context of a functional language.

In this book, we'll look at the most important functional programming concepts and we'll demonstrate them using real-world examples from .NET. We'll start with the description of the ideas and then turn to the aspects that make it possible to develop large scale real-world .NET applications in a functional way. We'll use both F# and C# 3.0 in this book, because many of these ideas are directly applicable to C# programming. You certainly don't need to write in a functional language to use functional concepts and patterns. However, seeing the example in F# gives you a deeper understanding of how it works and F# often makes it easier to express and implement the solution.

We'll start this chapter by looking at the functional concepts that make you more productive, and then explore several examples that demonstrate what those ideas look like in real source code. We won't go into any details in this chapter, however–the goal is just to show you an interesting and elegant example that we'll discuss more fully later in the book.

1.1 Being productive with functional programming

Many people find functional programming more elegant or even beautiful, but that's hardly a good reason to use it in a commercial environment. Elegance doesn't pay the bills, sadly. The key reason why for coding in a functional style is that it makes you and your team more productive.

In this section, we'll look at the key benefits that functional programming gives you and how it solves some of the most important problems of modern software development. We'll start by looking at the declarative programming style, which gives us a richer vocabulary for describing our intentions.

1.1.1 Declarative programming style

When writing a program, we have to explain our goals to the computer using the vocabulary that it understands. In imperative languages, this consists of commands. We can add new commands, such as "show customer details", but the whole program is a step by step description saying how the computer should accomplish the overall task. An example of a program is "Take the next customer from a list. If the customer lives in UK, show their details. If there are more customers in the list, go to the beginning."

Once the program grows, the number of commands in our vocabulary becomes too high, making it very difficult to use. This is where object-oriented programming makes our life easier, because it allows us to organize our commands in a better way. We can associate all commands that involve customer with some customer entity (a class), which makes the description a lot clearer. However, the program is still a sequence of commands specifying how it should proceed.

Functional programming provides a completely different way of extending the vocabulary. We're not limited to adding new primitive commands; we can also add new control structures–primitives that specify how we can put commands together to create a program. In imperative languages, we were able to compose commands in a sequence or using a limited number of built in constructs such as loops, but if you look at typical programs, you'll still see many recurring structures; common ways of combining commands.

In our example we can see a pattern (or a control structure), which could be expressed as "Run the first command for every customer for which the second command returns true." Using this primitive, we can express our program simply by saying "Show customer details of every customer living in UK." In this sentence the part "living in UK" specifies the second command and the part "show customer details" represents the first command.

SAYING "WHAT" RATHER THAN "HOW"

If you compare these two sentences, you can see that the first describes exactly *how* to achieve our goal while the second describes *what* we want to achieve. This is the essential difference between imperative and declarative styles of programming. Hopefully you'll agree that the second sentence is far more readable and better reflected the aim of our "program".

So far I've just been using an analogy, but we'll see how this idea maps to actual source code later in this chapter. However, this isn't the only aspect of functional programming that makes life easier. In the next section, we'll look at another concept that makes it much easier to understand what a program does.

1.1.2 Understanding what a program does

In the usual imperative style, the program consists of objects that have some internal state that can be changed either directly or by calling some method of the object. This means that when we call a method, it can be hard to tell what state is affected by the operation. For example, in the C# snippet in listing 1.1 we create an ellipse, get its bounding box and then call a method on the returned rectangle. Finally, we return the ellipse to whatever has called us.

Listing 1.1 Working with ellipse and rectangle (C#)

```
Ellipse el = new Ellipse(new Rectangle(0, 0, 100, 100));
Rectangle rc = el.BoundingBox;
rc.Inflate(10, 10);                                        #1
return el;
```

#1 Is the original ellipse changed here?

How do we know what the state of the ellipse `el` will be after the code runs, just by looking at it? This is really hard, because `rc` could be a reference to the bounding box of the ellipse and `Inflate` (#1) could modify the rectangle, changing the ellipse at the same time. Or maybe the `Rectangle` type is a value type (declared using the `struct` keyword in C#) and it's copied when we assign it to a variable. Perhaps the `Inflate` method doesn't actually modify the rectangle at all, and returns a new rectangle as a result, so the third line has no effect at all.

In functional programming, most of the data structures are immutable, which means that we cannot modify them. Once the `Ellipse` or `Rectangle` is created, we can't change it. The only thing we can do is to create a new `Ellipse` with a new bounding box. This makes it easy to understand what a program does. In listing 1.2 you can see how we could rewrite the previous snippet if `Ellipse` and `Rectangle` were immutable. As you'll see, understanding the program's behavior becomes much easier.

Listing 1.2. Working with immutable ellipse and rectangle (C#)

```
Ellipse el = new Ellipse(new Rectangle(0, 0, 100, 100));
Rectangle rc = el.BoundingBox;
Rectangle rcNew = rc.Inflate(10, 10);                          #1
return new Ellipse(rcNew);                                     #2
```

#1 Returns a new rectangle
#2 Return a new ellipse with the new bounding box

When writing program using immutable types, the only thing a method can do is to return a result. It cannot modify state of any objects. You can see that for example `Inflate` returns a new rectangle as a result (#1) and that we construct a new ellipse to return an ellipse with a modified bounding box (#2). This may feel a bit unfamiliar for the first time, but keep in mind that this isn't a new idea to .NET developers. `String` is probably the best known immutable type in the .NET world, but there are many examples such as `DateTime` and other value types.

Functional programming takes this idea further, which makes it a lot easier to see what a program does, because the result of method gives us full specification of what the method does. We'll talk about immutability in a more detail later, but let's first look at one area where it is extremely useful: implementing multi-threaded applications.

1.1.3 Concurrency-friendly application design

When writing a multi-threaded application using the traditional imperative style we have to face two problems. First of all, it is difficult to turn existing sequential code into parallel code, because we have to modify large portions of the code-base to use threads explicitly. The second problem is that using shared state and locks is difficult. You have to carefully consider how to use locks to avoid race conditions and deadlocks, but leave enough space for parallel execution. Functional programming gives us answers to these two problems:

1) A declarative programming style makes it easier to introduce parallelism into existing code. We can just replace a few primitives that specify how to combine commands with a version that executes commands in parallel.

2) Thanks to the immutability, we cannot introduce race conditions and we can write lock-free code. This style makes it easy to see which parts of the program are independent and we can easily modify the program to run those tasks in parallel.

These two aspects influence how we design our applications and as a result make it a lot easier to write code that executes in parallel, taking full advantage of the power of multi-core machines. This isn't the only change you should expect to see in your design when you start thinking functionally, either…

1.1.4 Elegant thought leads to elegant code

The functional programming paradigm no doubt influences how you design and implement applications. This doesn't mean that you have to throw away anything from your existing knowledge, because many of the programming principles that you're using today are applicable to functional applications as well. This is true especially at the design level in the way how you structure the application.

On the other hand, functional programming can cause a radical transformation of how you approach problems at the implementation level. However, when learning how to use functional programming ideas, you don't have to make any radical steps. In C# you just learn how to efficiently use the new features. In F#, you can often use direct equivalents of C# constructs while you're still getting your feet wet. As you become a more experienced functional developer, you'll learn more efficient and concise ways to express yourself.

The following list summarizes how functional programming influences your programming style, working down from a design level to actual implementation.

3) Functional programs on .NET still use object-oriented design as a great way for structuring applications and components. Larger number of types and classes are designed as immutable, but it is still possible to create standard classes especially when collaborating with other .NET libraries.

4) Thanks to functional programming, you can simplify many of the standard OO design patterns, because some of them correspond to language features in F# or C# 3.0. Also, some of the design patterns simply aren't needed any more when the code is implemented in the functional way. We'll see many examples of this throughout the book, especially in chapters 7 and 8.

5) Perhaps the larger influence of functional programming is at the lowest level. Thanks to the combination of a declarative style, succinct syntax and type inference, functional languages make it easier to concisely express algorithms in a more readable way.

We'll talk about all of these aspects later in the book - but building up from the lowest level. We'll start with the functional values used to implement methods and functions, before raising our sights to design and architecture. We'll see new patterns that are specific to functional programming, as well as looking at how the object-oriented patterns you're already familiar with either fit in with the functional world or are no longer required. The functional world from the previous sentence isn't a strictly delimited technology, because the functional ideas can appear in different forms.

1.1.5 The functional paradigm

Functional programming is a programming paradigm. This means that it defines the concepts that we can use when thinking about problems. However, it doesn't precisely specify how exactly these concepts should be represented in the programming language. As a result, there are many functional languages and they put more emphasis on different features and aspects of the functional style.

We can use an analogy with a paradigm you're already familiar with: object-oriented programming. In the object-oriented style, we think about problems in terms of objects. Each object-oriented language has some notion of what an object is, but the details vary between languages. For instance C++ has multiple inheritances and JavaScript has prototypes. Moreover, you can still use an object-oriented style in language which isn't object-oriented such as C. It is less comfortable, but you'll still get some of the benefits.

However, programming paradigms are not exclusive. The C# language is primarily object-oriented, but in the 3.0 version it supports several functional features, so we can use some techniques from the functional style directly. On the other side, F# is primarily a functional language, but it fully supports the .NET object model. The great thing about combining paradigms is that we can choose the approach that best suits the problem.

Finally, learning the functional paradigm is worthwhile even if you're not planning to use a functional language. By learning a functional style, you'll gain concepts that make it easier to think about and solve your daily programming problems. Interestingly, many of the standard object-oriented patterns describe how to encode some clear functional concept in the object-oriented programming style.

So far, we have only talked about functional programming in a very general sense. It's important to have some broad idea about what makes functional programming different and why it's worth learning, but there's nothing like seeing actual code to bring things into focus. In the next section, we'll take a quick look at a couple of more specific examples.

1.2 Functional programming by example

The goal of the upcoming few examples is to show you that functional programming isn't by any means a theoretical discipline. Instead, you'll see that you've already seen and maybe even used some functional ideas. Reading about functional programming will help you to understand these technologies at a deeper level and use them more efficiently. We'll also look at a couple of examples from later parts of the book that show important practical benefits of the functional style. In the first set of examples, we'll look at declarative programming.

1.2.1 Expressing intentions using declarative style

In the previous section, I described how a declarative coding style makes you more productive. Programming languages that support a declarative style allow us to add new ways of composing basic constructs. When using this style, we're not limited to basic sequences of statements or built-in loops, so the resulting code describes more "what" the computer should do rather than "how" to do it.

I'm talking about this style in a general way because the idea is universal and not tied to any specific technology. However, it's best to demonstrate it using a few examples that you may know already to show how it's applied in specific technologies. In the first two examples, we'll look at the declarative style of LINQ and XAML. If you don't know these technologies, don't worry. The examples are simple enough to understand without background knowledge. In fact, the ease of understanding code–even in an unfamiliar context–is one of the principal benefits of a declarative style!

WORKING WITH DATA IN LINQ

If you're already using LINQ then this example will be just a reminder. However, I'll use it to show something more important. Let's first look at an example of code that works with data using the standard imperative programming style.

Listing 1.3 Imperative data processing (C#)

```
List<string> res = new List<string>();                                 #1
foreach(Product p in Products) {                                       #2
   if (p.UnitPrice > 75.0M) {
      res.Add(String.Format("{0} - ${1}",
         p.ProductName, p.UnitPrice));                                 #3
   }
}
return res;
```

#1 Create resulting list
#2 Iterate over products
#3 Add information to list of results

You'll probably need to read the code carefully to understand what it does, but that's not the only aspect we want to improve. The code is written as a sequence of some basic imperative commands. For example, the first statement creates new list (#1), the second iterates over all products (#2) and a later one adds element to the list (#3). However, we'd like to be able to describe the problem at a higher level. In more abstract terms, the code just filters a collection and returns some information about every returned product.

In C# 3.0, we can write the same code using query expression syntax. This version is closer to our real goal–it uses the same idea of filtering and transforming the data. You can see the code in listing 1.4.

Listing 1.4 Declarative data processing (C#)

```
var res = from p in Products
          where p.UnitPrice > 75.0M                                    #1
          select string.Format("{0} - ${1}",
             p.ProductName, p.UnitPrice);                              #2
return res;
```

#1 Filter products using predicate
#2 Return information about product

The expression that calculates the result (`res`) is composed from basic operators such as `where` or `select`. These operators take other expressions as an argument, because they need to know exactly what we want to filter or select as a result. Using the previous analogy, these operators give us a new way for combining pieces of code to express our intention with less writing. It is worth noting that the whole calculation in the listing 1.3 is written just as a single expression that describes the result rather than a sequence of statements that constructs it. You'll see this become a trend repeated throughout the book. In more declarative languages such as F#, everything you write is an expression.

Another interesting aspect is that many technical details of the solution are now moved to the implementation of the basic operators. This makes the code simpler, but also more flexible, because we can easily change implementation of these operators without making larger changes to the code that uses them. As we'll see later, this makes it much easier to parallelize code that works with data. However, LINQ is not the only mainstream .NET technology that relies on declarative programming. Let's turn our attention to Windows Presentation Foundation and the XAML language.

DESCRIBING USER INTERFACES IN XAML

Windows Presentation Foundation is a .NET library for creating user interfaces that supports the declarative programming style. It separates the part that describes the user interface from the part that implements the imperative program logic. However, the best practice in WPF is to minimize the program logic and create as much as possible in the declarative way.

The declarative description is represented as a tree-like structure created from objects that represent individual GUI elements. It can be created in C#, but WPF also provides a more comfortable way using an XML based language called XAML. Nevertheless, we'll see that there are many similarities between XAML and LINQ. The listing 1.5 shows how the code in XAML compares with code that implements the same functionality using the imperative Windows Forms library.

Listing 1.5 Creating user interface using imperative and declarative style (C#)

```
<Canvas Background="Black">        protected override void OnPaint
   <Ellipse x:Name="el"                  (PaintEventArgs e) {
      Width="75"                      Graphics gr = e.Graphics;
Height="75"                           Brush lg = Brushes.LightGreen;
      Canvas.Left="0"                 Brush bl = Brushes.Black;
      Canvas.Top="0"                  gr.FillRectangle(bl,
      Fill="LightGreen" />         ClientRectangle);
</Canvas>                             gr.FillEllipse(lg, 0, 0, 75, 75);
                                   }
```

It isn't difficult to identify what makes the code on the left side more declarative. The XAML code describes the user interface by composing various primitives and specifying their properties. The whole code is a single expression that creates a black canvas containing a green ellipse. On the other hand, the imperative version specifies how to create the user interface. It is a sequence of statements that specify what drawing operations should be executed to get the required GUI. This example clearly demonstrates the difference between saying "what" using the declarative style and saying "how" in the imperative style.

Also, in the declarative version we don't need as much knowledge about the underlying technical details. If you just look at the code, you don't really need to know how WPF will represent and draw the GUI. On the other hand, when looking at the WinForms example, all the technical details such as representation of brushes and order of the drawing are visible in the code. In the example above, the correspondence between XAML and the drawing code was quite clear, but we can use XAML with WPF to describe more complicated runtime aspects of the program. Let's look at the following example:

```
<DoubleAnimation
   Storyboard.TargetName="el"
   Storyboard.TargetProperty="(Canvas.Left)"
   From="0.0" To="100.0" Duration="0:0:5" />
```

This single expression creates an animation that changes the `Left` property of the ellipse (specified by the name `el`) from value 0 to value 100 in 5 seconds. The code is implemented using XAML, but we could as well write it by constructing the object tree explicitly in C#. Under the hood, `DoubleAnimation` is a class, so we would just specify its properties. The XAML language adds a more declarative syntax for writing the specification. In either case, the code would be declarative thanks to the nature of WPF. On the other hand, the traditional imperative version of code that implements an animation would be rather complex. It would have to create some timer, register an event handler that would be called every couple of milliseconds and it would have to calculate new location of the ellipse.

DECLARATIVE CODING IN .NET

WPF and LINQ are two main-stream technologies that use a declarative style, but there are many others. The goal of LINQ is to simplify working with data in a general-purpose language. It draws on ideas from many data manipulating languages that use the declarative style, so you can find the declarative approach for example in SQL or XSLT.

Another area where the declarative style is used in C# or VB.NET is when using .NET attributes. Attributes give us a way to annotate a class or its members and specify how they can be used in specific scenarios, such as editing a GUI control in a designer. This is declarative, because we just specify what we expect from the designer when working with the control and we don't have to write the code that would imperatively configure the designer.

So far we've seen several technologies that are based on the declarative style and how they make problems easier to solve. However, you may be asking yourself how we use it for solving our own kinds of problems. In the next section we'll take a brief look at an example from chapter 15 that demonstrates this.

DECLARATIVE FUNCTIONAL ANIMATIONS

Functional programming gives you the ability to write your own library that allows you to solve problems in the declarative style. We've seen how LINQ does that for data manipulation and how WPF does that for user interfaces, but in functional programming, we'll often create libraries for our own problem domain.

When I earlier mentioned that declarative style makes it possible to ignore implementation details, I wasn't really saying the full truth. When we're designing our own declarative library, we of course need to implement all the technical details. However, the great thing about the functional style is that it allows us to hide the implementation from developers (just like LINQ does) and makes it possible to solve the general problem once and for all.

The listing 1.6 shows a code that uses a declarative library for creating animations that we'll develop in chapter 15. You don't have to fully understand the code to see the benefits that we get thanks to the declarative style. It is similar to WPF in a sense that it describes how the animation should look rather than how to draw it using a timer.

Listing 1.6 Creating functional animation (C#)

```
var green = Anims.Circle(Brushes.OliveDrab, 100.0f.Anim());             #A
var blue  = Anims.Circle(Brushes.SteelBlue, 100.0f.Anim());             #A

var animatedPos = Time.Wiggle * 100.0f.Anim();                          #1

var greenMove = green.MoveXY(animatedPos, 0.0f.Const());                #B
var blueMove = blue.MoveXY(0.0f.Const(), animatedPos);                  #B

var animation = Anims.Compose(greenMove, blueMove);                     #C
```

#A Create green and blue ellipse
#1 Value animated from -100 to +100
#B Animate X or Y coordinates of ellipses
#C Compose animation from both ellipses

We'll explain everything in detail later in chapter 15. However, you can probably guess that the animation creates two ellipses. Later, it creates animated ellipses and composes them into an animation (represented as `animation` value). If we render this animation to a form, we get a result that is displayed in figure 1.1.

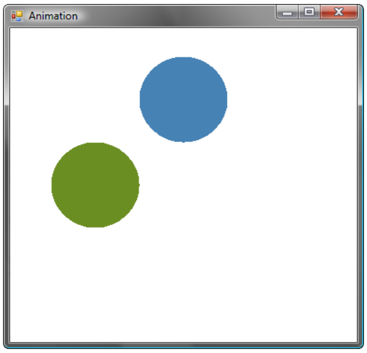


Figure 1.1 The green ellipse is moving from the left to the right and the blue ellipse is moving from the top to the bottom.

The entire declarative description is based on animated values. There is a primitive animated value called `Time.Wiggle`, which has a value that swings between -1 and +1. Another primitive construct is `x.Anim()` creates an animated value that has always the same value. If we multiply `Wiggle` by 100, we'll get an animated value that ranges between -100 and +100 (#1). These animated values can be used for specifying animations of graphical objects such as our two ellipses. The screenshot shows them in a state where X coordinate of the green one and Y coordinate of the blue one are close to the -100 state.

In the code we wrote, we don't need to know anything about the representation of animated values, because we're describing the whole animation just by calculating with the primitive animated value. Another aspect of the declarative style that you can see in the code is that the animation is in principle described using a single expression. We made it more readable by declaring several local variables, but if you replaced occurrence of the variable with its initialization code, the animation would remain exactly the same.

COMPOSITIONALITY

An important feature of declarative libraries is that we can use them in a compositional manner. In LINQ, you can move a part of a complex query into a separate query and reuse it. Similarly, our previous sample is very compositional. We can declare animated values such as `animatedPos` and compose primitive animated objects using `Anim.Compose`.

On the last couple of pages, we looked at the declarative programming, which is an essential aspect of the functional style. The last example shows how this style can be used in an advanced library for describing animations. In the next section, we'll turn our attention to more technical, but also very interesting functional aspect which is immutability.

1.2.2 Understanding code using immutability

We discussed immutability before when talking about benefits of the functional style. We used an example with bounding box of an ellipse, where it wasn't clear how the code behaved. Once we rewrote the code using immutable objects, it became easier to understand. We'll talk about this topic in detail in later chapters. The purpose of this example is just to satisfy your curiosity and show how an immutable object would look in practice.

Again, don't worry if you won't understand everything in detail, because we'll talk about everything more fully later. Now, let's imagine we're writing a game with some characters that we can shoot at. Listing 1.7 shows a part of the class that represents the character.

Listing 1.7 Immutable representation of a game character (C#)

```
class GameCharacter {
      readonly int health;                                         #1
      readonly Point location;                                     #1

      public Character(int health, Point location) {
         this.health = health;                                     #2
         this.location = location;                                 #2
      }
      public Character HitByShooting(Point target) {
         int newHealth = CalculateHealth(target);
         return new GameCharacter(newHealth, this.location);       #3
      }
      public bool IsAlive {
         get { return health > 0; }
      }
      // Other methods and properties omitted
   }
```

#1 All fields are declared as readonly
#2 Initialize immutable fields only once
#3 Return a game character with updated health

In C#, we can explicitly mark a field as immutable using the `readonly` keyword. This means that we cannot change the value of the field, but we could still modify the target object if the field is a reference to a mutable class. When creating a truly immutable class, we need to make sure that all fields are marked as `readonly` and also that the types of these fields are also primitive types, immutable value types or other immutable classes.

According to these conditions, our `GameCharacter` class is immutable. All its fields are marked using the `readonly` modifier (#1), `int` is a primitive type and `Point` is an immutable value type. When a field is read-only it can be set only when creating the object, so we can only set the health and location of the character only in the constructor (#2). This means that we can't modify the state of the object once it is initialized. So, what can we do when an operation needs to modify the state of the game character?

You can see the answer when you look at the `HitByShooting` method (#3). It implements a reaction to a shot being fired in the game. It uses the `CalculateHealth` method (not shown in the sample) to calculate the new health of the character. In an imperative style, it would then update the state of the character, but that's not possible since the type is immutable. Instead, the method creates a new `GameCharacter` instance to represent the modified character and returns it as a result.

The class from the previous example represents a typical design of immutable C# classes and we'll use it (with minor modifications) throughout the book. Now that we know what immutable types look like, let's see some of the consequences.

READING FUNCTIONAL PROGRAMS

We've already seen an example that used immutable types when looking at the code with bounding box of an ellipse. However, that was very briefly and we just concluded that it makes the code more readable. In this section, we're going to look at two snippets that we could find somewhere in our functional game.

Listing 1.8 shows two separate examples, each with two game characters: `player` and `monster`. The first one shows how we could execute the monster AI to perform a single step and then test whether the player is in danger and the second shows how we could handle a gunshot.

Listing 1.8 Code snippets form a functional game (C#)

```
// Move the monster & test if the player is in danger
var movedMonster = monster.PerformStep();                    #1
var inDanger = player.IsCloseTo(movedMonster);               #2

// Did gunshot hit a monster or the player?
var hitMonster = monster.HitByShooting(gunShot);             #3
var hitPlayer = player.HitByShooting(gunShot);               #3
```

#1 Move the monster
#2 Test distance from the moved monster
#3 Create new monster and player

All objects in our functional game are immutable, so when we call method on an object, it cannot modify itself or any other object. If we know that, we can make several interesting observations about the previous examples. In the first snippet, we start by calling the `PerformStep` method of the monster (#1). The method returns a new monster and we assign it to a variable called `movedMonster`. On the next line, we use this monster to check whether the player is close to it and so is in danger.

One interesting point to note here is that we can see that the second line of the code relies on the first one. If we changed the order of these two lines, the program wouldn't compile because `movedMonster` wouldn't be declared on the first line. On the other hand, if you implemented this in the imperative style, the method would modify the state of the `monster` object. In that case, we could rearrange the lines and the code would compile, but it would change the meaning of the program and it could start behaving incorrectly.

Now, what can we learn by looking at the second snippet? It consists of two lines that create a new monster and a new player objects with updated health property when a shooting occurs in the game. The two lines are independent, meaning that we could change their order. Can this operation change the meaning of the program? It appears that it shouldn't and when all objects are immutable it doesn't. Surprisingly, it might change the meaning in the imperative version if `gunShot` were mutable. The first of those objects could change some property of the gunshot and the behavior would depend on the order of these two statements.

The previous example was quite simple, but it already shows how immutability eliminates many possible difficulties. In the next section, we'll see another great example, but let me just briefly review what you'll find later in the book.

REFACTORING AND UNIT TESTING

We've already seen that immutability helps us to understand what a program does. This is very helpful when refactoring the code. Another interesting functional refactoring is changing when some code actually executes. It may run when the program hits it for the first time, but it may as well execute when its result is actually needed. As we'll see in a few pages, this way of evolving programs is very important in F# and immutability makes refactoring easier in C# too. We'll talk about refactoring later in chapter 11.

Another area where immutability helps a lot is when creating unit tests for functional programs. The only thing that a method can do in an immutable world is to return a result, so we only have to test whether a method returns the right result for specified arguments. You'll find more information about this topic in chapter 18.

When discussing how functional programming makes you more productive, I mentioned immutability as an important aspect that makes it easier to write parallel programs. In the next section we'll briefly look at that and also at other related topics.

1.2.3 Writing efficient parallel and asynchronous programs

I said earlier that functional programming makes it easier to write parallel programs. This is one of the most important aspects of this paradigm nowadays and maybe it is also the reason why you picked this book. In this section, we'll look at a couple of samples demonstrating how functional programs can be easily parallelized. In the first two examples, we'll use Parallel Extensions to .NET. This is a new technology from Microsoft for writing parallel applications, shipping as part of .NET 4.0. As you might expect, it lends itself extremely well to functional code. As always in this chapter, we won't go into the details. I just want to demonstrate that parallelizing functional programs is significantly easier and more importantly, less error prone than it is for the imperative code.

PARALLELIZING IMMUTABLE PROGRAMS

First we'll take another look at the previous example. We've seen two snippets from a game written in a functional way. In the first snippet, the second line uses the outcome of the first line (state of the monster after movement). Thanks to the use of immutable classes, we can see that this doesn't give us any space for introducing parallelism.

On the other hand, the second snippet consists of two independent lines of code. I said earlier that in functional programming, we can run independent parts of the program in parallel. Now you can see that immutability gives us a great way to spot which parts of the program are independent. Even without knowing any details, we can look at the change that makes these two operations run in parallel. The change to the source code is minimal:

```
var hitMonster = Future.Create(() =>
   monster.HitByShooting(gunShot));
var hitPlayer = Future.Create(() =>
   player.HitByShooting(gunShot));
```

The only thing that we did is that we wrapped the computation in a `Future` type from the Parallel Extensions library. We'll talk about `Future` in detail in chapter 14. Interestingly, the benefit isn't only that we have to write less code, but also that we have a guarantee that the code is correct. If you did a similar change in an imperative program, you'd have to carefully review the `HitByShooting` method (and any other method it calls) to find all places where it accesses some mutable state and add locks to protect the code that modifies shared state. In functional programming everything is immutable, so we don't need to add any locks.

The example in this section is a form of lower a level *task based parallelism*, which is one of three approaches that we'll see in chapter 14. In the next section we'll take a brief look at the second approach, which benefits from the declarative programming style.

DECLARATIVE PARALLELISM USING PLINQ

Declarative programming style gives us another great technique for writing parallel programs. I have already stated that the code written using the declarative style is composed using primitives. In LINQ, these primitives are query operators such as `where` and `select`. In the declarative style, we can easily replace the implementation of these primitives and that's exactly what PLINQ does. It allows us to replace standard query operators with query operators that run in parallel.

In the listing 1.9, you can see a query that updates all monsters in our fictive game and remove those that died in the last step of the game. The change is extremely simple, so I can show you both of the versions in a single listing.

Listing 1.9 Parallelizing data processing code using PLINQ (C#)

```
var updated =                            var updated =
   from m in monsters                       from m in monsters.AsParallel()   #1
   let nm = m.PerformStep()                 let nm = m.PerformStep()
   where nm.IsAlive select nm;              where nm.IsAlive select nm;
```

The only change that we made in the parallel version on the right side is that we added a call to `AsParallel` method (#1). This call changes the primitives that are used when running the query and makes the whole fragment run in parallel. We'll see how this works in chapter 11, where we'll talk about declarative computations like this in general and in chapter 14 which focuses on parallel programming specifically.

You may have already seen this demo and you were perhaps thinking that you don't use LINQ queries that often in your programs. This is definitely a valid point, because in imperative programs, LINQ queries are used less frequently. However, functional programs do most of their data processing in the declarative style. In C#, this can be written using query expressions whereas F# provides higher order list processing functions that we'll see in chapters 5 and 6. This means that after you'll read this book, you'll be able to use declarative programming more often when working with data. As a result, your programs will be more easily parallelizable. The technique I just described also inspired an algorithm used internally by Google for massive parallel data processing.

Microsoft PLINQ and Google MapReduce

Google has developed a framework called MapReduce [Dean, Ghemawat, 2004] for processing of massive amounts of data in parallel. This framework distributes the work between computers in large clusters and uses exactly the same ideas as PLINQ. The basic idea of MapReduce is that the user program describes the algorithm using two operations (somewhat similar to `where` and `select` in PLINQ). The framework takes these two operations and the input data, and runs the computation. You can see a diagram visualizing the computation in figure 1.2.

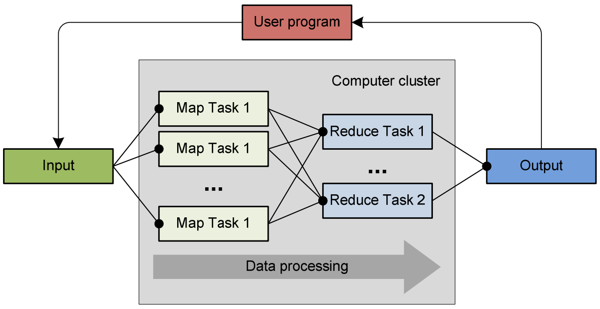


Figure 1.2 In the MapReduce framework an algorithm is described by specifying map task and a reduce task. The framework automatically distributes the input across servers and processes the tasks in parallel

The framework splits the input data into partitions and executes the map task (using the first operation from the user) on each of the partitions. For example, a map task may find the most important keywords in a web page. The results returned by map tasks are then collected and grouped by a specified key (for example the name of the domain) and the reduce task is executed for each of the groups. In our example, the reduce task may summarize the most important keywords for every domain.

We've briefly seen two ways in which functional programming makes parallelization simpler. However, there is one more related area where functional programming helps us to write more efficient and scalable code with respect to multi-threading. It is important especially when the code uses long running I/O operations.

WRITING NON-BLOCKING CODE USING F#

Long running operations are quite frequent in modern software. Many applications use HTTP requests to load some data from the internet or communicate using web services. When an application performs an operation like this, it is very hard to predict when the operation will complete, and if this is not handled properly the application will become unresponsive.

However, writing the code that performs I/O operations without blocking is very difficult using the current techniques. In F#, this is largely simplified thanks to a feature called *asynchronous workflows*. Interestingly, this is one of the F# features that are really hard to implement in C#, so it's a good reason for looking at F#. We'll talk about asynchronous workflows in detail later in chapter 13, but I can show you at least a brief example to demonstrate how interesting this feature is. Let's start by looking at listing 1.10, which shows a C# example that downloads source of a web page.

Listing 1.10 Downloading web pages (C#)

```
var req = HttpWebRequest.Create("http://manning.com");
var resp = req.GetResponse();                                   #1
var stream = resp.GetResponseStream();
var reader = new StreamReader(stream);
var html = reader.ReadToEnd();                                  #2
Console.WriteLine(html);
```

#1 Initialize HTTP connection
#2 Download the web page content

The listing shows a fairly simple code that downloads HTML source code of a specified web page. You'd also have to add some `using` directives to reference the necessary .NET namespaces if you wanted to compile the code, but we'll show this properly in later chapters. The program needs to perform HTTP communication in two places. In the first (#1) it needs to initialize HTTP connection with the server and in the second (#2) it downloads the web page.

Both of these operations could potentially take quite a long time and each of them could block the active thread, causing our application to become unresponsive. We could run the download on a separate thread, but using threads is expensive, so this would limit the number of downloads we can run in parallel. Also, most of the time, the thread would be just waiting for the response, so we'd be consuming thread resources for no good reason. To implement this properly, we need to use asynchronous .NET methods that allow us to trigger the request and call some code that we provide when the operation completes. This version of code is quite difficult to write. Even if we use anonymous delegates from C# 2.0, the code still looks quite complicated:

```
var req = HttpWebRequest.Create("http://manning.com");
req.BeginGetResponse(delegate(IAsyncResult ar) {
   var rsp = req.EndGetResponse(ar);
   // TODO: Use the response to read the HTML
});
```

Anonymous delegates or lambda expressions make this a bit nicer, because we don't have to write a method to handle the response, but we still have to change the structure of the code. In fact, if we decide to change a synchronous version of the code into asynchronous, we'll have to rewrite it almost completely.

The previous snippet isn't complete, but if we tried to finish it, we'd find another issue. There is no `BeginReadToEnd` method, so we'd have to implement this functionality ourselves. This is quite difficult, because we need to download the page in a buffered way. If we want to write this in an asynchronous style, we can't use any of the built-in constructs such as `while` loop.

In F#, it's common to start with the simplest possible solution to a problem and then turn it into a more sophisticated version. We'll talk about this principle later in this chapter, but we can see it in action right now. One of the things you may want to do is to take synchronous code that downloads a web page and make it asynchronous. When working with .NET libraries, the F# code is quite similar to C#, so you can just imagine that the listing 1.10 was in F# (you'd just delete semicolons and change the `var` keyword to `let`). The listing 1.11 shows an asynchronous version using F# asynchronous workflows.

Listing 1.11 Downloading web page asynchronously (F#)

```
let op =
   async {                                                      #1
      let req = HttpWebRequest.Create("http://manning.com")
      let! resp = req.AsyncGetResponse()                        #2
      let stream = resp.GetResponseStream()
      let reader = new StreamReader(stream)
      let! html = reader.AsyncReadToEnd()                       #A
      Console.WriteLine(html)
   }
Async.Run(op)                                                   #B
```

#1 Wrap in an asynchronous workflow
#2 Run operation asynchronously
#A Asynchronous download
#B Run the workflow

The process of turning a synchronous code into asynchronous in F# is quite easy. First of all, we wrap the whole computation into an `async` block (#1). The next thing to do is to identify all asynchronous operations in the block and to change the method to a corresponding asynchronous version. The workflow needs to know which of the methods should be executed in a non-blocking way, so we also change the usual value declaration using `let` into a workflow-specific declaration that uses `let!` syntax (#2). What is even more interesting is that methods like `AsyncReadToEnd` are quite easy to implement, because asynchronous workflows can use `while` loops and other basic constructs*.

This feature is very easy to use but it isn't easy to see how the code actually executes at first glance. We'll explain everything in detail in chapter 13, but it's worth noting that asynchronous workflows aren't a built-in feature of the F# language. It is just a very useful instance of a more general feature that allows you to write non-standard computations. This feature is also covered in this book and we'll talk about it in chapter 12.

* There are projects that attempt to simplify this problem in C# such as the Concurrency and Coordination Runtime (CCR), but all of them rely on using some C# language features in an unexpected and slightly unnatural ways. We'll mention a couple of these projects briefly when discussing asynchronous workflows in chapter 13.

Asynchronous workflows are very important. They allow us to write programs that wait for an operation to complete without using a dedicated thread (which consumes valuable resources). This also enables us to use different models for concurrency, such as the message passing style which is used in a successful functional language called Erlang.

Message passing in the Erlang language

Erlang is a language developed and heavily used by Ericsson for developing large scale real-time systems. It can be found in many of the Ericsson's telecommunication equipment. Erlang has been used commercially by Ericsson for programming their network hardware used concurrently by hundreds of users as well as by other companies.

Concurrent applications in Erlang are described using independent processes (written in a functional way) that can communicate with each other using messages. The process waits for a message and when a message arrives, it processes it. We'll see how to use this style in F# using asynchronous workflows in chapter 13.

Before we take a look at the F# language and talk about the F# programming style, let's briefly talk about the history of functional programming, which is surprisingly rich.

1.3 The path towards real-world functional programming

The history of functional programming goes as far back as the 1930s when Alonzo Church and Stephen C. Kleene introduced a theory called Lambda calculus as part of their investigation of the foundations of mathematics. Even though it didn't fulfill their original expectations, it is still used in some branches of logic and has evolved into a very useful theory of computation. For curiosity and to show the basic principles of functional programming, you'll find a brief introduction to lambda calculus in the next chapter. However, lambda calculus escaped its original domain when computers were invented and served as an inspiration for the first of functional programming languages.

1.3.1 Functional languages

The LISP language, created by John McCarthy in 1958, was based on lambda calculus. LISP is an extremely flexible language, and it pioneered many programming ideas that are still used today, including data structures, garbage collection and dynamic typing.

In the 1970s, Robin Milner developed a language called ML. This was the first of a family of languages which now includes F#. Inspired by typed lambda calculus, it added the notion of types and even allowed writing "generic" functions in a same way as we can do now with .NET generics. ML was also equipped with a powerful type inference mechanism, which is now essential for writing terse programs in F#. OCaml, a pragmatic extension to the ML language appeared in 1996. It was one of the first languages that allowed the combination of object-oriented and functional approaches. OCaml was a great inspiration for F#, which has to mix these paradigms in order to be a first-class .NET language *and* a truly functional one.

Other important functional languages include Haskell (a language with surprising mathematical purity and elegance) and Erlang, which I have already mentioned in a sidebar. We'll learn more about some of these languages in the rest of the book, when talking about topics where they have some interesting benefits over F#–but first, let's finish our story by looking at the history of F#.

1.3.2 Functional programming on the .NET platform

The first version of .NET was released in 2002 and the history of the F# language dates back to the same year. F# started off as a Microsoft Research project by Don Syme and his colleagues, with the goal of bringing functional programming to .NET. F# and typed functional programming in general gave added weight to the need for generics in .NET and the designers of F# were deeply involved in the design and implementation of generics in .NET 2.0 and C# 2.0.

With generics implemented in the core framework, F# began evolving more quickly and the programming style used in F# also started changing. It began as a functional language with some support for objects, but as the language matured, it seemed more and more natural to take the best from both of these styles. As a result F# can be now more precisely described as a multi-paradigm language, which combines functional and object-oriented approach, together with a great set of tools that allow using F# interactively for scripting.

F# has been a first-class .NET citizen since its early days. This means that not only can it access any of the standard .NET components, but equally importantly any other .NET language can access code developed in F#. This makes it possible to use F# to develop standalone .NET applications as well as parts of larger projects. F# has always come with support in Visual Studio, and in 2007 a process was started to turn F# from a research project to a full production-quality language. In 2008 it was announced that F# will become one of the languages shipped with Visual Studio 2010. Now we know its origins, let's take a look at the language itself.

1.4 Introducing F#

We'll introduce F# in stages throughout the book, as and when we need to. In this section we'll just look at the very basics, writing a couple of short examples so you can start to experiment for yourself. We'll examine F# more carefully after summarizing important functional concepts in chapter 2. Our first real-world F# application will come in chapter 4.

After discussing the "Hello world" sample, I'll talk a little bit about F# to explain what you can expect from the language. We'll also discuss the typical development process used by F# developers, because it is quite different to what you're probably used to with C#.

1.4.1 Hello world in F#

The easiest way to start using F# is to create a new script file. Scripts are lightweight F# sources that don't have to belong to a project and usually have an extension "fsx". In Visual

Studio, you can go to "File" - "New" - "File…" (Ctrl + N) and then select "F# Script File" from the "Scripts" category. Once we have the file, we can jump directly to the "Hello world" code.

Listing 1.11 Printing hello world (F#)

```
let message = "Hello world!"         #1
printfn "%s" message                 #2
```

#1 Value binding for 'message'
#2 Call to the 'printfn' function

I admit that this isn't the simplest possible "Hello world" in F#, but it would be fairly difficult to write anything interesting about the single line version. The listing 1.11 starts with a value binding (#1). This is similar to variable declaration, but there is one important difference - the value is immutable and we cannot change its value later. This matches with the overall functional style to make things immutable and we'll talk about this in detail in the next two chapters.

After assigning a value "Hello world" to a symbol `message`, the program continues with a call to a `printfn` function. It is important to note that arguments to F# functions are usually just separated spaces with no commas between them or surrounding parentheses. We'll sometimes write parentheses when it makes the code more readable, such as when writing `cos(1.57)`, but even in this case the parentheses are optional. I'll explain the convention that I'll use as we learn the core concepts of F# in the next couple of chapters.

The first argument to the `printfn` function is a format string. In our example, it specifies that the function should take only one additional parameter, which will be a string. The type is specified by the `%s` in the format string (the letter "s" stands for "string") and the types of arguments are even checked by the compiler. Now that we understand the code, let's look how we can run it.

INTERACTIVE PROGRAMMING IN F#

The easiest way to run the code is to use the interactive tools provided by F# tool chain. These tools allow you to use the interactive style of development. This means that you can easily try what code would do and verify whether it behaves correctly by running it with a sample input. Some languages have an interactive console, where you can paste code and execute it. This is called read-eval-print loop (REPL), because the code is evaluated immediately.

In F#, we can use a command prompt called F# interactive, but the interactive environment is also integrated inside the Visual Studio environment. This means that one can write the code with the full IDE and IntelliSense support, but also select a block of code and execute it immediately to test it.

Let's have a look at the results that we get when we run the code. If you're using F# interactive from command line, you'd just paste the previous code and type ";;" to execute it. If you're using Visual Studio, you can select the code and hit Alt + Enter to send it to the interactive window. Listing 1.12 shows the result that you'll get.

Listing 1.12 Running the Hello world program (F# interactive)

```
MSR F# Interactive, (c) Microsoft Corporation, All Rights Reserved
F# Version 1.9.4.10, compiling for .NET Framework Version v2.0.50727

> (...);;                                                        #A

val message : string                                             #1
Hello world!                                                     #2
```

#A Source code goes here
#1 Information about value binding
#2 Printed output of 'printfn' call

The first line (#1) is generated by the value binding. It reports that a value called `message` was declared and that the type of the value is string. We didn't explicitly specify the type, but F# uses a technique called type inference to deduce the types of values, so the program is statically typed, just as in C#. The second line (#2) is an output from the `printfn` function, which prints the string and doesn't return any value.

Writing something like "Hello world" example doesn't demonstrate how working with F# looks at the larger scale. Let's now briefly look at the usual development process, because is quite interesting.

1.4.2 From simplicity to robustness

When starting a new project, you don't usually know at the beginning how the code will look at the end. At this stage, the code evolves quite rapidly. However, as it becomes more mature, the architecture becomes more solid and we're more concerned with the robustness of the solution rather than with the flexibility. Interestingly, these requirements aren't reflected in the programming languages and tools that you use. F# is very appealing from this point of view, because it reflects this in both tools and the language.

F# development process in a nutshell

I have already mentioned the F# interactive tool. It allows you to verify and test your code immediately while writing it. This tool is extremely useful at the beginning of the development process, because it encourages you to quickly try various different approaches and choose the best one. Also, when solving some problem where you're not 100% sure, you can immediately try the code. When writing F# code, you'll never spend a large time debugging the program. Once you first compile and run your program, you've already tested substantial part of it interactively.

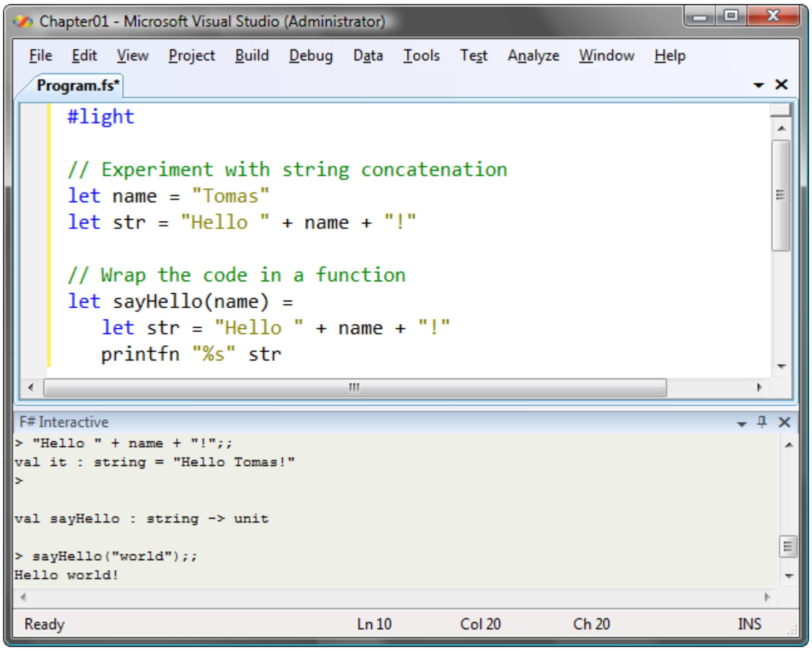
When talking about "testing" in the early phase, I mean that you tried to execute the code with various inputs a couple of times to interactively verify that it works. In the later phase, we can turn these snippets into unit tests, so the term "testing" means a different thing in the later phase. When working with the mature version of the F# code, we can use tools such as Visual Studio debugger or various unit testing frameworks.

Moreover, F# as a language reflects this direction as well. When you start writing a solution to some problem, you start with only the most basic functional constructs, because they make writing the code as easy as possible. Later, when you find the right way to approach the problem and you face the need to make the code more polished, you end up using more advanced features that make the code more robust, easier to document and also accessible from other .NET languages like C#.

Let's see what the development process might look like in action. I'll use a few more F# constructs, but we won't focus primarily on the code. The more important aspect is how the development style changes as the program evolves.

STARTING WITH SIMPLICITY

When starting a new project, you'll usually create a new script file and try implementing the first prototype or experiment with the key ideas. At this point, the script file contains sources of various experiments, often in an unorganized order. The figure 1.3 shows how your Visual Studio IDE might look like at this stage.


Chapter01 - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools Test Analyze Window Help
Program.fs*
F# Interactive
Ready Ln 10 Col 20 Ch 20 INS

```
#light

// Experiment with string concatenation
let name = "Tomas"
let str = "Hello " + name + "!"

// Wrap the code in a function
let sayHello(name) =
   let str = "Hello " + name + "!"
   printfn "%s" str
```

```
> "Hello " + name + "!";;
val it : string = "Hello Tomas!"
>

val sayHello : string -> unit

> sayHello("world");;
Hello world!
```

Figure 1.3 Using F# interactive we can first test the code and then wrap it into a function.

The screenshot shows only the editor and the F# interactive window, but that's really all we need now, because we don't yet have any project. As you can see, I first wrote a few value bindings to try how string concatenation works in F# and entered the code to the F# interactive window to verify that it works as expected. Once I knew how to use string concatenation, I wrapped the code in a function. We'll talk about functions in chapter 3.

Next, I selected the function and hit Alt + Enter to send it to the F# interactive. After that, I entered an expressions `sayHello("world")` to test the function I just wrote. Note that the commands in F# interactive are terminated with "`;;`". This allows you to easily enter multi-line commands.

Once we start writing more interesting examples, you'll see that the simplicity is greatly supported by using of the functional concepts. Many of them allow you to write the code in a surprisingly terse way and thanks to the ability to immediately test the code F# is very powerful in the first phase of the development. We'll talk about the easy-to-use functional constructs mostly in the part 2 of this book. However, as the program grows larger, we need to write it in a more polished way and make it coherent with the usual .NET techniques. Fortunately, F# helps us to do this too.

ENDING WITH ROBUSTNESS

Unlike many other languages that are popular for their simplicity, F# lives on the other side as well. In fact, it can be used for writing very mature, robust and safe code. The usual process is that you start with very simple code, but as the codebase becomes larger you refactor it in a way that makes it more accessible to other F# developers, enables writing better documentation and supports better interoperability with .NET and C#.

Perhaps the most important step in order to make the code well accessible from other .NET languages is to encapsulate the functionality into .NET classes. The F# language supports the full .NET object model, and classes authored in F# appear just like ordinary .NET classes with all the usual accompaniments such as static type information and XML documentation.

We'll talk about F# object types in chapter 9 and you'll see many of the robust techniques in part 4, but let me just shortly demonstrate this, to prove that you can use F# in a traditional .NET style as well. The listing 1.13 shows how to wrap the `sayHello` function in a C# style class and add Windows Forms user interface.

Listing 1.13 Object-oriented Hello world using Windows Forms (F#)

```
open System.Drawing                                              #1
open System.Windows.Forms                                        #1

type HelloWindow() =                                             #2
   let frm = new Form(Width = 400, Height = 140)                 #A
   let fnt = new Font("Times New Roman", 28.0f)                  #A
   let lbl = new Label(Dock = DockStyle.Fill, Font = fnt,        #A
                       TextAlign = ContentAlignment.MiddleCenter)   #A
   do frm.Controls.Add(lbl)                                      #A
```

```
      member x.SayHello(name) =                                          #3
         let msg = "Hello " + name + "!"
         lbl.Text <- msg                                                 #A

      member x.Run() =                                                   #4
         Application.Run(frm)
```

#1 Import necessary .NET namespaces
#2 F# class declaration
#A Constructor initializes the user interface
#3 Builds and displays the hello message
#4 Method that runs the application
#A Modify property of a .NET type

The example starts with several `open` directives (#1) that import types from .NET namespaces. Next, we declare the `HelloWindow` class (#2), which wraps the code to constructs the user interface and exposes two methods. The first method (#3) wraps the functionality for concatenating hello world messages that we interactively developed earlier. The second one runs the form as a standard windows forms application (#4). The class declaration appears just like ordinary C# class, with the difference that F# has a more lightweight syntax for writing classes. The code that uses the class in F# will look just like your usual C# code:

```
let hello = new HelloWindow()
hello.SayHello("dear reader")
hello.Run()
```

At this stage, we're developing the application in a traditional .NET style, so we'll run it as a standalone application. However, the interactive style helped us, because we had already interactively tested a part of the application. You can see how the resulting application looks in figure 1.4.

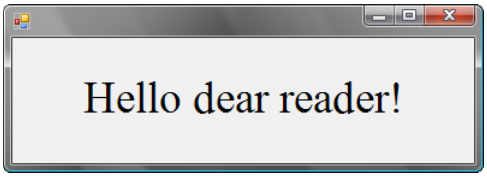

Hello dear reader!

Figure 1.4 Running WinForms application created using object-oriented programming style in F#

In this section, we've had a quick taste of what the typical F# development process feels like. I haven't explained every F# construct we've used, because we'll see how everything works in detail in later chapters. We used a very simple example, so the second version of the code was still quite simple. However, it demonstrated that you can use F# language for writing a pretty standard .NET programs.

What can F# offer to a C# developer?

As I said earlier, F# is well suited for writing code using simple concepts at the beginning and turning it into a traditional .NET version later, while C# is largely oriented towards the traditional .NET style. If you're C# developer, creating real-world applications you can easily take advantage of F# in two ways.

The first option is to use F# for rapid prototyping and experimenting with the code as well for exploring how .NET libraries work. As you've seen, using F# interactively is very easy, so writing a first sketch of the code can be done in F# and you save a lot of time when trying several approaches to a problem or exploring how a new library works. If you require code written in C#, then you can rewrite your prototype to C# later and still save a lot of development time.

However, F# is a fully compiled .NET language, so there are no technical reasons for preferring C# source code. This means that you can simply make sure that your library can be easily accessed from C# by turning the code from a simple to a traditional .NET version and use F# for example for writing parts of a larger .NET solution.

That should be enough about F# for now. It's possible that you're still finding some of the F# language constructs puzzling, but the purpose of this introduction wasn't to teach you F# in 4 pages, but I wanted to show you how F# looks and feels, so you can experiment with it as we'll look at more interesting examples in the subsequent chapters.

1.5 Summary

This chapter gave you a very brief overview of what makes functional programming interesting. We've talked about the declarative programming style, which is used when writing applications and libraries in a functional style. We've seen that this is already used in many successful technologies such as WPF and LINQ, but I also demonstrated that we can use it for writing functional solutions to other kinds of problems in C# 3.0.

We've also looked at parallel and asynchronous programming, which is a big challenge for modern software development. Using a functional approach makes it significantly easier thanks to the use of immutability and declarative programming. The first one gives us guarantees about the code and helps us writing correct and safe code and the second one is more expressive when solving problems.

In the next chapter, you'll see a much broader picture of functional programming. We'll look at all of the important ideas from a high-level perspective and you'll also see how they relate to each other. Even though we won't look at much real code yet, the next chapter will give you a solid foundation we can build on in the rest of the book.

2 Core concepts in functional programming

If you ask two functional programmers what they consider the most essential aspect of the functional paradigm, you’ll probably get three different answers. The reason for this is that functional programming has existed for a long time and there is a wide range of diverse programming languages. Every language emphasizes a different set of aspects while giving less importance to others. However, most of the concepts are to some extent present in all functional languages.

The central part of this chapter focuses on these common ideas, discussing the basic features and techniques that functional programmers have in their toolset. We'll look at the concepts from a high level perspective, but you'll see how they fit together to form one coherent way of thinking about and tackling problems.

We'll start by discussing how functional programs represent program state and how they change it. In object-oriented programming, the state is carried by objects while in functional programming the key role is played by functions and data types. Next, we'll look at language features that support the declarative programming style we looked at in the first chapter. Finally, we'll talk about types and how they help to verify program correctness. This aspect isn't shared by all functional languages, but is essential for many of them including F# and others such as Haskell. Their implementation of type checking is very advanced and is different to what you may be used to from C# in many ways.

We won't go into much programming yet. Instead you'll get a general understanding of the key concepts and a bit better feeling about how functional programs look. The first group of concepts that we'll talk about are related to the representation of data in functional programs. These concepts heavily influence how a program works with data.

2.1 How functional programs calculate

In the first chapter we saw that functional programs use immutable data structures to represent their state. The functional approach to make things immutable doesn't just influence data structures (or classes in C#), but also extends to local variables.

I wouldn't be surprised if you were wondering how the program can do anything at all when everything is immutable. The short answer is that functional programs aren't described as a sequence of statements that change the state but rather as computations. In this section, we'll shed some light on how these calculations are written. Let's start from the simple code that works with variables.

2.1.1 Working with immutable values

The first of the common features is that functional programs rarely have typical variables as we know them from other programming languages. The key difference is that functional languages prefer immutable variables, meaning the variable can't change its value once it is initialized. Thus using a term *variable* is quite inappropriate and functional programmers prefer the term *value*.

Let me demonstrate what I mean using an example. Let's say we want to write a program that takes some initial value, reads two numbers from the console, adds the first number to the initial value and multiplies the result by the second number. A typical implementation of something like this in C# would look like this (we'll use hypothetical methods `GetInitialValue`, `ReadInt32` and `WriteInt32`, but you could easily implement them if you want to play with this example):

```
int res = GetInitialValue();
res = res + ReadInt32();
res = res * ReadInt32();
WriteInt32(res);
```

As you can see, we declared a variable `res` to hold the initial value. Then we modified it two times, using an input value read from the console. Now, let's look at the same code implemented without modifying the value of any variables:

```
int res0 = GetInitialValue();
int res1 = res0 + ReadInt32();
int res2 = res1 * ReadInt32();
WriteInt32(res2);
```

Because we couldn't modify the value of the first variable we declared a new variable every time we wanted to calculate a new value (`res0`, `res1`, `res2`). The key difference is that in the second example, we didn't use the assignment operator (written as an equal sign in C#). The only occurrence of this symbol in a second example is when initializing a variable value, which has a different meaning then assignment operator. Instead of changing a value of an existing variable it creates a new variable with the specified initial value.

I already mentioned that the term "variable" is inappropriate. Working with values is different in many ways, so it isn't just a change in the terminology, but a different concept. For this reason we'll use the functional terminology in the rest of the book, but you may sometimes find the analogy between variables and values useful. We'll also use a term *value binding*, which refers to a declaration of a value, which assigns (binds) the value to a symbol.

Of course, using immutable values instead of variables requires us to express many problems in a different way. We'll get back to this topic in the section 2.1.3. First, let's look at how immutable values relate to the concept of immutable types that we discussed in chapter 1.

2.1.2 Using immutable data structures

When representing data in functional programs, we'll work with data structures. We'll discuss data structures in chapters 5 and 7. For now, you can imagine that I'm talking about any composite data type, for example a C# value type or even a class, even though data structures are generally a simpler concept. As mentioned in chapter 1, in functional programming these data structures are immutable.

The concept of immutable data structures is very closely related to the concept of immutable value bindings. A typical data structure contains field declarations. If we extend the idea of immutability from variable declarations to field declarations, we get a world where everything is immutable. In C#, you can write immutable class fields using the `readonly` modifier, whereas in F# all data structures are immutable by default. However, F# isn't a *strictly* functional language, so it allows you to create mutable types too.

We've already seen how to work with immutable data structures and how to create an immutable class in C#. Methods of a class or functions working with the data structure cannot modify its state. The only thing that they can do is to return something, so all the operations that work with the data structure return a new value as the result. In C# the `string` type behaves exactly like this. If you write for example `str.Substring(0, 5)`, you'll get a new string value as the result and the original string remains unchanged.

Another thing that I briefly mentioned in the first chapter is that functional code is often written as a single expression rather than a sequence of statements. This makes the code more declarative, so the use of immutable data structures supports this aspect of functional style as well. Let's say we have a class that represents a functional collection. It'll come with an operation that creates an empty list and an operation that "adds" a number to the list. As the list is immutable, adding an element to a list cannot change the original list. Instead, the operation returns a new list containing the items from the original list and the newly added element. If we wanted to create a list and add some elements to it, we could write something like this:

```
var res = ImmutableList.Empty().Add(1).Add(3).Add(5).Add(7);
```

If we wanted to do the same thing with a mutable list, we'd have to create it and then modify it by calling the imperative `Add` method that would modify the list. As a result we'd write one variable declaration and four statements (perhaps 5 lines of source code in total). This example shows that the immutable data structures often help you to write more succinct code. Of course, there are ways for getting similar benefits in imperative languages, but in the functional style, you'll get them without any additional effort.

So far, we've seen that functional languages use immutable data structures and immutable values instead of mutable variables. You can probably imagine how to write some extremely simple programs without using traditional variables and the assignment operator, but once you start thinking about slightly more complicated problems, things become difficult until you change the way you look at the world. In the next section, we'll look how to encode some more sophisticated calculation in the functional style.

2.1.3 Changing program state using recursion

Now, let's look how to write more complicated functions just using values. For example, we'll implement a function that sums numbers in a specified range. This could be of course calculated directly, but we'll use it as an example of a calculation which uses a loop and later we'll also see how to change this code into a more generally useful function:

```
int SumNumbers(int from, int to) {
    int res = 0;
    for (int i = from; i <= to; i++)
        res = res + i;
    return res;
}
```

In this case, we just can't directly replace variable with value bindings, because we need to modify the value during every evaluation of the loop. The program needs to keep certain state, and that state changes on each iteration of the loop, so we can't declare a new value for every change of the state as we did in our earlier example. This means that we need to do a fundamental change in the code and use a technique called *recursion* instead of using loops:

```
int SumNumbers(int from, int to) {
    if (from > to) return 0;
    int sumRest = SumNumbers(from + 1, to);
    return from + sumRest;
}
```

As you probably already know, recursion means that a function (`SumNumbers` in our case) calls itself. In our case this is when we calculate the value of the `sumRest` variable. In this code we're using only value bindings, so it is purely functional. The state of the computation, which was originally stored in a mutable variable, is now expressed using recursion. When I originally mentioned that we can't declare a new variable for every change of the state I was in some sense wrong, because this is what our new recursive implementation does. Every time the function recursively calls itself, it skips a first number and calculates a sum of the remaining numbers. This result is bound to a variable `sumRest`, which is declared as a new variable during every execution of the recursive function.

Of course, writing the recursive part of computation every time would be difficult, so functional languages provide a way for "hiding" the difficult recursive part and expressing most of the problems without explicitly using recursion. We'll get back to this topic in section 2.2.1 after we finish our discussion of calculation of functional programs.

2.1.4 How is the calculation written?

In imperative languages, an expression is simply a piece of code that can be evaluated and yields a result, so for example a method call or any use of a boolean or integer operator is an expression. Conversely, a statement is usually a piece of code that affects the state of the program and doesn't have any result. For example a call to a method that doesn't return any value is a statement, because it just affects the state of the program, depending on whatever the method does. An assignment also changes the state (by changing a value of a variable), but in the simplest version, it doesn't return any value.

NOTE

Actually, an assignment in C# returns a value, so you can write for example `a = (b = 42);` but in the most simple form, which we're discussing here, it is a statement that assigns a value to a variable, without returning anything (e.g. `b = 42;`).

Another example of a typical statement may be returning from a function using `return` or escaping a loop using `break`. Both of these constructs do not have any "return value" and instead, their only meaning is that they change the state of the program – in case of `return` and `break` they change the currently executing statement of the code (`return` by jumping to back to the code which the method and `break` by jumping to just after the end of the loop).

As we already said, in functional languages the state is represented by what a function returns and the only way to modify a state is to return a modified value. Following this logic, in functional languages everything is interpreted as expression with some return value. The practical consequence of this can be nicely demonstrated with the previous example that sums numbers in a specified range. Here is the original version of the code, which uses recursion, but is still not fully functional, because it is written as a series of two statements:

```
int SumNumbers(int from, int to) {
   if (from > to) return 0;
   return from + SumNumbers(from + 1, to);
}
```

We can turn this into a more functional version using the C# conditional operator (?:). This is of course possible only for relatively simple code samples, but we can use it to demonstrate a couple of important points about the syntax of functional language. The listing 2.1 shows the function above rewritten in a more functional way from the syntactical point of view.

Listing 2.1 Summing numbers in the specified range in a "functional C#"

```
int SumNumbers(int from, int to) {
   return                                             #1
```

```
          (from > to)
             ? 0                                              #2
             : (from + SumNumbers(from + 1, to));             #3
    }
```

#1 The method body contains only 'return'
#2 Value for the 'then' case
#3 Expression calculating the 'else' case

There are quite a lot of restrictions that we need to obey to write the code only using expressions in C#, because most of the control flow constructs such as conditional or loops are statements. Even though the example is quite minimalistic, it gives us many useful hints about what we can write in a functional language:

6) The whole body of the method is a single expression which returns a value. In C# this means that the body has to start with `return` (#1). Also, we can't use `return` anywhere else in the code, because that would require jumping to the end of the method from a middle of an expression, which isn't possible.

7) Since "if-then-else" is a statement, we have to use the conditional operator instead. This also means that we have to provide code for both of the cases ((#2) and (#3)). The expression returns a value, but if we omitted the "else" branch and the condition was false, we wouldn't know what to return!

8) The biggest limitation is that a variable declaration in C# is a statement, so we don't have any way for creating variables. The F# language treats value bindings differently and the `let` keyword isn't alone a valid expression. It always has to be attached to some other expression.

We'll get back to value bindings in the beginning of the chapter 3, so you'll see how F# solves the problem we had with variables. Another notable difference in F# is that there is a type that represents "nothing". The `void` keyword in C# isn't an actual type, so you can't for example declare a variable of type `void`. On the other hand, the F# type `unit` is a real type, which has only a single value that doesn't carry any information. All the imperative constructs in F# use this type, so when calling for example the `Console.WriteLine` method, F# treats it as an ordinary expression that returns a value of type `unit`. The fact that everything is an expression makes it easier to reason about the code. We'll take a look at one very interesting technique in the next section.

2.1.5 Computation by calculation

The approach discussed in the last two sections gives us a completely new way of thinking about program execution. To understand how an imperative program executes, we have to understand how its state changes. In a program written using an object-oriented imperative language, the state is not only the internal state of all the objects, but also the currently executing statement (in each thread!) and the state of all the local variables in every stack frame. The fact that the currently executing statement is part of the state is important, because it makes it hard to trace the state when writing the program execution down on paper.

In functional programming, we can use an approach called *computation by calculation*. This is particularly important for Haskell and is described in more detail in The Haskell School of Expression [Hudak, 2000]. Using this approach we start with the original expression (for example a call to a function) and perform a single step, such as replacing the call with a declaration of the function or calculating a result of a primitive mathematical operation. By repeating this step several times, we can directly analyze how the program executes and it is also very intuitive to write this process down.

Listing 2.2 demonstrates how we can use this mechanism to analyze how the `SumNumbers` function computes its result.

The following listing is only pseudo-code displaying sequence of steps, so I'd like to use [CA] (or some other arrow character) here not as a line continuation, but as a continuation of the example.

Listing 2.2 Functional evaluation of an expression `SumNumbers(5,5)`

```
[CA]Start with call "SumNumbers(5,5)"
  SumNumbers(5, 5)

[CA]Expand SumNumbers(5,5)
  if (5 > 5) 0 else { int sumRest = SumNumbers(6, 5) in 5 + sumRest }

[CA]Evaluate the condition, expand the "else" branch
  int sumRest = SumNumbers(6, 5) in 5 + sumRest

[CA]Expand SumNumbers(6, 5)
  int sumRest =
     if (6 > 5) 0
     else { int sumRest = SumNumbers(7, 5) in 1 + sumRest } in
  5 + sumRest

[CA]Evaluate the condition, expand the "then" branch
  int sumRest = 0 in 5 + sumRest

[CA]Replace occurrences of "sumRest" with the actual value
  5 + 0

[CA]Evaluate the expression
  5
```

As you can see, this way of writing down the computation of some functional code is very easy and even though functional programmers don’t spend their lives by writing down how their program executes, it is very useful to get used to this kind of computations, because it gives us a very powerful way of thinking about functional code.

Of course, this example is very simple and it didn’t discuss many important details. Rest assured, we will get to all of these problems in the next chapter. Another interesting aspect of the computation shown in listing 2.2 is deciding which part of the expression should be evaluated next. In this example we used the innermost part of the expression, so we evaluate all arguments of an expression first. This is how most functional languages work, including F#, and is similar to executing the code statement by statement. Before we continue, let me briefly talk a little more about Haskell, a popular functional language known for its close relation to mathematics.

Mathematical purity in Haskell

Haskell appeared in 1990 and has been very popular in the academic community. In this section we've seen that in functional languages, we work with immutable data structures and use immutable values rather than mutable variables. This isn't strictly true in F# because we can still declare mutable values. This is particularly useful for .NET interoperability, because most of the .NET libraries rely on mutable state.

On the other hand, Haskell very strictly enforces mathematical purity. This means it can be very flexible about the order in which programs execute. In the example above, I mentioned that F# evaluates the innermost part of an expression first. In Haskell, there are no side effects so the order of evaluation doesn't (and can't) matter. As long as we're reordering parts of the code that don't depend on each other, it will not change the meaning of the program. As a result, Haskell uses a technique called *lazy evaluation*, which doesn't evaluate the result of an expression until it is actually needed (for example to be printed on a console).

The ability to make a change in the program without changing its meaning is very important in F# too and we'll see how we can use it to refactor F# programs in chapter 11. We'll also see that lazy evaluation can be used in F# as well and it can be very useful optimization technique.

In the last few sections, we were talking about program state and writing calculations using recursion. I promised that we'd see how to write the difficult part of the code in a reusable way, so that's the primary topic for our next section.

2.2 Writing declarative code

In the first chapter, we saw what it means to use a declarative programming style from a high level perspective. Now, we'll talk about more technical concepts of the functional style that enable declarative programming. From this point of view, there are two important aspects that lead to the declarative style. We talked about the first one in the preceding section - we've seen that every language construct is an expression. This shows that functional languages try to minimize the number of built-in concepts and are very succinct and extensible. When talking about recursion, I said that writing every operation using explicit recursion would be difficult. The second aspect gives the answer to this problem, so let's start by looking how to write a single function that can be used in many variations for different purposes.

2.2.1 Functions as values

The question that motivates this section is: "How can we separate the functionality that will vary with every use from the recursive nature of the code which always stays the same?" The answer is simple – we will write the recursive part as a function with parameters and these parameters will specify the "unique operation" that the function should perform.

Let me demonstrate the idea on the `SumNumbers` function. We wrote a function that takes an initial value, looping through a specified numeric range. It calculates a new "state" of the calculation for each iteration using the previous state and the current number from the range. So far we have used zero as an initial value and we used addition as an operation that is used to aggregate the numbers in the range, so a resulting computation for a range from 5 to 10 would look like 5 + (6 + (7 + (8 + (9 + (10 + 0))))).

What if we now decided to modify this function to be more general and allow us to perform computations using different operations? For example, we could then multiply all the numbers in the range together, generating the following computation: 5 * (6 * (7 * (8 * (9 * (10 * 1))))). If you think about the differences between these two computations, you'll see that there are only two changes. First, we changed the initial value from 0 to 1 (because we don't want to multiply any result of a call by zero!) and we changed the operator used during the computation from + to *. Let's see how we could write a function like this in C#:

```
int AggregateNumbers(Func<int, int, int> op, int init, int from, int to) {
    if (from > to) return init;
    int sumRest = AggregateNumbers(op, init, from + 1, to);
    return op(from, sumRest);
}
```

We added two parameters to the function – the initial value (`init`) and an operation (`op`) that specifies how to transform the intermediate result and a number from the range into the next result. To specify the second parameter, we're using a delegate `Func<int, int, int>`, which represents a function which has two parameters of type `int` and returns an `int`. This delegate type is available in .NET 3.5 and we'll talk about it in chapter 3.

In functional languages, we don't have to use delegates, because they have a much simpler concept - a function. This is exactly what the term "functions as values" refers to – the fact that we can use functions in the same way as any other data type available in the language. We can write functions that take functions as parameters (as we did in this example), but also return a function as the result or even create a list of functions and so on. Functions are also very useful as a mental concept when approaching a problem.

Thinking about problems using functions

For many people who know functional programming, the most important thing isn't that functional languages have some particular useful features, but that the whole environment encourages you to think differently and more simply about problems that you encounter when designing and writing applications regardless of the language you use.

The idea of using functions as ordinary values is one of these very useful concepts. Let me demonstrate this using an example. Suppose we have a list of customers and we want to sort it in a particular way. The classical object oriented way to think about this problem is to use a `Sort` method that takes a parameter of some interface type (in .NET this would be `IComparer<Customer>`). The parameter specifies how to compare two elements. Now, if we want to sort the list using customer name, we'd create a class that implements this interface and we'd use it as an argument.

In functional programming, we can use the concept of a function. We've seen that C# can represent similar idea using a delegate, which is definitely simpler than interfaces, but functions are even simpler. They don't have to be declared in advance and the only thing that matters about them is what arguments they take and what results do they return. The generic `Func` delegate in .NET 3.5 is very close to the idea of a function, but once you get used to think about functions, you'll see them more often than when thinking about delegates.

The argument of the functional `Sort` method would be a function that takes two customers as arguments and returns an integer. This is quite brief way to specify the argument. On the other hand, when using an interface or a delegate, we have to declare some type in advance and then refer to it whenever we want to use the object-oriented `Sort` method. Using a function is more straightforward, because when you look at the functional `Sort` method, you immediately see what argument it expects. However, the concept of a function is useful even if you end up implementing the code using interfaces. It gives you a terser way to think about the problem, so the number of elements that you'll have to keep in mind will be lower.

In the first chapter, I said that the declarative style gives us a new way for extending the vocabulary we can use to specify a solution to a class of problems. This is usually best approached by using functions that take other functions as parameters. We'll talk about these in the next section.

2.2.2 Higher order functions

We've seen that we can treat functions as values and also write functions that take other function as parameters. There are several terms that are often used when talking about these kinds of functions. The first concept is treating a function as an ordinary value. This includes the fact that functions have a type (in C# this is a delegate type) and that you can use function as an argument to another function. Frankly, you can just use a function in any place where you can use for example an integer or a string including returning a function as a result or storing functions in a list. This language feature is usually called *first-class functions*, meaning that a function is a value just like any other.

The second important term is *higher order function*. This refers to a function that takes a function as a parameter or returns it as a result. In the C# examples in this book, this will often be a method. For example the method `AggregateNumbers` from the previous section is higher order function. This kind of parameterization of code is used very often in functional languages, so as you'll see, many of the useful functions in the F# library are higher order functions. Let's look at an example that shows how higher order functions make our code more declarative.

EXTENDING THE VOCABULARY USING HIGHER ORDER FUNCTIONS

The best example of how higher order functions make your code more declarative is working with collections. This can be done in C# using the extension methods such as `Where` and `Select` that are provided as part of LINQ, because everything you can write using LINQ query can be also written using a method that takes a `Func` delegate as an argument.

However, in this section we'll look how to write the same code using lists in F# to demonstrate a few interesting aspects of F#. We haven't yet seen enough from F# to fully explain what the code does, but we know enough to see the high-level picture. The first example in listing 2.3 shows how to filter only odd numbers from a list. The second one first filters numbers and then calculates square of every returned number.

Listing 2.3 Working with lists using higher order functions (F# interactive)

```
> let numbers = [ 1 .. 10 ]
  let isOdd(n) = (n%2 = 1)                                 #A
  let square(n) = n * n                                    #B
  ;;
val numbers : int list                                     #C
val isOdd : int -> bool                                    #C
val square : int -> int                                    #C

> List.filter isOdd numbers;;                              #1
val it : int list = [1; 3; 5; 7; 9]

> List.map square (List.filter isOdd numbers);;            #2
val it : int list = [1; 9; 25; 49; 81]
```

#A Is the number 'n' odd?
#B Returns square of a number
#C Type signatures inferred by F# interactive
#1 Filter numbers using 'isOdd' function
#2 Filter and apply 'square' to every number

We first implemented two functions, which we'll use later when working with lists. As I mentioned in chapter 1, the F# compiler automatically deduces the types of expressions that we enter, so it also deduced the type of those functions. However, this isn't important for now. We'll talk about types in F# later in this chapter and we'll talk about the printed type signatures and function declarations in detail in chapter 3.

What I wanted to show with this example is how higher order functions extend our vocabulary when expressing some problem. In the first example, we're using a higher order function `List.filter`, which takes a function as the first argument and a list as the second argument (#1). We give it our function that tests whether a number is odd and a list of numbers from 1 to 10. As you can see on the next line, the result is a list containing all odd numbers in that range.

In the usual imperative style, this could be implemented using a `for` loop or similar construct. As I wrote in the first chapter, imperative languages give us only a limited way to compose basic commands and for loop is one of them. The example we've just seen is interesting because it implements a new control structure for composing commands. The `List.filter` function is an abstract way for describing certain pattern for working with lists, but makes it reusable, because we can specify the behavior of the filter using a function. Higher order functions are essential concept of functional programming and we'll talk about them in chapter 6. We'll see that we can write very useful higher order functions for working with most of the data structures.

In the second example (#2), we use the entire expression from the first example as an argument to another function. This time we use List.map, which applies the function given as the first argument to all values from the given list. In our example this means that it calculates squares of all odd numbers. The code is still very declarative, but it isn't as readable as it should be. One of the reasons for this is that the first construct of the expression is `List.map`, but that's actually the operation that's performed as the last one. However, F# is a very flexible language and it gives us ways to deal with this problem. Let's see how we can use another feature–*pipelining*–to make the code clearer.

LANGUAGE ORIENTED PROGRAMMING

The language oriented programming can be viewed as another programming paradigm, but it is less clearly defined. The principle is that we're trying to write the code in a way that makes it reads more naturally. This can be achieved in languages that provide more flexibility in how you can write the code.

In this section, we'll see that a relatively simple syntactical change can give us a different point of view when thinking about the code. Listing 2.4 shows the new way of writing the same code–we're still returning squares of odd numbers. The example only demonstrates the idea, so you don't have to fully understand it. We'll talk about language oriented programming and list processing in later chapters. The point of this example is to show a different way of thinking about the task.

Listing 2.4 Elegant way for working with functions (F# interactive)

```
> let squared =
     numbers                                        #A
     |> List.filter isOdd                           #B
     |> List.map square                             #C
val it : int list = [1; 9; 25; 49; 81]
```

#A Take the list of numbers
#B Select odd numbers
#C Calculate square of each number

Instead of nesting function calls, we're now using the *pipelining operator* (|>). This construct allows us to write expressions as a series of operations that are applied to the data. The code is of course still written in the usual F# language, but if you didn't know that you could almost think that it is written in some data processing language. It is worth noting that from the F# point of view there is nothing special about the code. F# allows you to write custom operators and the pipelining operator is just an operator that we can define ourselves. The rest of the code is written just using the appropriate parameterized higher order functions.

However, we can look at the list processing constructs (such as `|>`, `List.map` and others) as if it was a separate list processing language embedded in F#. This is what the term "language oriented programming" refers to. Even though the code is completely standard F# library, it looks like a language designed for this particular problem, which makes the code more readable. In fact, many well designed functional libraries look like declarative languages.

The fact that functional libraries look like declarative languages for solving problems in some specific area is a very important aspect of the declarative style. Its great benefit is that it supports division of work in larger teams. You don't have to be an F# guru to understand how to use the list processing "language" or any other library that is already available. This means that even novice F# programmers can quickly learn how to solve problems using an existing library. Implementing the library is more difficult, so this is a task that would be typically done by more experienced F# developers in the team.

Of course, this book aims to train functional masters, so we'll talk about this problem in later chapters. In chapter 6, we'll look at writing higher order functions for working with lists and other basic types. This is a basic technique used when designing functional libraries, but as we've seen in this section, it makes the code look very natural. In chapter 15, we'll make the next step and we'll design a library for creating animation with the goal to make the syntax as natural as possible. Language oriented programming in LISP

LISP appeared in 1958 and is the oldest high-level language still in common use, other than FORTRAN. There are also some popular LISP dialects including Common Lisp and Scheme. The languages from this family are widely known for their extremely flexible syntax which allows LISP to mimic many advanced programming techniques. This includes object-oriented programming, but also some less widely known approaches that you may or may not have heard about, like aspect oriented programming (available today in languages like AspectJ or libraries such as PostSharp) or prototype-based object systems (also seen in JavaScript).

Anything you write in LISP is either a list or a symbol, so you can for example write `(- n 1)`. This is a list containing three symbols: `-`, `n` and `1`. However, it can be viewed as program code: a call to a function "`-`" (binary minus operator) with two arguments: `n` and `1`. This makes the code a little bit difficult to read if you're not used to the syntax, but I wanted to show it here just to demonstrate how far the idea of making the language uniform can be taken. When solving some difficult problem in LISP, you almost always create your own language (based on LISP syntax), which is designed for solving the problem. You can simply define your own symbols with a special meaning and specify how the code written using these symbols executes.

We've seen something slightly similar when talking about declarative animations in chapter 1, so you've seen that you can use a language oriented approach even when writing the code in C#. We'll talk about this example in chapter 15, where we'll see how language oriented programming looks in both C# and F#.

In the declarative programming style, we're extending the vocabulary, which we can use to express our intentions. However, we also need to make sure that the primitives we're adding will be used in a correct way. In the next section, we'll briefly look at types, which serve as "grammar rules" for these primitives.

2.3 Functional types and values

The C# language is a statically typed programming language[†]. This means that every expression has a type known during the compilation. The compiler uses static typing to verify that when the program runs, it will use values in a consistent way. For example, it can guarantee that the program won't attempt to add a DateTime with an integer, because the "+" operator cannot be used with these two types.

In C#, we have to specify the types explicitly most of the time. For example, when writing a method, we have to specify what the types of its parameters are and what the return type is. On the other hand, in F# we don't typically write any types. However, the F# language is also statically typed. In F#, every expression has type as well, but F# uses a mechanism called *type inference* to deduce the types automatically. In fact, static typing in a functional language such as F# guarantees even more than it does in C#.

Types in functional programming

[†] The C# 4.0 adds support for some of the dynamic language features, but even with these features, C# is still a mostly statically typed language.

I said that functional languages treat any piece of code as an expression. As a result, saying that every expression has a type is a very strong statement. It means that any syntactically correct piece of F# code has some type. The type says what kind of results we can get by evaluating the expression, so the type gives us valuable information about the expression.

I also mentioned that types can be viewed as grammar rules for composing primitives. In functional languages, a function (such as the `square` function from the last example) has a type. This type specifies how the function can be used - we can call it with an integer value as an argument to get an integer as the result.

More importantly, the type also specifies how we can compose the function with higher order functions. For example, we couldn't use `square` as an argument for `List.filter`, because filtering expects that the function returns a Boolean value and not an integer. This is exactly what I mean by a grammar rule–the types verify that we're using the functions in a meaningful way.

We'll talk about values and their types primarily in chapter 5. In chapter 6, we'll see how types of higher order functions help us to write correct code. We'll also see that type information can often give us a good clue about what the function does. In the next section, we'll briefly look at the mechanism which allows us to use types without writing them explicitly.

2.3.1 Type inference in C# and F#

When most of the types have a simple name such as `int` or `Random`, there is only a small need for type inference, because writing the type names by hand isn't difficult. However, C# 2.0 supports generics, so you can construct more complicated types. The types in functional languages like F# are also quite complicated, particularly because you can use functions as a value, so there must also be a type that represents a function.

A simple form of type inference for local variables is now available in C# 3.0. When declaring a local variable in earlier versions of C# you had to specify the type explicitly. In C# 3.0 you can very often replace the type name with a new keyword `var`. Let's look at a couple of basic examples:

```
var num = 10
var str = "Hello world!"
```

The first line declares a variable called `num` and initializes its value to 10. The compiler can easily infer that the expression on the right-hand side is of type `int`, so it knows that the type of the variable must also be `int`. Note that this code means exactly the same thing as if you had written the type explicitly. During the compilation, the C# compiler just replaces `var` with the actual type. As I already mentioned, this is particularly useful when working with complex generic types. We can for example write the following:

```
var dict = new Dictionary<string, List<IComparable<int>>>();
```

Without the `var` keyword, you'd have to specify the type twice on a single line - when declaring the variable and when creating the instance of `Dictionary` class. The type inference in C# is limited to local variable declarations. On the other hand, in F# you often don't write any types at all. If the F# type inference fails to deduce some type, then you can specify it explicitly, but this is a relatively rare.

To give you a better idea of how this works, we'll look at a single example. Listing 2.5 shows a simple function that takes two parameters, adds them and formats the result using the `String.Format` method. The listing first shows valid F# code, and then how you could write it in C# if implicit typing were extended to allow you to use the `var` keyword in other places.

Listing 2.5 Implementing methods with type inference

```
let add a b =                                                   #1
   let res = a + b                                              #2
   String.Format("{0} + {1} = {2}", a, b, res)                  #3

var Add(var a, var b) {                                         #4
   var res = a + b;                                             #4
   return String.Format("{0} + {1} = {2}", a, b, res);          #4
}
```

#1 F# version with no types
#2 Add two numbers
#3 Format the returned string
#4 Pseudo-C# version using 'var'

As you can see, the F# syntax is designed in a way that you don't have to write any types at all in the source code (#1). In the pseudo-C# version (#2), we just used the `var` keyword instead of any types and this is in principle what the F# compiler sees when you enter the code. If you paste the code for this function into F# interactive, it will be processed correctly and the F# interactive will report that the function takes two integers as arguments and returns a string. Let's now look how can the F# compiler figure this out.

The first hint which it has is that we're adding the values `a` and `b`. In F#, we can use "+" to add any numeric types or to concatenate strings, but if the compiler doesn't know anything else about the types of values, it assumes that we're adding two integers. From this single expression, the compiler can deduce that both `a` and `b` are integers. Using this information, it can find the appropriate overload of the `String.Format` method. The method returns `string`, so the compiler can deduce that the return type of the add function is also a `string`.

Thanks to the type inference, we can avoid many errors and use all other benefits of static typing (like hints to developers when writing the code), but for almost no price, as the types are inferred automatically in most of the cases. When using F# in Visual Studio, the type inference is running in the background, so when you hover over a value with a mouse pointer, you'll instantly see its type. The background compilation also reports any typing errors instantly, so you'll get the same experience as when writing C# code.

You may be used to using types from other programming languages and you probably already know that there are primitive types (like integer, character or floating point number) and more complicated types composed from these primitive types. Functional languages have usually slightly different set of composed types. We'll talk about all these types in detail in chapter 5, but let me briefly talk about one type which is particularly interesting and used quite frequently.

2.3.2 Introducing the discriminated union type

In this section, we'll talk about the discriminated union type, which is one of the basic functional types. Let's start by looking at a sample where it would be useful. Imagine that you're writing an application that works with graphical shapes. We'll use a simple representation of shape, so it will be a rectangle, an ellipse (defined by the corners of bounding rectangle) or a shape composed from two other shapes.

If you try to think about this problem using the object oriented concepts, you'll probably say that we need an abstract class to represent a shape (let's call it `Shape`) and three derived classes to represent the three different cases (`Ellipse`, `Rectangle` and `Composed`). Using the OO terminology, we now have in our mind four classes that describe the problem. Also, we don't yet know what we'll want to do with shapes. We'll probably want to draw them, but we don't know yet what arguments will we need to do the drawing, so we can't yet write any abstract method in the `Shape` class.

However the original idea was simpler than this full-blown type hierarchy: we just needed to have a representation of a shape with three different cases. We want to define a very simple data structure that we could use to represent the shape–and F# allows us to do exactly that:

```
type Shape =
   | Rectangle of Point * Point       #A
   | Ellipse of Point * Point         #B
   | Composed of Shape * Shape        #C
```

#A Rectangle with left-upper and right-lower point
#B Ellipse with the bounding rectangle
#C A shape composed from two shapes

This code creates a discriminated union type called `Shape`, which is closer to the original intention we had when describing the problem to start with. As you can see, the type declaration contains three different cases that cover three possible representations of the shape. When working with values of this type in F#, we'll write code such as `Rectangle(pt1, pt2)` to create a rectangle. Unlike unions in the C language, the value is tagged, which means that we always know which of the options it represents. As we'll see in the next section, this is quite important for working with discriminated union values.

The usual development process in F# starts by designing data structures needed to keep the program data. We'll talk about this problem in more detail in chapters 7 to 9. In the next section, we'll talk about pattern matching, which is a concept that makes many typical functional programming tasks easy. Even though pattern matching doesn't look like a concept related to types, we'll see that there are some very important connections. Among other things, we can use pattern matching for implementing functions that work with discriminated unions.

2.3.3 Pattern matching

When using functional data types, we know much more about the structure of the type that we're working with. A nice demonstration of this property is a discriminated union – when working with this type, we always know what kind of values we can expect to get (in our previous example, it could be either a rectangle, an ellipse or a composed shape).

When writing functions that work with discriminated unions, we need to specify what the program should do for each of the case. This is in many ways similar to the `switch` statement from C#, but there are several important differences. First let's see how we could use the `switch` statement to work with a data structure mimicking a discriminated union in C#. Listing 2.6 shows how we could print some information about the given shape.

Listing 2.6 Testing cases using 'switch' statement (C#)

```
switch(shape.Tag) {                                                #1
   case ShapeType.Rectangle:
      var rc = (Rectangle)shape;                                   #2
      Console.WriteLine("rectangle {0}-{1}", rc.From, rc.To);
      break;
   case ShapeType.Composed:
      Console.WriteLine("composed");
      break;
}
```

#1 Switch over the type of the shape value
#2 Cast to the appropriate type

The code assumes that the shape type has a property `Tag` (#1), which specifies what kind of shape it represents. This corresponds to F# discriminated unions, where we can also test which of the possible cases the value represents. When the value is a rectangle, we want to print some information about the rectangle. To do this in C#, we first have to cast the shape (which has a type of the abstract base class `Shape`) to the type of the derived class (in our example, it's `Rectangle`) and then we can finally access the properties that are specific for the rectangle. In functional programming we use this type of construct more often than in regular C#, so we'll need an easier way for accessing properties of the specific cases.

The last thing that is worth noting about the example above is that it contains code only for two of the three cases. If the shape represents an ellipse, the `switch` statement won't do anything. This may be the right behavior in C#, but it is not appropriate for functional programs. I said that everything is an expression in functional program, so we could return some value from the functional version `switch`. In that case, we definitely need to cover all cases, because otherwise the program wouldn't know what value to return.

In the listing 2.7, we'll look at the F# alternative to the C# `switch` statement. The construct is called `match` and we'll use it to calculate the area occupied by the shape.

Listing 2.7 Calculating area using pattern matching (F#)

```
match shape with
| Rectangle(pfrom, pto) ->
    rectangleArea(pfrom, pto)                                          #1
| Ellipse(pfrom, pto) ->
    ellipseArea(pfrom, pto)
| Composed(Rectangle(from1, to1), Rectangle(from2, to2))               #2
        when isNestedRectangle(from2, to2, from1, to1) ->              #2
    rectangleArea(from1, to1)                                          #A
| Composed(shape1, shape2) ->                                          #3
    let area1 = shapeArea(shape1)                                      #B
    let area2 = shapeArea(shape2)                                      #B
    area1 + area2 - (intersectionArea(shape1, shape2))                 #B
```

#1 Calculate area of a rectangle
#2 Case for a rectangle nested inside another
#A Optimized version
#3 Remaining case
#B Calculate area of composed shape

The first important difference from the C# switch construct is that in F#, we can deconstruct the value that we're matching against the patterns. In the listing above, it is used in all the cases. The different cases (denoted using the "|" symbol) are usually called *patterns* (or *guards*).

When calculating area of a rectangle (#1), we need to get the two points that specify the rectangle. When using `match`, we can just provide two names (`from` and `to`) and the match construct assigns a value to these names when the shape is represented as a rectangle and the branch is executed. The listing above is very simplified, so it just uses a utility function to calculate the actual number.

The second case is for an ellipse and it is very similar to the first one. However, the next case is more interesting (#2). The pattern that specifies conditions under which the branch should be followed (which is specified between the bar symbol and the arrow "->") is quite complicated for this case. The pattern only matches when the shape is of type `Composed`, *and* both of the shapes that form the composed shape are rectangles. Instead of giving names for values inside the `Composed` pattern, we specify another two patterns (two times `Rectangle`). This is called a *nested pattern* and it proves very useful. Additionally, this pattern also contains a `when` clause which allows us to specify any arbitrary condition. In our example, we call `isNestedRectangle` function, which tests whether the second rectangle is nested inside the first one. If this pattern is matched, we get information about two rectangles. We also know that the second one is nested inside the first one, so we can optimize the calculation and just return the area of the first rectangle.

The F# compiler has full information about the structure of the type, so it can verify that we're not missing any case. If we forgot the last one (#3) it would warn us that there are still valid shapes that we're not handling (for example a shape composed from two ellipses). The implementation of the last case is more difficult, so if our program often composes two rectangles, the optimization in the third case would be quite useful. Similar to first-class functions, discriminated unions and pattern matching are other functional concepts that allow us to think about problems in simple terms.

Thinking about problems using functional data structures

Even though there is no simple way to create a discriminated union type in C#, the concept is still very useful even for C# developers. Once you become more familiar with them, you'll find that many of the programming problems that you face can be represented using discriminated unions.

If you know object oriented design pattern called *Composite*, than you may recognize it in the example above. The shape can be composed from two other shapes, which represents the composition. In functional programming, we'll use discriminated unions more often to represent program data, so in many cases the Composite design pattern will disappear.

If you end up implementing the problem in C#, you can encode a discriminated union as a class hierarchy (with a base class and a derived class for every case). However, mentally, you can still work with the simple concept, which makes thinking about the application architecture easier. In functional programming, this kind of data structure is used more frequently, which is also a reason why functional languages support more flexible pattern matching constructs. The example above demonstrated that the F# `match` expression can simplify implementation of rather sophisticated constructs. We'll see this type of simplification repeatedly throughout the book: an appropriate model and a bit of help from the language can go a long way to keeping code readable.

I mentioned that the F# compiler can verify that we don't have any missing cases in the pattern matching. This is one of the benefits that we get thanks to the static typing of the F# language, but there are many other areas where it helps too. In the next section, we'll briefly review the benefits and we'll look at one example that highlights the goals of compile-time checking in F#.

2.3.4 Compile-time program checking

The well known benefits of using types are that it prevents many of the common mistakes and that the compiled code is more efficient. However, in functional languages there are several other benefits. Most importantly, types are used to specify how functions can be composed with each other. This is not only useful for writing correct code, but it serves as valuable information to the developer as part of the documentation or to the IDE, which can use types to provide useful hints when writing the code. Types in functional languages tell us even more than they do in imperative languages such as C#, because the functional code uses generics more often. In fact, most of the higher order functions are generic. We've seen that thanks to type inference, the types can be very non-intrusive and you often don't have to think about them when coding.

In the next section, I'll show you one example of a feature which nicely demonstrates the goal of types and compile-time program checking in F#. The goal is to make sure that your code is correct as early as possible and to provide useful hints when writing it.

UNITS OF MEASURE

In 1999 NASA's Climate Orbiter was lost because part of the development team used the metric system and another part used imperial units of measure. This was one of the motivations for a new F# feature called *units of measure* which allows us to avoid this kind of issue. We'll talk about units of measure later in chapter 17, but in this section I want to use it to demonstrate how type checking helps when writing F# code. I chose this example because it is easy to explain, but the compile time checking is present when writing any F# code.

The listing 2.8 shows a brief session from the F# interactive. The code shows a calculation that tests whether an actual speed of a car is violating a specified maximum speed.

Listing 2.8 Calculating with speed using units of measure (F# interactive)

```
> let maxSpeed = 50.0<km/h>                                          #A
  let actualSpeed = 40.0<mile/h>                                     #B
  ;;
val maxSpeed : float<km/h>                                           #1
val actualSpeed : float<mile/h>                                      #1

> if (actualSpeed > maxSpeed) then                                   #2
     printfn "Speeding!";;
Error FS0001: Type mismatch.
Expecting a float<mile/h> but given a float<km/h>.                   #3
The unit of measure 'mile/h' does not match the unit of measure 'km/h'  #3

> let mphToKmph(speed:float<mile/h>) =                               #4
     speed * 1.6<km/mile>;;                                          #4
val mphToKmph : float<mile/h> -> float<km/h>                         #4

> if (mphToKmph(actualSpeed) > maxSpeed) then                        #5
     printfn "Speeding!";;
Speeding!
```

#A Maximal allowed speed in km/h
#B Actual speed in mph
#1 Units are part of the type
#2 Is the speed larger?
#3 The types are not compatible
#4 Implement conversion of units
#5 Correct comparison using conversion

The listing starts by declaring two values (`maxSpeed` and `actualSpeed`). The declaration annotates these values with units, so you can see that the first is in kilometers per hour and the second is in miles per hour. This information is captured in the type (#1), so the type of these two values isn't just a `float`, but it is a `float` with additional information about the units.

Once we have these values, we try to compare the actual speed with the speed limit (#2). In a language without units of measure, this would be perfectly valid and the result would be false (because 40 is less than 50), so the driver would escape without a penalty.

However, the F# compiler reports (#3) that we cannot compare these numbers, because `km/h` is a different unit than `mile/h`.

To solve the problem, we have to implement a function that converts the speed from one unit to another. The function takes an argument of type `float<mile/h>`, which means that the speed is measured in miles per hour and returns a float representing speed in kilometers per hour. Once we use this conversion function in the condition (#5) the code compiles correctly and it reports that the actual speed is in fact larger than the allowed speed. If we implemented this as a standalone application (without using F# interactive) we'd get an error complaining about units during the compilation. Additionally, you can see the units in Visual Studio, so it helps you to verify that your code is doing the right thing. If you see that a value that should represent the speed has a type `float<km^2>`, then you very quickly realize that there is something wrong with the equation.

As I mentioned earlier, static type checking isn't present in all functional languages, but it's extremely important for F#. In the last few sections, we quickly looked at the concepts that are important for functional languages and we've seen how some functional constructs differ from similar constructs in the common imperative and object-oriented languages. Some of the features may still feel a bit unfamiliar, but we'll discuss every concept in detail later in the book, so you may return to this overview to regain the "big picture" after you learn more about functional programming details. To round off this overview, I'll briefly talk about lambda calculus which is a foundation of functional programming and the source of many of the concepts we've just seen.

2.4 The foundation of functional programming

As I mentioned in the first chapter, lambda calculus originated in 1930s as a mathematical theory. Nowadays, it is a very important part of theoretical computer science. In logic it is used in tools that assist with the proving and verification of systems (for example in CPU design). It is also used as a simple formal programming language that can be used for explaining precisely how other languages behave.

In the next section, I'll show you a few sample "programs" written in lambda calculus. We'll see that many of the concepts that we just introduced are appear in this "language" in this chapter appear there in their purest and cleanest form. In lambda calculus, the whole "program" is an expression, and functions can take other functions as parameters. By now, both of these should sound very familiar.

I've included this background material because it demonstrates some of the ideas in their purest form. Hopefully you'll find it as interesting as I do–but it's not essential in order to understand the rest of the book.

2.4.1 Introduction to λ-calculus

When Alonzo Church introduced lambda calculus, he attempted to formalize every mathematical construct just using the most essential mathematical concept, a function.

When you write a mathematical function (let's call it for example *f*), which adds ten to any given argument, you write something like this:

f(x) = x+10

However, Church wanted to use functions everywhere. In fact, everything in his formalism was a function. Assigning a name to every function would be impractical, because when everything is written as a function, many functions are used only once. He introduced a notation that allowed a function to be written without giving it a name:

(λx.x+10)

This expression represents a function that takes a single parameter, denoted by the Greek letter lambda followed by the variable name (in our case "*x*"). The declaration of the parameters is followed by a dot and by the body of the function (in our case "*x+10*"). Actually, in the pure lambda calculus, numerals (such as 10) and mathematical operators (such as "+") are also defined using functions, so there is really nothing except functions, which is quite surprising. To make the discussion clear, we'll just use standard numbers and operators. Let's continue with our example function that adds 10 to a given number. Let's say we want to set 32 as an argument and see what the result will be:

(λx.x+10) 32 = 32 + 10 = 42

As we can see, giving an argument to a function (which is called the *application* of a function in lambda calculus) is done by writing the function followed by the argument. When a function is called with some value as an argument, it simply replaces all occurrences of the variable (in our case "*x*") with the value of the argument (in our example "*32*"). This is the expression that follows the first equal sign. Finally, if we look at "+" as a built-in function, then it will be called in the next step, yielding 42 as a result.

The most interesting aspect about lambda calculus–and the cornerstone of functional programming languages–is that any function can take a function as an argument. This means that we can write a function that takes a function (binary operator) and a value as parameter and calls the binary operator with the value as both arguments:

(λop.λx.(op x x))

As you can see, we wrote a function that takes "op" and "x" as arguments. When writing a function with more arguments, we just use the lambda symbol multiple times to declare more variables. In the body of the lambda function we use "*op*" in a position of a function and "*x*" in a place of the first and second argument to the "*op*" function. Let's see what the code does if we give it plus operator as a first argument and 21 as a second argument:

(λop.λx.(op x x)) (+) 21 = (λx.((+) x x)) 21 = (+) 21 21 = 42

A function with multiple arguments is actually a function that takes the first argument (in our case "*op*") and returns a lambda expression, which may be again a function. This means that in the first step, we apply the function (which takes "*op*" as an argument) to the argument "*(+)*". This yields a result that you can see after the first equals sign - as you can see, the "*op*" variable was replaced with a plus sign. The result is however still a function with arguments, so we can continue with the evaluation. The next step is to apply the function with "*x*" as a parameter to a value *21*. The result is an expression "*(+) 21 21*", which is just a little bit odd notation for adding two numbers and it means exactly the same thing as "*21 + 21*", so our final result of this calculation is 42. As you can see, the calculation in lambda calculus continues until there is no function application (a function followed by its arguments) that could be evaluated. Lambda calculus is interesting from a theoretical point of view or to see where the functional ideas came from, but we'll now turn our attention back to the real world and I'll summarize how functional programming looks in F#.

Note: equations in this section look quite ugly using the default font, so please find some nice way to format this. Thanks!

2.5 The F# point of view

Even though F# has its roots in traditional functional languages, it follows a very pragmatic approach. It was influenced primarily by OCaml, but also by Haskell and C# and it was designed as a functional language intended for the .NET platform. This means that it can interoperate very easily with the outside world and also that it can fully access the object oriented features of .NET if you need them. Another implication is that F# can use a large number of libraries available for .NET including, but not limited to advanced 3D graphics and game development technologies (DirectX, XNA), web development tools (ASP.NET, ASP.NET "MVC" Framework) and windows user interface frameworks (Windows Forms, WPF), but is also compatible with the Mono runtime and can be used to develop Mono based applications (for example using Gtk#).

When thinking about F#, you should be aware of the fact that it is primarily a functional language. This means that most of the typical aspects of functional languages discussed in the earlier sections are essential for F# and well written F# programs usually use most of these features together.

The only aspect where F# isn't as strict as other languages is that it doesn't strictly enforce the use of immutable values and immutable data types, even though their use is still the preferred way where possible. This means that you can define a mutable value and also work with objects that have an internal changing state. The reason for this is to allow easy access to all .NET functionality and libraries, which don't always follow the functional style of thinking; using them fluently from a purely functional language would be a bit cumbersome.

This means that F# allows *side-effects* and doesn't have any mechanism for controlling them. In practice, this means that when you want to rely on the mathematical purity of a part of F# program for some reason, you have to explicitly think about side-effects and make sure that they will not cause any problems.

As already mentioned, F# also supports full .NET object-oriented features, but in a way which is orthogonal to the functional approach, so you can take the best from both worlds. In this book, we focus more on functional programming, so we'll discuss only those object-oriented features that are often used together with functional programming and we'll omit some of the advanced object-oriented features of F# that are not used frequently in a well designed F# program.

2.6 Summary

In this chapter, we talked about functional programming in general terms, including its mathematical foundation in lambda calculus. You've learned about the elements that are essential for functional programming languages such as immutability, recursion and using functions as values. We briefly introduced the ideas that influenced the design of these languages and that are to some extent present in almost all of them. These ideas include making the language extensible, writing programs using a declarative style and avoiding mutable state to make it easier to read and also parallelize programs. Even though all of the languages we've discussed are primarily "functional", there are still important differences between them. This is because each of these languages puts emphasis to slightly different combination of the essential concepts mentioned earlier. Some of the languages are extremely simple and extensible, while others give us more guarantees about the program execution.

In the next chapter, we'll see how some of the functional concepts look in practice in F# and how the same ideas can be expressed in C#, so you can see familiar C# code with a functional F# equivalent side-by-side. In particular, we'll talk about functional data structures and look at tuple, which is a basic F# immutable data structure as well at its equivalent in C#. We'll also look at collections of data (called lists in functional languages) and how you can work with them using recursion. We've seen that a single recursive function can be used for various purposes when it takes another function as an argument, so we'll use this technique for writing a universal list processing code.

3
Meet tuples, lists and functions in F# and C#

In the previous chapter we looked at the most important concepts of functional programming, but we did this from a high-level perspective. You haven't seen any real functional code yet, aside from quick examples to demonstrate the ideas. The purpose of the introduction was for you to see how various concepts relate to each other and how the result is a very different approach to programming.

In this chapter we'll finally look at some real functional F# code, but we'll focus on examples that can be also nicely explained and demonstrated using C#. We will not yet go into the deep details of everything; you'll see more information about most of the concepts in the second part of the book.

First we'll look at value bindings in F# and how we can use them to declare a value or a function, so we can write some real F# code later. After this intermezzo we'll turn our attention to aspects that are language neutral starting with immutability–the fact that values cannot be changed after they've been initialized. Next we'll look at the humble list, which proves to be a very useful data structure. I'll demonstrate how you can work with lists recursively–as you might remember from the introduction, recursion is another key aspect of functional programming. Aside from that, we'll also use pattern matching in several examples, so I'll introduce it along the way. Finally, we'll look at how we can treat functions as values, which is the feature that gave the name to the whole functional programming paradigm.

3.1 Value and function declarations

We've already seen several examples of value binding (written using the `let` keyword in F#) in Chapter 1. As you'll see, value binding isn't just a value declaration. It is a very powerful and common construct, used for declaring both local and global values as well as functions. Before writing examples that show functional programming in F#, we need to look at other uses of value binding as well.

3.1.1 Value declarations and scope

As we already know, the `let` keyword can be used for declaring immutable values. We haven't yet talked about a scope of the value, but it's easier to do that with a concrete example. Listing 3.1 is extremely simple, but it's amazing how many nuances can hide in just four lines of code.

Listing 3.1 The scope of a value (F#)

```
let num = 42                                #1
printfn "%d" num
let msg = "Answer: " + (num.ToString())     #2
printfn "%s" msg
```

The code is quite straightforward. It declares two values, where the second (#2) is calculated using the first (#1); it then prints them to the console. What is important for us is the *scope* of the values–that is, the area of code where the value can be accessed. As you would probably expect, the value `num` is accessible after we declared it on the first line (#1) and the value `msg` is accessible only on the last line. You can look at the code and verify that we're using the values only when they are in scope, so our code is correct.

I'll use this code to demonstrate one more thing. The example in listing 3.1 looked quite like C# code, but it's important to understand that F# treats the code very differently. We already touched this topic in previous chapter in section 2.1.4 (How is the calculation written?) where we attempted to write some code in C# only using expressions. We've seen that value bindings have to be treated specially, if we want every valid F# code to be an expression. Indeed, if you wrote code to do the same thing as listing 3.1 in C#, the compiler would see it as a sequence of four statements. Let's now look how F# understands the code. To demonstrate this, I've made few syntactical changes in the code to produce listing 3.2.

Like vertical hedgehog; [Ch_03_Listing_2.2.png] shows what I mean; I can do it in Visio, but I'd prefer leaving it to professionals.

Listing 3.2 Example with let binding with explicit scopes (F#)

```
let num = 42 in
(                                                #1
   printfn "%d" num;                          #2 #1
   let msg = "Answer: " +                        #1 #3
             (num.ToString()) in                 #1 #3
   (                                             #1 #3
      printfn "%s" msg                           #1 #3
   )                                             #1 #3
)                                                #1
```

Annotations below the code with bullets on the left side (as in the guide; [WritingDevices_BulletsSample.vsd] shows an example)

#1 The whole block in the parentheses is an expression inside the first let binding. The binding declares a value 'num', which is in scope in the expression enclosed by parentheses.
#2 Sequencing of expressions can be done explicitly using semicolon. By linking two expressions in a sequence using semicolon, we get a single expression.
#3 The whole let binding is an expression. It declares a value 'msg' which is in scope in the nested expression in parentheses.

There are several obvious changes to the layout, but it's also worth noting the introduction of the `in` keyword after every let binding. This is required if you turn off the default syntax where whitespace is significant[‡]. The other change is that a block of the code following the let binding is enclosed in parentheses. This example is closer to how F# compiler actually understands the code that we wrote. Interestingly, the code in listing 3.2 is still valid F# code with same meaning as earlier. This is because sometimes you may want to be more explicit about the code and using `in` keywords and braces enable this.

What becomes more obvious in listing 3.2 is that the let binding actually assigns a value to a symbol and specifies that the symbol can be used inside of an expression. The first let binding states that the symbol `num` refers to a value `42` in the expression following the in keyword, which is enclosed in braces (#1). The whole let binding is treated as an expression, which returns the value of the inner expression, so for example the whole let binding that defines the value `msg` (#3) is an expression that returns a result of `printfn`. This function has `unit` as a return type, so the result of the whole expression will be a `unit`.

The expression (#3) is preceded by another expression (#2) and as you can see, we added semicolon between these two. The semicolon works as a sequencing operator in F# and when using the lightweight syntax, we don't have to write it. It specifies that the expression preceding the semicolon should be evaluated before the one following it. In our example that means #2 will be evaluated before #3. The expression preceding the sequencing operator should also return a `unit`, because otherwise the returned value would be lost.

So far we have only seen ordinary bindings that declare an ordinary value, but the same let binding is also used for declaring functions and for nested bindings as we'll see in the next section.

‡ The default setting is sometimes called "lightweight syntax". However, F# also supports OCaml-compatible syntax, which is more schematic and which we use in the example. We will not use it in the rest of the book, but in case you want to experiment with it, you can turn it on by adding `#light "off"` directive to the beginning of F# source file.

3.1.1 Function declarations

As noted earlier, we can use let bindings to declare functions. Let's demonstrate this on a fairly simple function that multiplies two numbers given as the arguments. This is how you would enter it to the F# interactive tool:

```
> let multiply n1 n2 =
      n1 * n2;;
val multiply : int -> int -> int
```

To write a function declaration, the name of the symbol has to be followed by one or more argument names. In our example, we're writing a function with two arguments, so the name of the function (`multiply`) is followed by two arguments (`n1` and `n2`). Let's now look at the body of the function. It can be simply viewed as an expression that is bound to the symbol representing a name of the function (`multiply` in our case), with the difference that the symbol doesn't represent a simple value, but instead represents a function with several arguments.

In the previous chapter, we've seen that functions in F# are also just values. This means that when using the `let` construct, we're always creating a value, but if we specify arguments, we declare a special type of value-a function. From a strictly mathematical point of view, an ordinary value is just a function with no arguments, which also sheds more light on the F# syntax. If you omit all the arguments in function declaration, you'll get a declaration of a simple value.

When writing a function, the body of the function has to be properly indented. The indentation is required so that you don't have to use other, more explicit, ways to specify where the function declaration ends, which are used in the OCaml-compatible syntax.

FUNCTION SIGNATURES

One part of the previous example that we haven't discussed yet is the output printed by the F# interactive. It reports that we declared a new value and its inferred type. Because we're declaring a function, the type is a function type written as `int -> int -> int`. This type represents a function that has two arguments of type `int` (two `int`s before the last arrow sign) and returns a result of type `int` (the type after the last arrow sign). We've already seen that F# uses type inference to deduce the type and in this example, it used the default type for numeric calculations (which is an integer). We'll get back to function types in chapter 5 and we'll also explain why parameters are separated using same symbol as the return value.

NESTED FUNCTION DECLARATIONS

Let's now look at slightly more complicated function declaration in listing 3.3, which also demonstrates another interesting aspect of let bindings - the fact that they can be nested.

Listing 3.3 Nested let bindings (F# interactive)

```
> let printSquares msg n1 n2 =
      let printSqUtility n =                     #1
         let sq = n * n                  #3 #2 #1
         printfn "%s %d: %d" msg n sq    #4 #2 #1
```

```
          printSqUtility n1                          #1
          printSqUtility n2;;                        #1
    val printSquares : string -> int -> int -> unit

    > printSquares "Square of" 14 27;;
    Square of 14: 196
    Square of 27: 729
```

One more vertical hedgehog diagram (#3, #4 are ordinary bullets)

The code shows an implementation of a function `printSquares`. As you can see from its signature (`string -> int -> int -> unit`), it takes a `string` as its first argument (`msg`) and two numbers (`n1` and `n2`) as the second and third arguments. The function prints squares of the last two arguments using the first argument to format the output. It doesn't return any value, so the return type of the function is `unit`.

The body of the `printSquares` function (#1) contains a nested declaration of a function `printSqUtility`. This utility function takes a number as an argument, calculates its square and prints it together with the original number. Its body (#2) contains one more nested let declaration which declares an ordinary value called `sq` (#3) which is assigned the square of the argument, just to make the code more readable. It ends with a `printfn` call that prints the message, the original number and the squared number. The first argument specifies the format and types of the arguments (`%s` stands for a string and `%d` stands for an integer).

There is one more important aspect about nested declarations that can be demonstrated with this example. I have already mentioned that the parameters of a function are in scope (meaning that they can be accessed) anywhere in the body of a function. For example, the parameter `msg` can be used anywhere in the range (#1). This also means that it can be used in the nested function declaration and this is exactly what we do inside `printSqUtility` on the fourth line (#4) when we output the numbers using the `msg` value. The nested declarations are of course accessible only inside the scope where they are declared–for example, you cannot use `printSqUtility` directly from other parts of the program. This also guarantees that the `msg` argument will always have a value.

One last aspect of value declarations in F# is that they can be used for declaring mutable values as well. Even though we usually work with immutable values in functional programs, it is sometimes useful to be able to create a mutable value as well.

3.1.2 Declaring mutable values

In the earlier section, we declared a value of type integer by writing `let num = 10`. If you were curious and tried to modify it, you may have tried writing something like `num = 10`. This doesn't work because a single equal's sign outside a let binding is used to compare values in F#. It would be *valid* code, but it would probably return false (unless `num` happened to have the value 10). This makes it seem that modifying an existing value in F# isn't even possible.

This isn't actually true, since F# is very pragmatic and sometimes you may need to use mutable values in F#. This is most likely to happen when optimizing code or using mutable .NET objects. Listing 3.4 shows how immutable and mutable values can be used in F# and what the operator for mutating values looks like.

Listing 3.4 Declaring mutable values (F# interactive)

```
> let n1 = 22;;                          #1
val n1 : int

> n1 <- 23;;                             #2
error FS0027: This value is not mutable.  #2

> let mutable n2 = 22;;                  #3
val mutable n2 : int

> n2 <- 23;;                             #4
> n2;;                                   #4
val it : int = 23
```

Annotations below the code with bullets on the left side

#1 Declare standard value
#2 Immutable values cannot be modified
#3 Declare mutable variable
#4 Modify and show the new value

All values in F# are immutable by default, so when we declare a value using the usual let binding syntax (#1) and then try to modify it using the assignment operator ("`<-`") we get a compile-time error message (#2). To declare a mutable variable, we have to explicitly state this using the `mutable` keyword (#3). We can later change this value using the assignment operator and when we print it, we can see that the value has changed (#4).

You should try to get into the habit of using immutable values wherever possible in F# - only use mutable values when you really have to. This is not because they're necessarily *wrong* as such, but they're not idiomatic. If you can "think functionally" it will lead to more concise code which will be easier to read and reason about. Don't expect this to happen overnight, but the more you work *with* the language instead of fighting its normal idioms, the more you're likely to get out of it.

As I said in chapter 1, the default use of immutability doesn't just influence local value declarations, but also extends to data structures. In the next section we'll look at the most basic immutable types that we use in functional programming.

3.2 Using immutable data structures

An immutable data structure (or object) is a structure whose value doesn't change after it is created. When declaring a data structure that contains some values, these values are stored in slots such as field or value declaration. In functional programming, all these slots are immutable, which leads to the use of immutable data structures. In this section, we'll demonstrate the simplest built-in immutable data type. You'll see more common functional data structures in the upcoming chapters.

I pointed out earlier how we can write a function for processing data using immutable data types or objects. Instead of changing the internal state of the object (which isn't possible, because it is immutable) the processing function simply creates and returns a new object. The internal state of this new object will be initialized to a copy of the original object with a few differences in places where we wanted to change the state. This sounds a little abstract, but you'll see what I mean shortly in an example.

3.2.1 Introducing tuple type

The simplest immutable data structure in F# is a *tuple*[§] type. Tuple is a simple type that groups together several values of (possibly) different types. The following example shows how to create a value (called `tp`), which contains two values grouped together:

```
> let tp = ("Hello world!", 42)
val tp : string * int
```

Creating a tuple value is fairly easy: we just write a comma separated list of values enclosed in parentheses. But let's look at the code in more detail - on the first line, we create a tuple and assign it to a `tp` value. The type inference mechanism of the F# language is used here, so you don't have to explicitly state what the type of the value is. The F# compiler infers that the first element in the tuple is of type string and the second is an integer, so the type of the constructed tuple should be something like "a tuple containing a string as the first value and an integer as the second value". Of course, we don't want to lose any information about the type and if we represented the result just using some type called for example `Tuple`, we wouldn't know that it contains string and integer.

The inferred type of the expression is printed on the second line. You can see that a type of a tuple is in F# written as `string * int`. In general, a tuple type is written as types of its members separated by an asterisk. In the next few sections, we'll see how tuples can be used in F#, but I'll also show you how you can implement exactly the same functionality in C#. If you don't immediately understand everything after reading the F# code, don't worry; just continue with the C# examples, which should make everything clearer.

So, how can we implement the same type in C#? The answer is that we can use C# 2.0 generics and implement a generic `Tuple` type with two type arguments. The C# equivalent

[§] The word “tuple” is usually pronounced with “u” such as in “cup”.

of the F# type `string * int` will then be `Tuple<string, int>`. We'll get to the C# version shortly after discussing one more F# example.

WORKING WITH TUPLES IN F#

Let's now look at some more complicated F# code that uses tuples. In listing 3.5, we use tuples to store information about a city. The first member is a string (the name of the city) and the second is an integer, containing a number of people living there. We implement a function `printCity` which outputs a message with the city name and its population, and finally we create and print information about two cities.

Listing 3.5 Working with tuples (F# interactive)

```
> let printCity cityInfo =                                #1
     printfn "Population of %s is %d."                    #1
             (fst cityInfo) (snd cityInfo)                #1
  ;;
val printCity : string * int -> unit                      #2

> let prague  = ("Prague", 1188126)                       #3
  let seattle = ("Seattle", 594210)                       #3
  ;;
val prague : string * int                                 #4
val seattle : string * int                                #4

> printCity prague                                        #5
  printCity seattle;;                                     #5
Population of Prague is 1188126.
Population of Seattle is 594210.
```

#1 Function that prints information about the city
#2 Inferred type of the function
#3 Create tuples representing Prague and Seattle
#4 Types of created tuples
#5 Print information about the cities

The listing shows a session from the F# interactive, so you can easily try it for yourself. The first piece of code (#1) declares a function `printCity`, which takes information about the city as an argument and prints its value using the standard F# `printfn` function. The formatting string specifies that the first argument is a string and the second is an integer. To read first and second element of the tuple, we use two standard F# functions, `fst` and `snd` respectively (which are obviously acronyms for "first" and "second").

The next line (#2) shows the type of the function deduced by the F# type inference. As we can see, the function takes a tuple as an argument (denoted using asterisk as `string * int`) and doesn't return any value (denoted as `unit` type on the right side of functional arrow symbol). This is exactly what we wanted.

Next, we create two tuple values (#3) that store population information about Prague and Seattle. After these lines are entered, the F# interactive shell prints the types of the newly declared values (#4) and we can see that the values are of the same tuple type that the `printCity` function takes as argument. That means we can pass both of these two values as an argument to our printing function and get the expected result (#5).

The fact that types of the tuple match the parameter type of the function is important, because otherwise the two types would be incompatible and we wouldn't be able to call the function. To demonstrate this, you can try entering the following code in the F# interactive console:

```
let newyork = ("New York", 7180000.5)
printCity newyork
```

I'm not sure how New York could have 7180000 and half of inhabitants, but if this were the case then the type of the tuple `newyork` wouldn't be `string * int` anymore and would instead be `string * float`, as the type inference would correctly deduce that the second element of the tuple is a floating-point number. If you try it, you'll see that the second line isn't valid F# code and the compiler will report an error saying that the types are incompatible.

WORKING WITH TUPLES IN C#

I promised that we'd implement exactly the same code as the previous example in C# as well, so now it's the time to fulfill this promise and write some C#. As I already mentioned, we will represent tuples in C# using a generic type with two type arguments `Tuple<TFirst, TSecond>`, where `TFirst` and `TSecond` are generic type parameters.

The type will have a single constructor with two parameters of types `TFirst` and `TSecond` respectively, so that we can construct tuple values. It will also have two properties for accessing the values of its members, so unlike in F# where we accessed the elements using functions `fst` and `snd`, in C# we'll use properties `First` and `Second`. We skip the implementation for a minute, and instead look at how we can use the type. Listing 3.6 has the same functionality as listing 3.5, but written in C#.

Listing 3.6 Working with tuples (C#)

```
void PrintCity(Tuple<string, int> cityInfo) {             #1
   Console.WriteLine("Population of {0} is {1}.",
      cityInfo.First, cityInfo.Second);                   #2
}

var prague  = new Tuple<string, int>("Prague", 1188000);  #3
var seattle = new Tuple<string, int>("Seattle", 582000);  #3

PrintCity(prague);                                        #4
PrintCity(seattle);                                       #4
```

#1, #2 The 'PrintCity' method takes a tuple of string and int as an argument; in C# we the types of method arguments have to be specified explicitly, so you can see that the type of 'cityInfo' is 'Tuple<string, int>' (#1). The method prints the information using .NET 'Console.WriteLine' method and uses properties of the tuple type ('First' and 'Second') to read its value (#2).

#3, #4 Declares two variables ('prague' and 'seattle') and creates a tuple that stores information about the cities using a constructor with two arguments (#3); The city information are later printed using the 'PrintCity' method (#4)

Annotations with bullets on the left as in previous cases.

The translation from F# code to C# is very straightforward once we have an equivalent for the F# tuple type in C#. The code is slightly more verbose, mainly because we have to explicitly specify the type several times, whereas in the F# example, the type inference mechanism was able to infer the type everywhere. However, we'll shortly see that this can be improved a little bit. We used a new C# 3.0 feature (`var`), which at least lets us use type inference when declaring the `prague` and `seattle` variables (#3), because we're initializing the variables and C# can automatically infer the type from the right-hand side of the assignment.

Just like in the F# code, if we declared a tuple with an incompatible type (for example `Tuple<string, double>`) we wouldn't be able to use it as an argument to the `PrintCity` method. This is more obvious in C#, because we have to explicitly state what the type arguments for the generic parameters of the `Tuple` type are.

3.2.2 Implementing a tuple type in C#

The implementation of the tuple type in C# is quite straightforward. As already mentioned, we're using generics, so that one can create a tuple containing values of any two types. The complete code is shown in listing 3.7.

Listing 3.7 Implementing the tuple type (C#)

```
public sealed class Tuple<TFirst, TSecond> {
   private readonly TFirst  first;                     #1
   private readonly TSecond second;                    #1

   public TFirst  First  { get { return first;  } }
   public TSecond Second { get { return second; } }

   public Tuple(TFirst first, TSecond second) {
      this.first = first;                              #2
      this.second = second;                            #2
   }
}
```

Probably the most notable thing is that the type is immutable. We've already seen how to create an immutable class in C# in the first chapter. In short, we mark all fields of the type using the `readonly` modifier (#1) and provide only getter for both of the properties. Interestingly, this is somewhat opposite to F# where you have to explicitly mark values as mutable. Read-only fields can be set only from the code of the constructor (#2), which means that once the object is created, its internal state cannot be mutated as long both of the values stored in the tuple are immutable as well.

Better type inference for C# tuples

Before moving forward, I'd like to show you one C# trick that makes our further examples that use tuples much more concise. In the earlier examples, we had to create instances of our tuple type using a constructor call which required explicit specification of type arguments. We used the new C# 3.0 `var` keyword, so that the C# compiler inferred the type of variables for us, but we can do even better.

There is one more place where C# supports type inference and that is when calling a generic method. If you're calling a generic method and its type parameters are used as types of the method parameters then the compiler can use the compile-time types of the method arguments when the method is called to infer the type arguments**. To clarify this, let's look at the code showing this in listing 3.8.

Listing 3.8 Improved type inference for tuples (C#)

```
public static class Tuple {
   public static Tuple<TFirst, TSecond>
         Create<TFirst, TSecond>(TFirst first, TSecond second) {
      return new Tuple<TFirst, TSecond>(first, second);
   }
}

var prague  = Tuple.Create("Prague", 1188000);  #1
var seattle = Tuple.Create("Seattle", 582000);  #1
```

The code shows an implementation of a static method `Create`, which has two generic parameters and creates a tuple with values of these types. We need to place this method in a non-generic class, because otherwise we would have to specify the generic parameters explicitly. Luckily, C# allows us to use the name `Tuple`, because types can be overloaded by the number of their type parameters (so `Tuple` and `Tuple<TFirst, TSecond>` are two distinct types).

The body of the method is very simple and its only purpose is to make it possible to create a tuple by calling a method instead of calling a constructor. This allows the C# compiler to use type inference as shown at (#1). The full syntax for calling a generic method includes the type arguments, so using the full syntax we would have to write `Tuple.Create<string, int>(...)`. As the types can be inferred automatically we can omit the type arguments. In the next section, we'll look at writing code that calculates with tuples and since we've just implemented the tuple type in C# we'll start with the C# version of the code and then move on to the F# alternative.

** Sincere apologies for the mess of "type arguments", "method arguments" and so forth in this sentence. Sometimes the terminology defined in specifications just doesn't allow for elegant but accurate prose.

3.2.3 Calculating with tuples

In the examples so far we have just created several tuples and printed the values, so let's perform some calculation now. For example we might want to increment the number of inhabitants by adding a number of newborns for the last year.

As already discussed, the tuple type is immutable, so we cannot set the properties of the C# tuple class. In F#, we can read the values using two functions (`fst` and `snd`), but there are no functions for setting the value, so the situation is similar. This means that our calculation will have to return a new tuple formed by the original name of the city copied from the initial tuple and the incremented size of population.

Let's first see how this can be done in C#. The listing 3.9 shows a new method that we'll add to the generic `Tuple<TFirst, TSecond>` class and several lines of C# code that show how to use this new functionality.

Listing 3.9 Incrementing population of a city (C#)

```
class Tuple<TFirst, TSecond> {
   // ...
   public Tuple<TFirst, TSecond> WithSecond(TSecond nsnd) {  #1
      return Tuple.Create(this.first, nsnd);
   }
}

var prague0 = Tuple.Create("Prague", 1188000);                #A
var prague1 = prague0.WithSecond(prague0.Second + 13195);     #B
PrintCity(prague1);                                           #C
```

#1 Returns tuple with the second value changed
#A Create city information about Prague
#B Create information with incremented population
#C Print the new information

The `WithSecond` method (#1) takes a new value of the second element as an argument and uses the `Tuple.Create` method to create a new tuple with the first element copied from the current tuple (`this.first`) and the second element set to the new value `nsnd`.

Now we'd like to do the same thing in F#. Here, we will write a function `withSecond`, which will do the same thing as a `WithSecond` method from our earlier C# example. It will take a tuple and a new value of the second element and return a new tuple with the first element copied from the original tuple and the second element set to a given value. The code for F# is shown in listing 3.10.

Listing 3.10 Incrementing population of a city (F#)

```
let withSecond tuple nsnd =
   let (f, s) = tuple                                       #1
   (f, nsnd)                                                #2

let prague0 = ("Prague", 1188000)                           #A
let prague1 = withSecond prague0 ((snd prague0) + 13195)    #A
printCity prague1                                           #A
```

#1 Decompose a tuple into two values: 'f' and 's'
#A Increment population and print the new information

The code first shows an implementation of the function `withSecond`. We could implement it simply using the `fst` function, which reads a value of the first element in the tuple, but I wanted to demonstrate one more F# feature that can be used with tuples: *pattern matching*. You can see that inside the function, we first decompose the tuple given as the argument into two separate values (#1) and we call these two values `f` and `s` (for first and second). This is where the pattern matching occurs; on the left-hand side of the equals sign you can see a language construct called a *pattern* and on the right-hand side we have an expression that is matched against the pattern. Pattern matching takes the value of an expression and decomposes it into a values used inside the pattern.

On the next line (#2) we can use the value `f` extracted from the tuple using pattern matching. We reconstruct the tuple using the original value of the first element and the new value of the second element given as an argument (`nsnd`). We will look at more examples of pattern matching on tuples in the next section. Aside from using pattern matching, the code doesn't show anything new, but pattern matching is an important topic and F# provides other ways of using it with tuples, too. Let's take a closer look.

3.2.4 Pattern matching with tuples

In the last example we decomposed a tuple using pattern matching in a let binding. We can slightly improve the code in listing 3.10. Since we didn't actually *use* the second element of the tuple, we only need to assign a name the first one. To do this, we can write an underscore for the second value in the pattern like this:

```
let (f, _) = tuple
```

The underscore is a special pattern that matches any expression and ignores the value assigned to it. Using pattern matching in let bindings is often very useful, but there are other places you can use it too. In fact, patterns can occur almost anywhere an expression is assigned to some value. For example, another place where pattern matching is extremely useful is when we're specifying the parameters of a function. Instead of parameter names, we can use patterns. This makes our `setSecond` function even simpler:

```
let withSecond (f, _) nsnd = (f, nsnd)
```

Now we've shortened our declaration from three lines to one. The result doesn't use any unnecessary values and clearly shows how the data flows in the code. Just from looking at the code, you can see that the first element of the original tuple is copied (by tracing the use of symbol `f`) and that the second function argument is used as a second element of the returned tuple (by following uses of `nsnd`). This is the preferred way of working with tuples in most of the F# functions that we'll write.

One other common use for pattern matching is in an F# `match` expression, which we saw earlier in section 2.3.3. We could rewrite our `withSecond` function to use a `match` expression like this:

```
let withSecond tuple nsnd =
   match tuple with
```

```
    | (f, _) -> (f, nsnd)
```

The `match` construct lets us match the specified expression (`tuple`) against one or more patterns starting with the bar symbol. In our example, we have only one pattern and because any tuple with two elements can be deconstructed into two values containing its elements, the execution will always follow this single branch. The F# compiler analyzes the pattern matching to deduce that the argument `tuple` is a tuple type containing two elements.

NOTE

Keep in mind that you cannot use pattern matching for example to determine whether a tuple has two or three elements. This would lead to a compile-time error, because the pattern has to have the same type as the expression that we're matching against the pattern and the type of a tuple with three elements (for example `int * int * int`) isn't compatible with a tuple that has two elements (for example `int * int`). Pattern matching can be used only for determining run-time properties of values; the number of elements in a tuple is specified by the type of the tuple, which is checked at compile time. If you're wondering how to represent some data type that can have several distinct values then you'll have to wait until chapter 5, where we'll look at unions.

In the previous example we used a pattern that cannot fail, because all tuples of two elements can be deconstructed into individual elements. This is called a *complete pattern in F#*. The `match` construct is particularly useful when working with patterns that are not complete and can fail, because we can specify several different patterns (every pattern on a new line, starting with the bar symbol) and if the first pattern fails, the next one is tried until a successful pattern is found.

What would be an incomplete pattern for tuples? Well, we could write a pattern that matches only when the first element (a city name) is some specific value. Let's say for example there are 100 people in New York that are never counted by any statistical study, so when setting the second element of a tuple (the population of the city) we want to add 100 when the city is New York. You could of course write this using an `if` expression, but listing 3.11 shows a more elegant solution using pattern matching:

Listing 3.11 Pattern matching with multiple patterns (F# interactive)

```
> let setSecond tuple nsnd =
     match tuple with
     | ("New York", _) -> ("New York", nsnd + 100)    #1
     | (f, _) -> (f, nsnd)                            #2
  ;;
val setSecond : string * 'a -> int -> string * int

> let prague = ("Prague", 123)
  setSecond prague 10;;
val it : string * int = ("Prague", 10)                #A
```

```
> let ny = ("New York", 123)
  setSecond ny 10;;
val it : string * int = ("New York", 110)             #B
```

#1 Pattern that matches only New York
#2 Pattern that matches all other values
#A The expected result for Prague
#B Returned population is incremented by 100

You can see that in this example, the match expression contains two distinct patterns. The first pattern contains a tuple with a string "New York" as the first element and underscore as a second (#1). This means that it only matches tuples with a first element set to "New York" and with any value for the second element. When this pattern is matched, we return a tuple representing New York, but with a population which is 100 more than the given argument. The second pattern (#2) is the same as in previous examples and it just sets the second element of the tuple.

The examples following the function declaration shows the code behaving as expected. If we try to set a new population of Prague, the new value of population is used, but when we try to do this for New York, the new value of population is incremented by one hundred.

Tuples are used particularly frequently during the early phase of development, because they are so simple. In the next section, we'll look at another elementary immutable data type: a list. We've seen that a tuple represents a known number of elements with diverse types. Lists work the other way round: a list represents an unknown number of elements of the same type.

3.3 Lists and recursion

Tuple is a very good example of an immutable functional data type, but there is one more property of many functional data types that is worth discussing in this chapter and that is recursion. Let's start with a classic programming joke: What's the dictionary definition of recursion? "Recursion. See recursion."

Recursion appears in functional programming in different forms. It can be present in the structure of the type such as lists. The type that represents functional list is either an empty or it is composed from an element and a list. You can see that the type "list" that we're describing is recursively used in its definition. The second form of recursion is probably more widely know and is used when writing recursive functions. Let's start by looking at one example of the second form and then we'll focus on lists to demonstrate the first form.

3.3.1 Recursive computations

The most common example of a recursive function is probably calculating the factorial of a number. If you're not already familiar with it, here's a short definition: the factorial of a non-negative number *n* is 1 if *n* is one or zero; for larger *n*, the result is factorial of *n* - 1 multiplied by *n*. This function can be implemented essentially in two ways. In C# you can do it using a `for` loop, which iterates over numbers in the range between 2 and n and multiplies some temporary variable by the number in each iteration:

```
int Factorial(int n) {
    int res = 1;
    for(int i = 2; i <= n; i++)
        res = res * i;
    return res;
}
```

This is a correct implementation, but it isn't easy to see that it corresponds to the mathematical definition of the function. The second way to implement this function is, of course, to use a recursion and write a method in C# or a function in F# that recursively calls itself. These two implementations are surprisingly similar, so you can see both of them side-by-side in listing 3.12.

Listing 3.12 Recursive implementation of factorial in C# and F#

```
int Factorial(int n) {              #1       let rec factorial(n) =
    if (n <= 1)                              #1
        return 1;                   #2           if (n <= 1) then
    else                                             1
        return n * Factorial(n-1); #3       #2
}                                                else
                                                     n * factorial(n - 1)
                                             #3
```

#1 Declaration of recursive function or method; In F# we have to explicitly declare that it is recursive by using the 'let rec' binding instead of ordinary 'let'
#2 A case which terminates the recursion and returns 1 immediately
#3 A case which performs the recursive call to a 'factorial' function or 'Factorial' method

Annotations below the code with bullets on the left as earlier

The C# version of the code is very straightforward. The F# version is also quite clear, but as noted in the code comments, we have to explicitly state that the function is recursive using the `rec` keyword. This specifies that the let binding is recursive, making it possible to refer to the name of the value (`factorial`) within the declaration of the function.

In general, every recursive computation should have two branches - a branch where the computation performs a recursive call and a branch where the computation terminates. You can see both of them marked in the previous code listing. Usually, the recursive calculation performs the recursive call several times until a termination condition occurs (in our case this is when we're calculating the factorial of 1) and then returns some constant value or calculates the result using non-recursive code. If the termination condition is incorrect, then the code can keep looping forever or can eventually crash with a stack overflow exception.

Since recursion is absolutely essential for functional programming, functional languages have developed several ways for avoiding stack overflows even for very deep recursive calls and some other optimization mechanisms. This and other advanced topics will be discussed later in the book in Chapter 10.

3.3.2 Introducing functional lists

Now that we're a bit more comfortable with the general principle of recursion, we can look at functional lists in more detail. Earlier I wrote that a list is either empty or composed from an element and another list. This means that we need a special value to represents an empty list, and a way of constructing a list by taking an existing list and prepending an element at the beginning. The first option (an empty list) is sometimes called *nil* and the second option produces a *cons cell* (short for *constructed list cell*). You can see a sample list constructed using an empty list and cons cells in Figure 3.1.

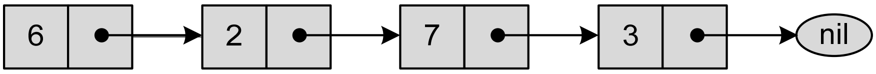


Figure 3.1 Functional list containing 6, 2, 7 and 3. Rectangle represents cons cell, which contains a value and a reference to the rest of the list. Last cons cell references a special value representing an empty list.

[Functional_Ch_03_Figure_3.1.vsd]

As you can see in the figure, every cons cell stores a single value from the list (called *head*) and a reference to the rest of the list (called *tail*), which can be either another cons cell or an empty list (*nil*). Let's now look at several ways that F# offers for creating lists:

```
> let ls1 = []
val ls1 : 'a list = []

> let ls2 = 6::2::7::3::[]
val ls2 : int list = [6; 2; 7; 3]

> let ls3 = [6; 2; 7; 3]
val ls3 : int list = [6; 2; 7; 3]

> let ls4 = [1 .. 5]
val ls4 : int list = [1; 2; 3; 4; 5]

> let ls5 = 0::ls4
val ls5 : int list = [0; 1; 2; 3; 4; 5]
```

At first, we created an empty list, which is written as `[]` in F#. If you look at the result, you can see that F# created a value containing empty list. The type of the list is a bit unclear, because we don't know yet what is the type of values contained in the list, so F# infers that the type is a list of "something". This is called a generic value and we'll talk about it in chapter 5. The second example is much more interesting for now - you can see how lists are created under the covers: we take an empty list and use an operator for creating a cons cell "`::`". Unlike many other operators such as "+", the "`::`" operator is right associative, which means that it composes values from the right to the left. If you read the expression in that direction, you can see that we construct a list cell from a value 3 and an empty list, than use the result together with a value 7 to construct another cell and so on. After entering the expression, F# interactive reports that we created a list of type `int list`. This means that the type of the `ls2` value is a list which contains integers. This is again done using generic types that you may know from C# and we will see how to use them in F# in detail later. In the next two examples, we use a piece of syntactic sugar F# provides for creating lists. The first one uses square braces with list elements separated by a semicolon and the second uses dot-dot to create a list containing a sequence of numbers. Finally, the last example shows how we can use cons cell to create a list by appending values at the beginning of another list. You can see that `ls5` contains 0 at the beginning and then all elements from the `ls4` list.

An important fact about functional lists which I've already mentioned but is worth repeating, is that they are immutable. This means that we can construct a list (as in the previous example) but we cannot take an existing list and modify it, for example by adding or removing an element. Functions that need to add new elements or remove existing ones always return a new list without modifying the original one, because modifying a list is in fact impossible. We'll see more examples of these functions in chapter 8, but for now, let's look at processing the elements in an existing list.

When working with lists in functional languages, the typical code to process a list contains two branches - one branch that performs something when the given list is an empty list and a second branch which performs an operation when the argument is a cons cell. The latter branch generally performs a calculation using the head value and recursively processes the tail of the list. We will see all these common patterns later in this chapter, but first let's see how we can write code that chooses between these two branches using pattern matching.

DECOMPOSING LISTS USING PATTERN MATCHING

When talking about pattern matching on tuples in section 3.3.4, we saw two distinct ways for using it. One method was to write the pattern directly in the let binding, either when assigning the result of an expression to a value, or in the declaration of function parameters. The other method was using the `match` keyword. The important difference between these two is that using `match` we can specify multiple patterns with multiple branches. For lists, we'll need to use the second option, because we need to specify two distinct branches every time we write list processing code (one for an empty list and one for a list which was created using cons cell).

The following code demonstrates pattern matching on lists and prints a message with the value of the first element or "Empty list" when the list is empty:

```
match l with
| []     -> printfn "Empty list"
| hd::tl -> printfn "List starting with %d" hd
```

You can see the pattern that matches an empty list on the second line and a pattern that extracts a head (the value of the first element) and a tail (the list appended after the head) on the third line. Both of these patterns are written with exactly the same syntax that we used earlier for creating the list. An empty list is matched using `[]` and a cons cell is deconstructed using `::` operator. The second pattern is much more interesting, because it assigns a value to two new symbols, `hd` and `tl`. These will contain a number and the rest of the list obtained by decomposing the first cons cell. An empty list doesn't carry any value, so the first pattern doesn't bind a value to any symbol; it just informs us that the original list was empty.

If you look back to figure 3.1, you can see that the first pattern corresponds to the "nil ellipse", which doesn't contains any value. The second pattern matches the "cons cell rectangle" and takes out the contents of its two parts.

As in the example with tuples, the list of patterns is complete, meaning that it can't fail to choose a branch for any given list. Let's now see what happens if we try using an incomplete pattern in listing 3.13.

Listing 3.13 An incomplete pattern matching on lists (F# interactive)

```
> let squareFirst l =
     match l with
     | hd::_ -> hd * hd
  ;;
Warning FS0025: Incomplete pattern matches on this     #1
expression. The value '[]' will not be matched.        #1
val squareFirst : int list -> int                       #A

> squareFirst [4; 5; 6];;                              #2
val it : int = 16

> squareFirst []                                       #3
Exception of type 'Microsoft.FSharp.Core.              #B
  MatchFailureException' was thrown.                   #B
(...)
```

#1 F# detects possible failure
#A Takes list and returns an integer
#2 Success for a non-empty list
#3 Failure for an empty list
#B Exception is thrown on failure

We start by declaring a function called `squareFirst`, which contains a pattern match that matches a cons cell and returns square of the first element from the list. However, this pattern doesn't handle the situation when a list is empty. We can see that the F# compiler is quite smart and when we write a pattern match that can possibly fail it detects this situation and even gives us an example when the match will fail (#1). You shouldn't ignore this warning unless you're absolutely sure that the situation can never occur. Even if the function doesn't have any reasonable meaning for empty lists, it is better to add a handler for the remaining case (you can use underscore character as a pattern that matches any value) and either throw an exception with additional information or just do nothing. (Of course, if the function's return type is anything other than `unit`, you'll have to work out a suitable value to return if you do nothing. Throwing an exception is generally a better idea if the function really shouldn't be called with an empty list.)

Even though there was a warning, F# interactive is willing to crunch the function, so we can try calling it. First, we try a case that should work (#2) and we can see that it behaves as expected. If we call the function with an empty list as an argument (#3) the `match` construct doesn't contain any matching pattern, so it throws an exception. This is a normal .NET exception and can be caught using `try` construct in F#.

You should have some idea what we can expect from functional lists, so in the next section we'll turn our attention to C# and we'll use it to explain lists in detail as well as to write our first list processing code.

3.3.3 Functional lists in C#

To show you how a functional list type works, let's now look how we can implement the same functionality in C#. There are several ways for representing the fact that list can be either empty list or a list with a *head* and a *tail*. The clear object oriented solution, would be to write an abstract class `FuncList` with two derived classes for representing the two cases - for example `EmptyList` and `ConsList`. However, to make the code as simple as possible, we'll use just a single class, with a property `IsEmpty` that will tell us whether the instance contains a value or not. Note that every instance of the `FuncList` type contains just a single value, when it is a cons cell or no value at all, when it is an empty list. You can see the implementation in listing 3.14.

Listing 3.14 Functional list (C#)

```
public class FuncList<T> {
   public FuncList() {                                         #1
      IsEmpty = true;
   }
   public FuncList(T head, FuncList<T> tail) {                 #2
      IsEmpty = false;
      Head = head;
      Tail = tail;
   }
   public bool IsEmpty { get; private set; }                   #3
   public T Head { get; private set; }                         #A
   public FuncList<T> Tail { get; private set; }               #A
}

public static class FuncList {                                 #4
    public static FuncList<T> Empty<T>() {
        return new FuncList<T>();
    }
    public static FuncList<T> Cons<T>(T head, FuncList<T> tail) {
        return new FuncList<T>(head, tail);
    }
}
```

#1 Constructor that creates an empty list
#2 Constructor that creates a cons cell
#3 Empty list or a cons cell?

#A Properties of the cons cell
#4 Utility class for constructing lists

The `FuncList<T>` class is a generic C# class, so it can store values of any type. It has a property called `IsEmpty` (#3), which is set to `true` when we're creating an empty list using the parameter-less constructor (#1). The second constructor (#2) takes two arguments, creates a cons cell and sets `IsEmpty` to `false`. The first argument (`head`) is a value that we're storing in the cons cell. The second argument (`tail`) is a list following the cons cell that we're creating. The tail has the same type as the list we're creating, which is written as `FuncList<T>`. The first constructor corresponds to the F# empty list (written as `[]`) and the second one creates cons cell in the same way as the double colon operator (`head::tail`).

As already mentioned, functional lists are immutable, so all properties of the class are read-only. We're implementing all of them using C# 3.0 *automatic property* feature, which generates getter and setter of the property for us, but we're specifying that the setter should be private, so they cannot be modified from outside. To make the type truly read-only, we set the values of the properties only in constructors, so once a list cell is created, none of its properties can change. This demonstrates that immutability is really just a concept that we can use in different ways and not a language feature. When using automatic properties, we will lose the checking that the C# compiler can do when we use fields marked using `readonly` as a tradeoff for a more convenient syntax.

Just like with our previous tuple example, I've included a non-generic utility class `FuncList` (#4) with static methods that simplify creation of generic lists by providing methods for creating an empty list (`Empty`) and one for creating a cons cell (`Cons`). The advantage of using this class is that C# can infer the type arguments for a method call, so we don't have to specify what type is carried by the type if it is obvious from the context. Now that we have a C# implementation of the list, we can write some code that uses lists to perform some computation.

3.3.4 Functional list processing

So far we have discussed what the functional list type looks like and how it can be implemented in C#. Now it's the time to write code that actually does something with functional lists. Suppose that we wanted to implement a method `SumList` in C# (or a `sumList` function in F#) that sums all the numbers in a list.

SUMMING NUMBERS IN A LIST WITH C#

If you were used to imperative programming in C# and were working with the standard .NET array or the `List<T>` class from `System.Collections.Generic`, you'd probably create a variable called `total` initialized to zero and write a for loop that iterates over all the elements adding every element to the total (something like `total += list[i]`). Alternatively, you could do this using `foreach` loop, which is a syntactic sugar that makes this a bit easier to write, but the idea is still the same.

But how can we do this using our functional list, where we can't access elements by index and which doesn't support `foreach`[††]? To do this, we can use recursion and write a method with code for the two cases - when the list is empty and when the list is a cons cell. You can see the code for the C# version of `SumList` in listing 3.15.

Listing 3.15 Summing list elements (C#)

```
int SumList(FuncList<int> numbers) {
   return numbers.IsEmpty ? 0 :                                     #1
      numbers.Head + SumList(numbers.Tail);                         #2
}

var list = FuncList.Cons(1, FuncList.Cons(2, FuncList.Cons(3,       #A
   FuncList.Cons(4, FuncList.Cons(5, FuncList.Empty<int>())))));    #A

int sum = SumList(list);                                            #B
Console.WriteLine(sum);                                             #B
```

#1 Sum of empty list is 0
#2 A branch for a cons cell #A Create a list storing 1,2,3,4,5
#B Calculate the sum and prints '15'

The `SumList` method first checks whether the list is empty. If the list is non-empty, the branch that matches cons cell (#2) is executed. It recursively calls `SumList` to calculate the sum of elements in the tail (which is a list) and adds this result to the value stored in the head. This recursive call is performed until we reach the end of the list and find an empty list as a tail. For an empty list (#1), the function terminates and returns zero.

Later in the listing, we create a list using the utility methods `Cons` and `Empty` from the non-generic `FuncList` class. The creation is a bit cumbersome, but you could make it simpler by implementing a method to create a functional list from a normal .NET collection, for example.

SUMMING NUMBERS IN A LIST WITH F#

Now that we know how the code looks in C#, we can try implementing exactly the same functionality in F#. Let's look at the listing 3.16, which shows an F# function `sumList` and a few F# interactive commands for testing it.

Listing 3.16 Summing list elements (F# interactive)

```
> let rec sumList(lst) =
     match lst with                                       #A
     | []         -> 0                                    #1
```

[††] We could of course add support for the `foreach` statement to our code and it would be desirable to do so for a real-world `FuncList<T>` type. However, let's first look at the key concepts for list processing from functional programming.

```
     | hd::tl -> hd + sumList(tl)                        #2
val sumList : int list -> int                            #3

> let list = [ 1 .. 5 ]                                  #B
val list : int list

> sumList(list)                                          #C
val it : int = 15
```

#A Pattern matching on the list
#1 Sum of empty list is 0
#2 A branch for a cons cell
#3 Takes list of integers and returns an integer
#B Create list for testing
#C Calculate the sum and print it

If you compare the code with the previous C# implementation you'll find many similarities. As in the previous case, there are two branches, one for an empty list (#1) and one for a cons cell (#2), which is implemented using recursion. The notable difference is that in F# we can use pattern matching for selecting an execution path. Pattern matching also extracts values from the cons cell, so once the execution enters the second branch, `head` and `tail` values are already available. This adds to the robustness of the code: you can't use values which haven't been matched by a pattern. It sounds trivial, but it prevents the code from accidentally trying to access the (non-existent) elements of an empty list. Pattern matching is a very natural construct in functional languages and there is no corresponding feature in C#, so we had to use an `if` statement to implement the same behavior.

Also, F# type inference was helpful once again: we didn't have to specify the types explicitly anywhere in the code. As you can see it correctly inferred that the function takes a list of integers and returns an integer (#3). It used the fact that we're testing whether `lst` value is an empty list or a cons cell to deduce that it is a list. Because one branch returns zero it knows that the whole function returns an integer and because we're adding elements of the list together, it deduces that the argument is a list containing integers.

The recursion which we used in this section is very important, but writing everything using recursion explicitly would be difficult. In the next section we're going to look at a mechanism that allows us to hide the difficult recursive parts of the code.

3.4 Using functions as values

In the last section, we were talking about immutable lists and we've seen how to write a function that processes a list recursively. In this chapter, we'll look at one more essential concept of functional programming and that is treating functions as values. In this section, we'll see why it is so useful to work with functions this way and what it actually means to treat a function as a value. More information about functions will follow later in chapter 5.

3.4.1 Processing lists of numbers

Imagine that we wanted to write a method similar to the `SumList` discussed in the previous section, but which instead of adding all the numbers together, would multiply them.

Making this change looks quite easy. We can just copy the SumList method and then tinker with it a bit. There are in fact only two changes in the modified method:

```
int MultiplyList(FuncList<int> numbers) {
   if (numbers.IsEmpty) return 1;                          #1
   else return numbers.Head * MultiplyList(numbers.Tail);  #2
}
```

The first change is that we're using multiplication instead of addition in the branch that does the recursive call (#2) and the second change is that the value returned for an empty list is now one instead of zero (#1). As I mentioned in chapter 2, this solution works, but copying blocks of code is a bad practice. Instead, we'd like to write a parameterized method or function that can do both adding and multiplying of the list elements depending on the parameters. This allows us to hide the difficult recursive part of the list processing routine in a re-usable function and writing SumList or MultiplyList will become a piece of cake.

This example is similar to one that we discussed in section 2.2.1. The solution is to write a method or a function that takes two arguments: the initial value and the operation which should be performed when aggregating the elements. Let's look how we can implement this idea in C#.

PASSING A FUNCTION AS AN ARGUMENT IN C#

We've seen that in C#, this can be done using delegates and in particular using the Func delegate. In listing 3.17, the delegate will have two arguments of type int and will return an int as a result. The code shows how we can implement the aggregation as a recursive method that takes a delegate as a parameter.

Listing 3.17 Adding and multiplying list elements (C#)

```
int AggregateList(FuncList<int> list, int init, Func<int,int,int> op) {
   if (list.IsEmpty)
      return init;                                               #1
   else {
      int rest = AggregateList(list.Tail, init, op);              #2
      return op(rest, list.Head);                                 #2
   }
}

static int Add(int a, int b) { return a + b; }                   #A
static int Mul(int a, int b) { return a * b; }                   #A

var list = FuncList.Cons(1, FuncList.Cons(2, FuncList.Cons(3,     #B
    FuncList.Cons(4, FuncList.Cons(5, FuncList.Empty<int>())))));  #B

Console.WriteLine(AggregateList(list, 0, Add));                  #C
Console.WriteLine(AggregateList(list, 1, Mul));                  #C
```

#1 Return initial value for empty list
#2 Branch for a non-empty list
#A Methods for testing 'AggregateList'
#B Initialize a sample list
#C Summing prints 15 and multiplying 120

Let's look at the `AggregateList` method in a detail first. It takes the input list to process as the first parameter and the next two parameters of specify what should be done with the input. The second parameter is the initial value, which is an integer. It is used in the case when a list is empty (#1) and we just want to return the initial value from the method.

The last parameter is a delegate and is used in the other branch (#2). Here we first recursively calculate the aggregate result for the rest of the list and then call the `op` delegate to calculate the aggregate of that result and the head of the list. In the examples later, it would either add or multiply the given parameters. The delegate type that we're using here is generic `Func<T1, T2, TResult>` delegate from .NET 3.5, which is further discussed in chapter 5. Briefly, it allows us to specify what the number and types of the arguments as well as the return type using .NET generics. This means that when we call `f` (#2) the compiler knows we should provide two integers as arguments and it will return an integer as a result.

Later in the code, we declare two simple methods that are compatible with the delegate type - one for adding two numbers and one for multiplying them. The rest of the code shows how to call the `AggregateList` method to get the same results as those returned by `SumList` and `MultiplyList` in the earlier examples.

Of course, writing the helper methods this way is a bit tedious, because they are not used anywhere else in the code. In C# 2.0, you can use anonymous methods to make the code nicer and in C# 3.0 we have even more elegant way for writing this code using lambda expressions. Lambda expressions and the corresponding feature in F# (called lambda functions) are used almost everywhere in a real functional code, so we'll discuss them much more fully in chapter 5. In the next section, we're going to look at the last code example in this chapter and we'll see how to implement the same behavior in F#.

PASSING A FUNCTION AS AN ARGUMENT IN F#

The function `aggregateList` in F# will be quite similar to the method that we've already implemented. The important distinction is that F# supports passing functions as arguments to other functions naturally, so we don't have to use delegates for this.

Function is a special kind of type in F#. Similarly to tuples, the type of a function is constructed from other basic types. In case of tuple, the type was specified in code using an asterisk between the types of the elements (e.g. `int * string`). In the case of functions, the type is specified in terms of the types of arguments and the return type. This gives type safety in the same way as delegates do in C#. For example a function that takes a number and adds 1 to it would be of type `int -> int`, meaning that it takes integer and returns an integer. The type of a function that takes two numbers and returns a number would be of type `int -> int -> int` and this is exactly the type of the first parameter in our `aggregateList` function. Listing 3.18 shows the F# version of the example.

Listing 3.18 Adding and multiplying list elements (F# interactive)

```
> let rec aggregateList (f:int -> int -> int) init list =           #1
```

```
        match list with
        | []     -> init                                         #2
        | hd::tl ->
            let rem = aggregateList f init tl                    #3
            f rem hd                                             #3
      ;;
val aggregateList : (int -> int -> int) -> int -> int list -> int   #4

> let add a b = a + b                                            #A
  let mul a b = a * b                                            #A
  ;;
val add : int -> int -> int                                      #B
val mul : int -> int -> int                                      #B

> aggregateList add 0 [ 1 .. 5 ];;                               #C
val it : int = 15                                                #C
> aggregateList mul 1 [ 1 .. 5 ];;                               #C
val it : int = 120                                               #C
```

#1 The 'f' argument is a function
#2 Empty list branch
#3 Non-empty list branch
#4 Inferred signature of the function
#A Functions for addition and multiplication
#B Signature is compatible with the 'f' argument
#C Test the function immediately

Just like the C# version of the code, the first two parameters of the function specify how the elements in the list are aggregated. The second parameter is the initial value and the first one is an F# function. In this example, we wanted to make the function only work with integers to make the code more straightforward, so we added a type annotation for the first parameter (#1). It specifies that the type of the `f` function is a function taking two integers and returning an integer.

Next we see the familiar pattern for list processing: one branch for an empty list (#2) and one for a cons cell (#3). After entering the code for the `aggregateList` function in the F# interactive, it prints a signature of the function (#4). This kind of signature may look a bit daunting the first time you see it, but you'll soon become familiar with them. In figure 3.2 you can see what each part of the signature means in a graphical form.

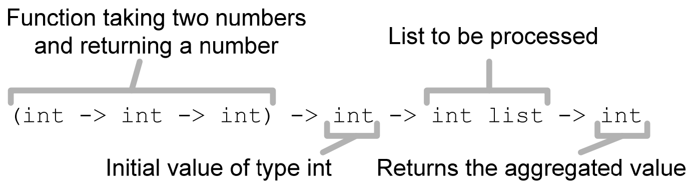

Function taking two numbers and returning a number
List to be processed
(int -> int -> int) -> int -> int list -> int
Initial value of type int
Returns the aggregated value

Figure 3.2 Type signature of the aggregateList function in detail. The first argument specifies how two numbers are aggregated, the second is an initial value and the third argument is a list to be processed.

Finally, we write two simple functions (add and mul) that both have a signature corresponding to the type of the first parameter of aggregateList and verify that the function works as expected. I wrote these two functions just to make the sample look exactly like the previous C# version, but F# allows us to take any binary operator and work with it as if it were an ordinary function. This means that we don't need to write the add function and we can instead just use the plus symbol directly:

```
> aggregateList (+) 0 [ 1 .. 5 ];;
val it : int = 15
```

This is often quite helpful and working with operators makes F# code very succinct. There are also several operators for working with lists that don't have any corresponding equivalent in C#; we'll see some of them in chapter 8. Note that when using an operator in place of a function, it has to be enclosed in parentheses, so instead of just writing "+", we had to write "(+)".

You may be thinking that aggregateList isn't a particularly useful function and that there aren't many other uses for it other than adding and multiplying elements in a list, but the next section shows one surprising example.

BENEFITS OF PARAMETERIZED FUNCTIONS

Let's look at one additional example that will use this function for something very different–something that at first glance seems very different to adding or multiplying the elements of a list. Let's see if we can work out the largest value…

```
> aggregateList max (-1) [ 4; 1; 5; 2; 8; 3 ];;
val it : int = 8
```

The function that we used as a first argument (max) is a built-in F# function that returns the larger from two numbers given as arguments. We used -1 as an initial value, because we expect that the list contains only positive numbers. The program first compares -1 with 3 and returns the larger of these two. In the next iteration it takes the current value (the result of the previous comparison, which is 3), compares it with 8 and returns the larger. In the next step, 8 is compared with 2, then with 5 and so on. Similarly, you could easily find the smallest element in a list by using min as a first argument and some large number (for example Int32.MaxValue) as the second argument.

In fact, the function can be made even more useful by allowing the caller to use something other than an integer during the aggregation. You can see that the body of aggregateList function doesn't state anywhere that the aggregated value should be integer and the only place where this is specified is in the type annotation for the f parameter. It specifies that the function returns an integer, so F# knows that the aggregated value will be an integer, but we could simply remove the type annotation and make the code more general. This is a powerful feature of the F# language called generalization and we'll see how to use it in chapter 6.

3.5 Summary

In this chapter we've looked at some of the essential functional constructs and techniques in practice. We started with value and function declarations using let bindings, showing how F# minimizes the number of concepts that you have to work with–from a strictly mathematical point of view, an immutable value is just a function with no arguments.

Next, we looked at the simplest immutable data structure used in functional languages: the tuple. We used it to demonstrate how you can work with immutable data structures–when you perform a calculation with an immutable data structure, you can't modify the existing instance, but you can create a new instance by copying the original values and replacing those that were newly calculated. The next interesting immutable data type that we encountered was a list. This helped us to explore recursion, both in terms of how to construct one list from another and in using pattern matching to process a list recursively.

Writing the same recursive processing whenever we want to perform an operation on lists would be inconvenient, so we looked at a mechanism that allows us to make the code general and useful for a broader range of similar use cases. The mechanism is called higher order functions. It means that a function can be simply parameterized by another function which is given to it as an argument.

Altogether, this chapter was just a sneak preview showing some of the most important functional techniques in action in their most simplistic form. We've also seen that most of them can be quite well written in C# too. Now that you have an idea of the "look and feel" of functional programming, we'll examine the F# language and tools in more detail, so that you can play with them and try writing some code on your own.

The examples from this chapter we're just a brief overview, so we'll get back to all of the concepts mentioned here later in the book. Other common functional data types will be discussed in chapter 5 and in chapter 6 we'll talk mostly about higher order functions that can be used for working with them. In these two chapters, we'll also see how to make the code more general by using not only generic types, but also generic functions.

4 Exploring F# and .NET libraries by example

Even though we've looked only at the most basic F# language features so far, we already know enough to write a simple application. In this chapter we won't introduce any new functional language constructs; instead we'll look at practical aspects of developing .NET applications in F#. You probably already know how to write a similar application in C#, so all code in this chapter will be in F#.

As we write our first real-world application in F#, we'll explore several functions from the F# library and also learn how to access .NET classes. The .NET platform contains many libraries and all of them can be used from F#. In this chapter we'll look at several examples, mainly in order to work with files and create the user interface for our application. We'll come across several other .NET libraries in the subsequent chapters, but after reading this one you'll be able to use most of the functionality provided by .NET from your F# programs, because the technique is usually the same.

4.1 Drawing pie charts in F#

The application we'll develop can be used for drawing pie charts. You can see the screenshot of the finished program in figure 4.1. It loads data from a CSV file and performs some pre-processing in order to calculate percentage of every item in the data source. Then it plots the chart and allows the user to save it as a bitmap file. We could of course use some library to display the chart (and we'll do exactly that in chapter 13), but by implementing the functionality ourselves, we'll learn quite a lot about F# programming and using .NET libraries from F# code.

The implementation of the application is divided into three parts. In section 4.2 we implement loading information from a file and performing basic calculations on the data. In this first section, we'll use the tuple and list types that we introduced in the previous chapter. Next, in section 4.3 we add some simple console-based output, so we can see the results of the calculations in a human-readable form. Finally, in section 4.4 we add a graphical user interface, drawing charts of the data. We'll use the standard .NET Windows Forms library to implement the user interface, and the `System.Drawing` namespace for drawing.

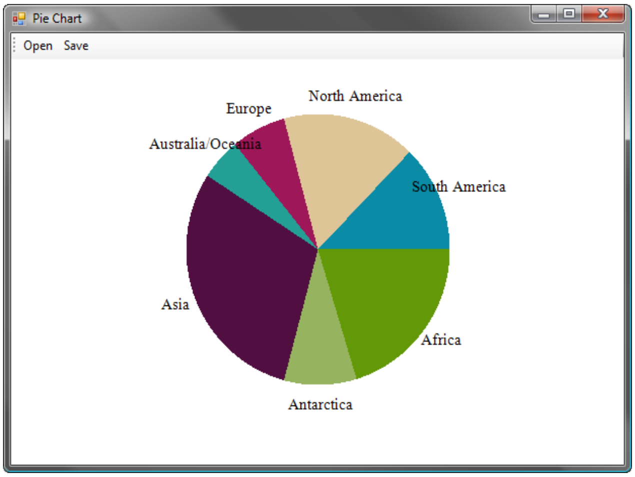


Figure 4.1 Running F# application for drawing pie charts developed in this chapter. It shows distribution of the world population between continents.

Even though you're only reading the fourth chapter out of sixteen, the code that we'll write will be very close to what you'd do if you wanted to develop an application like this after reading the entire book. You wouldn't probably use recursion explicitly as often, because this can be achieved in a simpler way which we'll see in the next chapters but the rest would be the same. This is because F# code is often developed in an iterative way: you start with the simplest possible way to solving the problem and later refine it to fit your advanced needs. Many people prefer developing F# code like this because it allows you to get interesting results as soon as possible. Of course, unless you're writing the code just as a script for a single use, you have to do some refactoring later to make the code well organized and more readable, but the ability to quickly write a working prototype for a problem is very useful.

One benefit of iterative development is that you can easily test your application when writing the first version as we'll see in this chapter. Another benefit is that it is much easier to correctly design the whole application if you already know how the core parts look in the prototype. Also, F# and the Visual Studio IDE are perfect tools for this kind of development. You can simply start writing the code in Visual Studio and execute it using F# interactive to see whether it works as you expected and later start wrapping this experimental code in modules or types.

4.2 Loading and processing data

From the previous description you probably already have a good idea of what kind of data we'll use in our application. It works with a series of elements containing a title to be displayed in the chart and a number. It will load the data from a simplified CSV file which contains a single element per line in the format which you can see in listing 4.1.

Listing 4.1 CSV file with population information

```
Asia,44579000
Africa,30065000
North America,24256000
South America,17819000
Antarctica,13209000
Europe,9938000
Australia/Oceania,7687000
```

CSV files like this one are supported by many spreadsheet editors including Excel, so if you save the file with CSV extension, you can easily edit it. Our application will only support basic files. We'll assume that values are separated using commas and that there are no commas or quotation marks in the titles. This would make the file format more complicated, leading to more complex parsing code.

Let's start off by writing F# functions to read the file in this format and perform basic calculations on the loaded data. We'll develop the code interactively, which will allow us to test every single function immediately after writing it.

4.2.1 Writing and testing code in FSI

As a first step, we'll implement a function `convertDataRow`, which takes a single row from the CSV file as a string and returns two components from the row in a tuple. Immediately after implementing the function, we test it by giving it a sample input that should be correctly parsed (a string "Testing reading,1234"). You can see the code for this function and the result of our test in listing 4.2.

Listing 4.2 Parsing a row from the CSV file (F# interactive)

```
> open System;;
> let convertDataRow(str:string) =                         #1
     let cells = List.of_seq(str.Split(','))         #2
     match cells with
     | lbl::num::_ ->                                       #A
        let numI = Int32.Parse(num)
        (lbl, numI)
     | _ -> failwith "Incorrect data format!"              #B
  ;;
val convertDataRow : string -> string * int                 #3
```

```
> convertDataRow("Testing reading,1234");;              #4
val it : string * int = ("Testing reading", 1234)
```

#1 Type annotation specifies the type of the parameter
#2 Split the string into a list
#A It should have two or more cells
#B Otherwise report an error
#3 Inferred signature of the function
#5 Test the function immediately

After starting the F# interactive, we first import functionality from the .NET `System` namespace. We need to open the namespace because the code uses the .NET functions `Int32.Parse and String.Split`. These have to be imported explicitly, whereas the functions from the core F# libraries, such as `List.of_seq` are available implicitly.

The function `convertDataRow` takes a string as an argument and splits it into a list of values using comma as a separator. We're using standard the .NET `Split` method to do this (#3). The F# compiler needs to know that `str` is a string before we can use this method and in this case, type inference doesn't have any way to infer this, so we're using type annotation (#2) to explicitly state the type of `str`.

The method is declared using the C# `params` keyword and takes a variable number of characters as arguments. We specify only a single separator and that's the comma character. The result of this method is an array of strings, but we want to work with lists, so we convert the result to list using the `of_seq` function from the F# `List` module. We'll talk about arrays and other collection types later in chapters 10 and 12.

Once we have the list, we use the `match` construct to test whether it is in correct format. If it contains two or more values it will match the first case (`lbl::num::_`). The title will be assigned to a value `lbl`, the numeric value to `num` and the remaining columns (if any) will be ignored. In this branch we use `Int32.Parse` to convert a string to an integer and return a tuple containing the title and the value. The second branch throws a standard .NET exception.

If you look at the signature (#4), you can see that the function takes a string and returns a tuple containing a string as the first value and an integer as the second value. This is exactly what we expected - the title is returned as a string and the numeric value from the second column is converted to an integer. The next line demonstrates how easy it is to test the function using F# interactive (#5). The result of our sample call is a tuple containing "Testing reading" as a title and 1234 as a numeric value.

Working with .NET strings in F#

When working with strings in F#, you'll usually use the normal .NET methods. Let's see how we can use them from F#, starting with a few selected static methods available in the `String` class. We can use these as if they were ordinary F# functions (using the `String` prefix). The arguments to these functions should be specified in parentheses as a comma separated tuple. In the type signatures, tuples are written using asterisks:

9) **String.Concat** (overloaded) Accepts variable number of arguments of type string or

object and returns a string obtained by concatenating all of them:

```
> String.Concat("1 + 3", 3);;
val it : string = "1 + 33"
```

10) **String.Join** (sep:string * strs:string[] -> string) Concatenates an array of strings supplied as the strs parameter using a separator specified by sep; we can use the [| ... |] syntax to construct an array literal:

```
> String.Join(", ", [| "1"; "2"; "3" |]);;
val it : string = "1, 2, 3"
```

Strings in .NET are also objects and they also have instance members too. These can be used from F# using the typical dot-notation. We've already seen this in the previous example when splitting a string using str.Split. The following examples assume that we have a string value str containing "Hello World!":

11) **str.Length** Property that returns the length of the string; properties are accessed in F# a same way as in C#, so the call reading the property is not followed by braces:

```
> str.Length;;
val it : int = 12
```

12) **str.[**int index**]** Indexing into a string, which can be written using square braces; returns the character at the location specified by index idx. Note that you still need the dot before the opening brace, unlike in C#:

```
> str.[str.Length - 1];;
val it : char = '!'
```

We can also use various functions that are available in the FSharp.PowerPack.dll library. These are partly available for compatibility with OCaml, but some of them are still useful, because they are designed with F# in mind. However, most of the string processing code in F# can be implemented using .NET methods.

In the previous listing we implemented the convertDataRow function, which takes a string containing a line from the CSV file and returns a tuple containing a label and a numeric value. As a next step we'll implement a function that takes a list of strings and converts each string to a tuple using convertDataRow. Listing 4.3 shows the function–and a test immediately afterwards, of course, parsing a sample list of strings.

Listing 4.3 Parsing multiple lines from the input file (F# interactive)

```
> let rec processLines(lines) =                          #1
     match lines with
     | [] -> []                                          #2
     | str::tail ->                                      #3
        let row = convertDataRow(str)                    #A
        let rest = processLines(tail)                    #B
        row :: rest
  ;;
```

```
val processLines : string list -> (string * int) list #4

> let tst = processLines ["Test1,123"; "Test2,456"];; #5
val tst : (string * int) list =
    [("Test1", 123); ("Test2", 456)]
```

#1 Recursive function
#2 A branch for an empty list
#3 A branch for a cons cell
#A Process the head of the list
#B Recursively process the tail
#4 Inferred type signature
#5 Test the processLines function

This function is in many ways similar to functions for processing lists that we implemented in the previous chapter. As you can see, the function is declared using `let rec` keyword (#1), so it is recursive. It takes a list of strings as an argument (`lines`) and uses pattern matching to test whether the list is an empty list or a cons cell. For an empty list, it directly returns an empty list of tuples (#2). If the pattern matching executes the branch for a cons cell (#3), it assigns a value of the first element from the list to value `str` and list containing the remaining elements to value `tail`. The code for this branch first processes a single row using the `convertDataRow` function from previous listing and then recursively processes the rest of the list. Finally the code constructs a new cons cell: it contains the processed row as a head and the recursively processed remainder of the list as a tail. This means that the function executes `convertDataRow` for each string in the list and collects the results into a new list.

To understand better what the `processLines` function does, we can also look at the type signature printed by F# interactive (#4). It says that the function takes a list of strings (`list string` type) as an argument and returns a list containing tuples of type `string * int`. This is exactly the type returned by the function that parses a row, so it seems that the function does the right thing! Of course, we verify this by calling it with a sample list as an argument (#5). You can see the result of the call printed by F# interactive - it is a list containing two tuples with a string and a number, so the function works well.

Now we have a function for converting a list of strings to a data structure that we'll use in our chart drawing application. Before writing the code to read data from a file and print labels together with the proportion of the chart occupied by each item (as a percentage), we need to implement one more utility function. The function `countSum` in listing 4.4 sums the numeric values of all the items in the list. Later, we'll need this sum when calculating percentage of each item.

Listing 4.4 Calculating a sum of numeric values in the list (F# interactive)

```
> let rec countSum(rows) =
      match rows with
      | [] -> 0                                #A
      | (_, n)::tail ->                        #1
         let sumRest = countSum(tail)          #B
         n + sumRest
  ;;
```

```
val countSum : ('a * int) list -> int           #2

> let sum = countSum tst;;                       #C
val sum : int = 579
> 100.0 / float(sum) * 123.0;;                   #3
val it : float = 21.24352332
```

#A For an empty list return zero
#1 Pattern to extract the current value
#B Recursively sum elements of the tail
#2 Inferred type signature
#C Test the function
#3 Calculating percentage

This function exhibits the recurring pattern for working with lists yet again. Of course, writing code that follows the same pattern over and over may suggest that we're doing something wrong (as well as being boring–repetition is rarely fun). Ideally, we should only write the part that makes each version of the code unique without repeating ourselves. This objection is valid for the previous example and we can write it in a more elegant way. We'll learn how to do this in the upcoming chapters and you can find the improved version (as you'd write it after reading the whole book) on the book's web site. Nevertheless, you'll still need both recursion and pattern matching in many functional programs, so it's useful to look at one more example and become familiar with these concepts.

For an empty list, the function `countSum` simply returns 0. For a cons cell, it recursively sums values from the tail (the original list minus the first element) and adds the result to a value from the head (the first item from the list). The pattern matching in this code demonstrates one interesting pattern that is worth discussing. In the second branch (#1), we need to decompose the cons cell, so we match the list against `head::tail` pattern. However, the code is more complicated than that, because at the same time, it also matches the head against pattern for decomposing tuples, which is written as `(first, second)`. This is because the list contains tuples storing title as the first argument and numeric value as the second argument. In our example, we want to read the numeric value and ignore the title, so we can use the underscore pattern to ignore the first member of the tuple. If we compose all these patterns into a single one, we get `(_, n)::tail`, which is what we used in the code.

If we look at the function signature printed by the F# interactive (#2), we can see that the function takes a list of tuples as an input and returns an integer. The type of the input tuple is `'a * int`, which means that the function is generic and works on lists containing any tuple whose second element is an integer. The first type is not relevant, because the value is ignored in the pattern matching. The F# compiler makes the code generic automatically in situations like this using a feature called *automatic generalization*. We'll learn more about writing generic functions and automatic generalization in chapters 5 and 6.

The last command from listing 4.3 prepared the way for the test in listing 4.4–why enter test data more than once? Having calculated the sum to test the function, we finally calculate the percentage occupied by the record with a value 123. Because we want to get the precise result (21.24%), we convert the obtained integer to a floating point number using a function called `float`.

Converting and parsing numbers

F# is a .NET language, so it works with the standard set of numeric types available within the platform. The following list shows the most useful types that we'll work with. You can see the name of the .NET class in bold and the name used in F# in braces:

13) **Int32**, **UInt32** (int, uint32) Standard 32-bit integer types; literals are written in F# as `42` or `42u` (unsigned); there are also 16 and 64bit variants written as `42s` and `42us` for 16bit and `1L` or `1UL` for 64bit
14) **Double**, **Single** (float/float32) Represent a double precision and a single precision floating point number; the literals are written as `3.14` and `3.14f` respectively. Note the difference between F# and C# here - `double` in C# is `float` in F#; `float` in C# is `float32` in F#.
15) **SByte**, **Byte** (sbyte/byte) Signed and unsigned 8-bit integers; the literals are written as `1y` (signed) and `1uy` (unsigned)
16) **Decimal** (Decimal) Floating decimal point type, appropriate for financial calculations requiring large numbers of significant integral and fractional digits. Literals are written as 1M.
17) **BigInteger**, **Math.BigNum** (bigint/bignum) Types for manipulating with numbers of an arbitrary size; literals are written as `1I` (for an integer) and `1N` (for a rational). The BigInt type is new in .NET 4.0, the BigNum type is available in the F# library.

Conversion from a string to a standard .NET numeric type can be done using the `Parse` method. This method is available in a .NET class corresponding to the numeric type that can be found in a `System` namespace. For example, to convert a string to an integer you can write `Int32.Parse("42")`. This method throws an exception on failure, so there is also a second method called `TryParse`. Using this method, we can easily test whether the conversion succeeded or not as you can see in the following example:

```
let (succ, num) = Int32.TryParse(str)
if succ then
   Console.Write("Succeeded: {0}", num)
else
   Console.Write("Failed")
```

Unlike C#, the F# compiler doesn't insert automatic conversions between distinct numeric types when precision cannot be lost. F# also doesn't use a type-cast syntax for explicit conversions, so we have to write all conversions as function calls. The F# library contains a set of conversion functions that typically have a same name as the F# name of the target type. The following list shows a few of the most useful conversion functions:

18) **int** - Converts any numeric value to an integer - the function is polymorphic, which means that it works on different argument types; we can for example write `(int 3.14)` for converting float value to an integer or `(int 42uy)` for converting a byte value
19) **float**, **float32** - Convert a numeric value to a double-precision or a single-precision floating point number; it is sometimes confusing that `float` corresponds to .NET `Double` type and `float32` to .NET `Single` type

This is by no means a comprehensive reference for working with numbers in F# and .NET. It should contain information about the most commonly used numeric types and functions. For more information you can refer to the standard .NET reference or the F# online reference [F# Website].

In listing 4.4 we ended with an equation that calculates the percentage of one item in our test data set. This is another example of iterative development in F#, because we'll need exactly this equation in the next section. We tried writing the difficult part of the computation to make sure we could do it in isolation: now we can use it in the next section. We'll start by writing code to read the data from a file and then use this equation as a basis for code to print the data set to the console.

4.3 Creating a console application

Writing a simple console-based output for our application is a good start, because we can do it relatively easily and we'll see the results quickly. In this section, we'll use several techniques that will be important for the later graphical version as well. Even if you don't need console-based output for your program, you can still start with it and later adapt it into a more advanced, graphical version as we'll do in this chapter.

4.3.1 Working with input and output

We have already finished most of the program in previous section by writing common functionality shared by both the console and graphical versions. We have a function `processLines` that takes a list of strings loaded from the CSV file and returns a list of parsed tuples and a function `countSum`, which sums the numerical values from the data set. In the last listing, we also tried to write the equation for calculating the percentage, so the only remaining tasks are reading data from a file and printing output to a console window. You can see how to put everything together in the listing 4.5.

Listing 4.5 Putting the console-based version together (F# interactive)

```
> open System.IO;;

> let lines = List.of_seq(File.ReadAllLines(@"C:\Ch03\data.csv"));;    #1
val lines : string list

> let data = processLines(lines);;                                      #A
```

```
val data : (string * int) list =
  [("Asia", 44579000); ("Africa", 30065000); ("North America", 24256000);
   ("South America", 17819000); ("Antarctica", 13209000);
   ("Europe", 9938000); ("Australia/Oceania", 7687000)]

> let sum = float(countSum(data));;                                   #B
val sum : float = 147553000.0

> for (lbl, num) in data do                                           #3
     let perc = int((float(num)) / sum * 100.0)                       #C
     Console.WriteLine("{0,-18} - {1,8} ({2}%)",                      #C
                        lbl, num, perc)                               #C
  ;;
Asia                 - 44579000 (30%)
Africa               - 30065000 (20%)
North America        - 24256000 (16%)
South America        - 17819000 (12%)
Antarctica           - 13209000 (8%)
Europe               -  9938000 (6%)
Australia/Oceania    -  7687000 (5%)
```

#1 Read the content as a list of lines
#A Convert lines to a list of tuples
#B Sum the numeric values
#3 Iterate over all elements
#C Calculate the percentage and print it

The listing starts by opening the `System.IO` namespace, which contains .NET classes for working with file system. Next, we use the class `File` from this namespace and its method `ReadAllLines` (#1), which provides a very simple way for reading text content from a file, returning an array of strings. Again we use the `of_seq` function to convert the array to a list of strings. The next two steps are fairly easy, because they just use the two functions we implemented and tested in previous sections of this chapter–we process the lines and sum the resulting values.

Let's now look at the last piece of code (#3). It uses a `for` loop to iterate over all elements in the parsed data set. This is similar to the `foreach` statement in C#. The expression between keywords `for` and `in` isn't just a variable though, it's a pattern. As you can see, pattern matching is more common in F# than you might expect! This particular pattern decomposes a tuple into a title (the value called `lbl`) and the numeric value (called `num`). In the body of the loop, we first calculate the percentage using the equation that we tested in listing 4.4 and then output the result using the familiar .NET `Console.WriteLine` method.

Formatting strings in F# and .NET

String formatting is an example of a problem that can be solved in two ways in F#. The first option is to use functionality included in the F# libraries. This is compatible with F# predecessors (the OCaml language), but it's also designed to work extremely well with F#. The other way is to use functionality available in .NET Framework, which is sometimes richer then the corresponding F# functions. The `printfn` function, which we've used in earlier examples, represents the first group and Console.WriteLine from the last listing is a standard .NET method.

When formatting strings in .NET we need to specify a *composite format string* as the first argument. This contains placeholders which are filled with the values specified by the remaining arguments. The placeholders contain index of the argument and optionally specify alignment and format. Two of the most frequently used formatting methods are Console.WriteLine for printing to console and String.Format, which returns the formatted string:

```
> let s = String.Format("Hello {0}! Today is: {1:D}", name, date);
val s : string = "Hello Tomas! Today is: Sunday, 15 March 2009"
```

The *format string* is specified after the colon. For example {0:D} for date formatted using the long date format, {0:e} for scientific floating point or {0:x} for hexadecimal integer). The alignment is specified after the comma and it is one of the cases where .NET formatting is used from F#:

```
> Console.WriteLine("Number with spaces: {0,10}!", 42);;
Number with spaces:         42!
> Console.WriteLine("Number with spaces: {0,-10}!", 42);;
Number with spaces: 42        !
```

Aside from the specification of alignment and padding, the .NET libraries are frequently used from F# when formatting standard .NET data types such as the DateTime type or the DateTimeOffset, which represents the time relatively to the UTC time zone. The following example briefly recapitulates some of the useful formatting strings:

```
> let date = DateTimeOffset.Now;;
val date : DateTimeOffset = 03/15/2009 16:37:53 +00:00
> String.Format("{0:D}", date);;
val it : string = "Sunday, 15 March 2009"
> String.Format("{0:T}", date);;
val it : string = "16:36:09"
> String.Format("{0:yyyy-MM-dd}", date);;
val it : string = "2009-03-15"
```

The F#-specific functions for formatting strings are treated specially by the compiler, which has the benefit that it can check that we're working correctly with types. Just like in .NET formatting, we specify the format as a first argument, but the placeholders in the format specify just the type of the argument. There is no index, so placeholders have to be in the same order as the arguments. In F#, you'll often work with printf and printfn that output the string to the console (printfn adds a line break) and sprintf, which returns a formatted string:

```
printfn "Hello %s! Today is: %A" name date
let s = sprintf "Hello %s! Today is: %A" name date
```

The following list shows the most common types of placeholders:

20) **%s** - the argument is of type `string`
21) **%d** - any signed or unsigned integer type (e.g. `byte`, `int`, `ulong`, …)
22) **%f** - floating point number of type `float` or `float32`
23) **%A** - outputs value of any type by calling the .NET `ToString` method

Choosing between the .NET and F# approach is sometimes difficult. In general it is usually better to use the F# function, because it has been designed to work well with F#. If you need some functionality that isn't available or is hard to achieve using F# functions, you can switch to .NET formatting methods, because both can be easily used from F#.

Instead of running everything from F# interactive, we could turn the code from the previous listing into a standard console application. If you we're writing the code in Visual Studio and executing it in F# interactive by hitting **Alt+Enter**, you already have the complete source code for the application. The only change that we can do to make it more useful is to read the file name from the command line. In F#, we can read command line arguments using the standard .NET `Environment.GetCommandLineArgs` method. The first element is the name of the running executable, so to read the first argument, we can write `args.[1]`.

In this section, we added a simple console-based output for our data processing application. Now, it is the time to implement the graphical user interface using the Windows Forms library and finally to draw the pie chart using classes from the `System.Drawing` namespace. Thanks to our earlier experiments and the use of F# interactive during the development, we already know that a significant part of our code works correctly! If we were to write the whole application from a scratch, we would quite possibly already have several minor, but hard to find bugs in the code. Of course, in later phase of the development process, we could turn these interactive experiments into unit tests. We'll talk about this topic briefly in chapter 11.

4.4 Creating a Windows Forms application

Windows Forms is a standard library for developing GUI applications for Windows and is nicely integrated with functionality from the `System.Drawing` namespace. This allows us, among other things, to draw graphics and display them on the screen. The .NET ecosystem is quite rich, so we could use other technologies as well. Windows Presentation Foundation (WPF) which is part of .NET 3.0 can be used for creating more visually attractive user interfaces that use animations, rich graphics or even 3D visualizations.

4.4.1 Creating the user interface

For this chapter we're using Windows Forms, which is in many ways simpler, but using other technologies from F# shouldn't be a problem for you. The user interface in Windows Forms is constructed using components (like `Form`, `Button` or `PictureBox`) so we're going to start by writing a code that builds the user interface controls. This task can be simplified by using a graphical designer, but our application is quite simple, so we'll write the code by hand. In some user interface frameworks including WPF, the structure of controls can be described in an XML-based file, but in Windows Forms, we're just going to construct the appropriate classes and configure them by specifying their properties.

Before we can start, we need to configure the project in Visual Studio. By default, the F# project doesn't contain references to the required .NET assemblies, so we need to add references to `System.Windows.Forms` and `System.Drawing`. This can be done using "Add Reference" option in the Solution Explorer. Also, we don't want to display the console window when the application starts. You can go to project properties and select "Windows application" option in the "Output type" drop-down list. After configuring the project, we can write the first part of the application as shown in listing 4.6.

Listing 4.6 Building the user interface (F#)

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

let main = new Form(Width = 620, Height = 450, Text = "Pie Chart")   #1

let menu = new ToolStrip()                                          #A
let btnOpen = new ToolStripButton("Open")                           #B
let btnSave = new ToolStripButton("Save", Enabled = false)          #B
menu.Items.Add(btnOpen)                                             #B
menu.Items.Add(btnSave)                                             #B

let img =
   new PictureBox                                                   #C
      (BackColor = Color.White, Dock = DockStyle.Fill,              #C
       SizeMode = PictureBoxSizeMode.CenterImage)                   #C

main.Controls.Add(menu)                                             #D
main.Controls.Add(img)                                              #D

// TODO: Drawing of the chart & user interface interactions         #2

[<STAThread>]                                                       #3
do
   Application.Run(main)                                            #F
```

#1 Create the main application form
#A Construct the application menu
#B Add two buttons to the menu
#C Construct control for displaying pie chart
#D Add controls to the main form
#3 Needed for all WinForms applications
#F Start application with a main form

#B, #C, #D inline annotations with vertical lines (if possible), #2 cueball without inline text

The listing starts by opening .NET namespaces that contain classes used in our program. Next, we start creating the controls that represent the user interface. We start with constructing the main window (also called *form*) (#1). We're using an F# syntax that allows us to specify properties of the object directly during the initialization. This makes the code shorter, but also hides side-effects in the code. Internally, the code first creates the object using a constructor and then sets the properties of the object specified using this syntax, but we can view it as single operation that creates the object. When creating the form, we're using parameterless constructor, but it is of course possible to specify arguments to the constructor too. You can see this later in the code when creating `btnSave`, whose constructor takes a string as an argument. A similar syntax for creating objects is now available in C# 3.0 as well and has an interesting history on the .NET platform.

The listing continues by constructing the menu and `PictureBox` control, which we'll use for showing the pie chart. We're not using F# interactive this time, so there is a placeholder in the listing (#2) marking the spot where we'll add code for drawing the charts and for connecting the drawing functionality to the user interface.

The final part of listing 4.6 is a standard block of code for running Windows Forms applications (#3). It starts with a specification of threading model for COM technology, which is internally used by Windows Forms. This is specified using a standard .NET attribute (`STAThreadAttribute`) so you can find more information about it in the .NET reference. In C#, we would place this attribute before the `Main` method, but in F# the source can contain code to be executed in any place. Since we need to apply this attribute, we're using a `do` block, which groups together the code to be executed when the application starts.

Constructing classes in F#, C# 3.0, and Cω

We already mentioned that some GUI frameworks use XML to specify how the controls should be constructed. This is a common approach, because constructing objects and setting their properties is very similar to constructing an XML node and setting its attributes. This similarity was a motivation for researchers working on a language Cω [Meijer et. al, 2003] in Microsoft Research in 2003, which later motivated many features that are now present in C# 3.0. In Cω, we could write a code to construct `ToolStripButton` control like this:

```
ToolStripButton btn = <ToolStripButton>
                        <Text>Save</Text>
                        <Enabled>True</Enabled>
                        <Image>{saveIco}</Image>
                      </ToolStripButton>
```

In Cω, the XML syntax was integrated directly in the language. The elements nested in the `ToolStripButton` node specify properties of the object and the syntax using curly braces allows us to embed usual non-XML expressions in the XML-like code. The ease of constructing objects in this way motivated C# 3.0 feature called *object initializers*:

```
var btn = new ToolStripButton("Save"){ Enabled = false, Image = saveIco };
```

It no longer uses XML based syntax, but the general idea to construct the object and specify its properties is essentially the same. Moreover, we can also specify arguments of the constructor using this syntax, because the properties are specified separately in curly braces. In listing 4.6 we've seen that the same feature is available in F# as well:

```
let btn = new ToolStripButton("Save", Enabled = false, Image = saveIco)
```

The only difference from C# 3.0 is that in F# we specify properties directly in the constructor call. The arguments of the constructor are followed by a set of key-value pair specifying the properties of the object.

Another way to parameterize construction of a class, but also any ordinary method call, is to use named arguments. The key difference is that names of the parameters are part of the constructor or method declaration. Named parameters can also be used to initialize immutable classes, because they don't rely on setting a property after the class is created. This feature is available in F# and you can find more information in the F# documentation. In C#, named arguments are being introduced in version 4.0 and the syntax is similar to specification of properties in F#. However, it is important to keep in mind that the meaning is quite different.

So far, we've implemented a skeleton of the application, but it doesn't actually *do* anything yet–at least, it doesn't do anything with our data. In the next section, we're going to fill in the missing part of the code to draw the chart and display it in the existing `PictureBox` called `img`.

4.4.2 Drawing Graphics

The application will draw the pie chart in a two steps. In the first step, it will draw the filled pie and in the second step it will add the text labels. This way we can be sure that the labels are never covered by the pie.

A large part of the code that performs the drawing can be shared by both of the steps. For each step, we need to iterate over all items in the list to calculate the angle occupied by the segment of the pie chart. The functional solution to this problem is to write a function that performs the shared operations and takes a drawing function as an argument. The code calls this function twice. The drawing function in the first step fills segments of the pie chart and the one in the second step draws the text label.

CREATING RANDOM COLOR BRUSHES

Let's start by drawing the pie. We want to fill specified segments of the pie chart using random colors, so first we'll write a simple utility function that creates a randomly colored brush that we can use for filling the region, as shown in listing 4.7.

Listing 4.7 Creating brush with random color (F#)

```
let rnd = new Random()                                                #1
let randomBrush() =                                                 #2
   let r, g, b = rnd.Next(256), rnd.Next(256), rnd.Next(256)        #A
   new SolidBrush(Color.FromArgb(r,g,b))                            #B
```

#1 Initialize random number generator
#2 Returns a brush with random color
#A Generate R,G,B components of a color
#B Return a solid brush

The code declares two top-level values. The first one is an instance of a .NET class `Random`, which is used for generating random numbers (#1). The second top-level value is a function `randomBrush` (#2). It has a `unit` type as a parameter, which is an F# way of saying that it doesn't take any meaningful arguments. The only possible unit value is `()`, so when calling the function later in the code, we're actually giving it unit as an argument, even though it looks like a function call with no arguments at all. The `randomBrush` function uses the `rnd` value and generates `System.Drawing` object, which can be used for filling of specified regions. It has side-effects and as you already know, we should be careful when using side-effects in functional programs.

Hiding the side-effects

The function `randomBrush` is an example of a function with side-effects. This means that the function may return a different result every time it is called, because it relies on some changing value that other the function arguments. In this example, the changing value is the value `rnd`, which represents a random number generator and changes its internal state after each call to the `Next` method. The previous code listing declares `rnd` as a global value despite the fact that it is used only in function `randomBrush`. Of course this is a hint that we should declare it just locally to minimize the number of global values. We could try rewriting the code as follows:

```
let randomBrush() =
    let rnd = new Random()
    let r, g, b = rnd.Next(256), rnd.Next(256), rnd.Next(256)
    new SolidBrush(Color.FromArgb(r,g,b))
```

But this code doesn't work! The problem is that we're creating a new `Random` object every time the function is called and the change of the internal state is not preserved. When created, `Random` initializes the internal state using the current time, but since the drawing is performed very quickly the "current time" doesn't change enough and we end up with the whole chart being drawn in the same color.

Of course, there is a way to write the code without declaring `rnd` as a global value, but which allows us to keep the mutable state represented by it between the function calls. To write this, we need two concepts that will be discussed in Chapter 5 - a closure and a lambda function. You can find this improvement in a more evolved version of the application available online.

Now that we know how to create brushes for filling the chart, we can finally take a look at the first of the drawing functions.

DRAWING THE PIE CHART SEGMENTS

Listing 4.9 implements a function called `drawPieSegment`. It fills the specified segment of the chart using a random color. This function will be used from a function that performs the drawing in two phases later in the application. The processing function will call it for every segment and it will get all the information it needs as arguments.

Listing 4.8 Drawing a segment of the pie chart (F#)

```
let drawPieSegment(gr:Graphics, lbl, startAngle, angle) =          #A
    let br = randomBrush()
    gr.FillPie(br, 170, 70, 260, 260, startAngle, angle)
    br.Dispose()                                                    #B
```

#A Fill the segment using random color
#B Free resources used by the brush

The function parameters are written as one big tuple containing 4 elements, because this helps to make the code more readable. The first argument of the function is written with a type annotation specifying that its type is `Graphics`. This is a `System.Drawing` class which contains functionality for drawing. We use its `FillPie` method within the function, but that's all that the compiler can tell about the `gr` value. It can't infer the type from just that information, which is why we need the type annotation. The next three tuple elements specify the title text (which isn't used anywhere in the code, but will be important for drawing labels), the starting angle of the segment and the total angle occupied by the segment (in degrees). Note that we also dispose the brush once the drawing is finished. F# has a nicer way to do this and we'll talk about it in chapter 9.

Choosing which syntax to use when writing functions

We've seen two ways for writing functions with multiple arguments so far. We can write the function arguments either as a comma separated list in parentheses or as a list of values separated just by spaces. Note that the first style isn't really special in any way:

```
let add(a, b) = a + b
```

This is actually just a function that takes a tuple as an argument. The expression `(a, b)` is the usual pattern, which we used for deconstructing tuples in chapter 3. The question is which of these two options is better. Unfortunately there isn't an authoritative answer and this is a personal choice. The only important thing is to use the choice consistently.

In this book, we'll usually write function arguments using tuples, especially when writing some more complicated utility functions that work with .NET libraries. This will keep the code consistent with the syntax you use when calling .NET methods. On the other hand, we'll use spaces when writing simple utility functions that deal primarily with F# values.

We'll also write parentheses when calling or declaring a function that takes a single argument, so for example we'll write `sin(x)` even though parentheses are optional and we could write `sin x`. This decision follows the way functions are usually written in mathematics and also when calling .NET methods in C#. We'll get back to this topic briefly in chapters 5 and 6, when we discuss functions in more detail and also look at implementing and using higher order functions.

The `drawPieSegment` function from the previous listing is one of the two drawing functions that we'll use as an argument to the function `drawStep`, which iterates over all the segments of the pie chart and draws them. Before looking at the code for `drawStep`, let's briefly look at its type. Even though we don't need to write the types in the code, it is useful to see what the types of values used in the code are.

DRAWING USING FUNCTIONS

The first argument to this function is one of the two drawing functions, so we'll use a name `DrawingFunc` for the type of drawing functions for now and define what it exactly is later. Before discussing the remaining arguments, let's look at the signature of the function:

```
drawStep : (DrawingFunc * Graphics *
            float * (string * int) list  * int) -> unit
```

We're again using the tuple syntax to specify the arguments, so the function takes a single big tuple. The second argument is the `Graphics` object for drawing which will be passed to the drawing function. The next two arguments specify the data set used for the drawing - a `float` value is the sum of all the numeric values, so we can calculate the angle for each segment and a value of type `(string * int) list` is our familiar data set from the console version of the application. It stores the labels and values for each item to be plotted.

Let's now look at the `DrawingFunc` type. It should be same as the signature of the `drawPieSegment` function from the previous listing. The second drawing function (`drawLabel`), which we'll see shortly has exactly the same signature. We can look at the signatures and declare the `DrawingFunc` type to be exactly the same type as the types of these two functions:

```
drawPieSegment : (Graphics * string * int * int) -> unit
drawLabel      : (Graphics * string * int * int) -> unit

type DrawingFunc = (Graphics * string * int * int) -> unit
```

As I mentioned earlier, we don't need to write these types in the code, but it will help us understand what exactly the code does. The most important thing that we already know is that the `drawStep` function takes a drawing function as a first argument and we know what arguments should be given to this function, because this is specified by its type (`DrawingFunc`). The listing 4.9 shows the code of the `drawStep` function.

Listing 4.9 Drawing items using specified drawing function (F#)

```
let drawStep(drawingFunc, gr:Graphics, sum, data) =
   let rec drawStepUtil(data, angleSoFar) =                          #1
      match data with
```

```
        | [] -> ()                                                    #2
        | [lbl, num] ->                                               #3
           let angle = 360 - angleSoFar                               #A
           drawingFunc(gr, lbl, angleSoFar, angle)                    #B
        | (lbl, num)::tail ->                                         #4
           let angle = int((float(num)) / sum * 360.0)
           drawingFunc(gr, lbl, angleSoFar, angle)                    #C
           drawStepUtil(tail, angleSoFar + angle)                     #D
     drawStepUtil(data, 0)                                            #5
```

#1 Nested recursive function that processes the data
#2 Matches an empty list
#3 Matches a list with a single element
#A Calculate the angle to add up to 360
#B Draw the segment
#4 Matches a list with non-empty tail
#C Draw the segment
#D Recursively draw the rest
#5 Run the local utility function

If #B+#C could be a single connected arrow it would be perfect (annotation is the same for these two)

To make the code more readable, we implement the function that does the actual work as a nested function (#1). It iterates over all items that should be drawn on the chart. The items are stored in a standard F# list, so the code is quite like the familiar list processing pattern. There is however one notable difference, because the list is matched against three patterns instead of the usual two cases matching an empty list and a cons cell.

The first branch in the pattern matching (#2) matches an empty list and doesn't do anything. As we've already seen, "doing nothing" is in F# expressed as a `unit` value, so the code just returns a `unit` value, written as `()`. This is because F# treats every construct as an expression and expressions always have to return a value. If the branch for the empty list were empty, it wouldn't be a valid expression.

The second branch (#3) is what makes the list processing code unusual. As you can see, the pattern used in this branch is `[lbl, num]`. This is a nested pattern composed from a pattern that matches a list containing a single item `[it]` and a pattern that matches the item with a tuple containing two elements: `(lbl, num)`. The syntax we're using is a shorthand for `[(lbl, num)]`, but it means exactly the same thing. The first pattern is written using the usual syntax for creating lists, so if you wanted to write a pattern to match lists with three items, you could write `[a; b; c]`. We included this special case, because we want to correct the rounding error: if we're processing the last item in the list, we want to make sure that the total angle will be exactly 360 degrees. In this branch we simply calculate the angle and call the `drawingFunc` function which was passed to us as an argument.

The last branch (#4) processes a list which didn't match any of the previous two patterns. The order of the patterns is important in this case, because any list matching the second pattern (#3) would also match the last one (#4) but with an empty list as the tail. The order of the patterns in the code guarantees that the last branch won't be called for the last item.

The code for the last branch calculates the angle and draws the segment using the specified drawing function. This is the only branch that doesn't stop the recursive processing of the list, because it is used until there is a last element in the list, so the last line of the code is a recursive call. The only arguments that change during the recursion is the list of remaining elements to draw and the `angleSoFar`, which is an angle occupied by all the already processed segments. Thanks to the use of local function, we don't need to pass along the other arguments that do not change. The only thing that is done in the `drawStep` function itself is that it invokes the utility function with all the data and the argument `angleSoFar` set to zero.

DRAWING THE WHOLE CHART

Before looking at the second drawing function, let's look at how to put things together. Figure 4.2 shows each of the steps separately: the code that we've already written draws the left part of the figure; we still need to implement the function to draw the labels as shown on the right part.

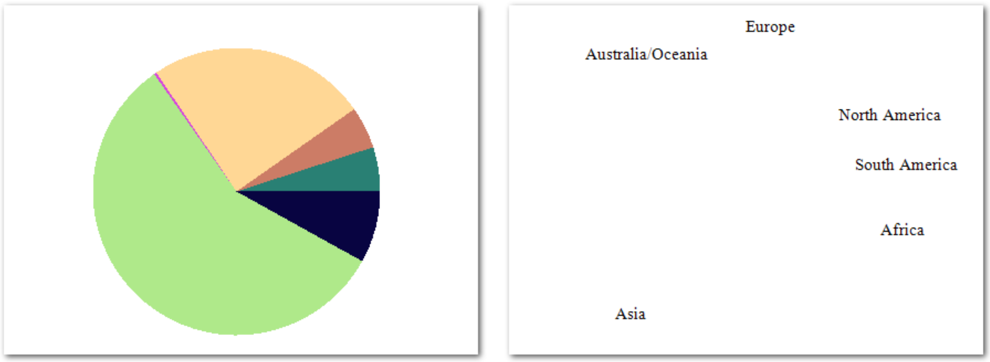


Figure 4.2 Two phases of drawing the chart - first pass using 'drawPieSegment' (left) and second pass using 'drawLabel' function (right). The chart shows distribution of world population in 1900.

The code that draws the chart first loads data from a file and processes it is the same as in the console application. Instead of printing data to the console, we now use the functions described above to draw the chart. You can see the function `drawChart` that does the drawing in listing 4.10.

Listing 4.10 Drawing the chart (F#)

```
let drawChart(file) =
   let lines = List.of_array(File.ReadAllLines(file))    #A
   let data = processLines(lines)                        #A
   let sum = float(countSum(data))                       #A

   let bmp = new Bitmap(600, 400)                        #B
   let gr = Graphics.FromImage(bmp)                      #C
   gr.FillRectangle(Brushes.White, 0, 0, 600, 400)
   drawStep(drawPieSegment, gr, sum, data)               #1
   drawStep(drawLabel, gr, sum, data)                    #2
   gr.Dispose()                                          #D
   bmp
```

#A Load and process the data
#B Create an in-memory bitmap
#C Create object for drawing on the bitmap
#1 Draw the pie chart
#2 Draw the text labels
#D Finalize the drawing

#A refers to all three lines, could we do it as a vertical line?

The function takes a name of the CSV file as an argument and returns an in-memory bitmap with the pie chart. In the code, we first load the file and process it using our existing `processLines` and `countSum functions`. We then draw the chart and on the last line we return the created bitmap as a result of the function.

In order to draw anything at all, we first need to create a `Bitmap` object and then an associated `Graphics` object. We've used `Graphics` for drawing in all the previous functions, so once it is created we can fill the bitmap with a white background and draw the chart using the `drawStep` function. The first call (#1) draws the pie using `drawPieSegment` and the second call (#2) draws the text labels using `drawLabel`. You can try commenting out one of these two lines to draw only one of the steps and get the same results as we've seen in figure 4.2. We haven't implemented the `drawLabel` function yet, because I wanted to show how the whole drawing works first, but now we're ready to finish this part of the application.

ADDING TEXT LABELS

We've already implemented the first drawing function and the second one should have the same signature, so that we can use each of them as an argument to the universal `drawStep` function. The only thing that we have to fill in is the code for drawing the label and calculating its position as you can see in listing 4.11.

Listing 4.11 Drawing text labels (F#)

```
let fnt = new Font("Times New Roman", 11.0f)                          #A

let centerX, centerY = 300.0, 200.0                                   #B
let textDistance = 150.0                                              #C
let drawLabel(gr:Graphics, lbl, startAngle, angle) =
   let lblAngle = float(startAngle + angle/2)                      #1
   let ra = Math.PI * 2.0 * lblAngle / 360.0                       #2
   let x = centerX + labelDistance * cos(ra)
   let y = centerY + labelDistance * sin(ra)
   let size = gr.MeasureString(lbl, fnt)                           #D
   let rc = new PointF(float32(x) - size.Width / 2.0f,             #D
                       float32(y) - size.Height / 2.0f)            #D
   gr.DrawString(lbl, fnt, Brushes.Black, new RectangleF(rc, size))    #D
```

#A Create font for drawing the text
#B Center of the pie chart
#C Distance of labels from the center
#1 Compute angle for the label
#2 Convert angle to radians
#D Get the bounding box and draw the label

We first declare a top-level font value used for drawing the text. We do this, because we don't want to initialize a new instance of the font every time the function is called. Since the font will be needed during the whole lifetime of the application, we don’t dispose it explicitly and we rely on .NET to dispose it when the application quits. The function itself starts with several lines of code that calculate location of the label.

Briefly, the first line (#1) calculates the angle in degrees that specifies center of the pie chart sector occupied by the segment. We take the starting angle of the segment and add half of the segment size to move the label to the center. The second line (#2) converts the angle to radians. Once we have the angle in radians, we can compute the X and Y coordinates of the label using trigonometric functions `cos` and `sin`. Finally, we use `MeasureString` method to estimate the size of the text label and calculate location of the bounding box in which the text is drawn. The X and Y coordinates calculated earlier are used as a center of the bounding box.

Now that we've finished the code for drawing text labels, we're done with the whole code for drawing the pie chart. We implemented the key function (`drawChart`), which performs the drawing of the chart earlier in listing 4.10. The function takes a file name of the CSV file as an argument and returns a bitmap with the chart. All we have to do now is add code which will call this function from our user interface.

4.4.3 Creating Windows Application

We started creating the GUI of the application earlier, so we already have code to create user interface controls. However we still have to specify user interaction logic for our controls. The user can control the application using two buttons. The first one (`btnOpen`) loads a CSV file and the second one (`btnSave`) saves the chart into an image file. We also have a `PictureBox` control called `img` which is where we'll show the chart. Listing 4.12 shows how to connect the drawing code with our user interface.

Listing 4.12 Adding user interaction (F#)

```
let openAndDrawChart(e) =                                #1
   let dlg = new OpenFileDialog(Filter="CSV Files|*.csv")
```

```
      if (dlg.ShowDialog() = DialogResult.OK) then
         let bmp = drawChart(dlg.FileName)                    #2
         img.Image <- bmp                                     #A
         btnSave.Enabled <- true                              #B

   let saveDrawing(e) =                                       #3
       let dlg = new SaveFileDialog(Filter="PNG Files|*.png")
       if (dlg.ShowDialog() = DialogResult.OK) then
           img.Image.Save(dlg.FileName)                       #C

   [<STAThread>]
   do                                                         #4
      btnOpen.Click.Add(openAndDrawChart)                     #D
      btnSave.Click.Add(saveDrawing)                          #D
      Application.Run(main)
```

#2 Draw the chart
#A Display the bitmap
#B Enable button for saving image
#C Save the current chart
#D Register event handlers

The code first declares two functions that will be invoked when the user clicks on the "open" and "save" buttons respectively. For opening a file, we have a function `openAndDrawChart` (#1). The function first creates an `OpenFileDialog`, which is a Windows Forms class that shows standard dialog for selecting a file. If the user selects a file, the function calls `drawChart` (#2), which we implemented earlier. A result of this call is an in-memory bitmap, which can be assigned to the `Image` property of the `PictureBox` control. The second function is simpler, because it doesn't need to draw the chart. It saves the image currently displayed in the `PictureBox` to a file, which is specified by the user using `SaveFileDialog`.

We've already talked about the code to execute a standard windows application, but listing 4.12 shows it again (#4), because we've added two lines of code. Before running the application, we specify that the `openAndDrawChart` function should be called when the user clicks on the `btnOpen` button and likewise for the second button. This is done by registering a function as a handler of the `Click` event using the `Add method`. Unlike in C#, where events are special language constructs, F# treats events as normal objects that have `Add` method. Events in F# also have `AddHandler` and `RemoveHandler` methods that serve exactly the same purpose as `+=` and `-=` operators for events in C#. We'll talk about this topic in more detail in chapter 16, but in most of the cases you can just use the `Add` method.

4.5 Summary

In this chapter we developed a simple but real-world application for drawing pie charts. We've seen basic F# and .NET numeric data types and explored both F# and .NET functionality for working with strings. We also demonstrated how to use usual .NET libraries from F# and we've seen examples using Windows Forms, `System.Drawing` as well as basic I/O.

What I really wanted to demonstrate in this chapter was a typical F# development process. In the beginning we started writing functions for working with the data and we immediately tested them in F# interactive. As we progressed, we implemented a function to load the real data from file and a simple console application to verify that the core functions work correctly. Finally, we added a graphical user interface and drew the chart using the functionality that we had already implemented and tested.

We were able to implement the application in this way so early in the book mainly because it doesn't work with data extensively. The only data structures that we've used are tuples and lists that were both introduced in chapter 2. However, most real-world applications need to work with more complex data sets. This is a topic for part 2, where we'll see how to represent more complicated and structured data in a functional way and how to process it elegantly.

Of course, the application is still quite simple and extending it (for example by adding different types of charts) would currently be difficult. To make the application more extensible, we need to perform one more iteration in our development approach. This requires many of the advanced functional techniques discussed in the rest of the book. After getting familiar with them, you can take a look at the book’s web site, which contains more evolved version of the application.

5 Using functional values locally

This chapter is about *values*. It's a term which is used a lot in different programming languages, so we ought to first define what we mean. When discussing the concepts of functional programming, you've seen that functional programs are described as a computation that takes some inputs and returns a result. In simple terms, a value is what you can use as an input or get as a result. This means that everything you'll work with inside the computations you implement is a value.

When writing a function that performs some calculation, we can give it all the input values as input parameters, but what if the function needs to return multiple values as a result? In C#, we could either use "out" parameters or define a new class to group the values into a single object. This feels a bit inconsistent, because handling of input and output in this scenario is quite different. What we need is a simple way for combining multiple different values (for example item name of type string and a count of type integer) into a single value that could be used both as an input argument and as a result. In chapter 3, we've briefly talked about tuples that can be used exactly for this purpose, so we'll look at tuples in some more detail.

Another example is when a computation can take one of various options as an input. A search function could for example take a name or an ID of the item. In C#, we would probably write a function that takes two parameters and set one of the argument to some invalid value (-1 as an ID or `null` instead of a name). However, there is a more elegant solution to this problem as well. We'll see how to combine values into an alternative value that can carry one of several options, but not both.

Finally, in functional languages, functions are treated as values meaning that function is another (very important) kind of value. As you can tell, values are pretty fundamental to understanding functional programming, which is why I'm starting part 2 of the book with them.

5.1 What are values?

Before we start by looking at various ways to create values and how to use them for controlling the program flow, let's try to clarify what a value is. Unfortunately, there is no simple definition, so the best way to understand that is to read this chapter. However, this section should make it a bit easier by drawing a distinction between values and data and also by explaining how values in functional languages relate to primitive types, value types and objects in languages like C#.

15.1.1 Primitive types, value types and objects

In C#, we can work with primitive value types such as integers or characters, a custom value types declared using the `struct` keyword (such as `DateTime`) and classes. The difference between value types and reference types is primarily in their behavior, but that's observable only when the class is mutable. For example string is a reference type that appears like a value type, because it is immutable. This means that by using only immutable types, we can almost eliminate the difference between value types and reference types. There are only differences in the performance, but the behavior will be the same.

However, there is another measure that we can use to look at the differences between various types and that's their complexity. In C#, this distinction isn't that obvious, because even primitive types are standard value types that have methods and can implement interfaces. However, immutable value types are still considerably simple than objects that have virtual methods and mutable state.

In functional languages, we start with a set of primitive types and we can then build more complicated types simply by composing the primitive types in various ways. This is different to object-oriented languages where we create types by defining their state in terms of primitive types and specifying their behavior using methods.

The functional approach makes the whole type system a lot easier, because there is in principle no distinction between value types and reference types. It also makes the transition from simple values to complex composed types very smooth. In this range, values are all the primitive types and also most of the simple composed types. To understand when a composed type becomes too complex to be considered as a value, we need to look at what we'll call data.

15.1.2 Recognizing values and data

Values are usually used locally and you need to create and use them all the time. I've already mentioned tuple as one of the composed values that is used very frequently. Another example is `option` type that we'll discuss in section 5.3.3. It consists of two alternatives - one is some actual value and the other specifies that the value is missing. When working with `option` values, we have to explicitly check for both of the cases, so there is no danger of getting `NullReferenceException`.

This should give you an idea that values are usually quite general purpose and are used for solving general programming tasks such as expressing that some argument is missing.

They can also be very simple (and locally used) utilities such as a value that would contain either ID or a name given as an argument to searching function. On the other hand, data is usually something large and represents information that is shared between several parts of the program. The programming language doesn't differentiate between the two, but we will occasionally our description.

In this chapter, we're going to look at ways of working with values locally, which will include some basic F# type declarations. We'll come back to this discussion in chapter 7 when we introduce the remaining type declarations that are typically used to represent data for the whole application.

VALUES AND THEIR TYPES

I've been using the terms *value* and *type* quite vaguely until now, so let me specify what I mean. To take a numeric example, "integer" is a type, whereas 43 is a value of that type. A type specifies an entire domain of values and value is always an element within the domain specified by its type.

That's enough theory for now: let's look at our first way of composing values together. It should be familiar by now–it's time to revisit tuples.

5.2 Multiple values

You already know that the only thing a function can do is return a single value as its result. Once you start writing practical code, you're likely to face a common problem almost immediately: the need to return multiple values. This is the primary motivation for tuples, although as we've seen they can also be used to combine several values into a single argument for a function as well.

5.2.1 Multiple values in F# and C#

When we were discussing tuples earlier, we implemented a `Tuple` class in C# with the same behavior as F# tuples. This isn't the normal way to return multiple values from a C# method, although you may still find it very useful when writing C# code in a functional way. If you wanted to write this in C# without using tuples or declaring a new class for every method that returns multiple values, you would probably use "out" parameters. You can see both approaches side by side in listing 5.1, where we implement a simple function performing division with a remainder.

Listing 5.1 Division with a remainder (F# and C#)

```
> let divRem(a, b) =
     (a / b, a % b);;
val divRem : int * int ->
             int * int
```

```
int DivRem(int a, int b,
           out int rem) {
   rem = a % b;
   return a / b;
}
```

```
> let (res, rem) = divRem(10, 3);;       int rem;
val res : int = 3                        int res = DivRem(10, 3, out rem);
val rem : int = 1
```

The F# version of the code shows the F# interactive output, but if you ignore it you can see that the code is shorter. This is because returning multiple values from a function is much more important in F# than in C#. However, C# 3.0 adds one more way for representing multiple values called *anonymous types*. It is somehow limited, because it can be used only locally inside a single method, but it is still interesting.

Anonymous types in C# 3.0

The key feature added by LINQ is the ability to write queries. We'll talk about them later in chapter 11. Queries work with collections, so for example we might filter a collection of products and select only products from a particular category, then just return the name and price of each product. This is where anonymous types are needed, because when returning the name and price, we effectively need to return multiple values:

```
var query = from p in data.Products
            where p.CategoryID == 1
            select new { Name = p.ProductName, Price = p.UnitPrice };
foreach(var result in query)
   Console.WriteLine(result.Name);
```

The difference between anonymous types and tuples is that elements of an anonymous type are named. The names are specified by the code creating the anonymous type in the query (#1) and can be used later to read the values of elements (#2). We could of course rewrite the example from previous listing using anonymous types:

```
int a = 10, b = 3;
var r = new { Remainder = a % b, Result = a / b };
Console.WriteLine("result={0}, remainder={1}", r.Result, r.Remainder);
```

However, this isn't particularly useful, because anonymous types can be used only locally. When we return them from the method, we lose the compile-time type information and we can't easily access the properties any more.

We've seen that in C# "out" parameters are often used for the same purpose as tuples in F#. You may be wondering how to use existing .NET methods with "out" parameters from F#… but fortunately the language has a nifty feature for exactly this purpose.

USING TUPLES INSTEAD OF OUT PARAMETERS

Even though you can use "out" parameters from F# if you really want to, tuples are generally preferred and so F# automatically exposes .NET methods with "out" parameters as methods that return a tuple. You don't have to do anything - it's just transparent. This means that your F# code can still look like idiomatic functional code even if it's calling into .NET code which has no concept of tuples. The most widely used method with an "out" parameter in .NET is probably `TryParse`, which is available in all of the numeric types such as `Int32`. Let's look at examples of using it from C# and F#:

```
// C# version using an "out" parameter
int num;
bool succ = Int32.TryParse("41", out num);

// F# version using tuples
let (succ, num) = Int32.TryParse("41");;
```

The F# version is quite easy to write and if you want to use the default value when the parsing fails, you can simply use the `snd` function to extract the second element from the tuple, or use an underscore instead of the `succ` value in the pattern. When talking about functions for working with values in the next chapter, we'll also see a very simple function that allows us to specify the default value to use if the parsing within `TryParse` fails.

Now, before looking at the best practices for using tuples in F#, let me just quickly get back to the discussion about values and types and revisit how tuple types and values of these types are constructed.

5.2.2 Tuple type and value constructors

You already know what the type of a tuple value looks like and we've seen it again in the previous code listing. The type is written using asterisk, so for example a type of a tuple storing an integer and a string is written as `int * string`. In the introduction, we talked about values and their types and I wrote that a type is a domain of all possible values. Let's use this point of view to look at the tuple type: how does this notation reflect the fact that tuple type is composed from several primitive types?

The asterisk symbol plays a key role in this notation, because it serves as a *type constructor*. This means that you can use the asterisk symbol to construct tuple types from any other types. This means that the domain specifying values of the type `int * string` contains all possible combinations of integers and strings. You don't have to explicitly write types very often thanks to the wonders of type inference, but it's useful to see how types are constructed.

On the other hand, you'll work with *value constructors* when writing any code that uses a tuple. This is the syntax that allows you to create values of tuple types from other, simpler values. For example `(1, "hello")` demonstrates the use of a value constructor. It creates one particular value that belongs to the domain specifying all possible combinations of integers and strings. To demonstrate the correspondence between value and type constructors, let's look at one more example. The following code snippet shows how we could use tuples to represent a message and X, Y coordinates saying where on the screen it should be displayed:

```
> let msgAt1 = (50, 100, "Hello world!")
val msgAt1 : int * int * string

> let msgAt2 = ((50, 100), "Hello world!")
val msgAt2 : (int * int) * string
```

The code shows two different representations. In the first case we're using a single tuple with three elements to store all the basic values together. As you can see, the printed type signature reflects this and shows three basic types separated by asterisks. In the second case, we first construct a tuple to store the X and Y coordinates and then we compose another tuple from this value and the message. As you can see, the type again reflects this construction. You can also see that the types are different. The first one is a tuple of three elements, while the second one is a tuple containing tuple and a string. This means that you should always consider the available options when you construct tuples. In this case, I prefer the second option, because it seems logical that the X and Y values form a single coordinate value. Let's look at a few more guidelines of how to use tuples appropriately.

5.2.3 Using tuples compositionally

The key concern when thinking about what kind of a tuple should be returned from a function is *compositionality*. How do you expect the tuple to be used? What other functions might use a tuple of the same type? Is this consistent with similar situations in the rest of the program?

Let me demonstrate this using an example. We'll use the two ways of representing message and screen coordinates from the previous example and we'll assume that we already have a function for printing the message. Our `printMessage` function has the following signature:

```
val printMessage : int * int -> string -> unit
```

The signature tells us that the function takes two arguments. The first argument is a tuple containing the coordinates and the second argument is the message. Now, we want to print the string "Test!" to a location specified by tuple that we used earlier. Listing 5.2 shows two different ways of doing this, depending on which representation we use for the message and coordinates.

Listing 5.2 Different representations of a message with coordinates (F#)

```
> let msgAt1 = (50, 100, "Hello!");;
val msgAt1 : int * int * string                  #1

> let (x, y, _) = msgAt1                         #2
  printMessage (x, y) "Test!";;                  #2

> let msgAt2 = ((50, 100), "Hello!");;           #3
val msgAt2 : (int * int) * string

> let (pt, _) = msgAt2                           #4
  printMessage pt "Test!";;                      #4

> printMessage (fst(msgAt2)) "Message Test!";;   #5
```

#1 Tuple of three elements
#2 We have to extract all elements
#3 Using a nested tuple
#4 We need to extract only the first element
#5 The simplest way using the second representation

As you can see, the tuple that we created in the first case (#1) isn't compatible with the `printMessage` function, so when we want to compose the code, we first have to deconstruct the tuple into elements (#3) and then build a new tuple value when calling the function. Using the second representation, we can do much better. The first element of the tuple is itself a tuple (#2) and is compatible with the first parameter of `printMessage`.

This is very helpful, because when we're deconstructing the tuple later (#4), we can just take the first element and use it directly as the first argument. Actually, as the last line demonstrates (#5), we can do even better and use the `fst` function to get the first element of the tuple directly when calling the function. I think this clearly shows why it is important to structure tuples logically. However, you also need to consider the complexity of the tuples you create…

AVOIDING COMPLICATED TUPLES

Clearly, returning the results as tuples with extremely large number of elements makes the code hard to read. In F#, you can replace tuples with too many elements with record types, which provide a simple way to create a type with labeled members. Records are usually used for storing program data, so we'll talk about them in chapter 7.

The point at which a function becomes hard to use based on the number of elements in its return type will vary from person to person, but I recommend avoiding returning tuples with more than 3 or 4 elements. Of course there are exceptions, and in the early phases of development it may be worth prototyping with large tuples, refactoring later when you have a clearer idea of how the values should be structured. Also, if the tuple is only used internally, using a larger tuple may be a better option than declaring a record type for a single use.

Now that you know everything you need to about tuples, let's move to another topic. In the next section, we're going to talk about a way of constructing values that can be used for representing types with several alternative values.

5.3 Alternative values

In the previous section, we looked at how to create values that combine several values into one. For example, we took a string value and a numeric value and created a composed value that contains both string and a number. In this section, you'll see how to construct a value that can contain *either* a string *or* a number.

First, let's look at an example of when this could be useful. Imagine that you're writing an application to schedule tasks and meetings, and you want to have several ways for specifying the schedule. For an event that happens only once, we'd like to store the date and time. However, we also want to allow events that occur repeatedly. For this kind of event, we'll need to store the date and time of the first occurrence and the time span between repetitions of the event. Finally, we'll also support events that don't have specified time yet, which we'll call unscheduled events.

This means we want to create a value with three different options to specify the schedule - once, repeatedly, or never. A typical way to represent several options in object-oriented programming is to use a hierarchy of classes. In our case, we'd have an abstract class Schedule with an inherited class for every of the three options. You can see a diagram showing the object-oriented solution in figure 5.1.

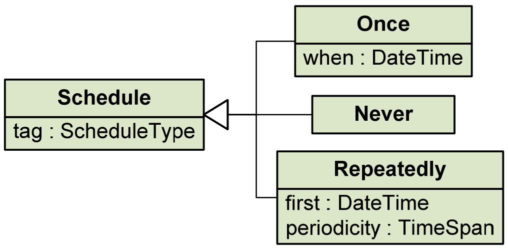


Figure 5.1 Class hierarchy for representing three different types of schedule with different properties for every case.

In this example, we don't expect to add new types of schedules later during the development. On the other hand, we'll probably add new modules to the application that will need to work with the schedule and will need to access the properties of the inherited types. To make this easier, the base class has a property called tag, which specifies what type of schedule the object represents. In this case, ScheduleType is a C# enumeration with three possible values (Once, Never and Repeatedly). This makes the code less extensible in one way (by adding new types of schedules), but it allows us to easily add methods that work with schedules. I'll talk about extensibility shortly, but first, we'll look at representing alternatives in F# using discriminated unions.

5.3.1 Discriminated unions in F#

Types like this crop up quite frequently in functional programming, so functional languages tend to make it easy to create and use them. In F# the supporting feature is called *discriminated unions*. Unlike tuples, discriminated unions have to be declared in advance, so before we can create a value representing the schedule, we first have to declare the type with its name and, most importantly, the options it can represent. The code in listing 5.3 shows a type for representing schedules in F#.

Listing 5.3 Schedule type using discriminated union (F#)

```
type Schedule =
    | Never                               #A
    | Once of DateTime                    #B
    | Repeatedly of DateTime * TimeSpan   #C
```

#A Unscheduled event with no arguments
#B Event with single occurrence
#C Repeated event with first occurrence and periodicity

When creating the Schedule type, we combine several alternatives. We need to be able to distinguish between the alternatives, so we also specify a name for each of them

(`Never`, `Once` and `Repeatedly`). These names are usually called *discriminators*, because they discriminate between the options. This means that every value of the `Schedule` type will carry its discriminator and the values stored for the selected option (such as `DateTime` and `TimeSpan` in the last case of our example). As you can see, we're using asterisk symbol when storing multiple values for a single option. This is exactly analogous to the syntax for creating tuples, so you can see how the two concepts (multiple and alternative values) play nicely together.

We'll also need discriminators when creating values of the `Schedule` type, because the discriminator specifies which option we are using. Listing 5.4 shows several examples.

Listing 5.4 Creating values of discriminated union (F# interactive)

```
> open System;;                                          #A

> let tomorrow = DateTime.Now.AddDays(1.0);;             #B
val tomorrow : DateTime                                  #B
> let noon = new DateTime(2008, 8, 1, 12, 0, 0);;        #B
val noon : DateTime                                      #B
> let daySpan = new TimeSpan(24, 0, 0);;                 #B
val daySpan : TimeSpan                                   #B

> let schedule1 = Never;;                                #C
val schedule1 : Schedule = Never
> let schedule2 = Once(tomorrow);;                       #D
val schedule2 : Schedule = Once(2.8.2008 17:29:07)
> let schedule3 = Repeatedly(noon, daySpan);;            #E
val schedule3 : Schedule = Repeatedly(1.8.2008 12:00:00, 1.00:00:00)
```

#A Open namespace with DateTime and TimeSpan
#B Create values representing times and periodicity
#C Create schedule using discriminator 'Never'
#D Event occurring once at specified time
#E Event occurring repeatedly every day

As you can see, creating values of the `Schedule` type is quite easy. We're using discriminators as value constructors. This is similar to our previous use of value constructors for creating tuples such as `(7, "seven")`. In this case, the syntax looks almost like calling a function. For an option with no additional arguments, we just write the discriminator name and for option with more arguments, we write the arguments as if they were a single tuple.

Of course, creating a value is pointless unless we can actually *use* it. Let's try calculating something useful with a schedule…

5.3.2 Working with alternatives

So far we've seen how to declare a discriminated union type and how to create values using discriminators. Now we'll look how to write code that reads the value. When working with discriminated unions, we always have to write code for all possible alternatives, because we don't know which one is represented by the value.

You can remember a similar situation from earlier on – we had to test whether a list was an empty list or a cons cell. You may also remember that we've used pattern matching to do this: the `match` construct allows us to test the value against several patterns. We can use the same feature to work with discriminated unions, except this time we use discriminators to write the patterns. Listing 5.5 shows an example that tests whether a scheduled event occurs during a week following the current date.

Listing 5.5 Does the event occur next week? (F#)

```
let occursNextWeek(schedule) =
   let isNextWeekDate dt =                                       #1
      dt > DateTime.Now && dt < DateTime.Now.AddDays(7.0)

   match schedule with
   | Never -> false
   | Once(dt) -> isNextWeekDate(dt)
   | Repeatedly(dt, ts) ->
      let q = (DateTime.Now - dt).TotalSeconds / ts.TotalSeconds  #A
      let q = max q 0.0                                           #B
      let next = dt.AddSeconds(ts.TotalSeconds *
                               (Math.Floor(q) + 1.0))             #C
      isNextWeekDate(next)                                        #D
```

#1 Nested utility function
#A How many times will it occur before today?
#B Only consider future occurrences
#C Calculate first occurrence after today
#D Test whether it happens next week

This example is quite complicated but it shows the typical structure of an F# program. We're using the standard .NET `DateTime` and `TimeSpan` structures to work with dates and times. First, we declare a utility function which tests whether a `DateTime` occurs during the next week from the present time. Next, we use pattern matching to test which of the alternative schedule representations has been given to us. For the first two cases, the calculation is quite simple, but for the last one (a repeated event), we calculate the first occurrence of the event after the present date and then test whether this occurs during the next week using the utility function written earlier. You can see that we declare the value `q` twice in the code. This is called hiding a value and it is useful if we want to split a complicated calculation into two or more steps and make sure that we won't accidentally use the intermediate values.

As you can see, the pattern used for testing whether a value matches a specific discriminator is exactly the same as we've used to construct the value in the first place. The pattern also extracts the values stored as arguments and assigns them to new values (called `dt` and `dt` with `ts` respectively), so we can immediately use them. Again, this is the same syntax we used when matching patterns with lists.

Next, we'll look at exactly the same functionality implemented in C#. We've already seen the classes involved in figure 5.1, so we'll assume they've already been implemented and just look at the code that uses them. We'll look at another example of alternative values later, including the complete C# implementation, so you'll see how we can write a C# class hierarchy with the same properties as an F# discriminated union later in the chapter.

TIP

If you want to see the complete source code for this example including class declarations, you can download it from the book's web site: www.functional-programming.net.

There is one important thing to note before we look at the C# version of the example in listing 5.6. It shows a situation when we already have the class hierarchy representing schedules implemented (for example in a shared library) and we're adding new functionality to a module in our application. This means that we can't easily add a virtual method to the base class `Schedule`. Also, we want to keep the functionality localized in a single place in the code, to keep everything related to the calculation in a same place and the same file.

Listing 5.6 Does the event occur next week? (C#)

```
bool IsNextWeekDate(DateTime dt) {
   return dt > DateTime.Now && dt < DateTime.Now.AddDays(7.0);
}
bool OccursNextWeek(Schedule schedule) {
   switch(schedule.Tag) {                                     #1
   case ScheduleType.Never:                                   #A
      return false;
   case ScheduleType.Once:                                    #B
      return IsNextWeekDate(((Once)schedule).When);           #C
   case ScheduleType.Repeatedly:                              #D
      var rp = (Repeatedly)schedule;                          #E
      double q1 = (DateTime.Now - rp.First).TotalSeconds
                  / rp.Periodicity.TotalSeconds;
      double q2 = Math.Max(q1, 0.0);
      DateTime next = rp.First.AddSeconds
          (rp.Periodicity.TotalSeconds * (Math.Floor(q2) + 1.0));
      return IsNextWeekDate(next);
   }
   throw new InvalidOperationException();                     #F
}
```

#1 Switch using the schedule type
#A Schedule is 'Never'
#B Schedule is 'Once'
#C Accessing property of the 'Once' class
#D Schedule is 'Repeatedly'
#E Extract properties of the 'Repeatedly' class
#F All code-path should return, but this one is unreachable

The algorithm used in the C# version is exactly the same as in the F# version, so the only difference is how we distinguish between the options and how we read values stored for the option. In F#, this was done using pattern matching. In the C# version we're using the `switch`, which is a C# analogy of the `match` construct from F#. This is possible because we have a `Tag` property in the base class and an enumeration that tells us what kind of schedule the object represents. Otherwise we would have to use `if` statement with a sequence of dynamic type tests. Also, reading of values, which was done automatically in F# is now a bit difficult. We have to cast the schedule to the concrete class to read its properties.

In fact, the C# version of the code is very close to the .NET representation used by the F# compiler for discriminated unions. This means that the two examples above are essentially the same after compilation. However, functional programming puts a stronger emphasis on this kind of data type, which is why it was much easier to write this code in F#.

ADDING TYPES VS. FUNCTIONS

As I mentioned earlier, our `Schedule` data type isn't extensible: it's difficult to add a new type of event. In F#, the difficulty occurs because you have to modify the type declaration; if it's in a shared library you have to recompile the shared library. Similarly, in the C# version, we have a `Tag` property which makes adding new types difficult. On the other hand, the benefit of this design is that it allows us to very easily add new functionality for working with schedules.

I'll explain this in more detail, but let's first look at the second way for representing a problem. This is the usual object-oriented way in which all the functionality is enclosed in virtual methods. You can see this version of class hierarchy in figure 5.2.

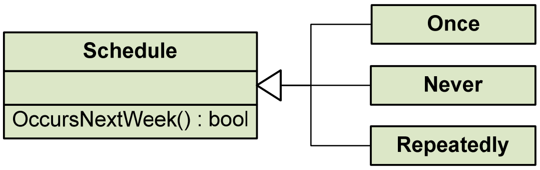


Figure 5.2 Representation of schedule using the usual object-oriented design with functionality implemented using virtual methods.

In this version, we'd implement the virtual method `OccursNextWeek` in each of the inherited classes. The following list shows the key differences between the functional programming style (FP) demonstrated earlier and the usual object-oriented style (OO) outlined in figure 5.2.

24) The FP version makes it easier to add new functionality that works with the data type. This is done by writing a function using pattern matching. On the other hand, adding a new kind of representation to the type is difficult.

25) The OO version makes it easier to add new types of representation. This is done by writing a new inherited class and implementing of its virtual methods. On the other hand, adding a new virtual method is difficult.

26) In the FP version the code for a single functionality is localized, so all code related to one kind of computation is in a single function.

27) In the OO version the code for a single type is localized, meaning that all code which works with the type is inside one class declaration.

As you can see, the key question is whether it you want to make it easier to add new types or new functions. Experience shows that in functional programming, it is more common to add new functionality to an existing type.

If you're familiar with common design patterns then you may remember the Visitor pattern, which is an object-oriented way of implementing data structures like discriminated unions. We'll talk about it when we look at recursive discriminated unions in chapter 7, because it is usually used when working with complex program data rather than simple values. I'll also delay the discussion of whether to choose a discriminated union or not until chapter 7, because this question is more relevant when talking about program data.

In this chapter, we're talking about simple values: for any simple value that is represented as limited set of alternatives, you should always choose discriminated union. This is because for simple values, you almost certainly want to add new functionality instead of adding new types. There's one discriminated union which is particularly useful in functional programming and is present in all functional languages - in F# it is called the `option` type.

5.3.3 Using the option type in F#

We often need to represent the idea that some computation may return an undefined value. In C#, this is usually done by returning `null`. Unfortunately using `null` is a frequent cause of bugs: you can easily write code that assumes that a method doesn't return `null` and when this assumption is false, you'll see the infamous `NullReferenceException`. Of course, properly written code always checks for `null` values where appropriate and when writing unit tests for the application, a large number of tests verify the behavior in this corner case.

In F# use of the `null` value is minimized and it is often used only when interoperating with .NET types. For representing computations that may return an undefined result, we instead use the `option` type. When we use this as the return type of a function, it is an explicit statement that the result may be undefined; this also lets the compiler force the caller to handle an undefined result.

The option type is a discriminated union with two alternatives. The discriminator `Some` is used for creating an option that carries a value and `None` is used for representing undefined value. Listing 5.7 shows a function which reads an input from the console and returns undefined value when the user doesn't enter a number.

Listing 5.7 Reading input as an option value (F# interactive)

```
> open System;;
> let readInput() =
     let s = Console.ReadLine()
     let (succ, num) = Int32.TryParse(s) #A
     if (succ) then
```

```
        Some(num)                         #1
     else
        None;;                            #2
val readInput : unit -> int option        #3
```

#A Try to parse the input
#1 Return a value using 'Some'
#2 Return an undefined value using 'None'
#3 Signature of the function

The code is quite simple - it first reads the input and uses the `TryParse` method to get a tuple representing whether the input was correct and the parsed number. If the parsing succeeded, we use `Some` discriminator (#1), which takes a single argument to return the value. Otherwise we return an undefined result using the `None` discriminator (#2). You can also see the signature printed by the F# interactive (#3) it says that the method returns `int option`. This means that the option type is generic and in this case it carries an integer as a value. We'll see how a generic type like this can be defined in section 5.3.2.

First, let's look at the code that uses this function. Here we'll see the real benefit of using option type, which is that we're forced by the language to write code to hande the undefined value. This is because the only way to access the value is using pattern matching. You can see the example in listing 5.8.

Listing 5.8 Processing input using option value (F# interactive)

```
> let testInput() =
     let inp = readInput()                              #1
     match inp with                                     #2
     | Some(v) ->                                       #A
        printfn "You entered: %d" v
     | None ->                                          #B
        printfn "Incorrect input!";;
val testInput : unit -> unit

> testInput();;                                         #C
42                                                      #C
You entered: 42                                         #C

> testInput();;                                         #D
fortytwo                                                #D
Incorrect input!                                        #D
```

#1 We cannot use the value directly!
#2 Check for alternatives using pattern matching
#A Branch for correct input
#B Branch for undefined input
#C Testing the first case interactively
#D Testing the second case interactively

As you can see, we cannot use the value directly after we call the `readInput` function (#1). This is the key difference that makes the program safer, because when a function returns a `null` value, you don't have to check this possibility. To read the value in F#, we have to use pattern matching (#2) and we write a branch for each of the discriminators that can be used to construct a value of the option type. We already seen that F# verifies whether pattern matching is complete; that is, whether it covers all possible options. This guarantees that we cannot accidentally write code that only contains a branch for the Some discriminator. Listing 5.8 also follows F# best practices by testing the code in F# interactive straight away, checking that it behaves correctly in both cases.

NULLABLE AND OPTION TYPES

The F# option type is in some ways similar to the Nullable<T> type in C#, but it is more universal and safer. When we want to represent a missing value in C#, we usually use the null value, but this is possible only for reference types. Nullable types can be used to create value type that also has null as a valid value.

On the other hand, in F# null isn't valid value of any type declared in F# (though it is still valid for existing .NET reference types). This means that whenever we need to create any value that may be empty we wrap the actual type into option type. Thanks to the pattern matching, the compiler can also ensure that we always implement code that handles the case when the value is missing.

Now that we've seen how to use option types and how they are important for F# programming, we'll discuss how to implement them.

IMPLEMENTING SIMPLE OPTION IN F#

In the previous example, we were working with option type carrying integers, so let's first look at somewhat simplified type IntOption which can carry only integer values. I'm sure you could write the declaration for the type on your own already, but here it is:

```
> type IntOption =                         #A
       | SomeInt of int
       | NoneInt;;
(...)
> SomeInt(10);;                            #B
val it : IntOption = SomeInt 10
```

#A Declare discriminated union with two alternatives
#B Create sample value of 'IntOption' type

There is one important difference between our declaration and the option type from the F# library. The library type is generic, which means that you can use it to store any type of value, including .NET object references such as Some(new Button()). Writing generic types is very important, because it makes the code more widely applicable. Let's take a closer look now.

5.4 Generic values

In this section we'll talk about generic type declarations and you'll see that in many ways generic types in F# are similar to generic types in C#. We've only seen one kind of type declaration so far–the discriminated union, declared with the type construct. We'll see other type declarations that can be written using the same construct later (in particular in chapters

7 and 9), but the syntax for making them generic is exactly the same as the syntax we'll see now.

Types that don't need prior declaration, such as tuples, are naturally generic because we can use the value constructor with any values. When creating tuples we can write `(12, 34)` as well as `("Pi", 3.14)`. In this section we'll see how to make your own type constructors generic as well. We'll start by looking at how we can implement a generic option type in C#.

5.3.1 Implementing the option type in C#

As we've just seen, option types are very important in functional programming and since we want to be able to code in a functional style in C# too, we need a proper C# implementation for `option` type. We've already discussed how to encode discriminated unions in object oriented languages, so the code has similar structure as the `Schedule` type we talked about earlier. In case of `Option<T>`, we could create a single class (or value type) with `HasValue` property, which would be a bit simpler. However, I want to demonstrate the idea of encoding discriminated unions in general, so we'll create a base class `Option<T>`, with `Tag` property and two inherited classes for the two possible alternatives.

TIP

We will use this type in some of the later chapters, so we'll also add several utility methods that make it easier to use in routine C# programming. This makes the code slightly longer, so you can download it from the book web site. Moreover, you can also download a .NET library with this and several other classes that are discussed in this book directly from: www.functional-programming.net/library.

To make the type generic, we'll implement it as a generic C# class `Option<T>`. An inherited class `Some<T>` represents an alternative with a value of type `T` and `None<T>` class represents an alternative with no value. You can see the source code in the following listing.

Listing 5.9 Generic option type using classes (C#)

```
enum OptionType { Some, None };                               #1

abstract class Option<T> {
   private readonly OptionType tag;
   protected Option(OptionType tag) {                         #A
      this.tag = tag;
   }
   public OptionType Tag { get { return tag; } }              #2
}
class None<T> : Option<T> {                                   #3
   public None() : base(OptionType.None) { }
}
class Some<T> : Option<T> {                                   #4
   public Some(T value) : base(OptionType.Some) {
      Value = value;
   }
   public T Value { get; private set; }                        #B
}

static class Option {                                          #5
   public static Option<T> None<T>() {                         #C
      return new None<T>();
   }
   public static Option<T> Some<T>(T value) {                  #D
      return new Some<T>(value);
   }
}
```

#1 Enumeration with possible alternative
#A Derived class will specify the alterative
#2 Alternative represented by the instance
#3 Inherited class representing empty option
#4 Inherited class representing option with value
#B Value carried by the option
#5 Utility class for creating options
#C Creates empty option
#D Creates option with a value

The generic base class contains only `Tag` property (#1), which can have one of two values specified by enumeration `OptionType` (#2). The tag is set in the constructors of the two derived classes, `None<T>` (#3) and `Some<T>` (#4). The second derived class carries a value, so it has a property called `Value` of type `T`. As usual in functional programming, this property is immutable, so it is set only once in the constructor.

Finally, the code also includes a non-generic utility class `Option`. We've already implemented similar classes in chapter 3 when implementing functional tuple and list types in C#. The purpose of this class is to simplify the construction of option values. Instead of using constructor directly (`new Some<int>(10)`), we can leverage C# type inference when calling generic methods and write just `Option.Some(10)`.

Now, how can we work with our option type in C#? The following snippet shows C# version of code from listing 5.5 which tries to read a number from the console:

```
   Option<int> ReadInput() {
      string s = Console.ReadLine();
      int num;
      if (Int32.TryParse(s, out num))
         return Option.Some(num);
      else
         return Option.None<int>();
   }
```

Thanks to the use of our option type, the method can simply return a single result, which may or may not contain a value. Before we see how we can work with the returned value, we're going to add two useful methods to the `Option<T>` class. In F#, we used pattern matching to tell the options apart; the methods in listing 5.10 allow us to write similar code in C#.

Listing 5.10 "Pattern matching" methods for Option class (C#)

```
public bool MatchNone() {
   return Tag == OptionType.None;                          #A
}
public bool MatchSome(out T value) {
   if (Tag == OptionType.Some) value = ((Some<T>)this).Value;  #1
   else value = default(T);
   return Tag == OptionType.Some;
}
```

#A Return true when the value is 'None'
#1 For 'Some' value, return the value using 'out' parameter

Both of the methods return boolean that tells us whether the instance represents the tested alternative. The second one has also one "out" parameter, which is set to the value carried by the option type when the object is an instance of the `Some` class (#1), otherwise the "out" parameter is set to a default value and false is returned. Listing 5.11 shows how we can work with `ReadInput` method using these two utility methods.

Listing 5.11 Working with option type (C#)

```
void TestInput() {
   Option<int> inp = ReadInput();
   int num;
   if (inp.MatchSome(out num))                      #A
      Console.WriteLine("You entered: {0}", num);
   else if (inp.MatchNone)                          #B
      Console.WriteLine("Incorrect input!");
}
```

#A Pattern matching for Some(num)
#B Pattern matching for None

Thanks to the `MatchSome` and `MatchNone` utilities, we don't have to explicitly cast the value to the inherited class (e.g. `Some<T>`) to access the value. However, it still lacks many useful features of pattern matching. First of all, the compiler doesn't verify that we're providing code for all of the branches. More importantly, it isn't possible to write nested patterns, which is a common pattern in F#. For example, you might want to create an option type carrying a tuple. This would be written simply as `Some(1, "One")` and the pattern used with `match` construct could read values directly from the tuple: `Some(num, str)`.

Now we've seen how to implement an option type using generics in C#, we can turn our attention back to F# and show how the built-in option type is declared in the F# library.

5.4.2 Generic option type in F#

Generic types in F# are essentially the same as generic classes in C#. They allow us to write more general and reusable types. We've seen this need in the case of the option type, because we'd like to be able to use exactly the same generic type for creating options that carry different types as a value. But of course, we want to write type-safe code, so we need to know what type is carried by the option type.

Just as in C#, we declare the type with a type parameter and then use that as the type of the value stored in the `Some` alternative:

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

The syntax for declaring generic type is similar to that used in C#–we write type parameters in angle brackets. Unlike in C#, we have to use special names for type parameters so the name of the type parameter always starts with an apostrophe.

When creating an instance of generic class in C# or a value of generic type in F#, the type parameter is "replaced" by the actual type used when creating the value. In C#, you have to specify the type explicitly when calling the constructor, but in F# the type argument is usually inferred by the compiler. Let's look at an example:

```
> Some("Hi there!");;
val it : Option<string> = "Hi there!"
```

In this example, the compiler infers that we're creating an option containing a string because we're giving it a string literal as the argument. It then deduces that the type argument is `string` and the inferred type is `Option<string>`. We'll talk about type inference in some more detail in the next section.

We've seen other syntax for writing generic types earlier. This is because F# is compatible with OCaml which uses different notation. We'll use the .NET syntax when writing generic types, but it's useful to understand both forms because you'll occasionally encounter the OCaml syntax.

OCaml syntax for writing generic types

In OCaml syntax, type parameters are written before the name of the type, so our pervious example of the generic option type could be written like this:

```
type 'T Option = (...)
```

When creating a value of this type, F# also prints its type using this notation. For example, the type of `Some(10)` would be displayed as `int option`. When declaring types with more than one type argument, the arguments are written in braces (which resembles the syntax for creating tuple values):

```
type ('T1, 'T2) OptLabeledTuple = (...)
```

It is important to note that this is just a syntactical difference and F# treats both declarations equally. If you declare a type using OCaml syntax and later use the .NET syntax when working with it (or vice versa), your code is still absolutely correct. It is just a matter of style–but it's a good idea to be consistent just for the sake of readability.

We can declare generic types with more than one type parameter in exactly the same way as in C#. The following example shows how to create a generic discriminated union with two discriminators that allows us to store two values and optionally specify labels for them:

```
> type OptLabeledTuple<'T1, 'T2> =
      | LabeledTuple of string * 'T1 * string * 'T2
      | UnlabeledTuple of 'T1 * 'T2
```

```
(...)

> LabeledTuple("Seven", 7, "Pi", 3.14);;
val it : OptLabeledTuple<int, float> =
            LabeledTuple ("Seven", 7, "Pi", 3.14)
```

You can see that when we create the value, the F# compiler correctly infers a type for both of the type arguments. Type inference is one of the cornerstones of F#, so let's look at some more examples and compare it with the inference available in C# 3.0.

5.4.3 Type inference for values

In general, type inference is a mechanism that deduces types from the code. Its purpose is to simplify code by removing the need to specify all types explicitly. In this section, we'll look at type inference for values, which lets us create values easily without writing their types. This isn't the only place where type inference occurs–especially in F#–so this is just the first part of description of the type inference. We'll talk about type inference for functions (and methods) and about automatic generalization in the next chapter.

TYPE INFERENCE IN C# 3.0

In C#, type inference for values is primarily represented by the `var` keyword, which is a new feature in C# 3.0. We've seen it already, but listing 5.12 shows a few examples so we can discuss it in more detail.

Listing 5.12 Type inference using 'var' keyword (C#)

```
var num = 10 + (2 * 16);                                       #A
var str = String.Concat(new string[] {"Hello ", "world!"});    #B
var unk = null;                                                #1
```

#A Infers type 'int'
#B Infers type 'string'
#1 Error CS0815!

The type inference mechanism simply looks at the right side of the assignment operator and works out the type of the expression. It has to do this even when you're not using `var`, to make sure that the variable you're assigning to is compatible with the value you're trying to assign. However, in the last case (#1), the C# compiler refuses to infer the type and reports an error message. While the `null` literal can be implicitly converted to any .NET reference type (or even a nullable value type) it doesn't actually have a real .NET type itself. The compiler doesn't know which type we want for the `unk` variable, so we have to specify the type explicitly. We've been using the `var` keyword with our option type earlier, so let's analyze several examples in detail:

```
var s1 = Option.Some<int>(10);
var s2 = Option.Some(10);           #A
var n1 = Option.None<int>();
var n2 = Option.None();             #B
```

#A Type inference for 'Some' method call
#B Error CS0411!

The first and third lines are unsurprising: we're calling a generic method and specifying its type arguments explicitly, so the compiler deduces the return type. On the second line, we're not specifying a type argument for the method, but the compiler knows that the type of the first argument has to be compatible with the type argument, and it correctly deduces that we want to create a value of type `Option<int>`[‡‡]. However, on the last line, we get an error saying "The type arguments for method '...' cannot be inferred from the usage." This is because here, the compiler doesn't have enough clues to know what the type should be.

Type inference in C# is limited in many ways, but it's still pretty useful. In F#, the algorithm is smarter and can infer the type in more cases, so let's look at some F# examples.

TYPE INFERENCE IN F#

In F#, we can often write large swathes of code without explicitly specifying any types, because the type inference mechanism is more sophisticated. When creating values, we use the `let` keyword and, in fact, we haven't yet seen any example where we would need to specify the type explicitly with a let binding. Listing 5.13 shows some examples that you'd probably expect to work.

Listing 5.13 Type inference for basic values (F#)

```
let num = 123                                    #A
let tup = (123, "Hello world")                   #B
let opt = Some(10)                               #C
let input = printfn "Calculating..."             #D
            if (num = 0) then None
            else Some(num.ToString())
```

#A int **#B int * string** **#C int option** **#D string option**

Out of these bindings, only the last example is particularly interesting or surprising. As we already know, everything in F# is an expression, so type inference has to work with any F# expressions (meaning any F# code, because everything is an expression). In this case, we have code that first prints something to screen and then returns an option type using a conditional expression. Note that whitespace is significant in F#'s lightweight syntax, so the `if` expression should start at the same offset as the `printfn` call.

It sees that the value assigned to `input` is returned from a conditional branch. From the true branch, it can see that it will be some option type (because we're returning `None`), but it doesn't yet know what is the type instantiation. That's inferred from the false branch, where we're returning `Some` value containing a string.

[‡‡] That's not the only type which could be valid here - we could want `Option<long>` for example. The rules for type inference with generic methods in C# 3.0 are long and complicated, but if situations where the compiler is willing to perform the inference, it usually gets the desired result.

I mentioned that the F# type inference is more sophisticated, so let's now look at several slightly tricky examples:

```
let a = null                      #1
let (a:System.Random) = null      #2
let a = (null:System.Random)      #2
```

#1 Inferred type is obj (System.Object)

#2 Different ways for adding type annotations

In the first case (#1), F# behaves differently to C#. Instead of reporting an error, F# automatically chooses the most general reference type, which is in this case .NET type `System.Object` abbreviated as `obj` in F#. The next two examples (#2) show two different ways for adding a type annotation. In general, you can place type annotation around any block of F# code if you need to. Now, let's look at one more interesting case:

```
> let n = None
val n : 'a option
```

Interestingly, this doesn't cause an error and instead F# creates a generic value. This construct doesn't have a C# equivalent; it's a value with only partially specified type. Instead of a concrete type (such as `int` or `obj`), F# uses a type parameter (and you can see that F# automatically names type parameters using letters starting with "a"). The type is fully specified later when using the value. We can for example compare the value with different types of option values without getting an error:

```
> Some("testing...") = n;;
val it : bool = false
> Some(123) = n;;
val it : bool = false
```

Now that we know how to declare and create generic values, we should also discuss how to write functions that use them! We'll talk about generic functions in detail them later, but for now I'll show you just one example to whet your appetite.

5.4.4 Writing generic functions

Most of the functions or methods that work with generic types are *higher order*, which means that they take another function as an argument. This is such an important topic that I've given it a whole chapter to itself, but we can already write a generic function without straying into higher order territory. We'll create a function that takes an option type and returns the contained value when it contains a value. If the option *doesn't* contain a value, the function throws an exception. We can start by looking at the C# version:

```
T ReadValue<T>(Option<T> opt) {                #A
   T v;
   if (opt.MatchSome(out v)) return v;
   else throw new InvalidOperationException();
}
```

#A Generic method with type parameter T

As you can see, we have created a generic method with a single type parameter. The type parameter is used in the method signature as a return value and also as a parameter to the generic `Option<T>` type. Inside the body, we use it once again to declare a local variable of this type.

This is exactly the kind of situation where F#'s type inference really shines. Take a look at the same thing implemented in F#. Interestingly, we still don't have to specify any types:

```
> let readValue(opt) =
     match opt with
     | Some(v) -> v
     | None -> failwith "No value!";;
val readValue : 'a option -> 'a        #A
```

#A Inferred signature with type parameter

As you can see from the inferred type signature, the function is generic in exactly the same way as the C# version. The feature that allows this is called *automatic generalization* and we'll discuss it in depth later, but for the moment, here's a 20 second description: The F# type inference algorithm searches for the most general way to assign the types and leaves everything else as a generic type parameter. In this case, it knows that the argument (opt) is an option type, because we're matching it against Some and None discriminators. It also knows that the function returns a value contained in the option type, but it doesn't know what type it is, so it makes this type a generic type parameter.

Hopefully this has piqued your interest and you're looking forward to hearing more about both automatic generalization and higher order functions–but first we should really finish our tour of common functional values. In other languages you wouldn't normally think of a function as a value, but that's one of the essential aspects that make functional programming so powerful and elegant.

5.5 Function values

We've already seen an example of using functions as values in chapter 3, where we wrote a function to aggregate list elements using another function given as an argument. In this way, we were able to use the same aggregation for different purposes - once we used it to calculate the sum of all the elements in a list and later we found the largest element in a collection.

Working with collections of data is probably the best way of showing why using functions as values is important. Having said that, it's far from the *only* scenario where this concept is useful, as you'll see in the rest of the book. Let's start by looking at an example of imperative code that selects even numbers from the given collection and returns them in another collection:

```
var nums = new int[] {4,9,1,8,6};
var evens = new List<int>();       #A
foreach(var n in nums)             #A
    if (n%2 == 0)                  #B
        evens.Add(n);              #A
return evens;
```

#A Boilerplate code
#B The important part

The annotation refers to all three lines marked with #A.

Imagine which lines of the above code you would need to modify if you wanted to filter the collection differently, for example to return all positive numbers. Perhaps surprisingly, 3 of the 4 lines shown (not counting the first one which just initializes data) are just boilerplate code that would stay exactly the same. By using a function as a value and by accepting it as a parameter we can extract the common parts of the code as a reusable method. The calling code then just has to specify an argument which describes the part which varies for different filters: the predicate to apply to each element.

In fact, many standard functions such as filtering are already available in F# and in .NET 3.5 LINQ added almost the same functions for working with collections. Some of them are named differently though. In F# a function which takes a predicate and performs filtering is called `filter`, whereas in LINQ it's called `Where` (similar to a SQL WHERE clause) Listing 5.14 shows an implementation of the previous example using these functions.

Listing 5.14 Filtering using predicate

```
// C# version
using System.Linq;                                        #A

var nums = new int[] {4,9,1,8,6};
var evens = nums.Where(n => n%2 == 0);                    #1
PrintNumbers(evens);                                      #B

// F# version with output from F# interactive
> let nums = [ 4; 9; 1; 8; 6 ]
val nums : int list
> let evens = List.filter (fun n -> n%2 = 0) nums     #2
val evens : int list = [ 4; 8; 6 ]
```

#A Required to find the 'Where' extension method
#1 Filtering using predicate
#B Print the results to the console
#2 Filtering using predicate

If we had to write the predicate as a normal method in C# or function (written using `let`) in F#, it wouldn't make the code any shorter. The key feature that makes the code brief is the ability to write the function (in this case the predicate) inline directly when calling `Where` method (#1) or `filter` function (#2).

In C#, this notation is called a *lambda expression* and in F# it's a *lambda function*. As most of this book is about F#, I'll use the F# name consistently throughout. In both cases, the word "lambda" refers to the Greek letter from lambda calculus, which I mentioned briefly in chapter 2.

What is a function value?

In functional programming languages, the existence of functions is motivated by mathematical notion of a function. This is in many ways different to the way that programmers with an imperative background intuitively think about functions. In imperative programming, a function is a routine that takes some arguments, executes some code and returns the result. However a function in this sense can do anything. Most importantly, it can use and modify global state, so the result of calling the same function with exactly the same arguments can differ. The most obvious example of this is probably a pseudo-random number generator - it wouldn't be very random if it always returned the same result!

In math, a function is more a relation between the arguments and the result. This means that a mathematical function always returns the same result given the same arguments. Clearly, this is the way our predicate from the previous example works. It always returns the same result for the same argument (`true` for even numbers and `false` for odd ones). Most of the functions we'll write will behave like this, but we'll see some interesting and useful exceptions from this rule at the end of the next chapter. You might like to think about what a mathematical pseudo-random number generator function would have to look like…

For those who come from an object-oriented background, there is one more way to look at functions. You can think of a function value as an object implementing a really simple interface with just a single method. Using this understanding, the predicate from the previous example corresponds to the following interface:

```
interface Function_Int_Bool {
    bool Execute(int arg);
}
```

In C#, delegates are somewhat similar to functions and C# 3.0 moves them very close to this simple concept. However, the concept of a function as it's used in F# and functional programming is based primarily on the notion from mathematics. In this sense, F# functions are a lot more straightforward then interfaces or delegates - they are just functions.

In the earlier examples, we've seen that lambda functions are a key element that makes concise functional style of programming possible. We'll work with them all the time through the entire book, so let's look at them in more detail.

5.5.1 Lambda functions

In F#, lambda functions create exactly the same function as the usual declaration using a `let` binding. In C#, there is no built-in concept of a function, so we work either with methods or with delegates. When you write a lambda function it is converted into a delegate or an expression tree, but you cannot use lambda function in C# to declare an ordinary method. Vitally, delegates can be also used like any other value in C#, so you can pass them as arguments to other methods, which in turn means we can use them to write higher order functions in C#. Let's start by looking at a short F# interactive session, then write similar code in C#. Listing 5.15 shows how we can write a function in F# using a let binding and lambda function syntax.

Listing 5.15 Using lambda functions and let bindings (F# interactive)

```
> let square1(a) = a * a;;            #A
val square1 : int -> int              #1

> let square2 = fun a -> a * a;;      #2
val square2 : int -> int              #3

> let add = fun a b -> a + b;;        #4
val add : int -> int -> int

> add 2 3                             #B
val it : int = 5
```

#A Square written using let binding
#1 Its signature
#2 Square written using lambda notation
#3 Signature is the same
#4 Adding two integers
#B Calling the function

We started off by writing a simple function called `square1` that calculates square of the given number, in the same way we've seen several times before. After we've entered it, F# prints its signature (#1) (the type of the value) which tells us that it takes an integer and returns an integer. Next, we declare another value called `square2` and initialize it to a function using lambda notation (#2). As you can see by looking at the output (#3), the two declarations are equivalent. Finally, we declare another value (#4) which shows the syntax for a lambda function with two parameters. After seeing these examples, you could probably rewrite any F# function written using a let binding to use the lambda notation and vice versa.

Now, let's see how we can write the same thing using lambda functions in C#:

```
Func<int, int> square =       #A
    a => a * a;
Func<int, int, int> add =     #B
   (a, b) => a + b;
```

#A Square as a delegate using lambda function
#B Lambda function with multiple parameters

We're using a delegate type called `Func`, which is available in .NET 3.5. This delegate represents a function and its type arguments specify the types of the parameters and the return type. Notable differences between the C# and F# syntax are that the F# version starts with `fun` keyword and that in C# you specify multiple arguments in parentheses and also that in C# you have to specify the type explicitly.

From delegates to functions in C#

As already mentioned, functions in C# are represented using delegates and in particular the new `Func` family of delegates. In one sense, lambda functions and this delegate are a revolutionary change adding functional programming to C#, but it can also be seen as just a natural evolution of features that were already available in C#. This book usually takes the former view, but we'll look at the evolutionary aspect just for a moment.

In the first version of C#, we already had delegates, but without generics, you had to declare a separate delegate for every combination of return and parameter types. When creating delegates, we also had to write the code inside a named method, so we could write a code like this:

```
delegate int FuncIntInt(int a, int b);
FuncIntInt add = new FuncIntInt(Add);
```

The code assumes that there is an `Add` method with two integer parameters and an integer return type. C# 2.0 was a big step forward. It added generics, so we could declare a generic delegate like `Func` and use the new anonymous methods feature to create them instead of writing named method:

```
delegate R Func<T1, T2>(T1 arg1, T2 arg2);
Func<int, int, int> add = delegate(int a, int b) { return a + b; }
```

Finally, .NET 3.5 and C# 3.0 came with several other changes. The `Func` delegate was added to the system libraries, so you no longer have to declare it yourself, and C# added lambda expressions that allow us to write the same code in a much more succinct way:

```
Func<int, int, int> add = (a, b) => a + b;
```

Lambda expressions have another interesting feature, which is that they can be converted into *expression trees* when we declare them as the `Expression` type. this allow us to treat the code of the lambda expression as data and obtain some representation of the source code of the lambda expression. This is very important for using LINQ with databases, but it isn't a key feature for us now. Also, due to this feature, we can't use `var` keyword when declaring lambda expressions, because the compiler needs to decide whether to compile it as a delegate (`Func`) or whether to store the expression tree (`Expression`). You can find more information on our web site http://www.functional-programming.net.

The `Func` delegate and lambda expressions in C# are very similar to functions in F#, but F# had functions right from its inception, so it has little need for delegates. It supports using delegates mainly for interoperability reasons, but you probably won't use them very often.

We've looked at a few examples of lambda functions in both F# and C#, but there are still a few important things to look at.

TYPE ANNOTATIONS, ACTIONS AND STATEMENT BLOCKS

In the previous examples, we didn't have to specify the parameter types explicitly. This is the normal behavior in F#, because its type inference capabilities are very powerful and in the previous examples it had enough clues to deduce the type. The situation in C# is quite interesting in a different way.

```
Func<int, string> toStr1 = num => num.ToString();
Func<int, string> toStr2 = (int num) => num.ToString();
```

Both lines show exactly the same code, with the sole difference being that the second line explicitly specifies the type of the `num` parameter. Both lines are correct, so how does C# know the type of `num` in the first line? The answer is that it uses the type from the variable declaration. It knows that `Func<int, string>` is a delegate that takes an integer as an argument, so it infers that the type of num should be integer.

Explicit parameter typing is rarely needed in C#. You can't use the `var` keyword to declare lambda functions anyway, so C# will usually be able to deduce the type. One exception is where we're using the lambda function as an argument to a specific generic method. Even in F# we may occasionally need to give the compiler more information, which we do with *type annotations*. Listing 5.16 shows lambda function with a typo annotation to explicitly state the type of its parameter.

Listing 5.16 Advanced lambda functions (F# interactive, C#)

```
// F# version of the code (using F# interactive)
> let sayHello =
     (fun (s:string) ->                                    #1
        let msg = String.Format("Hello {0}!", s)
        Console.WriteLine(msg)
     )                                                     #2
val sayHello : string -> unit                              #3

// C# version of the code
Action<string> sayHello =                                  #4
   s => {
      var msg = String.Format("Hello {0}!", s);
      Console.WriteLine(msg);
   };                                                      #5
```

#1 Using F# type annotation
#2 Complex lambda function is enclosed in braces
#3 Inferred function signature
#4 Function declared as an Action
#5 Lambda function written as a statement block

This example shows several interesting things. The first is the use of a type annotation in the F# version (#1). The syntax for type annotations in lambda functions is exactly the same as anywhere else in the F# code. The reason why we have to use it in this case is that there are several overloads of the `String.Format` method (with arguments of type `int`, `string`, and so forth); F# wouldn't be able to determine which overload to use without the type annotation.

Another notable thing is that the body of the lambda function isn't just a single expression. In F#, we added a single let binding and enclosed the whole lambda function in braces (#2). In the C# version, we added a variable declaration and changed the syntax to use *statement block*. A statement block means that the body of the lambda function is enclosed in curly braces (#5) which allows us to write several statements inside the body. To return a result from a lambda function using a statement block, you use the `return` keyword as if you were returning a result from a method.

However, in this example the lambda function doesn't return a result at all. In F# where `unit` is an ordinary type, the inferred signature of the function is `string -> unit` (#2). This is an ordinary F# function that returns `unit` (that is, nothing) as a result. In C#, we cannot write `Func<string, void>` because `void` isn't a real type. For this reason, C# has another family of delegate types called `Action`, which represents lambda functions with no return type. The `Action` and `Func` delegates are very useful and they are in many ways similar to the F# function type, so let's look at the type of a function value in more detail.

5.5.2 The function type

We've seen that the type of function values in F# is written using the arrow symbol. This is in many ways similar to the way tuples are constructed. Earlier, we've seen that a tuple type can be constructed from other simpler types using a type constructor with an asterisk (e.g. `int * string`). The function type is constructed in a similar way, but using the function type constructor (e.g. `int -> string`). Of course, there is no value constructor for functions. In some sense, a function is a relation that specifies return value for every possible input, so instead of specifying enormous number of all combinations of this relation, we specify code that calculates the result using lambda functions.

In C#, you can see this similarity as well. If we use our generic `Tuple` type and `Func` delegate, we can write the examples from previous examples as `Tuple<int, string>` and `Func<int, string>`. Instead of using built-in types as we can in F#, we have similar constructs implemented as ordinary C# types using generics. However, there is a very important difference between the F# function type and C# `Func` (or `Action`) delegate. The difference is that the type of an ordinary F# function is exactly the same as the type of an equivalent function written as a lambda function. In C#, lambda functions are converted into delegates and a delegate isn't the same thing as method. The distinction is subtle but important: we'll see it more clearly when we consider functions with multiple parameters in F#. Before that, let's look how we can use function value as an argument or a return value.

FUNCTIONS AS AN ARGUMENT AND RETURN VALUE

We've already used a function as an argument in C# and F# in chapter 3, so the basic idea this shouldn't be new to you. However, we haven't used lambda functions in that way yet. Lambda functions are the easiest way to write a function this is just used as an argument to another function. Listing 5.17 provides a simple example. The function at the start of the listing takes a number and a function as arguments and calls the function twice, using the result of the first call as an argument for the second.

Listing 5.17 Function as an argument in C# and F#

```
// Using C#                                  // Using F# interactive
```

```
int Twice                        #1
   (int n, Func<int,int> f) {    #1
   return f(f(n));
}

var r = Twice(2, n => n * n);    #2
// Result: r == 16
```

```
> let twice n (f:int -> int) =
     f(f(n))                    #3
val twice :
   int -> (int -> int)
   -> int                       #4

> twice 2 (fun n -> n * n)      #5
val it : int = 16
```

#1 C# method taking function as an argument
#2 Calling using lambda function
#3 F# function taking function as an argument
#4 Type of the function
#5 Calling using lambda function

In this example, we can see all the important features in a single place. It shows how to declare a C# method and an F# function that takes a function as an argument (#1, #3) and how to call them using lambda functions (#2, #5). In F# we use type annotations to tell the F# compiler that we want to work only with integers. As we'll see in the next chapter, without this annotation it would automatically make the function more general. This is usually desirable, but I wanted to keep this example as simple as possible.

In the C# version, `Twice` is a method with a delegate as a parameter, but in the F# version it is a function, so when we look at the F# signature (#4), we can see that it is constructed with just a function type constructor (arrow). The second parameter is a function taking an integer and returning an integer; the overall type is a function with two parameters.

Since a function is an ordinary value, we can also write a function (or method in C#) that returns a function as a result. Listing 5.18 shows a function that takes a number as an argument and returns a function that adds this number to any given argument.

Listing 5.18 Function as a return value in C# and F#

```
// Using C#

Func<int, int> Adder(int n) {
    return (a) => a + n;      #1
}

Func<int, int> add10 =        #3
    Adder(10);

var r = add10(15);            #5
// Result: r == 25
```

```
// Using F# interactive

> let adder(n) =
      (fun a -> a + n)          #1
val adder : int -> int -> int   #2

> let addTen = adder 10         #3
val addTen : int -> int         #4

> addTen 15                     #5
val it : int = 25
```

#1 `Adder` takes an `int` as an argument and returns a function as a result. In C# the return type is specified explicitly and it is a `Func` delegate, while in F# the return type is deduced by type inference: it's a function with type `int -> int`.
#2 As we'll see later, the printed type signature represents a function taking integer and returning a function. We can see it more clearly if we add braces to the printed signature. Then it would be written as `int -> (int -> int)`.
#3, #4 Calling the function (or C# method) that returns a function. In C# the result is a delegate and in F# it is an ordinary function. As the printed type signature shows (#4) it takes integer as an argument and returns also an integer.

#5 Calling the returned function (or Func delegate in C#); in F# we're using it as an ordinary function and in C#, we're calling it as a delegate.

Below the code with bullets [WritingDevices_BulletsSample.png]

The listing shows returning function as a result using a very simple example, but we'll see in the next few chapters that returning one function from another can be very useful. There is however one thing about the code that deserves further explanation. If we look at the type signature of the F# `adder` function, we can see that its type is `int -> int -> int`. This looks like a function with two arguments, but it's probably easier to think of it as `int -> (int -> int)`. They mean exactly the same thing, because F# and functional languages in general have a different notion of functions with multiple parameters to the normal object-oriented understanding.

5.5.3 Functions of multiple arguments

First of all, let's quickly review what options we have when writing a function. In F#, we can use tuples when writing functions with multiple arguments. Let's look at an example of a function that adds two integers written in this style. I'll use the lambda function syntax, but you could get exactly the same results using simple let-binding in F# as well.

```
> let add = fun (a, b) -> a + b;;
val add : int * int -> int
```

As you can see by looking at the type signature, the function takes a single argument which is a tuple of the form (`int * int`) and the return type is `int`. This corresponds to the C# lambda function written in this form:

```
Func<int, int, int> add =
    (a, b) => a + b
```

The `Func<int, int, int>` delegate represents a method which has two arguments of type `int` and returns an `int`, so this is very similar to the F# version written using tuples. You can see this similarity when calling the functions as well:

```
let n = add(39, 44)                    #A
var n = add(39, 44)                    #B
```

#A F#: Calling a function with type int * int -> int
#B C#: Calling a Func<int, int, int> delegate

The syntax is exactly the same to call an F# function with a tuple as an argument as it is to call a C# `Func` delegate. Now, let's write the same code in F# using the more idiomatic F# style for writing functions with multiple arguments:

```
> let add = fun a b -> a + b;;
val add : int -> int -> int
```

This is the same signature we saw earlier when we were returning a function. We can read it as int -> (int -> int). That would be a function that takes the first argument for the addition and returns a function. The result is then a function taking the second argument. We can rewrite the code in this way using two lambda functions, nesting one inside the other:

```
> let add = fun a -> fun b -> a + b;;
val add : int -> int -> int
```

If this is the first time you've come across this idea, it can seem very odd. How can a function returning another function be the same as a function returning an integer? How can a function with just one parameter be the same as a function with two parameters? Don't worry too much if it doesn't make sense right away–I promise it will make sense eventually.

Tuples with more than two elements

Earlier on we implemented a generic type `Tuple<A, B>` for representing tuples in C#, but this class supports only tuples with two elements–unlike the real F# tuple type. How could we use it to represent F# type `int * string * bool for example`? Instead of implementing another `Tuple` class with three elements, we can nest the tuples:

```
Tuple<int, Tuple<string, bool>> tup = (...);
```

When we declare a variable like this, it can carry three values. To get the integer value, we can write `tup.First`, string is stored in `tup.Second.First` and finally, boolean value in `tup.Second.Second`.

This is similar to nested functions, such as F# type `int -> (string -> bool)`. However, there is a difference between tuples and functions. The function type above means the same thing as `int -> string -> bool`, while an F# tuple with three elements (`int * string * bool`) is different to a nested tuple type such as `int * (string * bool)`.

You may be wondering if there's any way of rewriting our previous example in C# 3.0–and indeed we can. Instead of creating a delegate of type `Func<int, int, int>` we can create a delegate of type `Func<int, Func<int, int>>`. This is closer to the F# understanding of a function with a signature of `int -> (int -> int)`:

```
Func<int, Func<int, int>> add =
    a => b => a + b;                 #1
int n = add(39)(44);                 #2
```

#A Nested lambda functions
#B Adding numbers

The declaration is written using two lambda functions (#1) just like our previous F# example. When adding numbers using this delegate (#2), we first have to invoke the first delegate, which returns another delegate. We then invoke the second delegate. In F#, where this is an entirely normal way of working with functions, the compiler optimizes it to make it more efficient.

This is all very interesting, I hear you say–but what's the point of taking functions apart in this way? It turns out to be surprisingly powerful.

PARTIAL FUNCTION APPLICATION

To show a situation where this new understanding of functions is useful, let's turn our attention back to lists. Imagine that we have a list of numbers and we want to add 10 to every number in the list.

In F# this can be written using the `List.map` function; in C# we would use the `Select` method from LINQ:

```
list.Select(n => n + 10)            #A
List.map (fun n -> n + 10) list     #B
```

#A C# version
#B F# version

That's pretty brief already, but we can be even more concise if we already have the `add` function from the previous examples. The function that List.map expects as a first argument is of type `int -> int`; that is a function taking an integer as an argument and returning another integer. The technique that we can use is called *partial function application*:

```
> let add a b = a + b;;
val add : int -> int -> int                                     #1

> let addTen = add 10;;
val addTen : int -> int                                         #2

> List.map (addTen) [ 1 .. 10 ];;                               #3
val it : int list = [11; 12; 13; 14; 15; 16; 17; 18; 19; 20]

> List.map (add 10) [ 1 .. 10 ];;                               #4
val it : int list = [11; 12; 13; 14; 15; 16; 17; 18; 19; 20]
```

The add function has a type int -> int -> int (#1). Since we now know that it actually means that the function takes an integer and returns a function, we can simply create a function `addTen` (#2) that adds 10 to a given argument just by calling `add` with only the first argument. We can then use this function as an argument to the `List.map` function (#3). This is sometimes useful, but what is more interesting is that we can use partial function application directly when specifying the first argument for `List.map` (#4).

If we look at the types involved then the type of the `add` function is `int -> (int -> int)` and by calling it with a single number as an argument, we get the result of type `int -> int` which is exactly what the `List.map` function expects. Of course, we can write exactly the same code in C# as well if we declare the `add` function using nested lambda functions:

```
Func<int, Func<int, int>> add =     #A
    a => b => a + b;

list.Select(add(10));               #B
```

#A Declaration using nested lambda functions
#B Calling 'Select' using partial function application

Just as we saw in the F# version, we call the `add` delegate and get a result of type `Func<int, int>`, which is compatible with the `Select` method. However, in C# it is more convenient to use the `Func` delegate with multiple arguments and specify the argument to the `Select` method using another lambda function, because the language supports this better.

PARTIAL FUNCTION APPLICATION AND CURRYING

A term that you can sometimes hear when using the partial function application is *currying*. This refers to converting a function that takes multiple arguments (as a tuple) into a function that takes the first argument and returns a function taking the next argument and so on. So, for example the function of type `int -> int -> int` is a curried form of a function that has a type `(int * int) -> int`. Partial function application is then the use of a curried function without specifying all the arguments.

As I've already mentioned, choosing the right style in F# can be difficult. Code that is written using tuples is sometimes easier to read for a large number of arguments, but it can't be used with partial function application. In the rest of the book, we'll use the style that feels more appropriate in each case, so you can get the intuitive understanding of which one is better. Most importantly, we'll use tuples in cases where it makes the code more readable and the style allowing partial function application in situations where that gives us clear benefits. We'll see plenty of examples of the latter when we look at higher order functions in the next chapter.

5.6 Summary

In this chapter we've been talking about values–the fact that the discussion went into a lot of detail about functions just highlights the fact that in F# functions *are* values! We've seen several ways for creating different values and corresponding composed types. We started by looking at tuples, which gave us a way to store multiple values as one. Next, we've seen discriminated unions that allow us to represent values consisting of various alternatives. When declaring discriminated union, we specify what are the options and a value can then be one of the declared options. We also looked at generic types that are similar to generic classes in C#. We've used them to declare types that can be used for carrying different values, which makes the code more general and reusable.

As well as looking at the theory behind these types, we've looked at some of the common uses of them in F#. We've seen that multiple values (tuples) are useful for returning multiple results from a single function, and how this can be more appealing than using C# "out" parameters. A particularly interesting alternative value (discriminated union) is the option type, which can represent values that can be undefined. This is a very useful alternative to using `null` values, as the language forces the calling code to write a case that handles the "undefined" case when we use pattern matching.

Finally, we looked at the function type in F# and its equivalent in C# - the `Func` delegate. We've seen how functions can be created using lambda function syntax and how they can be used as arguments as well as return values from another function or a method.

In one last twist to function values, we've also seen a very useful technique called partial function application.

In this chapter we've seen only the basic ways for working with values. This is because many of the operations aren't usually written directly and are instead use higher order functions. Working with values in this way is the main topic for the next chapter. Using higher order functions, we'll be able to hide the logic for working with the value in a function and specify just the most important part of the operation using a function value given as an argument.

6 Processing values with higher order functions

In the previous chapter, we introduced the most common functional values. We've seen how they can be constructed and how we can work with them using pattern matching. However, expressing all the logic explicitly like this can be tedious, especially if the type has a complicated structure.

The types of values that are composed from one or more simpler values include tuples and options from the previous chapter, but also lists from chapter 3. Tuples are in general formed from values of different types, so they contain value of one type exactly once. Options can contain zero or one value and lists contain any finite number of elements. When working with these composed values, we often want to apply some operation to the underlying values. This involves recurring and boilerplate task of deconstructing the composed value into its components and reconstructing it after we apply the operation.

In this chapter, we'll see how to process values in an easier way. We'll do this by writing functions that abstract us from the underlying structure of the value and can be simply parameterized to perform a particular operation with some part of the value. We'll see that this approach gives us more concise way than using pattern matching explicitly.

We'll first look at higher order and generic functions from a technical point of view, to provide some background for our discussion about value processing. Then we'll talk about processing functions for all the values that we've discussed so far, and some interesting relationships between processing functions for different kinds of values.

6.1 Generic higher order functions

Higher order functions are a way for writing generic functional code, meaning that the same code can be reused for many similar but distinct purposes. This is a key of modern programming, because it allows us to write fewer lines of code by factoring out the common part of the computation.

Generic code in functional and object-oriented programming

When writing generic code, we usually want to perform some operation on the value that we obtain, but since the code should be generic, we don't want to restrict the type of the value too much: we want to allow further extension of the code.

The elementary (but not always the best) solution to this problem using object-oriented programming is to declare an interface. The actual value given to a method will have all operations required by the interface, so it will be possible to perform needed operations on the value. A trivial example in C# might look like this:

```
interface ITestAndFormat {
   bool Test();
   string Format();
}
void CondPrint(ITestAndFormat tf) {
   if (tf.Test()) Console.WriteLine(tf.Format());
}
```

In functional programming, the approach is usually to work with generic methods that use type parameters and can work with any type. However, we don't know what operations can be performed on the value, since the type parameter can be substituted by any actual type. As a result, functional languages use a different method for specifying operations - they pass functions for working with the value as additional arguments. The functional version of the previous example in C# would look like this:

```
void CondPrint<T>(T value, Func<T, bool> test, Func<T, string> format) {
   if (test(value)) Console.WriteLine(format(value));
}
```

For a small number of functions this is a very efficient method, because we don't need to declare the interface in advance. However, for more complicated processing functions, we can still use interfaces as we'll see in chapter 9. Also calling the function is easier, because we can implement the operations using lambda functions. As we'll see in section 6.5, writing a code like this in F# is also largely simplified by the use of type inference.

Higher order functions are very important for functional programming and we'll see how they can be used for working with several functional values shortly. Methods like `CondPrint` from the previous sidebar will be quite important for us, so let's look how we can implement the same functionality in F#.

6.1.1 Writing generic functions in F#

We've already seen a simple generic function in the previous chapter, but it only used a single argument which was a generic option type. Listing 6.1 shows an F# implementation of the CondPrint method from the sidebar above. It takes three arguments - a value, a function that tests whether the value should be printed and a function for formatting the value.

Listing 6.1 Generic function 'condPrint' (F# interactive)

```
> let condPrint value test format =                                 #1
     if (test(value)) then printfn "%s" (format(value))             #A
  ;;
val condPrint : 'a -> ('a -> bool) -> ('a -> string) -> unit        #2

> condPrint 10 (fun n -> n > 5)                                     #B
               (fun n -> "Number: " + n.ToString());;               #B
Number: 10
```

#1 Function with three arguments
#A Calling functions given as arguments
#2 Inferred type signature
#B Test the function

As you can see, we've declared a function with three parameters using a let binding (#1), but we didn't need to specify the type of any of the parameters. This is because F# type inference works for functions too. We'll see later just how sophisticated it can be. For now, we can just be content that it automatically infers the type signature of the function (#2), which corresponds to our previous generic method in C#.

SYNTAX FOR WRITING HIGHER ORDER FUNCTIONS

In chapter 4, we were discussing whether it is better to pass multiple pieces of data to a function as separate arguments (for example `add 2 3`) or as a tuple (for example `add(2, 3)`). When writing higher order functions, we'll use the first style, because this makes it easier to use lambda functions as arguments. It also supports the pipelining operator, which we'll see shortly.

Another way of representing generic functionality in F# is to write custom operators. We'll want to use these later, so let's take a brief look now, and also introduce the *pipelining operator*–a particularly useful operator from the F# library.

6.1.2 Custom operators

Custom operators are defined using let bindings in a similar way to functions. They can use any characters from the usual F# mathematical (`+/-*<>`) or logical operators (`!&|=`) and also several other characters (`$%.?@^~`). When declaring an operator, you enclose its name in braces, which is the only difference from a normal let binding. Be careful when using an asterisk, because `(*` is a beginning of multi-line F# comment-the solution in that case is to add additional space between the parenthesis and asterisk. Listing 6.2 shows how to declare and use a simple operator for working with strings.

Listing 6.2 Working with strings using custom operator (F# interactive)

```
> let (+>) a b = a + "\n>>" + b;;                  #A
val ( +> ) : string -> string -> string

> printfn "%s" ("Hello world!" +>                  #B
                "How are you today?" +>            #B
                "I'm fine!");;                     #B
>> Hello world!
>> How are you today?
>> I'm fine!
```

#A Operator for concatenating strings in a special way
#B Concatenate several messages

The benefit of using a custom operator instead of a function is that you can use it with infix notation. This means that instead of `concat "A" (concat "B" "C")`, we can write `"A" +> "B" +> "C"`. This is particularly useful when applying the operator several times as in our previous example, because then you don't have to wrap each call in braces.

SIMULATING CUSTOM OPERATORS IN C#

In C# you can't declare new operators, although you can overload existing ones. However, the same pattern can be achieved to some extent using extension methods. This is a new feature in C# 3.0, so we briefly introduce them in a sidebar.

Extension methods

In C#, every method has to be wrapped in a class and operations that work with object are part of the class declaration and can be called using dot-notation. Extension methods give us a way to add new method for working with an object, without modifying the original class declaration. Previously, this could be done by writing static method like this:

```
StringUtils.Reverse(str);
```

This is very impractical though, because finding a static method in some "Utils" class is quite difficult. In C# 3.0 we can implement `Reverse` as an extension method and then call it this way:

```
str.Reverse();
```

Implementing an extension method is quite easy, because it is just an ordinary static method, with a special modifier. The only difference is that it can be invoked as an instance method using dot-notation. However, it is still static method, so it can neither add new fields nor access private state of the object:

```
static class StringUtils {
   public static string Reverse(this string str) { /* ... */ }
}
```

All extension methods have to be enclosed in a non-nested static class and they have to be static methods. The keyword `this` is used before the first parameter to tell the compiler to make it an extension method.

If we implement the string concatenation in the previous example as an extension method, we'll get syntax very similar to the original F# version. The listing 6.3 shows the same code written using standard static method call and using extension methods.

Listing 6.3 Working with strings using extension methods (C#)

```
public static string AddLine(this string str, string next) { #A
   return str + "\n>>" + next;
}

Console.WriteLine(StringUtils.AddLine(                       #B
   StringUtils.AddLine("Hello world!",                       #B
                       "How are you today"),                 #B
   "I'm fine!"));                                            #B

Console.WriteLine("Hello world!"                             #C
         .AddLine("How are you today")                       #C
         .AddLine("I'm fine!"));                             #C
```

#A 'this' keyword precedes the first parameter
#B Using standard static method calls
#C Concatenate strings using extension method

The benefits are purely in terms of readability: we can write the method calls in the same order in which we want them to occur, we don't need to specify the class implementing the method, and we don't need extra bracing. As it is often the case, syntax makes quite an important difference.

THE F# PIPELINING OPERATOR

The pipelining operator (|>) allows us to write the first argument for a function on the left side; that is, before the function name itself. This is useful if we want to invoke a several processing functions on some value in sequence and we want to write the value that's being processed first. Let's look at an example, showing how to reverse a list in F# and then take its first element:

```
List.hd(List.rev [1 .. 5])
```

This isn't very elegant, because the operations are written in opposite order then in which they are performed and the value that is being processed is on the right side, surrounded by several braces. Using extension methods in C#, we'd write:

```
list.Reverse().Head();
```

In F#, we can get the same result by using the pipelining operator:

```
[1 .. 5] |> List.rev |> List.hd
```

Even though, this may look tricky, the operator is in fact very simple. It has two arguments - the second one (on the right side) is a function and the first one (on the left side) is a value. The operator gives the value as an argument to the function and returns the result.

In some senses, pipelining is similar to calling methods using dot-notation on an object, but it isn't limited to intrinsic methods of an object. This is similar to extension methods, so when we write a C# alternative of an F# function that's usually used with the pipelining operator, we'll implement it as an extension method.

Now that we've finished our short introduction about generic higher order functions and operators, we can finally look how they can be used for solving daily functional programming problems. The first topic that we'll discuss is using higher order functions for working with tuples.

6.2 Working with tuples

We've been working with tuples from our first functional code in chapter 3, so you're already quite familiar with them. However, we haven't looked at how we can work with them using higher order functions. Tuples are really simple, so you can often use them directly, but in some cases the code isn't as concise as it could be. Tuples are a good starting point for exploring higher order functions because they're so simple. The principles we'll see here are applicable to other types, too. In chapter 3, we used tuples to represent a city and its population. When we wanted to increment the population, we had to write something like this:

```
let (name, population) = oldPrague
let newPrague = (name, population + 13195)
```

This is very clear, but a bit longwinded. The first line deconstructs the tuple and the second one performs a calculation with the second element and then builds a new tuple. Ideally, we'd like to say that we want to perform a calculation on the second element deconstructing and re-constructing the tuple. First let's quickly look at the code we want to be able to write, in both F# and C#, and then we'll implement the methods which make it all work. This is what we're aiming for:

```
let newPrague = oldPrague |> mapSecond ((+) 13195)   #A
var newPrague = oldPrague.MapSecond(n => n + 13195); #B
```

#A F# version
#B C# version

This version removes all the additional code to re-construct the tuple and specifies the core idea - that is, we want to add some number to the second element from the tuple. The idea that we want to perform calculation on the second element is expressed by using the `mapSecond` function in F#. Listing 6.4 shows the implementation of both this and the similar `mapFirst` function.

Listing 6.4 Higher order functions for working with tuples (F# interactive)

```
> let mapFirst  f (a, b) = (f(a), b)                       #1
  let mapSecond f (a, b) = (a, f(b))                       #2
  ;;
val mapFirst  : ('a -> 'b) -> 'a * 'c -> 'b * 'c           #3
val mapSecond : ('a -> 'b) -> 'c * 'a -> 'c * 'b           #3
```

#1 Applies function to the first element
#2 Applies function to the second element
#3 Inferred type signatures

Listing 6.4 implements two functions: one that performs an operation on the first element of the tuple (#1) and one that acts on the second element (#2). The implementation of these functions is quite simple: we use pattern matching in the parameter list to deconstruct the given tuple, and then call the function on one of the elements. Finally, we return a new tuple with the result of the function call and the original value of the other element. Even though the body doesn't look difficult, the inferred type signatures (#3) look rather complicated when you see them for the first time. We'll come back to them shortly.

MAP OPERATION

I used the term *map* in the name of the functions above. A map (also called a *projection*) is a very common operation and we'll see that we can use it with many data types. In general, it takes a function as an argument and applies this function to one or sometimes more values that are stored in the data type. The result is then wrapped in a data type with the same structure and returned as a result of the map operation. The structure isn't changed, because the operation we specify doesn't tell us what to do with the composed value. It specifies only what to do with the component of the value and without knowing anything else the projection has to keep the original structure. This description may not be fully clear now, because it largely depends on the intuitive sense that you'll get after more similar operations later in this chapter.

The signatures of these functions are useful for understanding what they do. Figure 6.1 disassembles the signature of `mapFirst` and shows what each part of it means.

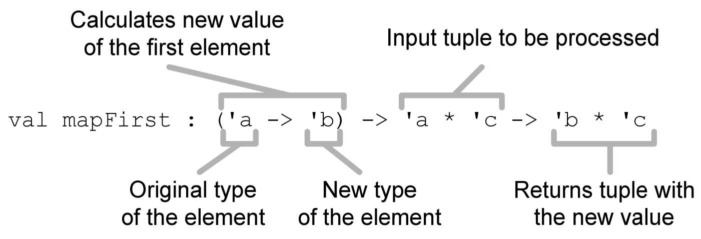

Calculates new value of the first element
Input tuple to be processed
val mapFirst : ('a -> 'b) -> 'a * 'c -> 'b * 'c
Original type of the element
New type of the element
Returns tuple with the new value

Figure 6.1 The 'mapFirst' function takes a function as the first argument and applies it to the first element of a tuple that is passed as the second argument.

Let's look at what the signature tells us about the function. First of all, it is a generic function and it has three type parameters, automatically named by F# compiler. It takes a function as the first parameter and a tuple containing values of types `'a` and `'c` as the second argument and the signature tells us that the returned tuple is composed from values of types `'b` and `'c`.

Since the function doesn't have any safe way of working with values of type `'c`, it is likely that the second element is just copied. The next question is how we can get a value of type `'b` in the result. We have a value of type `'a` (the first element of the tuple) and a function that can turn a value of type `'a` into a value of type `'b`, so the most obvious explanation is that `mapFirst` applies the function to the first element of the tuple.

Now that we've implemented the `mapFirst` and `mapSecond` functions, let's start using them. Listing 6.5 shows an F# interactive session demonstrating how they can be used to work with tuples.

Listing 6.5 Working with tuples (F# interactive)

```
> let oldPrague = ("Prague", 1188000);;
val prague : string * int

> mapSecond (fun n -> n + 13195) oldPrague;;       #1
val it : string * int = ("Prague", 1201195)

> oldPrague |> mapSecond ((+) 13195);;             #2
val it : string * int = ("Prague", 1201195)
```

#1 Using a lambda function as an argument
#2 Using partial application and pipelining

The example shows two ways for writing the same operation using the `mapSecond` function. In the first case, we directly call the function (#1) and give it a lambda function as the first argument and the original tuple as the second argument. If you look at the resulting tuple printed by F# interactive, you can see that the function was applied to the second element of the tuple as we wanted.

In the second version (#2) we're using two powerful techniques. We're using partial function application, which was introduced in the previous chapter, to create a function which adds `13195` to the second element. Instead of writing lambda function explicitly, we just wrote `(+) 13195`. If an operator is used in braces, it behaves like an ordinary function, which means that we can add two numbers by writing `(+) 10 5`. If we use partial application and give it just one argument, we obtain a function of type `int -> int` that adds the number to any given argument and is compatible with the type expected by the `mapSecond` function. The type is `'a -> 'b` and in this case `int` will be substituted for both `'a` and `'b`.

Thanks to pipelining, we can write the original tuple and then the function to apply. This makes the code more readable, describing first what's we're going to manipulate and then what we're going to do with it–just like in C# where operations are typically of the form `target.MethodToCall()`. The use of pipelining is also a reason why `mapSecond` takes the lambda function as the first argument and tuple as the second one and not the other way round.

I started this section by talking about F#, because showing the inferred type signature of a higher order function and using pipelining can be demonstrated very naturally in F#. Of course, we can use the same concepts in C# and we'll do so in the next section.

6.2.1 Methods for working with tuples in C#

In this section, we'll be working with the generic `Tuple` class from chapter 3 and we'll add similar functionality to what we've just seen in F#. Listing 6.6 shows C# alternatives to higher order functions `mapFirst` and `mapSecond`.

Listing 6.6 Extension methods for working with tuples (C#)

```
public static class Tuple {
   public static Tuple<B, C> MapFirst<A, B, C>
         (this Tuple<A, C> t, Func<A, B> f) {   #1
      return Tuple.New(f(t.First), t.Second);    #A
   }
   public static Tuple<C, B> MapSecond<A, B, C>
         (this Tuple<C, A> t, Func<A, B> f) {
      return Tuple.New(t.First, f(t.Second));    #B
   }
}
```

#1 Create extension method using 'this' modifier
#A Apply function to the first element
#B Apply function to the second element

The implementation of these methods is very straightforward, but we have to specify the types explicitly. I used the same names for the type parameters as in the previous F# version, so you can compare them. In C#, the type signature is mixed with the implementation, which makes the code harder to read, but we can look at the type signature separately:

```
Tuple<B, C> MapFirst(Tuple<A, C>, Func<A, B>)
```

This corresponds to the previous F# signature. You can see that the last argument is a function that turns a value of type A into a value of type B. We're using type A in the input tuple and B in the result. We also changed the order of parameters, so the original tuple is now the first argument. This is because we want to use the method as an extension method for tuples, so the tuple has to come first. (We also added `this` modifier to the first parameter (#1) to tell the compiler we wanted to make it an extension method.) Now we can use the method both directly and as an extension method:

```
var oldPrague = Tuple.New("Prague", 1188000);
var newPrague1 = Tuple.MapSecond(oldPrague, n => n + 13195); #A
var newPrague2 = oldPrague.MapSecond(n => n + 13195);        #B
```

#A Direct call
#B Calling extension method

When calling the method directly, the code is very similar to the first use in F#, because it calls a method with two arguments, and uses a lambda function for one of them. In F# we were then able to use the pipelining operator to write the original tuple first, and as you can see on the last line, extension methods play a similar role. Because `MapSecond` is written as an extension method, we can call it using dot-notation on the `oldPrague` object.

In this section, we've seen two useful higher order functions for working with tuples and I'm sure you'd be now able to write other functions such as applying the same operation on both elements of a tuple and so on. After discussing multiple values in the previous chapter, we talked about alternative values, so we'll follow the same pattern and look at writing higher order functions for alternative values now.

6.3 Calculating with schedules

In this section, we'll apply the techniques from previous section to alternative values. When working with tuples, we found it very helpful to write a function that works with one element from the tuple. Similarly, when working with alternative values, we'll need a higher order function that performs some operation on one or more of the alternatives. We'll follow the examples from the previous chapter, so we'll start with a schedule type and then we'll look at the option type.

In the previous chapter, we implemented a type for representing schedule of an event. In F#, it is a discriminated union called `Schedule` that can contain one of three options. The three discriminators for the alternatives are `Never`, `Once` and `Repeatedly`. In C#, we represented it as an abstract class `Schedule` with a property called `Tag` and one derived class for representing each of the three options. In this section we'll add a higher order function for working with schedules.

Now, imagine what the application might want to do with the schedule. The most common operation (especially in the today's busy world) could be rescheduling the events. For example, we may want to move all the events we know about by one week, or move events scheduled for Monday to Tuesday. Writing this explicitly would be difficult, because we'd have to provide code for each of the three different types of schedule.

However, if you think about the problem, we only want to calculate a new time based on the original time, without changing any other property of the schedule. In listing 6.7, we implement a function that allows us to do exactly this.

Listing 6.7 Map operation for schedule type (F# interactive)

```
> let mapSchedule f sch =
     match sch with
     | Never -> Never                                            #A
     | Once(dt) -> Once(f(dt))                                   #B
     | Repeatedly(dt, ts) -> Repeatedly(f(dt), ts)               #C
  ;;
val mapSchedule : (DateTime -> DateTime) -> Schedule -> Schedule #1
```

#A Unscheduled events remain unscheduled
#B Reschedule event occurring once
#C Reschedule repeated event

I called the operation `mapSchedule`, because it performs some operation for all the date and time information that the schedule contains. When the alternative is `Never`, it simply returns `Never` with no re-calculation. When it is `Once`, the function given as an argument is used to calculate the new time. When the schedule is represented using `Repeatedly`, the function is used to calculate new time for the first occurrence, keeping the original period between occurrences.

If you look at the type signature (#1), you can see that the first parameter is a function that takes `DateTime` as an argument and returns a new `DateTime`. This is used for calculating the new time of scheduled events. The original `Schedule` is the last parameter. This parameter ordering makes it possible to call this function using the pipelining operator, just as we did with the tuple projections earlier. Listing 6.8 shows how we can manipulate a collection of schedules using this function.

Listing 6.8 Rescheduling using 'mapSchedule' function (F# interactive)

```
> let schedules =
     [ Never; Once(DateTime(2008, 1, 1));
       Repeatedly(DateTime(2008, 1, 2), TimeSpan(24*7, 0, 0)) ];;    #1
val schedules : Schedule list

> for s in schedules do
     let ns = s |> mapSchedule (fun d -> d.AddDays(7.0))              #2
     printfn "%A" ns;;                                                #A
Never                                                                 #3
Once 8.1.2008 0:00:00                                                 #3
Repeatedly (9.1.2008 0:00:00,7.00:00:00)                              #3
```

#1 Create list of schedules for testing
#2 Add one week using 'mapSchedule'
#A Print the new schedule
#3 Schedules moved by one week

We start by creating a list of schedules for testing (#1). One interesting thing to note here is that I omitted the `new` keyword when constructing `DateTime` and `TimeSpan` .NET objects. This is just a syntactical simplification that F# allows when working with simple types like these two.

After creating the list, we iterate over all the schedules. In the next line (#2), we use the `mapSchedule` function to move each schedule by one week. The change in the date is specified as a lambda function that returns a new `DateTime` object. Of course, you could implement more complicated logic to perform different rescheduling inside this function. The original schedule is passed as the last argument using the pipelining operator. As you can see (#3) the operation changed the date of the `Once` schedule and the first occurrence of the schedule represented using `Repeatedly` option.

6.3.1 Processing a list of schedules

In the previous example we used an imperative `for` loop, because we just wanted to print the new schedule. If you wanted to create a list containing the new schedules, you could use `List.map` function and write something like this:

```
let newSchedules =
   List.map (fun s ->
         s |> mapSchedule (fun d -> d.AddDays(7.0)) #A
      ) schedules
```

#A Calculate new schedule

The first argument of the `List.map` function is another function that is used to obtain a new value using the original schedule. In this example, we calculate a new schedule called ns and return it as the result of the function. However, the previous code can be simplified by using pipelining and partial function application like this:

```
    let newSchedules =
        schedules |> List.map (mapSchedule (fun d -> d.AddDays(7.0)))
```

When we specify just the first argument (a function for calculating the date) to the `mapSchedule` function, we get a function of type `Schedule -> Schedule`. This is exactly what the `List.map` operation expects as the first argument, so we don't have to write lambda function explicitly. This example shows another reason why many higher order functions take the original value as the last argument. That way we can use both pipelining *and* partial application when processing a list of values.

Another option would be to use *sequence expressions* that are similarly succinct, but probably more readable for a newcomer. We'll look at sequence expressions in chapter 12, but now let's see how we could implement the same functionality in C#.

6.3.2 Processing schedules in C#

In C# we'll build a `MapSchedule` method which should be similar to the mapSchedule function in F#. Again, this will have two parameters: a function for calculating the new date, and the original schedule. As we're working with alternative values in C#, we'll use a `switch` block and the `Tag` property as shown in the previous chapter. Listing 6.9 shows the complete implementation.

Listing 6.9 Map operation for schedule type (C#)

```
    public static Schedule MapSchedule
         (this Schedule schedule, Func<DateTime, DateTime> calcDate) {    #1
       switch(schedule.Tag) {
          case ScheduleType.Never:
             return new Never();
          case ScheduleType.Once:
             var os = (Once)schedule;
             return new Once(calcDate(os.When));                          #A
          case ScheduleType.Repeatedly:
             var rs = (Repeatedly)schedule;
             return new Repeatedly(calcDate(rs.First), rs.Periodicity);   #B
          default:
             throw new InvalidOperationException();                       #C
       }
    }
```

#1 Extension method using 'this' modifier
#A Calculate new date
#B Calculate new date for the first occurrence
#C Unreachable code - no other option!

The method simply provides a branch for each of the possible representations and returns a new value in each branch. When the option carries a date that can be processed (`Once` and `Repeatedly`), it first casts the argument to the appropriate type and then uses `calcDate` argument to calculate the new date.

The method is implemented as an extension method inside a `ScheduleUtils` class (for simplicity, the listing doesn't include the class declaration). This means that we can call it as a static method, but also more readably using dot-notation on any instance of the `Schedule` class. The following snippet shows how we can move every schedule in a list by one week:

```
schedules.Select(s =>
    s.MapSchedule(dt => dt.AddDays(7.0)) )
```

This is similar to our earlier F# code. We're using the LINQ `Select` method (instead of the `List.map` function) to calculate a new schedule for each schedule in the original list. Inside a lambda function, we call `MapSchedule` on the original schedule, passing it an operation that calculates the new date.

When we have several similar operations that we need to perform with the value, it would be tedious to use the schedule type directly, because we'd have to provide the same unwrapping and wrapping code multiple times for each of the operations. In this section, we've seen that a well designed higher order function can simplify working with values quite a lot. Now, let's look at writing higher order functions for another alternative value that we introduced in the previous chapter: the option type.

6.4 Working with the option type

One of the most important alternative values in F# is the option type. To recap what we've seen in the previous chapter, it gives us a safe way to represent the fact that value may be missing. This safety means that we have to explicitly pattern match on option whenever we want to perform some operation with the actual value. In this section, we'll learn about two useful functions for working with the option type.

F# LIBRARY FUNCTIONS

The functions we saw earlier for working with tuples aren't part of the F# library, because they are extremely simple and using tuples explicitly is usually easy enough. However, the functions we'll see in this section for working with the option type are part of the standard F# library.

First of all, let's quickly look at an example that demonstrates why we need higher order operations for working with the option type. We'll use the `readInput` function from the previous chapter, which reads user input from the console and returns a value of type `option<int>`. When the user enters a valid number, it returns `Some(n)`; otherwise it returns `None`. Listing 6.10 shows how we could implement a function that reads two numbers and returns a sum of them or `None` when either of the inputs wasn't a valid number.

Listing 6.10 Adding two options using pattern matching (F#)

```
let readAndAdd1() =
   match (readInput()) with
```

```
      | None    -> None
      | Some(n) ->                        #A
         match (readInput()) with
          | None    -> None
          | Some(m) ->                    #B
             Some(n + m)                  #C
```

#A Extract value from the first input
#B Extract value from the second input
#C Add numbers and return the result

The function calls `readInput` to read the first input, extracts the value using pattern matching and repeats this for the second input. When both of the inputs are correct, it adds them and returns `Some`, in all other branches it returns `None`. Unfortunately, the explicit use of pattern matching makes the code rather long. Let's now look at two operations that will help us to rewrite the code more succinctly.

6.4.1 Using the map function

I'll first introduce both of the operations and first show you how to use them from F#, where they are already available in the F# library. Later we'll also look at their implementation and how we can use them from C#. As we've already seen, the best way to understand what a function does in F# is often to understand its type signature. Let's first look at `Option.map`:

```
> Option.map;;
val it : (('a -> 'b) -> 'a option -> 'b option) = (...)
```

I said that map operations usually apply a given function to values carried by the data type and wrap the result in the same structure. For the option type, this means that when the value is `Some`, the function given as the first argument (`'a -> 'b`) will be applied to a value carried by the second argument (`'a option`) and the result of type `'b` will be wrapped inside an option type, so the overall result has type `'b option`. When the original option type doesn't carry a value, the `map` function will simply return `None`.

We can use this function instead of the nested match. When reading the second input, we want to 'map' the carried value to a new value by adding the first number:

```
match (readInput()) with
| None        -> None
| Some(first) -> readInput() |> Option.map (fun second -> first + second
```

On the second line we already have a value from the first number entered by the user. We then use `readInput()` to read the second option value from the console. Using `Option.map`, we project the value into a new option value, which is then returned as the result. The lambda function used as an argument adds the first value to a number carried by the option value (if there is one).

6.4.2 Using the bind function

As a next step, we'd like to eliminate the outer pattern matching. Doing this using `Option.map` isn't possible, because this function always turns input value `None` into output value `None` and input value `Some` into output `Some` carrying another value.

However, in the case above, we want to do something quite different. Even when the input value is `Some`, we still want to be able to return `None` when we fail to read the second input. This means that the type of the lambda function we specify as an argument shouldn't be `'T1 -> 'T2`, but rather `'T1 -> option<'T2>`.

The operation like this is called *bind* in the functional programming terminology and it is provided by the standard F# library. Let's now take a look at the signature and at the specification of what this function actually does:

```
> Option.bind;;
val it : (('T1 -> option<'T2>) -> option<'T1> -> option<'T2>) = (...)
```

The difference in the type signature of *bind* and *map* is only in the type of the function parameter as we discussed it in the previous paragraph. Understanding is a behavior of a function just using the type is a very important skill of functional programmers. In this case, the type gives us a very good clue of what the function does if we assume that it behaves reasonably. We can analyze all the cases to infer the specification of the function's behavior:

28) When the input value is `None`, *bind* cannot run the provided function, because it cannot safely get value of type `'T1`, and so it immediately returns `None`.

29) When the input value is `Some` carrying some value `x` of type `'T1`, *bind* can call the provided function with `x` as an argument. It could of course still return `None`, but a more reasonable behavior is to call the function when possible. Now, there are two different cases what the function given as the argument can return:

30) If the function returns `None`, the *bind* operation doesn't have any value of type `'T2`, so it has to return `None` as the overall result.

31) If the function returns `Some(y)`, then *bind* has a value `y` of type `'T2` and only in this case it can return `Some` as the result, so the result in this case is `Some(y)`.

Using *bind* we can now rewrite the outer pattern matching, because it gives us a way to return undefined value (`None`) even when we successfully read the first input. Listing 6.11 shows the final version of `readAndAdd`.

Listing 6.11 Adding two options using bind and map (F#)

```
let readAndAdd2() =
   readInput() |> Option.bind (fun num ->     #1
      readInput() |> Option.map ((+) num) )   #2
```

#1 Process first input using 'bind'
#2 Process second input using 'map'

After reading the first input, we pass it to the bind operation (#1), which executes the given lambda function only when the input contains a value. Inside this lambda function, we read the second input and project it into a result value (#2). The operation used for projection just adds the first input to the value. In this listing, we've written it using the plus operator and partial application instead of specifying the lambda function explicitly. If you compare the code with listing 6.10, you can see that it is definitely more concise. Let's now analyze how it works in some more detail.

6.4.3 Evaluating the example step-by-step

It can take some time to become confident with higher order functions like these, especially when they are nested. We're going to examine how the code from the previous listing works by tracing how it runs for a few sample inputs. Moving from the abstract question of "what does this code do in the general case?" to the concrete question of "what does this code in this particular situation?" can often help to clarify matters.

First let's see what happens if we enter an invalid value as the first input. In that case, the first value returned from `readInput()` will be `None`. To see what happens, we can use computation by calculation and show how the program evaluates step-by-step. You can see how the calculation proceeds in listing 6.12.

Listing 6.12 Evaluation when the first input is invalid

```
[CA] Read the first input from the user:               #A
   None |> Option.bind (fun num ->
      readInput() |> Option.map ((+) num) )

[CA] Evaluate the "Option.bind" call:                  #B
   None
```

#A Replace the 'readInput()' call with the returned value
#B Lambda function isn't called and None is returned

In the first step, we simply replace the call with the `None` value that the function returns when we enter some invalid input (such as empty string). The second step is more interesting. Here, the `Option.bind` function gets `None` as its second argument. However, `None` doesn't carry any number, so `bind` cannot call the specified lambda function and the only thing it can do is to immediately return `None`.

Now, how would the function behave if we entered "20" as the first input? Obviously, there will be two different options - one when the second input is correct and one when it is invalid. Listing 6.13 shows what happens if the second input is "22".

Listing 6.13 Evaluation when both inputs are valid

```
[CA] Read the first input from the user:               #A
   Some(20) |> Option.bind (fun num ->
      readInput() |> Option.map ((+) num) )

[CA] Evaluate the "Option.bind" call:                  #1
   readInput() |> Option.map ((+) 20)

[CA] Read the second input from the user:              #B
   Some(22) |> Option.map ((+) 20)

[CA] Evaluate the "Option.map" call:                   #2
   Some( (+) 20 22 )

[CA] Evaluate the "+" operator:                        #C
   Some(42)
```

#1 Replace 'readInput()' with the first input
#2 'Option.bind' calls the lambda function and replaces 'n' with 20
#3 Read the second input value
#4 'Option.map' calls the provided function and wraps the result in 'Some'
#5 Calculate 20 + 22 and wrap the result

The first step is similar to the previous case, but this time, we call `Option.bind` with `Some(20)` as an argument. This option value carries a number that can be passed as the `num` argument to the lambda function we provided. `Option.bind` simply returns the result that it gets from this function, so the result in the next step will be the body of this function (#1). We also replace all occurrences of `num` with the actual value, which is 20.

We then read the next input value with `readInput()` which returns `Some(22)`. Having replaced `readInput()` with `Some(22)` we can evaluate the `Option.map` function. This operation evaluates the function it gets as an argument and in addition wraps the result in the `Some` discriminator, so our next step (#2) shows that we need to calculate the addition next and wrap the result in `Some`. After calculating the addition, we finally get the result, which is `Some(42)`.

After following this step-by-step explanation you should have pretty good idea how `Option.bind` and `Option.map` work. Equipped with this information, we can look at the implementation of these two operations in both F# and C#.

6.4.4 Implementing operations for the option type

The implementations of both `bind` and `map` have a similar structure, because they are both higher order functions that pattern match against an option value. We'll take a look at both F# and C# implementations, which is a good example of how to encode functional ideas in C#. Let's start with listing 6.14, which shows the implementation of map operation.

Listing 6.14 Implementing the map in F# and C#

```
> let map f opt =
     match opt with
     | Some(v) ->
Some(f(v))
     | None -> None
  ;;
val map :
   ('a -> 'b) ->
   'a option -> 'b option
```

```
static Option<R> Map<T, R>(this
Option<T>
      opt, Func<T, R> f) {
   T v;
   if (opt.MatchSome(out v))
      return Option.Some(f(v));
   else
      return Option.None<R>();
}
```

The implementation first examines the option value given as an argument. When the value is `None`, it immediately returns `None` as the result. Note that we cannot just return the `None` value that we got as an argument, because the types may be different. In the C# version the type of the result is `Option<R>`, but the type of the argument is `Option<T>`.

When the value of the argument matches the discriminated union case `Some`, we get the value of type `T` and use the provided function (or `Func` delegate) to project it into a value of type R. Since the value returned from the operation should have a type Option<R>, we need to wrap this value using the Some constructor again.

The source code of map and bind operations is quite similar, but there are some important differences. Let's now look at the second couple of operations in listing 6.15.

Listing 6.15 Implementing the bind operation in F# and C#

```
> let bind f opt =
     match opt with
     | Some(v) -> f(v)
     | None -> None
  ;;
val bind :
   ('a -> 'b option) ->
   'a option -> 'b option
```

```
static Option<R> Bind<T, R>(this
Option<T>
      opt, Func<T, Option<R>> f) {
   T v;
   if (opt.MatchSome(out v))
      return f(v);
   else
      return Option.None<R>();
}
```

Can we fit these two listings on a single page (so that the reader can compare them without turning pages)?

The bind operation starts similarly by pattern matching on the option value given as the argument. When the option value is None, it immediately returns None just like in the previous case. The difference is in the case when the option carries some actual value. We again apply the function that we got as an argument, but this time we don't need to wrap the result inside Some constructor. This is because the value returned from the function is already option and as you can see from the type signature, it has exactly the type that we want to return. This means that even in the Some case, the bind operation can still return None, depending on the function provided by the user.

As usual, the F# version takes the original value as a last argument to enable pipelining and partial application, while the C# version is an extension method. Let's now look how to rewrite the previous example in C# using the newly created methods.

USING THE OPTION TYPE IN C#

Extension methods give us a way to write the code that uses Bind and Map in a fluent manner. As the number of parentheses can be confusing, note that the call to Map is nested inside a lambda function that we give as an argument to Bind:

```
Option<int> ReadAndAdd() {
   return ReadInput().Bind(n =>
      ReadInput().Map(m => m + n));
}
```

In C# the difference between using higher order functions and working with option types explicitly is even more significant. This is because C# doesn't directly support types like discriminated unions, but if we supply our types with appropriate processing functions, the code becomes readable. This is the important point to keep in mind when writing functional-style programs in C#: some of the low-level constructs may feel unnatural, but thanks to lambda functions, we can write elegant functional code in C# too.

So far, we've seen how to use higher order functions to work with multiple values and alternative values. The last kind of value we talked about in the previous chapter was function. In the next section, we'll see that we can write surprisingly useful higher order functions for working with function values as well.

6.5 Working with functions

All the higher order functions we've discussed so far in this chapter have had a similar structure. They had two parameters: one was a value to be processed and the second was a function that specified how to process the value. When working with functions, the "value" parameter will be also a function, so our higher order functions will take two functions as arguments.

6.5.1 Function composition

Probably the most important operation for working with functions is function composition. Let's start by looking at an example where this will be very helpful. We'll use the example where we stored a name and population using a tuple. In listing 6.16 we create a function to determine the status of a settlement based on the size of the population. We also test it by determining the status of several places stored in a list.

Listing 6.16 Working with city information (F# interactive)

```
> let places = [ ("Grantchester", 552);
                 ("Cambridge", 117900);
                 ("Prague", 1188126); ];;  #A
val places : (string * int) list

> let statusByPopulation(population) =                          #B
      match population with
      | n when n > 1000000 -> "City"
      | n when n >    5000 -> "Town"
      | _                  -> "Village";;
val statusByPopulation : int -> string

> places |> List.map (fun (_, population) ->                    #1
      statusByPopulation(population));;                         #2
val it : string list = ["Village"; "Town"; "City"]
```

#A Create a list with test data
#B Returns status based on the population
#1 Iterate over settlements and read population information
#2 Calculate the status

The first parts of listing 6.16 (creating a list of test data and the declaration of the `statusByPopulation` function) are quite straightforward. The interesting bit comes in the last few lines. We want to obtain the status of each settlement using `List.map`. To do this we pass it a lambda function as an argument. The lambda function first extracts the second element from the tuple (#1) and then calls our `statusByPopulation` function (#2).

The code works well but it can be written more elegantly. The key idea is that we just need to perform two operations in sequence. We first need to access the second element from a tuple and then perform the calculation using the returned value. Since the first operation can be done using `snd` function, we just need to compose these two functions. In F#, this can be written using function composition operator (>>) like this:

```
snd >> statusByPopulation
```

The result of this operation is a function that takes a tuple, reads its second element (which has to be an integer) and calculates the status based on this number. We can understand how the functions are composed from table 6.1, which shows their type signatures.

Function value	Type
`Snd`	`('a * 'b) -> 'b`
`snd` (after specification)	`('a * int) -> int`
`statusByPopulation`	`int -> string`
`snd >> statusByPopulation`	`('a * int) -> string`

Table 6.1 Type signatures of 'snd', 'statusByPopulation' and a function obtained by composing these two functions using >> operator.

On the second line, the table shows a specific type of the `snd` function after the compiler figures out that the second element of the tuple has to be an integer. We can get this type if we substitute type parameter `'b` from the first row with a type `int`. Now we have two functions that can be composed, because the return type on the second row is same as the input type on the third row. Using composition, we join the functions together and get a function that calls the first one and passes the result of this call as an input to the second one. The resulting function has the same input type as the function on the second row and the same return type as the function on the third row. Listing 6.17 shows how we can rewrite the original code using function composition.

Listing 6.17 Using function composition operator (F# interactive)

```
> places |> List.map (fun x -> (snd >> statusByPopulation) x);;  #1
val it : string list = ["Village"; "Town"; "City"]
```

```
> places |> List.map (snd >> statusByPopulation);;              #2
val it : string list = ["Village"; "Town"; "City"]
```

#1 Calling composed function explicitly
#2 Using composed function as an argument

On the first line (#1), we call the composed function explicitly by giving it the tuple containing city name and population as an argument. This is just to demonstrate that a result of composition is a function that can be called. However, the reason for using function composition is that we can use the composed function as an argument to other functions. In this case, the composed function takes a tuple and returns a string, so we can immediately use it as an argument to `List.map` to get a list of the statuses of the sample settlements.

The implementation of the function composition operator is remarkably simple. Here's how we could define it if it didn't already exist in the F# library:

```
> let (>>) f g x = g(f(x))
val (>>) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

In this declaration, the operator has three parameters. However, when we were working with it earlier, we only specified only the first two parameters (the functions to be composed). We'll get better insight into how it works by looking at the two possible interpretations of the type signature in figure 6.2.

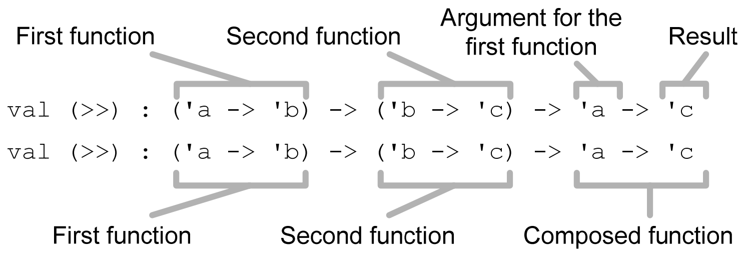

First function
Second function
Argument for the first function
Result
val (>>) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
val (>>) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
First function
Second function
Composed function

Figure 6.2 Type signature of the function composition operator. If we specify three arguments (annotations above), it returns the result of calling them in sequence. If we specify only two arguments, it returns a composed function (annotations below).

The operator behaves as function composition thanks to the partial application. If we specify just the first two arguments, the result is a composed function. When the operator receives the third argument, it uses it to call the first function and then calls the second function using the result. Clearly, specifying all three arguments to it isn't typically very useful - we could just call the functions directly, without using the operator!

Now that we've seen how function composition works in F#, let's look at what it might look like in C#.

FUNCTION COMPOSITION IN C#

Function composition in C# is possible, but it has only a very limited use. This is partly because partial application can't be used as easily in C#, but more importantly because most of operations are written as members instead of functions. However, we can at least demonstrate the same idea in C#. Listing 6.18 shows an implementation of the `Compose` method as well as an example of using it.

Listing 6.18 Implementing and using Compose method (C#)

```
static Func<A, C> Compose<A, B, C>(this Func<A, B> f, Func<B, C> g) {    #1
   return (x) => g(f(x));                                                  #2
}

// Using function composition in C#
Func<double, double> square = (n) => n * n;                               #A
Func<double, string> fmtnum = (n) => n.ToString("E");                     #A

var data = new double[] { 1.1, 2.2, 3.3 };
var sqrs = data.Select(square.Compose(fmtnum));                           #3

// Prints: "1.210000E+000"; "4.840000E+000"; "1.089000E+001"
foreach (var s in sqrs) Console.Write(s);
```

#1 Returns a function value
#2 Construct the composed function using lambda function
#A Two functions that can be composed
#3 Using the composed function

Function composition is implemented as an extension method for the `Func<T, R>` delegate, so we can call it on function values that take a single argument using dot-notation. In F# it was written as a function with three parameters, even though it's usually used just with two arguments. In C# we have to implement it as a method with two arguments that returns a `Func` delegate explicitly (#1). We construct a lambda function that takes an argument and calls functions that we're composing (#2), and then return this function as a delegate.

To test the method, we create two functions that we want to compose. We use the composed function when processing numbers in a collection using `Select`. Instead of using it with an explicit lambda function we just call `Compose` to create a composed function value that we can use as an argument.

Over the last few sections, we've seen that many of the useful processing functions are generic, some of them having even three type parameters. Writing functions like this in F# has been very easy because we haven't had to write the types explicitly: type inference has figured out the types automatically. It's time to take a closer look at how this mechanism works.

6.6 Type inference

We have already talked a little bit about type inference for values. We've seen it in C# 3.0 with the `var` keyword and in F# with let bindings. We'll start this section with another aspect that is shared by both C# and F#. When calling a generic method, such as `Option.Some` or `Option.Map` from listing 6.13 in C#, we *can* specify the type arguments explicitly like this:

```
var dt = Option.Some<DateTime>(DateTime.Now);
dt.Map<DateTime, int>(d => d.Year);
```

That's very verbose though, and we've almost never written code in this style in the previous examples, because C# performs type inference for generic method calls. This deduces type arguments automatically, so in the previous example we could have written just `dt.Map(d => d.Year)`.

The exact process of type inference in C# is quite complicated, but it works very well and it usually isn't important to understand it at an intimate level. If you ever really need the details, you can find complete documentation in the C# Language specification [C# Specification]. In general, it deduces types of all values given as an argument, which is always possible and then treats lambda functions specially. From lambda functions, it can obtain return type and also types of the arguments. For example, when calling `Option.Bind`, it needs to deduce the return type of the lambda function, because this is the type of the second type parameter and there is no other hint that could be used. Types of the arguments can be used when they are specified explicitly such as in lambda function `(DateTime d) => d.Year`. Also note that in C# 3.0, the order of parameters doesn't matter.

TYPE INFERENCE FOR FUNCTION CALLS IN F#

Even though it is possible to specify type arguments in F# using angle brackets in a same way as in C#, this is used only very rarely. The reason is that when the compiler cannot infer all the information and needs some aid from the programmer, we can use add type annotation to the particular location where more information is needed. Let's demonstrate this using an example:

```
> Option.map (fun v -> v.Year) (Some(DateTime.Now));;
error FS0072: Lookup on object of indeterminate type.

> Option.map (fun (v:DateTime) -> v.Year) (Some(DateTime.Now));;
val it : int option = Some(2008)
```

Unlike in C#, the order of arguments matters in F#, so the first case fails. The reason is that the F# compiler doesn't know that the value `v` is of type `DateTime` until it reaches the second argument and so it doesn't know whether the `Year` property exists when processing the first argument. To correct this, we added a type annotation in the second case, which specifies the type of the `v` value explicitly. However, this is one more interesting aspect of the pipelining operator: if we use pipelining to write the previous code snippet, we don't need type annotations either:

```
> (Some(DateTime.Now)) |> Option.map (fun v -> v.Year);;
val it : int option = Some(2008)
```

This works because the option value, which contains the `DateTime` value, appears earlier and so it is processed prior to the lambda function. When processing the lambda function, the compiler already knows that the type of `v` has to be `DateTime`, so it can find the `Year` property with no trouble.

So far, we've just looked at the similarities between C# and F#, but type inference goes further in F#. Let's see how the F# compiler can help us when we write higher order functions.

6.6.1 Automatic generalization

We've already implemented several higher order functions in F# in this chapter and we've seen a few side-by-side implementations in F# and C# as well. The interesting fact about the F# implementations is that we didn't need to specify the types at all. This is thanks to *automatic generalization*, which is used when inferring the type of a function declaration. I'll explain how this process works using an implementation of the `Option.bind` function as an example:

```
let bind func value =            #1
   match value with              #2
   | None    -> None             #3
   | Some(a) -> func(a)          #4
```

The type inference process for this function is described step by step below. It starts with the most general possible type and adds constraints as it processes the code, so the listing shows steps that are made while processing the function body.

32) Use the type signature (#1) to infer that `bind` is a function with two arguments and assign a new type parameter to each of the arguments and to the return type:

```
func  : 't1
value : 't2
bind  : 't1 -> 't2 -> 't3
```

33) Use the pattern matching (#2) to infer that `value` is an option type, because it is matched against `Some` and `None` patterns. Use (#3) to infer that the result of `bind` is also an option type, because it can have `None` as a value:

```
func  : 't1
value : option<'t4>
bind  : 't1 -> option<'t4> -> option<'t5>
```

34) Use (#4) to infer that `func` is a function, because we're calling it with a single parameter:

```
func  : ('t6 -> 't7)
value : option<'t4>
bind  : ('t6 -> 't7) -> option<'t4> -> option<'t5>
```

35) From (#4) we know that the parameter to the function has type `'t4` and that the result has the same type as the result of `bind` function, so we add two following constraints:

```
't6 = 't4
```

```
't7 = option<'t5>
```

36) Now, we can replace types `'t6` and `'t7` using the constraints obtained in the previous step:

```
func  : ('t4 -> option<'t5>)
value : option<'t4>
bind  : ('t4 -> option<'t5>) -> option<'t4> -> option<'t5>
```

37) Finally, we rename the type parameters according to the usual F# standards:

```
bind  : ('a -> option<'b>) -> option<'a> -> option<'b>
```

This shows a process; I'm not sure how to format it the best, so use whatever formatting you'll find appropriate. Thanks!

Even though implementing the F# type inference algorithm just using this description would be difficult, it should show you what kind of information F# can use when deducing a type of a higher order function. Probably the most interesting step in the process was deduction of the type of a function (`func`) used as a parameter. This is a very important step, because functions given as parameters represent operations that can be used on values. As we've seen earlier, these are in some sense similar to methods, but thanks to the type inference, writing code like this in F# doesn't require any additional type specification and still makes the code completely type-safe.

After that short interlude about type inference and automatic generalization, we'll get back to writing and using higher order functions. We've discussed them for most of the types from chapter 5, but we're still missing one important functional value. In the next section we're going to look at higher order functions for working with lists.

6.7 Working with lists

We have already talked about lists in chapter 3 where we learned how to process lists explicitly using recursion and pattern matching. We also implemented a functional list type in C#. In the sample application in chapter 4, we used lists in this way, but noted that writing list processing explicitly isn't very practical.

This is a recurring pattern of this chapter, so you probably already know what I'm going to say next. Instead of using pattern matching explicitly in every case, we can use higher order functions for working with lists. We've already seen some functions for working with F# lists such as `List.map` and similar methods for working with C# collections (`Select`). In this section, we'll look at these in some more detail, examining their type signatures and seeing how they can be implemented for a functional list.

6.7.1 Implementing list in F#

Even though we've been working with functional lists in F# and implemented the same functionality in C#, we haven't looked at how we might implement the list type in F#. When we talked about lists earlier, we saw that a list is represented as either a nil value (for an empty list) or a cons cell containing an element and a reference to the rest of the list.

Now, if we look at our gallery of values from the previous chapter, this is exactly like an alternative value with two options. There is one slight wrinkle, however: the list type is recursive, which means that a cons cell contains a value of type list itself. Listing 6.19 shows a type definition that creates a very similar list type to the one in the F# standard library.

Listing 6.19 Definition of a functional list type (F#)

```
> type List<'a> =                                  #1
      | Nil                                        #2
      | Cons of 'a * List<'a>                      #3
type List<'a> = (...)

> let list = Cons(1, Cons(2, Cons(3, Nil)));;  #A
val list : List<'a>
```

#1 The type is generic
#2 Represents an empty list
#3 List with head and a tail
#A Create list containing 1, 2, 3

The type is written as a generic type with a single type parameter (#1). The type parameter represents the type of the values that are stored in the list. Alternatives in F# are represented using discriminated unions and this particular union has two discriminators. The first one (#2) represents an empty list and the second one (#3) is a list with an element (of type `'a`) and a reference to the rest of the list, whose type is written recursively as `List<'a>`.

The last line in the code sample shows how we can create a list with three elements. The first argument to the Cons constructor is always a number and the second argument is a list, which in turn is either constructed using another `Cons` or the `Nil` discriminator. The built-in F# list type is declared in exactly this way. Earlier on we worked with lists using two operators. The `::` operator corresponds to `Cons` in our definition and `[]` represents the same value as `Nil`.

In general, creating a recursive discriminated union type is a very common way to represent program data as we'll see in the next chapter. However, the list type lies somewhere between simple values and complex program data. It can be interpreted in both ways, depending how it is used in the program. We'll also see how recursive unions can also express many of the standard design patterns, but for now let's get back to the higher order functions which make it easier to work with lists.

6.7.2 Understanding type signatures of list functions

As I mentioned earlier, we were already using functions for filtering and projecting lists, but we were using them quite intuitively. In this section, we'll look at their type signature and we'll see how we can deduce what a higher order function does just using this information.

Of course, you can't tell what a function does just by looking at its type in general, but for generic and higher order functions, such as those for working with lists, this is very often possible. As we've seen earlier in this chapter, functions for working with generic values cannot do much with the value alone, because they don't know anything about it. As a result, they usually take a function as an extra argument and use it to work with the value. However, the type of the function gives some clues as to how the result will be used. Let's demonstrate this using type signatures displayed in listing 6.20.

Listing 6.20 Types of functions and methods for working with lists (F# and C#)

```
// F# function signatures
List.map    : ('T -> 'R)    -> 'T list -> 'R list  #1
List.filter : ('T -> bool) -> 'T list -> 'R list  #2

// C# method declarations
List<R> Select<T, R> (List<T>, Func<T, R>)        #1
List<T> Where<T>     (List<T>, Func<T, bool>)     #2
```

#1 Projection
#2 Filtering

Let's look at projection (#1) first. As you can see, the input parameter is a list of values of type T and the result is a list of values of type R. However, the operation doesn't know what R is and so it cannot create values of this type alone. The only way to creating a value of type R is to use a function given as an argument that turns a value of type T into a value of type R. This suggests that the only reasonable way for the operation can work is to iterate over the values in the input list, call the function for each of the values and return a list of results. Indeed, this is exactly what the projection operation does.

It is also worth noting that the types of the input list and output list can differ. In the previous chapter we were adding a number 10 to a list of integers, so in that case, the input list had the same type as the output list. However, we could use a function that created a string from a number as an argument. In this case the input list would be a list of integers and the result will be a list of strings.

The second operation is filtering (#2). In this case, the input and the resulting lists have the same type. The function given as an argument is a predicate that returns true or false for a value of type T, which is the same type as the elements in the input list. This gives us a good hint that the operation probably calls the function for each of the list elements and uses the result to determine whether the element should be copied to the returned list or not.

Working with lists

Let's look at a larger example showing the use of filtering and projection. Both of them are available in the F# library for various collection types, but we'll use lists as we're familiar with them. In C#, these methods are available for any collection implementing `IEnumerable<T>`, so we'll use generic .NET `List<T>` class. Listing 6.21 shows initialization of the data that we'll be working with.

Listing 6.21 Data about settlements (F# and C#)

```
> let places =                  #1
    [
      ("Seattle", 594210);
      ("Prague", 1188126);
      ("New York", 7180000);
      ("Grantchester", 552);
      ("Cambridge", 117900);
    ];;

val places : (string * int) list
```

```
class CityInfo {                         #2
   public CityInfo(string n, int p) {
      Name = n; Population = p;
   }
   /* Properties omitted... */
}

var places = new List<CityInfo>    #3
  { new CityInfo("Seattle", 594210),
    new CityInfo("Prague", 1188126),
    /* more data... */ };
```

In F#, we'll use our usual example - a list with information about city with name and population (#1). Even though we could convert the F# tuple into the `Tuple` class that we've implemented, we're use a more typical C# representation this time. We declare a class `CityInfo` (#2) and use it to create a list containing city information (#3).

In C#, we can work with the data using the `Where` and `Select` methods that are available in .NET 3.5. Both of these are extension methods so we can call them using the usual dot-notation:

```
var names =
    places.Where(city => city.Population > 1000000)
          .Select(city => city.Name);
```

Again, this shows the benefits of using higher order operations. The lambda functions given as arguments specify what the condition for filtering is (in the first case), or the value to return for each city (in the second case). However, this is all we have to specify. We don't need to know the underlying structure of the collection and we're not specifying how the result should be obtained. This is all encapsulated in the higher order operations.

Let's perform the same operation in F#. We want to filter the data set first and then select only the name of the city. We can do this by calling `List.filter` and using the result as the last argument to the `List.map` function. As you can see, this looks quite ugly and hard to read:

```
let names =
    List.map fst
         (List.filter (fun (_, pop) -> 1000000 < pop) places)
```

Of course, F# can do better than this. The previous C# version was elegant because we could write the operations in the same order in which they are performed (filtering first, projection second) and we could write each of them on a single line. In F#, we can get the same code layout using pipelining:

```
let names =
    places |> List.filter (fun (_, pop) -> 1000000 < pop)
           |> List.map fst
```

In this case, the pipelining operator first passes the value on the left side (`places`) to the filtering function on the right side. In the next step, the result of the first operation is passed to the next operation (here projection). Even though we've been using this operator for quite some time already, this example finally shows why it is called "pipelining". The data elements are processed in sequence as they go through the "pipe" and the pipe is created by linking several operations using the pipelining operator.

Note that sometimes the order of operations is important and sometimes not. In this case we have to perform the filtering first. If we did the projection in the first step, we'd obtain a list containing only city names and we wouldn't have the information about population, which is needed to perform the filtering.

C# 3.0 queries and F# sequence expressions

You've probably already seen examples of data queries written in C# using *query expressions*. Using this feature, our previous code would look like this:

```
var names = from c in places
            where 1000000 < p.Population
            select p.Name
```

This is often demonstrated as a key new feature, but it wouldn't exist without the underlying machinery such as lambda functions and higher order operations. We've focused on using these explicitly, because when you learn to use them explicitly, you can use a similar functional approach for working with any data and not just collections.

However the simplified syntax is quite useful and a similar feature called *sequence expressions* is available in F# too. We'll talk about this later in chapter 12, but just for the curious, here is the same query written in F#:

```
let names =
   seq { for (name, pop) in places do
         if (1000000 < pop) then yield name }
```

It looks almost like ordinary code enclosed in a block and marked with the word `seq`. This is the intention, because in F#, it is a more general language construct and it can be used for working with other values too. In chapter 12 we'll see how to use it when working with option values, but we'll also see how C# query expressions can sometimes be used for similar purposes.

Having looked at how we can use two higher order list processing functions and seen how useful they are, let's take a deeper look at a third such function, and implement it ourselves.

6.7.3 Implementing list functions

Instead of showing how to implement the functions for filtering and projection which we've just seen, we'll look at a function that we started creating in chapter 3. Since all list processing functions have a very similar structure, you'll probably be able to implement any of the others after looking at the following example.

In chapter 3, we wrote a function that could either sum or multiply all elements in a list. We later realized that it is more useful than it first appeared: we saw that it could also be used to find the minimum or maximum elements as well. However, we hadn't covered generics at that point, so the function worked only with integers. In listing 6.22, we look at the same function without the type annotations which originally restricted automatic generalization.

Listing 6.22 Generic list aggregation (F# interactive)

```
> let rec foldLeft f init list =
     match list with
     | [] -> init                                                #A
     | hd::tl ->
        let rem = foldLeft f init tl                             #B
        f rem hd                                                 #C
  ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a          #1
```

#A Return initial value
#B Recursively process the tail
#C Aggregate using the given function
#1 Type signature

The implementation is the same as in chapter 3, but as we removed type annotations, the inferred signature is more general (#1). The function now takes a list with values of type `'b` and the value produced by aggregation can have a different type (type parameter `'a`). The processing function takes the current aggregation result (of type 'a) and an element from the list (`'b`) and returns a new aggregated result.

As we'll see very soon, the use of generics makes the aggregation far more useful. It is also available in the F# library under a name `fold_left` and the version that works with the immutable F# list type is located in the `List` module. The following snippet shows our original use from chapter 3, where we multiplied all the elements in a list together:

```
> [ 1 .. 5 ] |> List.fold_left (*) 5
val it : int = 120
```

As we're working with generic function, the compiler had to infer the types for the type parameters first. In this case, we're working with a list of integers, so parameter `'b` is `int`. The result is also an integer, so `'a` is `int` too. Listing 6.23 shows some other interesting examples using `fold_left`.

Listing 6.23 Examples of using fold_left (F# interactive)

```
> places |> List.fold_left (fun sum (_, pop) -> sum + pop) 0;;    #1
val it : int = 9080788

> places |> List.fold_left (fun s (n, _) -> s + n + ", ") "";;    #2
val it : string =
  "Seattle, Prague, New York, Grantchester, Cambridge, "

> places
    |> List.fold_left (fun (b,str) (name, _) ->                   #3
          let n = if b then name.PadRight(20) else name + "\n"
          (not b, str+n))
```

```
        ) (true, "")                                                    #A
     |> snd                                                             #4
     |> printfn "%s";;                                                  #B
Seattle              Prague
New York             Grantchester
Cambridge
```

#1 Sum population into 'int'
#2 Format into a 'string'
#3 Aggregation using 'bool * string' value
#A Initial tuple value
#4 Drop the helper element from a tuple
#B Print the formatted string

In all the examples, we're working with our collection of city information, so the type of the list is always the same. This means that the actual type of parameter `'b` is always `(string * int)` tuple. However, the result of aggregation differs. In the first case (#1), we're just summing population, so the type of the result is `int`. In the second example (#2), we want to build a string with names of the cities, so we start the aggregation with an empty string. The lambda function used as the first argument appends the name of the currently processed city and a separator.

In the last example (#3) we implement a version with improved formatting–it writes the city names in two columns. This means that the lambda function performs two alternating operations. In the first case, it pads the name with spaces (to fill the first column) and in the second case it just adds a newline character (to end the row). This is done using a temporary value of type `bool`, which is initially set to `true` and then inverted in every iteration. The aggregation value contains this alternating temporary value and the resulting string, so at the end, we need to drop the temporary value from the tuple (#4).

IMPLEMENTING FOLD IN C#

An operation with the same behavior as `fold_left` is available in the .NET library as well, although it has the name `Aggregate`. As usual, it is available as an extension method working on any collection type and we can use it in the same way as the F# function. Let's rewrite the last example from listing 6.21 in C# 3.0, where we used a tuple to store the information during the aggregation. When we were talking about tuples in the previous chapters, I mentioned that C# 3.0 anonymous types can be sometimes used for the same purpose. This is an example of where they're a really good fit:

```
var res =
   places.Aggregate(new { StartOfLine = true, Result = "" },
   (r, pl) => {
      var n = r. StartOfLine ? pl.Name.PadRight(20) : (pl.Name + "\n");
      return new { StartOfLine = !r.StartOfLine, Result = r.Result + n };
    }).Result;
```

In C#, the initial value is specified as the first argument. We create an anonymous type with properties `StartOfLine` (used as a temporary value) and `Result`, which stores the concatenated string. The lambda function used as the second argument does the same thing as in our previous F# example, but returns the result again as an anonymous type, with the same structure as the initial value. To make the code more efficient, we could also use the

StringBuilder class instead of concatenating strings, but I wanted to show the simplest possible example. Now that we know how to use the function in C#, we should also look how it is implemented. In listing 6.24 you can see two implementations. One is a typical functional implementation for the functional list from chapter 3 and the other is an imperative implementation for the generic .NET List type, which is in principle the same as the Aggregate extension method in .NET library.

Listing 6.24 Functional and imperative implementation of FoldLeft (C#)

```
R FoldLeft<T, R>                           R FoldLeft<T, R>
    (this FuncList<T> ls,        #1            (this List<T> ls,            #2
     Func<R, T, R> f, R init)    #1             Func<R, T, R> f, R init)    #2
{                                          {
  if (ls.IsEmpty) return init;   #3          R temp = init;                 #5
  else return f(                             foreach(var el in ls)
    ls.Tail.FoldLeft(f, init),   #4            temp = f(temp, el);          #6
    ls.Head);                                return temp;
}                                          }
```

#1, #2 The signature of both methods is the same and it corresponds to the earlier declaration in F#, although we have to write the type parameters explicitly. Also note that the list is used as the first parameter and both methods are implemented as extension methods
#3, #4 In the functional version, we have two branches - one to process the empty list case and another to recursively processes a cons cell and aggregate the result using the 'f' parameter
#5, #6 The imperative version declares a local mutable value to store the current result during the aggregation. When processing an element, the new value is calculated using the 'f' parameter

Annotations below the code with cueballs on the left.

As I've already mentioned, implementing the other operations is quite a similar process. In the functional version of map or filter, you'd return an empty list in (#3) and in the imperative version, you'd use mutable list as a temporary value (#5). The other change would be on lines (#4) and (#6). When performing a projection, we'd just call the given function, while for filtering we'd decide whether to append the current element or not.

To conclude our discussion of higher order functions, I'd like to highlight a few interesting relationships between the functions that we've used for manipulating lists and the functions available for working with option values.

6.8 Common processing language

We've seen a few recurring patterns over the course of this chapter, such as an operation called "map" which was available for both option values and lists. Actually, we also used it when we were working with tuples and implemented the mapFirst and mapSecond functions.

It turns out that many different values share a similar set of processing functions, so it makes sense to think about these operations as a common language. However, the name of the operation can vary for different values: similarities in type signatures are often better clues than similarities in names. Listing 6.23 shows the types of the `map` and `filter` functions for several types, including a function `Option.filter` that we haven't discussed yet.

Listing 6.23 Signatures of filter and map functions (F#)

```
mapFirst      : ('a -> 'b) -> 'a * 'c   -> 'b * 'c
List.map      : ('a -> 'b) -> 'a list   -> 'b list
Option.map    : ('a -> 'b) -> 'a option -> 'b option

List.filter   : ('a -> bool) -> 'a list   -> 'a list
Option.filter : ('a -> bool) -> 'a option -> 'a option
```

The map operation can perform the function given as the first argument on any elements that are somehow enclosed in the composed value. For tuples, it is used exactly once; for an option value it can be called never or once; for a list it is called for each element in the list. In this light, an option value can be viewed as a list containing zero or one element.

This also explains what the new `Option.filter` function does. For an option value with no elements it returns `None;` for an option with a single value it tests whether it matches the predicate and returns either `Some` or `None` depending on the result. Let's demonstrate this using an example that filters option values containing even numbers:

```
> Some(5) |> Option.filter (fun n -> n%2 = 0);;
val it : int option = None
```

If we use the analogy between lists and options then this code filters a list containing one value and the result is an empty list. The analogy can work the other way round as well - we've already seen the bind operation for options, and we can apply the same concept to lists.

6.8.1 The bind operation for lists

We've only discussed the bind operation for option values, but as we'll see in chapter 12, it is an extremely important functional operation in general. Listing 6.24 shows the type signature of the bind operation for option values and also what it would look like if we defined it for lists.

Listing 6.24 Signatures of bind operations (F#)

```
Option.bind :  ('a -> 'b option) -> 'a option -> 'b option
List.bind   :  ('a -> 'b list)   -> 'a list   -> 'b list
```

The function `List.bind` is available in the F# library under a different name, so let's try to figure out what it does, just using the type signature. The input is a list and for each element, it can obtain a list with values of some other type. A list of this type is also returned as a result from the bind operation.

In practice, this means that the operation calls the given function for each element and concatenates the lists returned from this function. The name that F# library uses reflects this use, so the function is called `List.map_concat`. We can use this function for example to get a list of all files from a given list of directories. Note that a single directory usually contains a list of files. Listing 6.25 shows how we can list all source files for this chapter.

Listing 6.25 Listing files using map_concat (F# interactive)

```
> open System.IO;;
> let directories =
     [ "C:\Source\Chapter06\Chapter06_CSharp";
       "C:\Source\Chapter06\Chapter06_FSharp";
       "C:\Source\Chapter06\FunctionalCSharp" ];;
val directories : string list

> directories |> List.map_concat (fun d ->
     Directory.GetFiles(d)                                   #A
     |> List.of_seq                                          #A
     |> List.map Path.GetFileName );;                        #A
val it : string list =
   [ "Chapter06_CSharp.csproj"; "Program.cs"; "Chapter06_FSharp.fsproj";
     "Script.fsx"; "FunctionalCSharp.csproj"; "List.cs";
     "Option.cs"; "Tuple.cs" ]
```

#A Get list of file names for the given directory

The `map_concat` operation calls the given lambda function for each of the directory in the input list. The lambda function then gets all files from that directory, converts them from an array into a list and uses `List.map` to get the file name from the full path. The results are then collected into a single list that is returned as the overall result. You probably won't be surprised to hear that this operation is also available in .NET 3.5, where it's represented by the `SelectMany` method. This is the method used when you specify multiple `from` clauses in a C# 3.0 query expression.

6.9 Summary

This chapter together with chapter 5 discussed functional values. As we saw in the previous chapter, values are important for controlling the flow of the program and they allow us to write code in a functional way: that is, composing it from functions that take values as an argument and return values as the result. In this chapter we've seen a more convenient way for working with values. Instead of directly using the structure of the value, we used a set of higher order functions that are defined in the F# library. We've seen how they are implemented and also how we can implement similar functionality for our own types.

In particular, we talked about functions that allowed us to perform an operation on the values carried by standard F# types such as tuples and option types, and also our type for representing schedules. We've learned how to construct a function from two functions using function composition and we've seen how all these features, together with partial application and the pipelining operator, can be used to write elegant and readable code that works with values.

Finally, we looked at several functions for working with lists and we also observed interesting similarities between some of the higher order functions acting on different types.

For example, we saw that the map operation is useful for many distinct kinds of values and that the bind operation for an option type looks similar to the `map_concat` function for working with lists. We'll talk more about this relationship in chapter 12.

When we started talking about using values in chapter 5, we made a distinction between *local values* and *program data*. In the next chapter, we'll turn our attention to program data, which represent the key information that the program works with. For example, this could be the structure of shapes in a vector graphics editor or the document in a text editor. In this chapter we introduced a convenient way for working with local values and we'll see that same ideas can be used for working with program data as well. We've already taken a step in this direction when we talked about lists, because many programs represent their data as a list of records.

7 Designing data-centric programs

The first thing to do when designing a functional program is to think about the data that the program works with. Since any non-trivial program uses data, this phase is extremely important in the application design. When implementing a program in a functional language, we also begin with the data structures that we'll use in the code and then write operations to manipulate the data as the second step.

This is different to the object-oriented design, where data is encapsulated in the state of the objects; processing is expressed as methods that are part of the objects and interact with other objects involved in the operation. Most of functional programs are data-centric, which means that data is clearly separated from operations and adding a new operation to work with the data is a matter of writing a single function.

DATA-CENTRIC AND BEHAVIOR-CENTRIC PROGRAMS

Even though most functional programs are data-centric, there are some applications and components where we can't just think about the data, because the primary concern is behavior. For example, in an application that allows batch processing of images using filters, the primary data structure would be a list of filters and from a functional point of view, a filter is just a function.

This shows that there are two primary ways of looking at functional code. These approaches are often combined together in different parts of a single application, but we'll talk about them separately. In this chapter, we'll look at data-centric programs and in chapter 8 we'll talk about behavior-centric programs.

The primary aim of this chapter is to teach you how to think about application design in a functional way. We'll demonstrate the ideas in the context of an application which works with simple documents containing text, images and headings. In this chapter, we'll use F# as our primary language. Although we can C# in a functional style, designing the structure of the application in a functional way would be somewhat inconvenient, because functional data structures rely heavily on data types like discriminated unions. However, I'll mention several related object-oriented design patterns and we'll also consider how we would work with immutable types in C#.

USING DIFFERENT DATA REPRESENTATIONS

In functional programming, it is very common to use multiple data structures to represent the same program data. This means that we design different data structures and then write transformations between the various representations. These transformations usually compute additional information about the data.

This has several benefits. First of all, different operations can be more easily implemented using different data representations. You'll see an example in this chapter where we'll work with two representations of documents. In section 7.2, we'll implement a flat data structure, which is suitable for drawing of the document. Later, in section 7.3, we'll add structured representation, which is more appropriate for storing and processing of the document. Moreover, this approach also supports sharing work, because different representations can be developed and maintained to some extent independently by different developers.

We'll start this chapter by talking about one more F# type that is important for representing program data and then we'll turn our attention to the example application.

7.1 Functional data structures

In functional programming, the data that the program manipulates is always stored in data structures. The difference between data structure and objects is that data structure exposes information about its structure (as the name suggests). The structure can be for example a list of some records, a list of alternative values (represented using discriminated unions in F#) or a recursive data structure such as tree. Knowing the structure of the data makes it easier to write code that manipulates with it, but as we'll see in chapter 9, F# also gives us a way to encapsulate the structure, just like in object-oriented programming, when we want to export the F# data structures from a library or make it available to C#. As we mentioned when we talked about functional concepts in chapter 2, these data structures are immutable.

We'll look at two of the most common representations of program data in this chapter. We'll start with a list of records and later use a recursive data structure. We've already used lists in various examples, and in chapter 4 we used a list of tuples to draw a pie chart, where each tuple contained a title and a value. Using tuples is simple, but it's impractical for more complicated data. In this section we'll look at the F# *record* type, which is the one remaining core F# data type left to discuss.

7.1.1 Using the F# record type

A simple description of records is that they are "labeled tuples". They store multiple different elements in a single value, but in addition, each of the elements has a name that can be used to access it. This is in many ways similar to records or `struct` constructs from C or to anonymous types in C#. Unlike anonymous types, records have to be declared in advance. Similarly to anonymous types, records in their basic form contain only properties to hold data; listing 7.1 shows one such declaration to represent a rectangle.

Listing 7.1 Representing rectangle using record type (F# interactive)

```
> type Rect =                                            #1
     { Left   : float32                                  #A
       Top    : float32                                  #A
       Width  : float32                                  #A
       Height : float32 };;                              #A
type Rect = (...)

> let rc = { Left = 10.0f; Top = 10.0f;                  #2
             Width = 100.0f; Height = 200.0f; };;        #2
val rc : Rect                                            #3

> rc.Left + rc.Width;;                                   #B
val it : float32 = 110.0f
```

#1 Declaration of 'Rect' record
#A Elements of the record with name and a type
#2 Creating a record value
#3 Inferred type is 'Rect'
#B Accessing elements using the name

When declaring a record type (#1) we have to specify the types of the elements and their names. In this example, we're using `float32` type, which corresponds to `float` in C# and the .NET `System.Single` type, because we'll need rectangles of this type later. To create a value of an F# record, we simply specify values for all its elements in curly braces (#2). Note that we don't have to write the name of the record type: this is inferred automatically using the names of the elements and as you can see, in our example the compiler correctly inferred that we're creating a value of type `Rect` (#3).

When working with records we'll need to read their elements, but we'll also need to "change" values of the elements - for example when moving the rectangle to the right. However, as a record is a functional data structure and it is immutable, so we'll instead have to create a new record with the modified value. For example, moving a rectangle record to the right could be written like this:

```
let rc2 = { Left = rc.Left + 100.0f; Top = rc.Top;
            Width = rc.Width; Height = rc.Height }
```

Writing all code like this would be awkward, because we'd have to explicitly copy values of all elements stored in the record. In addition, we may eventually need to add a new element to the record declaration, which would break all the existing code. Unsurprisingly, F# lets us express the idea of "copy an existing record with some modifications" in a succinct manner:

```
let rc2 = { rc with Left = rc.Left + 100.0f }
```

Using the `with` keyword, we can specify just a value of the elements that we're going to change and all the remaining elements will be copied automatically. This has exactly the same meaning as the previous code, but it's much more practical.

So far we've seen how to write "primitive" operations on records - but of course we're trying to write code in a functional style, so we really want to be able to manipulate records with functions.

WORKING WITH RECORDS

We'll use the `Rect` type later in this chapter and we'll need two simple functions to work with rectangles. The first function deflates a rectangle by subtracting the specified width and height from all its borders and the second one converts our representation to the `RectangleF` class from `System.Drawing` namespace. You can see both of them in listing 7.2.

Listing 7.2 Functions for working with rectangles (F# interactive)

```
> open System.Drawing;;
> let deflate(rc, wspace, hspace) =
     { Left = rc.Top + wspace                                  #A
       Top = rc.Left + hspace                                  #A
       Width = rc.Width - (2.0f * wspace)                      #A
       Height = rc.Height - (2.0f * hspace) };;                #A
val deflate : (Rect * float32 * float32) -> Rect               #1

> let toRectangleF(rc) =
     RectangleF(rc.Left, rc.Top, rc.Width, rc.Height);;        #B
val toRectangleF : Rect -> RectangleF                          #2

> { Left = 0.0f; Top = 0.0f;
    Width = 100.0f; Height = 100.0f; };;
val it : Rectangle = (...)

> deflate(it, 20.0f, 10.0f);;                                  #3
val it : Rectangle = { Left = 20.0f;  Top = 10.0f;
                       Width = 60.0f; Height = 80.0f;}
```

#A Create and return deflated rectangle
#1 Function signature
#B Return a new instance of 'RectangleF' class
#2 Function signature
#3 Test 'deflate' using rectangle from the previous command

As you can see from the printed type signatures (#1, #2), the F# compiler correctly deduced that the type of the `rc` parameter is of type `Rect`. The compiler uses the names of the elements that are accessed in the function body. However, if we had two record types and used only elements shared by both of them, we'd have to specify the type explicitly. We could use type annotations and write `(rc:Rect)` in the function declaration. As usual when working with F# interactive, we immediately test the function (#3). We didn't use a let binding when creating the value, so later we access it using the automatically created value called `it`.

If we were designing a functional data structure like this one in C# we would of course use classes or occasionally structs. However, F# record types are immutable and as we've seen, they can be easily cloned using the `{ x with ... }` construct. In the next section, we'll briefly look how to design similar type in C#.

7.1.2 Functional data structures in C#

We've already implemented several functional immutable data types in C# such as `FuncList` or `Tuple`. In C#, we do this by writing a class in a particular way. Most importantly all its properties have to be immutable. This can be done either by using `readonly` field, or by declaring a property which has a private setter and is set only in the constructor of the class. We use the second approach in listing 7.3.

Listing 7.3 Immutable 'Rect' type (C#)

```
public sealed class Rect {
   public float Left   { get; private set; }                        #A
   public float Top    { get; private set; }                        #A
   public float Width  { get; private set; }                        #A
   public float Height { get; private set; }                        #A

   public Rect(float left, float top, float width, float height) {   #B
      Left = left; Top = top; Width = width; Height = height;
   }

   public Rect WithLeft(float left) {                               #1
      return new Rect(left, this.Top, this.Width, this.Height);     #C
   }
   // Similarly: WithTop, WithWidth and WithHeight                  #2
}
```

#A Readonly properties of the type
#B Construct the value
#1 Returns 'Rect' with modified 'Left' property
#C Create a copy of the object
#2 'With' methods for other properties (Omitted)

The class contains the usual declarations of read-only properties using the C# 3.0 automatic properties feature and a constructor that initializes them. Since we're not modifying the value of the property anywhere from inside the class, they are all immutable.

The more interesting part is the `WithLeft` method (#1), which can be used to create a clone of the object with a modified value of the `Left` property. I've omitted similar methods for other properties (#2), because they are all very similar. These methods correspond to the `with` keyword that we've seen earlier for F# records. You can see the similarity yourself:

```
let moved = { rc with Left = 10.0f }   #A
var moved = rc.WithLeft(10.0f);        #B
```

#A F#: using 'with' keyword
#B C#: using 'WithLeft' method

The important thing is that we don't have to explicitly read all properties of the `Rect` class and we just mention the property that we want to change. This syntax is actually quite elegant even if we want to modify more than one of the properties:

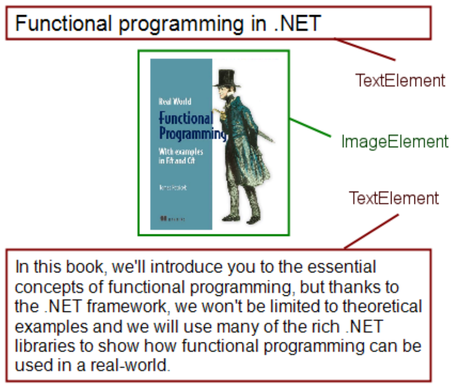
```
var moved = rc.WithLeft(10.0f).WithTop(10.0f);
```

Just as we've seen in this example, you'll often need to set two related properties at the same time. If this happens frequently, it is more convenient to add a new method that creates a clone and modifies all the related properties. In our example, we would likely also add methods `WithPosition` and `WithSize`, because they represent very common operations. This can also be necessary if each individual change would otherwise create an object in an invalid state, but the combined operation represents a valid state transition.

That's all we need to know about F# record types for now. We'll get back to functional data types in .NET in chapter 9. In the next section, we'll start working on a larger sample application, which is the heart of this chapter, and we'll talk about one usual way of representing program data.

7.2 Flat document representation

As I wrote in the introduction, we'll develop an application for viewing documents in this chapter. We'll start by designing a representation of the document that is suitable for drawing it on the screen. In this representation, the document will be just a list of elements with some content (either text or an image) and a specified bounding box in which the content should be drawn. You can see an example of a document with three highlighted elements in figure 7.1.



In this book, we'll introduce you to the essential concepts of functional programming, but thanks to the .NET framework, we won't be limited to theoretical examples and we will use many of the rich .NET libraries to show how functional programming can be used in a real-world.

Figure 7.1 Sample document that consists of three elements; two of them display some text with different fonts and one shows an image.

Let's look at the data structures that represent the document in F#. Listing 7.4 introduces a new discriminated union to represent the two alternative kinds of elements and a new record type for text elements. It uses the `Rect` type we defined earlier.

Listing 7.4 Flat document representation (F#)

```
open System.Drawing                               #A

type TextContent =                                #1
   { Text : string
     Font : Font }

type ScreenElement =                              #2
   | TextElement  of TextContent * Rect           #B
   | ImageElement of string * Rect                #C
```

#A Contains the 'Font' class
#1 Represents text with font
#2 Represents element of the document
#B Text content with bounding box
#C Image file name with bounding box

In this sample, we're defining two types. First of all, we define a record type called `TextContent` (#1) that represents some text and the font that should be used to draw it. The second type called `ScreenElement` (#2) is a discriminated union with two alternatives. The first alternative stores text content and the second one contains the file name of an image. Both of them also have a `Rect` to define the bounding box for drawing. Listing 7.5 shows the code to represent the sample document from figure 7.1 using our new data types.

Listing 7.5 Sample document represented as a list of elements (F#)

```
let fntText = new Font("Arial", 12.0f)  #A
let fntHead = new Font("Arial", 15.0f)  #A

let elements =                          #B
   [ TextElement
       ({ Text = "Functional programming in .NET"; Font = fntHead },
        { Left = 10.0f; Top = 0.0f; Width = 400.0f; Height = 30.0f });
     ImageElement
       ("cover.jpg",
        { Left = 120.0f; Top = 30.0f; Width = 150.0f; Height = 200.0f });
     TextElement
       ({ Text = @"In this book, we'll introduce you to the essential
           concepts of functional programming, but thanks to the .NET
           framework, we won't be limited to theoretical examples and we
           will use many of the rich .NET libraries to show how functional
           programming can be used in a real-world."; Font = fntText },
        { Left = 10.0f; Top = 230.0f; Width = 400.0f; Height = 400.0f }) ]
```

#A Create fonts for heading and for usual text
#B Create a list of 'ScreenElement' values

First we define fonts for the two different text elements, and then just construct a list containing the elements. When creating elements, we create several F# record type values using the syntax discussed earlier. You can see that this way of constructing documents is a bit impractical and we'll design a different representation, more suitable for creating documents later. Before that, we'll implement a function to draw a document stored using this representation.

7.2.1 Drawing documents

Just like in chapter 4 when we drew a pie chart, we'll use the standard .NET `System.Drawing` library. The point of this example is to demonstrate that using the previous representation, drawing is extremely simple, so the core function in listing 7.6 has just a few lines of code. It simply iterates over all elements in a list and contains drawing code for the two different kinds of elements.

Listing 7.6 Drawing document using flat representation (F# interactive)

```
> let drawElements elements (gr:Graphics) =
     for p in elements do                                                #A
        match p with
        | TextElement(te, rc) ->
           let rcf = toRectangleF rc                                     #B
           gr.DrawString(te.Text, te.Font, Brushes.Black, rcf)
        | ImageElement(img, rc) ->
           let bmp = new Bitmap(img)                                     #C
           let wsp, hsp = rc.Width / 10.0f, rc.Height / 10.0f            #D
           let rc = toRectangleF(deflate(rc, wsp, hsp))                  #D
           gr.DrawImage(bmp, rc);;
val drawElements : seq<ScreenElement> -> Graphics -> unit                #1
```

#A Imperative iteration over all elements
#B Convert 'Rect' to .NET RectangleF
#C Load image from the specified file
#D Add border 10% of the image size
#1 Function type signature

The function draws the specified list of elements to the given `Graphics` object. The type of the first parameter is `seq`, which represents any collection. So far we've been working with lists, but you'll see some other collections (such as arrays) in chapters 10 and 11. In the code, we only need to iterate over the elements in the collection using a `for` loop, so the compiler inferred the most general type for us. The type `seq<'a>` corresponds to the generic `IEnumerable<T>`, so in C# the type of the parameter would be `IEnumerable<ScreenElement>`.

The code also uses the functions from the previous section to work with the `Rect` values. We use `toRectangleF` to convert our `Rect` value to the type which the `DrawString` method needs, and `deflate` to add space around the image.

Our drawing function takes the `Graphics` object as an argument, so we need some way of creating one. As a final step, we'll write some code to create a form and draw the document onto it.

DRAWING TO A FORM

The drawing will be similar to the example from chapter 4. Because the drawing can take some time, we'll create an in-memory bitmap, draw the document there and then display the document on a form rather than drawing the document every time form is invalidated. However, let's first look at one very useful functional programming pattern that we'll use in this section.

The "Hole in the middle" pattern

One very common situation when writing a code is that you perform some initialization, then the core part of the function and then some clean-up at the end. When you repeat similar operation in multiple places of the program, the initialization and clean-up don't change and only the core part is different. A sample that draws on an in-memory bitmap written in C# would look like this:

```
var bmp = new Bitmap(width, height)
using(var gr = Graphics.FromImage(bmp)) {
   (...)                                #A
}
```

#A Core part: drawing using 'gr' object

The problem with this code is that using only object-oriented programming concepts, you can't simply wrap the code that performs the initialization and finalization into a subroutine and share it between all the places that do different drawing.

In functional programming, the solution is trivial. You can simply write higher order function and wrap the core part into a lambda function and use it as an argument:

```
var bmp = DrawImage(width, height, gr => {
      (...)                                #A
   });
```

#A Core part inside a lambda function

From a functional point of view, this is an uninteresting example of using a higher order function, but the case where we need to perform some initialization followed by the core part and then clean-up is very common, so it deserves a special name. The name was first used by Brian Hurt in a blog post "The 'Hole in the middle' pattern" [Hurt, 2007]. It nicely describes the fact that only the middle part needs to be filled in with a different functionality in every use of the code.

Listing 7.7 shows an F# implementation of a function similar to the `DrawImage` from the previous sidebar. In addition to the two parameters that specify the size of the created bitmap, it also allows specifying additional margins from the border of the image.

Listing 7.7 Function for drawing images (F# interactive)

```
> let drawImage (wid:int, hgt:int) space f =
      let bmp = new Bitmap(wid, hgt)
      let gr = Graphics.FromImage(bmp)
      let rc = Rectangle(Point(0,0), Size(wid,hgt))                  #A
      gr.FillRectangle(Brushes.White, rc)                            #A
      gr.TranslateTransform(space, space)                            #B
      f(gr)                                                          #1
      gr.Dispose()                                                   #C
      bmp;;
val drawImage : int * int -> float32 -> (Graphics -> unit) -> Bitmap #2
```

#A Fill the background with white color
#B All drawing on graphics will be shifted
#1 Call the core part of drawing
#C Clean-up
#2 Function type signature

When we use this function to draw an image, the core part of the drawing will be specified in a function given as the last argument. The type signature (#2) shows that the function takes a `Graphics` as an argument and doesn't return a result. It is invoked in the middle of the code (#1) after the bitmap and `Graphics` object are created. We also call `TranslateTransform` in the initialization phase, to provide some padding for the drawing. Finally, the function ends with the clean-up code to release the resources used for drawing before returning the bitmap. In the listing above, we call the `Dispose` method explicitly, which isn't entirely correct. We'll look how to fix this in chapter 9 when we'll talk about using `IDisposable` type from F#.

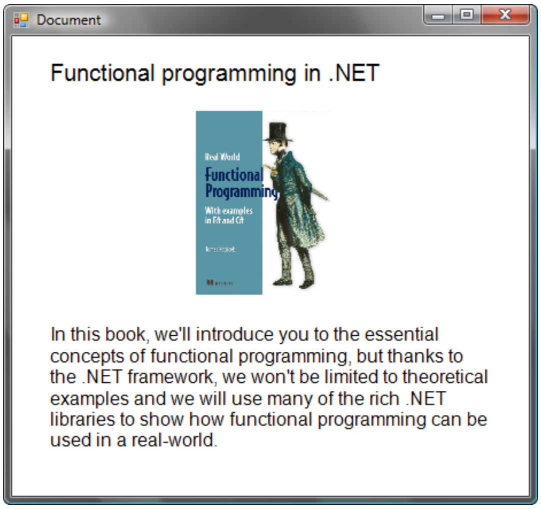
Finally we have everything we need to see our code in action. For now, we'll just create and test the form interactively. Listing 7.8 shows how to draw the screen elements from listing 7.5 and show the document on a form.

Listing 7.8 Drawing the document using WinForms (F# interactive)

```
> let docImg = drawImage (400, 450) 20.0f (drawElements elements)       #1
val docImg : Bitmap

> open System.Windows.Forms                                             #A
  let main = new Form(Text = "Document", BackgroundImage = docImg)      #A
  main.Show();;                                                         #A
```

#1 Draw the document
#A Create a form with the document

The line where we draw the bitmap (#1) may require a little explanation. We're calling `drawImage`, which takes a function specifying the core part of the drawing as the last argument. Since we've already implemented this in the `drawElements` function, you might expect us to just be able to pass it directly as the last argument. However, `drawElements` has two parameters - whereas `drawImage` expects a function with only one (the `Graphics` object to draw on). We use partial function application to specify the list with `ScreenElement` values. The result of the partial application is a function that takes a `Graphics` object and draws the document, which is exactly what we need. You can see the result of our work in figure 7.2.



Functional programming in .NET

In this book, we'll introduce you to the essential concepts of functional programming, but thanks to the .NET framework, we won't be limited to theoretical examples and we will use many of the rich .NET libraries to show how functional programming can be used in a real-world.

Figure 7.2 Sample document stored as a list of screen elements, drawn using 'drawElements' function to a Windows Forms form.

As we've seen, our previous representation of the document allowed us to implement drawing very easily. However, the code we had to use to create the document in the first place was somewhat awkward. In functional programming, you'll often find that different contexts suggest different data structures: the desired usage determines the ideal representation to some extent. It's not uncommon for a functional program to have different representations for the same information in a single program. Now that we've got a suitable form for drawing, let's try to design one which is suitable for construction and processing–and then write a transformation function to get from one representation to the other.

7.3 Structured document representation

The data structure that we'll design in this section is inspired by the HTML format, which is a familiar and successful language for creating documents. Just like HTML, our representation will have several different types of content and it will be possible to nest some parts within each other in appropriate ways. Figure 7.3 shows an annotated sample document, which should give you an idea of what the format will include.

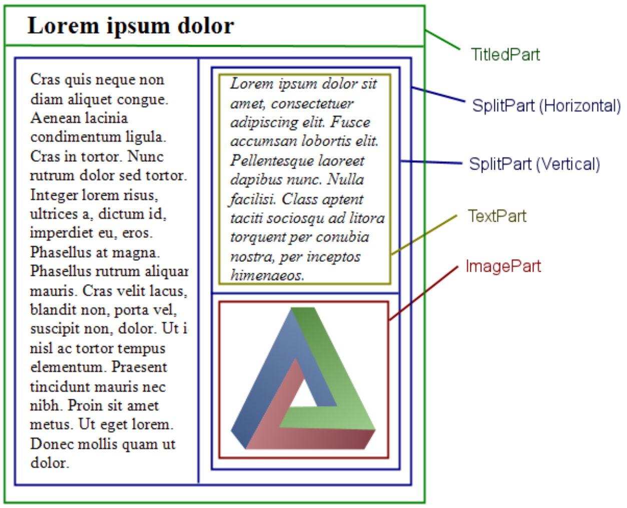

Lorem ipsum dolor
Cras quis neque non diam aliquet congue. Aenean lacinia condimentum ligula. Cras in tortor. Nunc rutrum dolor sed tortor. Integer lorem risus, ultrices a, dictum id, imperdiet eu, eros. Phasellus at magna. Phasellus rutrum aliquar mauris. Cras velit lacus, blandit non, porta vel, suscipit non, dolor. Ut i nisl ac tortor tempus elementum. Praesent tincidunt mauris nec nibh. Proin sit amet metus. Ut eget lorem. Donec mollis quam ut dolor.
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Fusce accumsan lobortis elit. Pellentesque laoreet dapibus nunc. Nulla facilisi. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.
TitledPart
SplitPart (Horizontal)
SplitPart (Vertical)
TextPart
ImagePart

Figure 7.3 Four different kinds of parts available in our document format; 'TitledPart' adds title to another part, and using 'SplitPart' we can create columns and rows. 'TextPart' and 'ImagePart' specify the actual content.

There are two different types of parts. Simple parts like `TextPart` and `ImagePart` contain some content, but cannot contain nested parts. On the other side, `TitledPart` contains one nested part and adds a title to it, while `SplitPart` contains one or more nested parts and an orientation. As you may have guessed, we'll represent the different parts using a discriminated union. Because two of the parts can contain nested parts, the type will be recursive. Listing 7.9 shows the type declaration, giving us something more concrete to discuss in detail.

Listing 7.9 Hierarchical document representation (F#)

```
type Orientation =                                          #1
   | Vertical
   | Horizontal

type DocumentPart =                                         #2
   | SplitPart  of Orientation * list<DocumentPart>        #A
   | TitledPart of TextContent * DocumentPart              #B
   | TextPart   of TextContent                             #C
   | ImagePart  of string                                  #C
```

#1 Represents orientation of the 'SplitPart'
#2 Recursive type representing the document
#A Columns or rows containing parts
#B Other part with a title
#C Simple content parts

The most important point to note is that the transcription of our informal specification to F# code is very straightforward. This is definitely one of the most attractive aspects of the standard F# type declarations. We first declare a simple discriminated union with just two options to represent an orientation for split parts (#1) and then declare the `DocumentPart` type with four alternative options.

Two of the options recursively contain other document parts. `SplitPart` contains several other parts in a list and also an orientation to determine how the area should be divided; `TitledPart` consists of a single other part and a title to decorate it with. As you can see, the text is stored using the `TextContent` type from the previous section, which is a record containing a string together with a font.

Note that the `DocumentPart` type represents the document as a whole. Because the type is recursive, we can nest any number of content parts inside a single document part. This is different to the previous approach, where we created a type for an element and then represented the document as a list of elements. In that representation, the list served as a "root" of the data structure and the elements were not further nested. Using the new data types, we can write the document from section 7.2 like this:

```
let doc =
   TitledPart({ Text = "Functional programming"; Font = fntHead },
      SplitPart(Vertical,
         [ ImagePart("cover.jpg");
           TextPart({ Text = "..."; Font = fntText }) ]
      )
   )
```

I omitted the content of the `TextPart` located below the image, but you can still see that the representation is terser, because we don't need to calculate bounding rectangles. However, we don't have an implementation of drawing for this data type. We're not going to write one, either–why would we, when we've already got a perfectly good drawing function for the earlier representation? All we need to do is provide a translation from the "designed for construction" form to the "designed for drawing" one.

7.3.1 Converting representations

There are two key differences between the data types that we've just implemented. The first is that the data type from section 7.2 explicitly contains the bounding boxes specifying location of the content. Compare that with the second data type, which only indicates how the parts are nested. This means that when we translate the representation, we'll need to calculate each location based on the nesting of the parts. The second difference is that in the new representation, the document is just a single (recursive) value, while in the first case it is a list of elements. These two differences affect the signature of the translation function looks, so let's analyze that before we study the implementation:

```
val documentToScreen : DocumentPart -> Rect -> list<ScreenElement>
```

The function takes the part of the document to translate as the first argument and returns a list of `ScreenElement` values from the section 7.2. This means that both the input argument and the result can represent the whole document. The function has also a second argument, which specifies the bounding rectangle of the whole document. During the translation, we'll need it to calculate positions of the individual parts. Listing 7.10 shows the implementation, which is (unsurprisingly) a recursive function.

Listing 7.10 Translation between document representations (F#)

```
let rec documentToScreen(doc, rc) =
   match doc with
   | SplitPart(Horizontal, parts) ->                           #1
      let width = rc.Width / (float32 parts.Length)            #A
      parts
         |> List.mapi (fun i part ->                           #2
               let left = rc.Left + (float32 i) * width        #2
               let rc   = { rc with Left = left; Width = width }  #2
            documentToScreen(part, rc))                        #2
         |> List.concat                                        #3
   | SplitPart(Vertical, parts) ->                             #4
      let height = rc.Height / (float32 parts.Length)
         parts
         |> List.mapi (fun i part ->
               let top = rc.Top + (float32 i) * height         #D
               let rc  = { rc with Top = top; Height = height }   #D
               documentToScreen(part, rc))                     #E
         |> List.concat
   | TitledPart(tx, doc) ->                                    #5
      let titleRc = { rc with Height = 35.0f }
      let restRc  = { rc with Height = rc.Height - 35.0f;
                              Top = rc.Top + 35.0f }
      TextElement(tx, titleRc)::(documentToScreen(doc, restRc))   #F
   | TextPart(tx)  -> [ TextElement(tx, rc) ]                  #6
   | ImagePart(im) -> [ ImageElement(im, rc) ]                 #6
```

#A Calculate the size of individual parts
#2 Recursively translate columns of part
#3 Concatenate lists of screen elements
#D Calculate bounding box of the row
#E Recursive call
#F Translate the body and append the title

Let's start from the end of the code. It's easy to process parts that represent content (#6) because we just return a list containing a single screen element. We can use the rectangle that we've been provided as an argument to indicate the position and size. No further calculation is required.

The remaining parts are more interesting, because they are composed from other parts. In this case, the function calls itself recursively to process all the sub-parts that form the larger part. This is where we have to perform some layout calculations, because when we call `documentToScreen` again, we give it a sub-part and the bounding box for the sub-part. We can't just copy the `rc` parameter, or all the sub-parts would end up in the same place! Instead we have to divide the rectangle we've been given into smaller rectangles, one for each sub-part.

`TitledPart` (#5) contains just a single sub-part, so we need to perform just one recursive call. Before that, we calculate one bounding box for the title (35 pixels at the top) and one for the body (everything except the top 35 pixels). Next, we process the body recursively and append a `TextElement` representing the title to the returned list of screen elements.

We process a `SplitPart` using a separate branch for each of the orientations (#1, #4). First we calculate size of each of the column or row and then convert all its parts. We use `List.mapi` function (#2), which is just like `List.map`, but in addition it gives us an index of the part that we're currently processing. We can use the index to calculate the offset of the smaller bounding rectangle from the left or from the top of the main rectangle. The lambda function then calls `documentToScreen` recursively and returns a list of screen elements for every document part. This means that we get a list of lists as the result of the projection using `List.mapi`. The type of the result is `list<list<ScreenElement>>` rather than the flat list we need to return, so we use the standard F# library function `List.concat` (#3), which turns the result into a value of type `list<ScreenElement>`.

TRANSLATION IN DETAIL

The translation between different representations of the document is the most difficult part of this chapter, so you may want to download the source code and experiment with it to see how it works. The most interesting (and difficult) part is calculating the bounding rectangle for each recursive call. Likewise it's worth making sure you understand the list returned by the function, and how it's built up from each of the deep recursive calls. You may find it useful to work through an example with a pencil and paper, keeping track of the bounding rectangles and the returned screen elements as you go.

Translation between different representations is often the key to the simplicity of a functional program, as it allows us to implement each of the other operations using the most appropriate data structure for the situation. We've seen that the first representation is perfect for drawing the document, but that the second make construction simpler. It turns out that the second form also makes manipulation easier, as we'll see in section 7.4. Before that though, we'll introduce one more representation: XML.

7.3.3 XML document representation

The XML format is very popular and is a perfect fit for storing hierarchical data such as our document from the previous section. Working with XML is important for many real-world applications, so in this section we'll extend our application to support loading documents from XML files. We'll use the .NET 3.5 *LINQ to XML* API to do most of the hard work–there's no point in writing yet *another* XML parser. LINQ to XML is a good example of how functional concepts are being used in mainstream frameworks: although it isn't a purely functional API (the types are generally mutable) it allows objects to be constructed in a recursive and declarative form. This can make the structure immediately apparent from the code, so it's much easier to read than typical code using the DOM API.

In some sense, this is just another translation from one representation of the data into another. In this case the source representation is a structure of LINQ objects and the target is our document data type from section 7.3.1. The translation is a lot easier this time because both of the data structures are hierarchical. Listing 7.11 demonstrates the XML-based format that we'll use for representing our documents.

Listing 7.11 XML representation of a sample document (XML)

```
<titled title="Functional Programming in .NET"
        font="Times New Roman" size="18" style="bold">          #1
   <split orientation="vertical">                               #2
      <text>In this book, we'll introduce you (...)</text>      #A
      <image url="C:\Tomas\Writing\Functional\Petricek.jpg" />   #A
   </split>
</titled>
```

#1 Properties of the font used for the title
#2 Vertical or horizontal split
#A Sub-parts are nested XML elements

Before looking at the core part of the translation, we need to implement some utility functions that parse the attribute values shown in the XML. In particular, we need a function for parsing a font name (#1) and the orientation of the `SplitPart` (#2). Listing 7.12 shows these functions and also introduces several objects from the LINQ to XML library.

Listing 7.12 Parsing font and orientation using LINQ to XML (F#)

```
open System.Xml.Linq

let attr(node:XElement, name, def) =                              #1
   let attr = node.Attribute(XName.Get(name))
   if (attr <> null) then attr.Value else def

let parseOrientation(node) =                                      #2
   match attr(node, "orientation", "") with
   | "horizontal" -> Horizontal
   | "vertical" -> Vertical
   | _ -> failwith "Unknown orientation!"                         #3

let parseFont(node) =                                             #4
   let str = attr(node, "style", "")
   let style =
      match str.Contains("bold"), str.Contains("italic") with     #A
      | true,  false -> FontStyle.Bold
      | false, true  -> FontStyle.Italic
      | true,  true  -> FontStyle.Bold ||| FontStyle.Italic       #B
      | false, false -> FontStyle.Regular
   let name = attr(node, "font", "Arial")
   new Font(name, float32(attr(node, "size", "12")), style)
```

#1 Reads attribute or returns the specified default value
#2 Parses value of the 'orientation' attribute

#3 Throw an exception
#4 Parse a font node with specified style
#A Tests whether the attribute contains specified strings
#B Combine two options of .NET enumeration

This code will only work with a reference to the System.Xml.Linq.dll assembly. In Visual Studio, you can use the usual "Add Reference" command from Solution Explorer. In F# interactive you can use the `#r "(...)"` directive and specify the path to the assembly as the argument, or just the assembly name if it's in the GAC.

The listing starts with the `attr` function (#1) that we use for reading attributes. It takes an `XElement` (the LINQ to XML type representing an XML element) as the first argument and then the name of the attribute. The final parameter is the default value to use when the attribute is missing. The next function (#2) uses `attr` to read the value of the "orientation" attribute of an XML node that is passed into it. If the attribute contains an unexpected value, then the function throws an exception using the standard F# function `failwith` (#3).

Finally, `parseFont` (#4) is used to turn attributes of an XML tag like `title` in listing 7.11 into a .NET `Font` object. The most interesting part is the way that we parse the "style" attribute. It tests whether the attribute value contains two strings ("bold" and "italic") as substrings and then uses pattern matching to specify a style for each of the four possibilities. The function also converts a string representation of the size into a number using the `float32` conversion function and creates an instance of the `Font`.

Now that we have all the utility functions we need, loading the XML document is quite easy. List 7.13 shows a recursive function `loadPart` which performs the complete translation.

Listing 7.13 Loading document parts from XML (F#)

```
let rec loadPart(node:XElement) =
   match node.Name.LocalName with                                        #1
   | "titled" ->
      let tx = { Text = attr(node, "title", ""); Font = parseFont node}
      let body = loadPart(Seq.hd(node.Elements()))                       #A
      TitledPart(tx, body)
   | "split"  ->
      let orient = parseOrientation node
      let nodes = node.Elements() |> List.of_seq |> List.map loadPart    #B
      SplitPart(orient, nodes)
   | "text"   ->
      TextPart({Text = node.Value; Font = parseFont node})
   | "image"  ->
      ImagePart(attr(node, "url", ""))
   | _ -> failwith "Unknown node!"                                       #2
```

#1 Select branch using element name
#A Recursively load the first child element
#B Recursively load all children
#2 Throw an exception

The function takes an XML element as an argument and we'll give it the root element of the XML document when we use it later. Its body is a single `match` construct (#1) that tests the name of the element against the known options and throws an exception if it encounters an unknown tag (#2).

Loading image and text parts is easy because we just need to read their attributes using our functions utility function and create appropriate `DocumentPart` values. The remaining two document part types involve recursion, so they are more interesting.

To create a `TitledPart` from a "titled" element we first parse the attributes for the title text and then recursively process the first XML element inside the part. To read the first child element, we call `Elements()` method, which returns all the child elements as a .NET `IEnumerable` collection. `IEnumerable<T>` is abbreviated as `seq<'a>` in F# and we can work with it using functions from the `Seq` module that are similar to functions for working with lists. In our example, we use `Seq.hd`, which returns the first element (the head) of the collection. If we were writing this code in C#, we could call `Elements().First()` to achieve the same effect.

Finally, to create a `SplitPart` from a "split" element we need to parse *all* the children, so again we call the `Elements()` method but this time we convert the result to a functional list of `XElement` values. Next, we recursively translate each one into a `DocumentPart` value using a projection with the `loadPart` function as an argument.

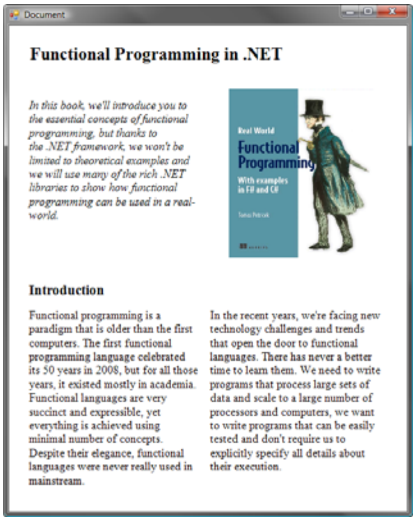
The function is very straightforward because it simply provides a few lines of code that parse the XML node for each of the supported tags. A lot of the simplicity is due to the fact that the XML document is hierarchical in the same way as the target representation. This lets us simply use recursion when a part has nested sub-parts.

Now that we have a function to load the document from an XML element, we can finally see how the application displays a larger document: designing the document in an XML editor is easier than creating values in F#. Listing 7.14 shows the final piece of plumbing used to combine all the code that we've developed so far into a normal Windows Forms application.

Listing 7.14 Putting the parts of the application together (F#)

```
open System.Windows.Forms

[<System.STAThread>]
do
   let doc = loadPart(XDocument.Load(@"C:\...\document.xml").Root)
   let parts = documentToScreen(doc, { Left = 0.0f; Top = 0.0f;
                                       Width = 500.0f; Height = 600.0f })
   let img = drawImage (550, 650) 25.0f (drawElements parts)
   let main = new Form(Text = "Document", BackgroundImage = img
                       ClientSize = Size(550, 650))
   Application.Run(main)
```

The code starts by loading the document from a XML file using the `XDocument` class. We pass the document's root element to our `loadPart` function which converts it into the hierarchical document representation. Next, we convert this into the flat representation using `documentToScreen` and then draw and display the document using the code we saw in listing 7.8. This time, we have also added the `STAThread` attribute which is needed for Windows Forms applications. The final line starts the application with the `Application.Run` method. You can see a screenshot showing the result in figure 7.3.



Functional Programming in .NET

In this book, we'll introduce you to the essential concepts of functional programming, but thanks to the .NET framework, we won't be limited to theoretical examples and we will use many of the rich .NET libraries to show how functional programming can be used in a real-world.

Introduction

Functional programming is a paradigm that is older than the first computers. The first functional programming language celebrated its 50 years in 2008, but for all those years, it existed mostly in academia. Functional languages are very succinct and expressible, yet everything is achieved using minimal number of concepts. Despite their elegance, functional languages were never really used in mainstream.

In the recent years, we're facing new technology challenges and trends that open the door to functional languages. There has never a better time to learn them. We need to write programs that process large sets of data and scale to a large number of processors and computers, we want to write programs that can be easily tested and don't require us to explicitly specify all details about their execution.

Figure 7.3 Finished application displaying document with all four different kinds of document parts.

Earlier I mentioned that the hierarchical representation is useful for manipulating the document as well as performing the initial construction. Let's take a look at that now, starting with a generally useful function for processing documents and then using it in a practical example.

7.4 Writing operations

There are many kinds of operations that we could perform with a document. We could capitalize all the titles in the document or perhaps merge text in multiple columns into a single column. All these operations have something in common and you may see a similarity between this and the map operation from the previous chapter. Just like mappings, they examine the document, perform some operation with certain parts of it and return a new document.

Another kind of operation would return just a single value of different type. For example we could implement a function to count the words in the document or return all the document text as a single string. Again, this should sound familiar: the `foldLeft` function from section 6.6.3 does exactly the same job, but working with lists instead of documents.

As we learned in the previous chapter, writing a separate function for each operation would be impractical and we can get better results if we write a single higher order function that can be reused for different purposes. We'll start by implementing the function discussed in the first paragraph: the one reminiscent of the map operation.

7.4.1 Updating using a map operation

Even though the operation is similar to map, we'll implement it a bit differently this time. The function which is used to process each part should be able to give two kinds of result. It could return a new part which we'll use to replace the original part, or it return an empty value, in which case we'll use the original part and recursively process all its sub-parts. Keep in mind that there are several variations of this design. For example, we could also return special value to denote that the currently processed part should be removed from the parent. However, we'll continue and implement the simpler version. When thinking about higher order functions, one of the first aspects to consider is the signature. Here's the signature of the function we're going to implement:

```
val mapDocument :
    (DocumentPart -> option<DocumentPart>) -> DocumentPart -> DocumentPart
```

Let's start from the end. The function takes the original document and returns an updated version. The first parameter is the processing function and as you can see, it returns an option value. This allows it to return `Some` value when replacing a part or `None` to leave the original part (and recursively map it). The function is implemented in listing 7.15.

Listing 7.15 Map operation for documents (F#)

```
let rec mapDocument f doc =
   match f(doc), doc with                                          #1
   | Some(newDoc), _ ->
      mapDocument f newDoc                                         #2
   | _, TitledPart(tx, cont) ->
      TitledPart(tx, mapDocument f cont)                           #3
   | _, SplitPart(orient, parts) ->
      let updated = parts |> List.map (mapDocument f)              #4
      SplitPart(orient, updated)
   | _ -> doc
```

#1 Tests the result and the original part
#2 Function returned new document part
#3 Recursively process the body
#4 Process all columns or rows

The code is implemented using handy pattern matching (#1). We combine the result of the call to the function given as an argument with the original document part, so we can write all the possible cases in a single `match` construct. When the function returns a new document part, we recursively process the part and then return it as the result (#2). This would be important for example if we wanted to change all titles in the document. In that case, we'd need to recursively process the body of the returned part to process all nested titles. When the function returns `None`, we look at the original document part. If the original part contains sub-parts, we need to process them recursively. For a titled part (#3), we just process the body and return a new TitledPart with the original title. For a split part, we use List.map to obtain a new version of each of the columns or rows and then use the result to construct a new SplitPart.

Now that we have a higher order function, let's try to use it. I mentioned earlier that we could merge several columns of text into a single part. This would be useful in an adaptive document layout: on a wide screen we want to view several columns, whereas on a narrow screen a single column is more readable. Listing 7.16 shows how to shrink a split part containing only text into a single part.

Listing 7.16 Shrinking split part containing text (F#)

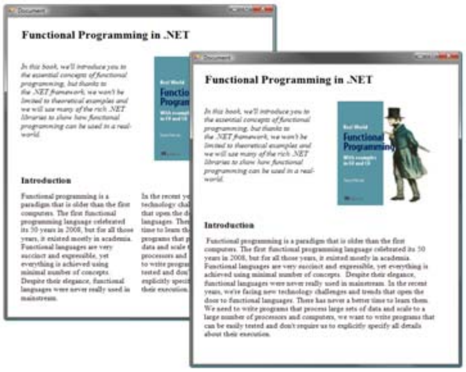
```
let isText(part) =                                                #A
   match part with | TextPart(_) -> true | _ -> false             #A

let doc = loadPart(XDocument.Load(@"C:\...\document.xml").Root)
let shrinkedDoc = doc |> mapDocument (fun part ->
   match part with
   | SplitPart(_, parts) when List.for_all isText parts ->        #1
      let res =
         List.fold_left (fun st (TextPart(tx)) ->                 #2
            { Text = st.Text + " " + tx.Text                      #B
              Font = tx.Font } )
            { Text = ""; Font = null } parts                      #C
      Some(TextPart(res))
   | _ -> None )                                                  #D
```

#A Utility testing whether part is a 'TextPart'
#1 Split part containing only text parts
#2 Aggregate all parts using fold
#B Concatenate text and return font
#C Start with empty string and 'null' font
#D Ignore other cases

In the processing function, we need to check whether the given part is a SplitPart containing only text parts. The first condition can be checked directly using pattern matching and the second one is specified in a when clause of the pattern. We write a utility function isText that tests whether a part is TextPart and then use it from List.for_all to test whether all parts fulfill the condition (#1).

Next, we use fold_left to aggregate all the parts in the list into a single part. We already know that each sub-parts is a TextPart, so we can use it directly as a pattern when we write the lambda function to aggregate the result (#2). However, the compiler cannot verify that this is correct, so it gives a warning. You should be always very careful when you spot a warning, but in this case we can safely ignore it. However, in larger projects where you want to eliminate all compiler warnings, you'd probably rewrite the code using match construct and call the failwith function in the unreachable branch. The aggregation implicitly uses the TextContent type and specifies an initial value with no text content and unset font. During every step, we concatenate the existing string with the value in the current part, and use the current font. We do not process fonts in a sophisticated manner, so we'll just end up with the font used by the last part.

You can see the final result of this operation in figure 7.5.



Functional Programming in .NET

In this book, we'll introduce you to the essential concepts of functional programming, but thanks to the .NET framework, we won't be limited to theoretical examples and we will use many of the rich .NET libraries to show how functional programming can be used in a real-world.

Introduction

Functional programming is a paradigm that is older than the first computers. The first functional programming language celebrated its 50 years in 2008, but for all those years, it existed mostly in academia. Functional languages are very succinct and expressible, yet everything is achieved using minimal number of concepts. Despite their elegance, functional languages were never really used in mainstream. In the recent years, we're facing new technology challenges and trends that open the door to functional languages. There has never a better time to learn them. We need to write programs that process large sets of data and scale to a large number of processors and computers, we want to write programs that can be easily tested and don't require us to explicitly specify all details about their execution.

Figure 7.5 Original and updated document; in the new document, split parts that contain only text are merged into a single text part

I mentioned earlier that this map-like operation is just one of several useful operations that we can provide for our documents. In the next section, we'll look at another one, which aggregates the document into a single value.

7.4.2 Calculating using an aggregate operation

The idea behind aggregation is that we maintain some state that will be passed around over the course of the operation. We start with an initial state and calculate a new one using the given processing function for each part in the document. This idea is reflected in the signature of the function:

```
    val aggregateDocument :
        ('a -> DocumentPart -> 'a) -> 'a -> DocumentPart -> 'a
```

The reason I've used the broad notion of "some state" is that the state can be anything. The type of the state in the function signature is a type parameter `'a`, so it depends on the user of the function. The processing function which calculates the new state based on the old state and a single document part is passed as the first argument; the initial state as the second argument, and the document as the third. Listing 7.17 shows the complete (and perhaps surprisingly brief) implementation.

Listing 7.17 Aggregation of document parts (F#)

```
    let rec aggregateDocument f state doc =
       let state = f state doc                                           #1
       match doc with
       | TitledPart(_, part) ->
          aggregateDocument f state part                                 #2
       | SplitPart(_, parts) ->
          List.fold_left (aggregateDocument f) state parts           #3
       | _ -> state
```

#1 Calculate new state for the current part
#2 Recursively process the body
#3 Aggregate state over all subparts

The code needs to walk over all the parts in the document. It first calls the function on the current part and then recursively processes all subparts. The ordering is relevant here: we *could* have designed the function to process all the sub-parts first and then the current part. The difference is that in the implementation above, the function is called on the "root" node of the tree, while in the other case it would first be called on the "leaf" nodes. For our purposes, both options would work fine, but for some advanced processing we'd have to consider what kind of traversal we wanted.

When we call the aggregation function with the current part (#1) we use the same name for the value to hold the new state. The new value hides the old one, and in this case that's a useful safety measure: it means we can't accidentally use the old state by mistake after we've computed the new state. Next, we process the parts that can contain sub-parts. For a titled part, we just recursively process the body (#2). When we get a split with a list of sub-parts, we aggregate it using normal aggregation on lists with the `List.fold_left` function (#3).

Aggregation can be useful for a variety of things. The following snippet shows how to use this operation for counting a number of words in the whole document:

```
let totalChars =
    aggregateDocument (fun count part ->
        match part with
        | TextPart(tx) | TitledPart(tx, _) ->           #A
            count + tx.Text.Split(' ').Length
        | _ -> count) 0 doc
```

#A Single case for both parts with text

The function that we use as an argument only cares about parts that contain text. We have two parts like this and both of them contain the text as a value of type `TextContent`. F# pattern matching allows us to handle both cases just using a single pattern. This syntax is called an or-pattern and it can be used only when both patterns bind value to the same identifiers with the same type. In our case, we only need a single identifier (`tx`) of type `TextContent`. Finally, in the body for the pattern matching, we split the text into words using a space as the separator and then add the length of the returned array to the total count.

TRY IT!

You can try extending this example in many ways. Here are a few ideas that you'll find solved on the book web www.functional-programming.net. You can use `mapDocument` to split text parts with more than 500 characters into two columns. Using aggregation, you can collect a list of images that are used in the document. Moreover, you can implement a filter-like operation that takes a function of type `(DocumentPart -> bool)` and creates document containing only parts for which the function returns true. Using this function, you can remove all the images from a document.

We've seen that the second representation is very convenient for various operations with the document, especially if we implement useful higher order functions first. In the last section we'll get back to C# for a while and we'll discuss which of the ideas that we've just seen are applicable to C# and also about related design patterns.

7.5 Object-oriented representations

Standard design patterns are divided into three groups - creational, structural and behavioral. In this section we'll look at few patterns from the last two groups and we'll see that they are very similar to some of the constructs that we used in F# earlier in this chapter. Of course, the functional version of the patterns will not be exactly the same as object-oriented, because OOP puts more emphasis on adding new types and FP puts more emphasis on adding new functionality, but the structure will be very similar.

TIP

This section assumes that you know a bit about some of the design patterns. You can find links to good introductory articles on the book's web site. We also don't have space to show all the data structures in C#, but you can find the full implementation online.

We'll start by discussing two structural patterns and later we'll look at one behavioral.

7.5.1 Representing data with structural patterns

If we talk about programs in terms of data structures instead of objects, we can say that structural patterns describe common and proved ways to design data structures. Design patterns as you know them are more concrete and specify how to implement these structures in object oriented languages using objects. In this chapter, we've seen functional ways to represent data. In the first representation we used a simple list of records, which is easy to write in any language, but the second representation using a discriminated union is more interesting. The first related pattern that we'll look at is the *composite* pattern.

THE COMPOSITE DESIGN PATTERN

This pattern allows us to compose several objects into a single composed object and work with it in a same way as with primitive objects. Figure 7.6 shows the usual object oriented way to implement this pattern.

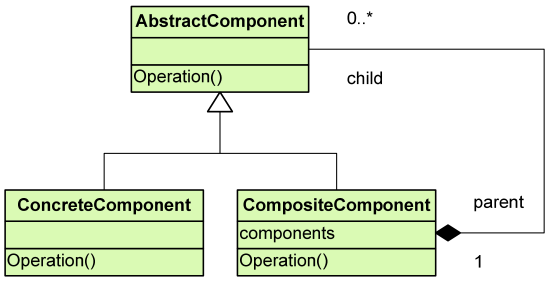


Figure 7.6 'CompositeComponent' is a class that contains collection of other components; it inherits from 'AbstractComponent', so it can be used in place of primitive component in a same way as other components, such as 'ConcreteComponent'

The composed object is represented by the `Composite` class. The program then works with objects just using the `AbstractComponent` class, so it doesn't need to understand the difference between primitive and composed object. You can also see an example of a virtual method, which is called `Operation`. In the `CompositeComponent` class, its implementation is usually very simple. It just iterates over all objects from the `components` collection and invokes `Operation` method on them.If you think about our document representation, you can find a very similar case there. When a part is split into multiple columns or rows using `SplitPart`, we treat it as an ordinary document part in exactly the same way as other parts. However, the part is just composed from other parts that are stored in a list. We can rewrite the general example from the figure 7.6 in the same way using recursive discriminated union type in F#:

```
type AbstractComponent
  | CompositeComponent of list<AbstractComponent> #A
  | ConcreteComponent of (...)
  | (...)                                      #B
```

#A Composite component

#B Other primitive components

In this example, the composite value is represented as one of the alternatives besides other primitive components. It recursively refers to the `AbstractComponent` type and stores values of this type in a list representing the composed object. When working with values of `AbstractComponent` type, we don't need to treat composed and primitive values separately, which is the primary aim of this design pattern.

As I said in the introduction for this section, there are some important differences between functional and object-oriented version of the pattern. Most importantly, in functional programming, the composition is public aspect of the type. As a result, any user of the type knows that there is a component created by composition and can use this fact when writing primitive processing functions, just like we did when implementing the `mapDocument` operation.

When using functional data structures, the focus is on the ability to easily add new functionality to existing types, so making the composition public is a valid design decision. This means that the functional version of the code also doesn't need to define the `Operation` method, which was part of the `AbstractComponent` type in the object-oriented representation. Any operation that uses the type can be implemented independently of the type as a processing function.

In fact, F# has an advanced feature called *active patterns* that allows us to encapsulate the composition to some extent. This allows us to publicly expose the composition, but the whole discriminated union type, which can be useful for evolving F# libraries. We don't discuss details of this feature in the book, but you'll find more information on the book's web site.

THE DECORATOR DESIGN PATTERN

Another pattern that is closely related to composite is called the *decorator* pattern. The goal of this pattern is to allow adding of new behavior to an existing class at runtime. As you can see in figure 7.7, the structure looks similar to the composite pattern.

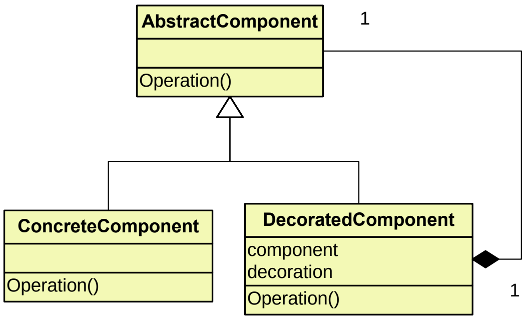


Figure 7.7 'DecoratedComponent' class wraps a component and adds new state to it; Implementation of the 'Operation' in decorated component calls the wrapped functionality and adds new behavior that uses the state of the decorated component

Even though the patterns look similar, their purposes are completely different. While the composite pattern allows us to treat composed values in a same way as primitive values, the purpose of the decorator pattern is to add new a feature to the existing object. As you can see, the `DecoratedComponent` class in the diagram wraps a single other component that is decorated and can carry additional state (such as the `decoration` field). The decorated component can also add some behavior that uses the additional state in the `Operation` method.

Again we can see a correspondence between this pattern and one of the parts in our document representation. The part that adds some decoration to another part in our application is `TitledPart`. The decoration is of course the title and the added state is the text and font of the title. We can write F# code that corresponds to the diagram of Decorator pattern similarly simply as for the Composite pattern:

```
type AbstractComponent =
    | DecoratedComponent of AbstractComponent * (...)    #A
    | ConcreteComponent of (...)
    | (...)                                               #B
```

#A Decorated component with additional state
#B Other primitive components

In this case, the data carried by the Decorator alternative is just a single decorated component (instead of list of components in the case of Composite) and also the additional state, which can vary between different decorators. I symbolized this using `(...)` syntax in the previous listing, but this is only pseudo-code. In real F# code you would specify the type of the actual state here, such as `TextContent` in our titled part. Just as with the composite pattern, the code that implements operations on the decorated component is located in the processing functions that we implement for our data structure. The code for the `DecoratedComponent` case in the processing function would call itself recursively to process the wrapped component and then execute the behavior added by the decorator, such as drawing a title of the document part.

The F# implementation of both of the patterns in this section in relied on using a recursive discriminated union type. In the next section, we'll work with it again, but in a different way. We'll look at the object-oriented way for adding new operations to existing data types.

7.5.2 Adding functions using the visitor pattern

Adding new operations to an existing data structure is the primary way of implementing any code that works with data in a functional language. In object-oriented languages, this is more difficult to do, but it is also needed less frequently. In this section we'll talk about the *visitor* pattern that is designed for this purpose and we'll sketch how we could use it to add operations to our representation of document. Figure 7.7 shows the basic classes that we'll use in this section.

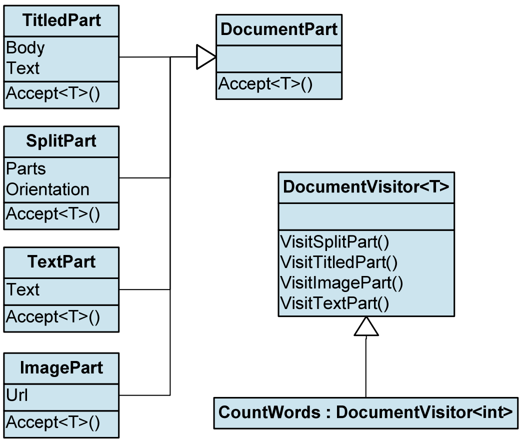


Figure 7.7 Diagram shows a class hierarchy that represents a document and a generic visitor class with state as a generic type parameter (T); all parts support the visitor via 'Accept' method

The hierarchy of classes that inherit from an abstract class `DocumentPart` is a usual way to represent alternatives in object-oriented programming and it corresponds to the discriminated union type that we've used in F#.

The main class of the visitor pattern is a generic `DocumentVisitor<T>` class. We're using a variant of the pattern that allows working with state, so the type parameter `T` represents the kind of state we need to maintain, such as arguments or the result of some computation performed by the visitor. The pattern also requires adding a virtual `Accept` method and implementing it in each of the derived classes. The method takes the visitor as an argument and calls it's appropriate `Visit` method, depending on which part it is. You can find the complete source code online, but let's briefly look at the code of the `Accept` method in `ImagePart`:

```
public override T Accept<T>(DocumentPartVisitor<T> visitor, T state) {
   return visitor.VisitImagePart(this, state);
}
```

The method only delegates the processing to the visitor. However, because it is implemented in every derived class, it can call `VisitImagePart` whose argument is a concrete class (in this case `ImagePart`). This means that when we'll implement a concrete visitor, we'll have an easy way to access properties of the different types that represent the document.

The listing 7.18 shows how we can add an operation that counts words in the document to the object oriented representation using the Visitor pattern.

Listing 7.18 Counting words in the document using Visitor (C#)

```
class CountWords : DocumentPartVisitor<int> {
   public override int VisitTitledPart(TitledPart p, int st) {
```

```
            return p.Text.Text.Split(' ').Length +
                   p.Body.Accept(this, st);                              #1
         }
         public override int VisitSplitPart(SplitPart p, int st) {
            return p.Parts.Aggregate(st, (n, p) =>                       #A
                      p.Accept(this, n));                                #2
         }
         public override int VisitTextPart(TextPart p, int st) {
            return p.Text.Text.Split(' ').Length + st;
         }
         public override int VisitImagePart(ImagePart p, int st) {
            return st;
         }
      }
```

#1 Recursively count words of the body
#A Aggregate the count over all subparts
#2 Count words in each part

This code corresponds to writing a recursive F# function that uses pattern matching to test which of the parts we are currently processing. In an object-oriented way, this choice is done in the `Accept` methods from the Visitor pattern. The `CountWords` class inherits from the visitor and uses a single `int` value as the state. Methods that process different types of document parts just add the number of words to the current state and there are also two methods (#1, #2) that have to recursively invoke the visitor on certain subparts. The invocation is done by calling the `Accept` method on the subpart. This is similar to the code that we need to run the processing on the entire document:

```
int count = doc.Accept(new CountWords(), 0);
```

Here we just call the `Accept` method and give it a new instance of the visitor as an argument. If we wanted to add another operation, we would implement a new class similarly as `CountWords` and execute it by giving it as an argument to the `Accept` method.

7.6 Summary

Working with data and designing data structures in a way that matches how we want to *use* the data is an important part of functional program design. In this chapter, we completed our toolset of basic functional data types by looking at the F# record type. We used records, lists and recursive discriminated unions together to design and implement an application for working with documents.

Functional programs often use multiple representations of data during processing and our application provided an example of this. One representation (a flat list of elements) allowed us to draw the document simply, whereas another (a hierarchy of parts) proved more useful for constructing and manipulating documents. We implemented a translation between these two representations, so the application could read the document from an XML file, process it in the hierarchical form, and then draw it using the flat form.

We've also looked at design patterns that you'd probably use if you wanted to implement the same problem in C#. In particular, we've seen that the composite and decorator patterns correspond closely with the alternative values we used in the document data structure. Finally, we've also seen a C# way to add a new "function" for processing an existing data structure using the visitor pattern.

This chapter was primarily about data-centric programs, where we design the data structures first. However, as I mentioned in the introduction, there are also programs that are primarily concerned with behavior. Of course, in more complex applications these two approaches are combined. In the next chapter, we'll turn our attention to the second important aspect of functional program design and talk about behavior-centric applications.

8 Designing behavior-centric programs

In the previous chapter, we discussed data-centric applications and I wrote that the first step when designing functional programs is to think about the relevant data structures. However, there are also cases where the data structure contains some form of behavior. For example, this might be a command that the user can invoke or some tasks that the program executes at some point. Instead of hard-coding every behavioral feature, we want to work with them uniformly, so we need to keep them in a data structure which can be easily modified, either before the compilation or even at run-time.

In the previous chapter, I gave the example of an application that processes images using graphical filters. The application needs to store the filters and add or remove them depending on what filters you want to apply. When representing this in the program, we could easily use a list for the actual collection of filters to apply–the harder question is what data structure we should use to represent the filters themselves? Clearly, a filter isn't really *data*, although it may be parameterized in some fashion. Instead, it denotes *behavior* and the simplest way for representing behavior in a functional language is to use a function.

As we've seen in chapter 5, functions can be treated as values, so we can work with them as with any other data types. This means that a list of functions is a perfectly reasonable data structure for representing graphical filters. The difference between behavior-centric and data-centric is more conceptual then technical. Understanding what kind of application you are designing is a helpful hint for creating a correct design.

EXAMPLES OF BEHAVIOR-CENTRIC APPLICATIONS

In applications of a significant size, both approaches are usually combined. A larger graphical editor that supports vector graphics as well as image filtering might use a data-centric approach for representing shapes and a behavior-centric approach for applying graphical filters to the image. Implementing graphical processing is beyond the scope of this chapter, but you can find a sample application for graphical processing on the book web www.functional-programming.net.

The design of functional data-centric applications from the previous chapters relied heavily on functional data types, most importantly discriminated unions. These aren't particularly idiomatic in C#, so we mostly talked about F#. On the other side, using functions for representing simple behavior is perfectly possible in C# 3.0. Thanks to the `Func` delegate, which represents a function in C#, most of the examples you'll see in this chapter will be written in both C# and F#.

In this chapter, we'll use a single example that we'll keep extending to demonstrate the look and feel of behavior-oriented applications. We're going to develop an application for testing the suitability of a client for a loan offer. Let's now look at probably the simplest way.

8.1 Using collections of behaviors

In this section, we'll write several conditions for testing whether a bank should offer a loan to the client or not and we'll store these conditions in a collection. This way, it is very easy to add new conditions later during the development, because we would just implement the condition and add it to the collection. One of the key aspects of behavior-oriented programs is the ability to add new behavior easily.

8.1.1 Representing behaviors as objects

This time, we'll start with the C# version, because working with collections of behaviors in a functional way is supported in C# 3.0 to a similar extent as in F#. However, before we'll look at the functional version, it is useful to consider how the same pattern might be written using a purely object oriented style.

We would probably start by declaring an interface with a single method to execute the test and return whether or not it failed. In our loan example, a return value of `true` would indicate that the test suggests the loan offer should be rejected. Later we would implement the interface in several classes to provide concrete tests. Listing 8.1 shows the interface and a very simple implementation.

Listing 8.1 Loan suitability tests using object oriented style (C#)

```
interface IClientTest {
   bool Test(Client client);             #1
}
class TestYearsInJob : IClientTest {     #A
   public bool Test(Client client) {
      return client.YearsInJob < 2;      #2
   }
}
```

#1 Method that tests the client
#A Each test is represented by a single class
#2 Body of the concrete test

When working with tests implemented like this, we would create a collection containing elements of the interface type (#1) (for example `List<IClientTest>`) and then add an instance of each class implementing the interface to this collection. It is worth noting that we have to create a separate class for every test, even though the condition itself is just a simple expression (#2).

8.1.2 Representing behaviors as functions in C#

I mentioned earlier that an object-oriented way to understand a function is to think of it as an interface with a single method. If we look at the code from the previous listing, we can see that `IClientTest` is declared exactly like this. That means the test can easily be represented as just a simple function. In C#, we can write tests using lambda functions:

```
Func<Client, bool> testYearsInJob =
   client => client.YearsInJob < 2;
```

Instead of using the interface type, we now use a type `Func<Client, bool>`, which represents a function that takes the `Client` as an argument and returns a Boolean value. By writing the code in this fashion, we have significantly reduced the amount of boilerplate code around the expression that represents the test.

Just like we could store objects that implement some interface in a collection, we can also create a collection that stores function values and we'll look how to do this using the `List<T>` type in the listing 8.2. Note that we're creating a completely standard collection of objects - we can iterate over all the functions in the collection or change the collection later by adding or removing some of the function values.

When initializing the collection, we can easily write the code to specify the default set of tests in a single method. We can add the tests using lambda function syntax without the need to declare the functions in advance and we can also use C# 3.0 feature called *collection initializer* that makes the syntax even more concise.

Listing 8.2 Loan suitability tests using a list of functions (C#)

```
class Client {                                      #1
   public string Name { get; set; }
   public int Income { get; set; }
   public int YearsInJob { get; set; }
   public bool UsesCreditCard { get; set; }
   public bool CriminalRecord { get; set; }
}

static List<Func<Client, bool>> GetTests() {        #A
   return new List<Func<Client, bool>> {            #2
      cl => cl.CriminalRecord,                      #B
      cl => cl.Income < 30000,                      #B
      cl => !cl.UsesCreditCard,                     #B
```

```
        cl => cl.YearsInJob < 2                      #B
      };
    }
```

#1 Stores information about the client
#A Returns a list of tests
#2 Create new list using collection initializer
#B Several test checking loan suitability

The listing uses many of the new C# 3.0 features and thanks to them it is quite similar to the F# implementation we're about to write. First we declare a class to store information about the client using automatic properties (#1). Next, we implement a method that returns a collection of tests. The body of the method is just a single return statement that creates a new .NET List type and initializes its elements using *collection initializer* (#2). This allows you to specify the values when creating a collection in the same way as for arrays. Under the cover, this calls the `Add` method of the collection, just as we did in the previous example, but it is clearer.

The values stored in the collection are functions written using the lambda function syntax. Note that we don't have to specify the type of the `cl` argument. This is because the C# compiler knows that the argument to the `Add` method is the same as the generic type argument, which in our case is `Func<Client, bool>`.

LOADING BEHAVIORS USING REFLECTION

One frequent requirement for behavior-centric programs is the ability to load new behaviors dynamically from a library. For our application that would mean that someone could write a .NET class library with a type containing a `GetTests` method. This would return a list of tests just as in the earlier code; our program would call the method to get the tests at execution time, and then execute the tests without needing to know anything more about them.

This can be done using the standard .NET classes from the `System.Reflection` namespace that support dynamic loading an assembly and executing a method based on its name. The sample application for working with graphical filters supports this functionality, so you can find more examples online.

Now that we have a class for representing clients and a collection of tests that advises us whether to offer a loan to the client or not, we should also look how we can run the tests.

8.1.3 Using collections of functions in C#

When considering a loan for a client, we want to execute all the tests and count the number of tests that returned true (meaning a high risk). If the count is zero or one then the program will recommend the loan. The normal imperative solution would be to declare a variable and enumerate the tests using a `foreach` statement. In the body of the loop, we'd execute the test and increment the variable if it returned `true`. However, as you can see in listing 8.3, this can be implemented more elegantly using the LINQ extension method `Count`.

Listing 8.3 Executing tests (C#)

```
void TestClient(List<Func<Client, bool>> tests, Client client) {
   int issuesCount = tests.Count(f => f(client));               #1

   bool suitable   = issuesCount <= 1;                          #A
   Console.WriteLine("Client: {0}\nOffer a loan: {1}",          #A
      client.Name, suitable ? "YES" : "NO");                    #A
}

var john = new Client {                                         #B
      Name = "John Doe", Income = 40000, YearsInJob = 1,
      UsesCreditCard = true, CriminalRecord = false
   };
TestClient(GetTests(), john);                                   #C
```

#1 How many tests does the client fail?
#A Print the results of testing
#B Create client using object initializer
#C Offer a loan to the client?

In functional terminology, `Count` is a higher order function. It takes a predicate as an argument and counts the number of elements for which the predicate returns `true`. We're using it to count how many tests consider the client to be unsuitable for a loan (#1). The element of the collection in our case is a function, so our predicate has to take a function and return a Boolean. The lambda function we wrote executes the function passed as *its* parameter, specifying it the client as the argument, and simply returns the result of the test as the predicate result. Once we count the tests that failed, calculating and printing the result is easy. Describing how it works (even in this relatively simple case) is complicated, but if you think about what you're trying to do with each element, it's not that hard to understand.

I mentioned earlier that the F# version of the example will be essentially the same. This is because all the necessary features such as higher order functions, lambda functions and the ability to store functions in a collection are now available in C# 3.0 as well. Let's see what the F# code looks like.

8.1.4 Using lists of functions in F#

First of all, we'll declare a type to represent information about the client. A client has quite a lot of properties, so the most natural representation will be an F# record type that we've seen in the previous chapter. You can see the type declaration and a code to create sample client in the listing 8.4

Listing 8.4 Client record type and sample value (F# interactive)

```
> type Client =                                                 #A
     { Name : string; Income : int; YearsInJob : int
```

```
       UsesCreditCard : bool; CriminalRecord : bool };;
type Client = (...)

> let john =                                                #B
     { Name = "John Doe"; Income = 40000; YearsInJob = 1
       UsesCreditCard = true; CriminalRecord = false };;
val john : Client
```
#A Declare 'Client' as an F# record type
#B Create a value of the 'Client' type

There's nothing new here - we're just declaring a type and creating an instance of it. To make the listing a bit shorter, I haven't used a separate line for each property, either when declaring the type or when creating the value. This is entirely valid F#, but we have to add semicolons between the properties. In the light-weight syntax, the compiler adds them automatically at the end of the line (when they are needed), but they have to be written explicitly when the line breaks aren't there to help the compiler.

Listing 8.5 completes the example. First it creates a list of tests and then decides whether or not to recommend offering a loan to the sample client (John Doe) from the previous listing.

Listing 8.5 Executing tests (F# interactive)

```
> let tests =                                                         #1
     [ (fun cl -> cl.CriminalRecord = true);
       (fun cl -> cl.Income < 30000);
       (fun cl -> cl.UsesCreditCard = false);
       (fun cl -> cl.YearsInJob < 2) ];;
val tests : (Client -> bool) list                                     #2

> let testClient(client) =
     let issues =  tests |> List.filter (fun f -> f client)           #3
     let suitable = issues.Length <= 1                                #A
     printfn "Client: %s\nOffer a loan: %s (issues = %d)" client.Name  #A
             (if (suitable) then "YES" else "NO") issues.Length;;     #A
val testClient : Client -> unit

> testClient(john);;
Client: John Doe
Offer a loan: YES (issues = 1)
```
#1 Create a list of tests
#2 Inferred signature of the list
#3 Filter tests and get a list of issues
#A Count the issues and print the result

This uses the normal syntax for creating lists to initialize the tests (#1) and the tests are written using lambda function syntax. Interestingly, we don't have to write any type annotations and F# still infers the type of the list correctly (#2). F# type inference is smart enough to use the names of the accessed members in order to work out which record type we want to use.

In the C# version, we used the `Count` method to calculate the number of the tests that failed. F# doesn't have an equivalent function; we could either implement it, or combine other standard functions to get the same result. We've taken the second approach in this case. First we get a list of tests that considered the client to be unsafe; these are the tests which return `true` using `List.filter` (#3). Then we get the number of issues using the `Length` property.

Point-free programming style

We've seen many examples where we don't have to write lambda function explicitly when calling a higher order function, so you may be wondering whether this is possible in the previous listing as well. This way of writing programs is called "point-free", because we're working with data structure that contains values (for example a list), but we never assign any name to the value ("point") from that structure. Let's demonstrate this using a couple of examples that we've seen already:

```
[1 .. 10] |> List.map ((+) 100)
places |> List.map (snd >> statusByPopulation)
```

In the first case, we're working with collection of numbers, but there is no symbol that would represent values from the list. The second case is similar, except we're working with list of tuples. Again, there are no symbols that would represent either the tuple or any element of the tuple.

The point-free style is possible thanks to several programming techniques. The first line uses partial function application, which is a way to create a function with the required number of parameters based on a function with larger number of parameters. In our example, we also treat an infix operator (plus) as an ordinary function. The second line uses function composition, which is another important technique for constructing functions without explicitly referencing the values that the functions work with.

Now, let's look how we could rewrite the example from listing 8.5. First of all, we'll rewrite the lambda function to use pipelining operator:

```
Instead of:  (fun f -> f client)
We'll write: (fun f -> client |> f)
```

These two functions mean exactly the same thing. We're almost finished now, because the pipelining operator takes the client as the first argument and a function as the second argument. If we use partial application to specify just the first argument ('client'), we'll obtain a function that takes a function ('f') as an argument and applies it to the 'client':

```
tests |> List.filter ((|>) client)
```

Point-free programming style should be always used wisely. Even though it makes the code more succinct and elegant, it may be harder to read and the reasoning that I've demonstrated here isn't trivial. However, the point-free style is important for some areas of functional programming and in chapter 12 we'll see how it can be very useful when developing a domain-specific language.

In this section, we've seen how to design and work with a basic behavior-oriented data structures - a list of functions - in both C# and F#. We've also seen a common functional technique called point-free programming. In the next section, we'll continue talking about common practices as we look at two object-oriented design patterns and related functional constructs.

8.2 Idioms for working with functions

In the previous chapter, we talked about data structures and several related design patterns. We've seen two examples of structural patterns that are related to the problem of designing functional data structures. We've also seen one behavioral pattern that describes how objects communicate, which corresponds to how functions call each other in functional terminology.

In this chapter, we're talking about behavior-oriented applications, so it seems natural that the relevant patterns will be behavioral ones. The first one is called the *strategy* pattern and is surprisingly simple from a functional point of view.

8.2.1 The strategy design pattern

The *strategy* pattern is useful if the application needs to choose between several algorithms or parts of an algorithm at run-time. One of the common situations is for example when several tasks that our application needs to perform differ only in one smaller subtask. Using the strategy pattern, we can write the common part of the task just once and parameterize it by giving it the subtask (primitive operation) as an argument. Figure 8.1 shows an object-oriented representation of the strategy pattern.

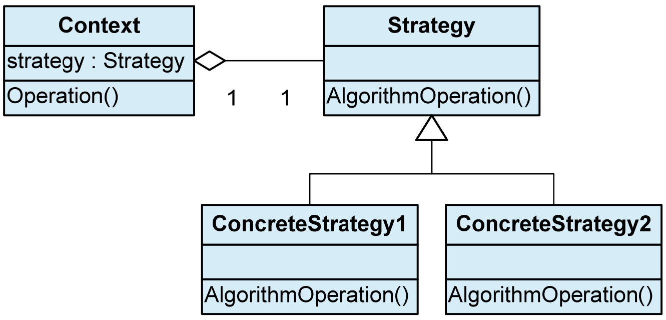


Figure 8.1 'Strategy' is an interface with a method representing the primitive operation. Two concrete strategies implement that operation differently and the 'Context' class can choose between the implementations.

The idea of "parameterizing a task by giving it subtask as an argument" has probably made it fairly clear what the strategy pattern looks like in functional programming: it's just a higher order function. The `Strategy` interface from the previous diagram has a single method which suggests that it is just a simple function; the two classes that implement it are effectively just concrete functions that can be created using lambda functions.

In a language that supports functions, we can replace the `Strategy` interface with the appropriate function (a `Func` delegate in C# or a function type in F#). Usually, we also don't need to store the strategy in a local field of the `Context` class: instead, we pass it directly to the `Operation` method as an argument. Using the abstract names from the previous diagram, we could write:

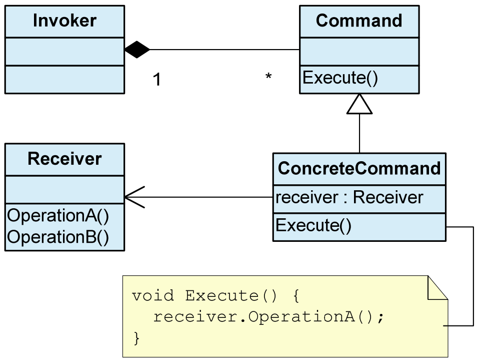
```
Context.Operation(arg => {
        // concrete strategy #1
    });
```

We've already seen a practical example of this pattern when filtering a list. In this case, the function that specifies the predicate is a concrete strategy (and we various different strategies to write different filters) and the `List.filter` function or the `Where` method is the operation of the context. This means that in a language that supports higher order functions, you can always replace the strategy pattern with a higher order function.

Our next pattern is somewhat similar, but more related to our earlier discussion of behavior-centric applications that work with a list of behaviors.

8.2.2 The command design pattern

The *command* pattern describes a way to represent actions in an application. As opposed to the previous pattern, which is used to parameterize a known behavior (e.g. filtering of a list) with a missing piece (predicate), the command pattern is used to store some "unit of work" that can be invoked at some later point in time. We often see collections of commands that specify steps of some process or operations that the user can choose from. If you look at figure 8.2, you'll quickly recognize an interface which looks like a good candidate for being replaced with a single function.



```
void Execute() {
  receiver.OperationA();
}
```

Figure 8.2 'Invoker' stores a collection of classes implementing the 'Command' interface. When invoked, the concrete command uses a 'Receiver' object, which usually carries and modifies some state.

The type that can be easily replaced with a function is the `Command` interface. Again, it has just a single method, which acts as a hint. The classes that implement the interface (such as `ConcreteCommand`) can be turned into a functions, either constructed using lambda function syntax or written as ordinary functions when they are more complex.

I mentioned that the difference between the command and strategy patterns is that the `Invoker` works with a list of commands and executes them as and when it needs to. This is very similar to the "client loan" example. We had a collection of tests for checking the suitability of the client. However, instead of declaring the `Command` interface, our functional version used the `Func<Client, bool>` delegate in C# and a function type `Client -> bool` in F#. The invoker was the `TestClient method`, which used the tests to check a client.

RECEIVER COMPONENT AND MUTABLE STATE

Figure 8.2 also shows a `Receiver` class; I explained that it usually represents some state that is changed when the command is invoked. In a typical object-oriented program, this might be a part of the application state. For example in a graphical editor, we could use commands to represents undo history. In that case, the state would be the picture on which the undo steps can be applied.

This is not the way in which you would use the pattern in a functional programming. Instead of modifying state, the command usually returns some result (such as the Boolean value in our client checking example). In purely functional programming, the `Receiver` can be a value captured by the lambda function.

Although mutable state should usually be avoided in functional programming, there is one example where it is useful, even in F#. We'll see that a technique similar to the command pattern can help us to hide the state from the outside world, which is important if we still want to keep most of the program purely functional. First look at a similar idea in C# and then study the usual implementation using lambda functions in F#.

CAPTURING STATE USING THE COMMAND PATTERN IN C#

As I've explained, the command pattern often works with mutable state, encapsulated in something like the `Receiver` class of our example. Listing 8.6 shows an example of this, creating a more flexible income test for our financial application. The goal is to allow the test to be configured later without updating the collection of tests.

Listing 8.6 Income test using the command pattern (C#)

```
class IncomeTest {                                                #1
   public int MinimalIncome { get; set; };                        #2
   public IncomeTest() {
      MinimalIncome = 30000;
   }
   public bool TestIncome(Client client)  {                      #3
      return client.Income < minimalIncome;
   }
}

// Usage of 'IncomeTest' later in the program
IncomeTest incomeTst = new IncomeTest();                            #B
Func<Client, bool> command = cl => incomeTst.TestIncome(cl);    #4
tests.Add(command)                                                  #C
```

#1 Corresponds to the 'Receiver' class
#2 Publicly accessible mutable state
#3 Operation used by the 'Command'
#B Create 'Receiver' with the state
#4 Create the 'Command' as a lambda function
#C Add command to the list of tests ('Invoker')

We start by creating a class that carries the mutable state and corresponds to the `Receiver` component from the Command design pattern (#1). The state is a recommended minimal income and the class has a method for modifying it (#2). The next method implements the test itself (#3) and compares whether the income of the given client is larger than the current minimal value stored in the test.

The later part of the listing shows how we can create a new test. First we create an instance of the `IncomeTest` class containing the state and then we create a lambda function that calls its `TestIncome` method (#4). This function corresponds to the `Command` component and we add it to the collection of tests. We can later configure the test using the `SetMinimalIncome` method. Listing 8.6 creates the function explicitly with lambda syntax, just to demonstrate that it corresponds to the design pattern, but we can write it more concisely:

```
IncomeTest incomeTst = new IncomeTest();
tests.Add(incomeTst.TestIncome);
```

The C# compiler automatically creates a delegate instance that wraps the `TestIncome` method and can be added to the collection if the method has the right signature. Now that we've added the test to the collection, we can see how it behaves:

```
TestClient(tests, john);            #A
incomeTst.SetMinimalIncome(45000);
TestClient(tests, john);            #B
```

#A Result is YES
#B Result is NO

This is a common pattern which is widely used in imperative object-oriented programming. From a functional point of view, it should be used carefully: the code and comments should clearly document what calls can affect the mutable state. In the previous example, the state is modified using the `incomeTst` object and this is the reason why the same line of code can give different results when called at different times. In the next section, we'll look how to implement similar functionality in a simpler way using F#.

8.2.2 Capturing state using closures in F#

In this section we're going to talk about *closures*, which is an important concept in functional programming. Closures are very common and most of the time they aren't used with mutable state. However, working with mutable state is sometimes needed for the sake of pragmatism and closures give us an excellent way to limit the scope of the mutable state.

First let's look at a very simple piece of F# which we saw in chapter 5:

```
> let createAdder num =
     (fun m -> num + m)
val createAdder : int -> (int -> int)
```

When we were discussing this example earlier, we didn't see any difference between a function written like this and a function called `add` taking two parameters and returning their sum. This is because we can call the `add` function with a single argument: thanks to partial application, the result is a function that adds the specified number to any given argument.

If you analyze what is returned in the previous example, it isn't just the code of the function! The code is just a bunch of instructions that add two numbers, but if we call `createAdder` twice with two different arguments the returned functions are clearly different, because they're adding different numbers. The key idea is that a function isn't just code, but also a *closure* which contains the values that are used by the function, but aren't declared inside its body. The values held by the closure are said to be *captured*. In the previous example, the only example of capturing is the `num` parameter.

Of course, we've been using closures when creating functions since we started talking about lambda functions. We didn't talk about them explicitly, because usually you don't need to think about them–they just work. However, what if the closure captures some value that can be mutated?

MUTABLE STATE USING REFERENCE CELLS

In order to answer this question, we'll need to be able to create some mutable state to be capture. We can't do that with `let mutable`, because that kind of mutable value can be used only locally–it can't be captured by a closure.

The second way to create mutable values is using a type called `ref`, which is a shortcut for a *reference cell*. Put simply, this is a small object (actually declared as an F# record type) that contains a mutable value. To understand how the `ref` type works, we can look how we could define exactly same type in C#. As you can see, it's fairly simple:

```
class Ref<T> {
   public Ref(T value) { Value = value; }
   public T Value { get; set; }
}
```

The important point about the type is that the `Value` property is mutable, so when we create an immutable variable of type `Ref<int>`, we can still mutate the value it represents. The listing 8.7 shows an example of using reference cells in F# and also shows the corresponding code using C# type `Ref<T>`. In F#, we don't access the type directly, because there is a function–again called `ref`–that creates a reference cell, along with two operators for setting and reading its value.

Listing 8.7 Working with reference cell in F# and C#

```
let st = ref 10                                var st = new Ref<int>(10);

st := 11                  #A                   st.Value = 11;
printfn "%d" (!st)        #B                   #A
                                               Console.WriteLine(st.Value);
                                               #B
```

#A Modify the value of reference cell
#B Prints 11

On the first line, we create a reference cell containing an integer. Just like the `Ref<T>` type we've just declared in C#, the F# `ref` type is generic, so we can use it to store values of any type. The next two lines demonstrate the operators which work with reference cells - assignment (`:=`) and dereference (`!`). The operators correspond to setting or reading value of the property, but give us a more convenient syntax.

CAPTURING REFERENCE CELLS IN A CLOSURE

Now we can write code that captures mutable state created using a reference cell in a closure. Listing 8.8 shows an F# version of the configurable income test. We create a `createIncomeTests` function that returns a tuple of two functions: the first changes the minimal required income and the second is the test function itself.

Listing 8.8 Configurable income test using closures (F# interactive)

```
> let createIncomeTest() =
      let minimalIncome = ref 30000                                #1
      (fun (newMinimal) ->
         minimalIncome := newMinimal),                             #A
      (fun (cl) ->
         cl.Income < (!minimalIncome))                             #B
val createIncomeTest : unit -> (int -> unit) * (Client -> bool)    #2

> let setMinimalIncome, testIncome = createIncomeTest()            #3
val testIncome : (Client -> bool)
val setMinimalIncome : (int -> unit)

> let tests = [ testIncome; (* more tests... *) ]                  #C
val tests : (Client -> bool) list
```

#1 Declare local mutable value
#A Set new minimal income
#B Test client using the current minimal income
#2 Returns a tuple of functions
#3 Create functions for setting and testing income
#C Store testing function in a list

Let's look at the signature of the `createIncomeTest` function (#2) first. It doesn't take any arguments and returns a tuple of functions as a result. In its body, we first create a mutable reference cell and initialize it to the default minimal income (#1). The tuple of functions to be returned is written using two lambda functions and both of them use the `minimalIncome` value. The first function (with signature `int -> unit`) takes a new income as an argument and modifies the reference cell. The second one compares the income of the client with the value stored by the reference cell and has the usual signature of a function used to testing a client (`Client -> bool`).

When we later call `createIncomeTest` (#3), we get two functions as a result. However, we created only one reference cell, which means that it is shared by the closures of both of the functions. We can use `setMinimalIncome` to change the minimal income required by the `testIncome` function.

Before moving to the next example, let's look at the analogy between the F# version and the command pattern with the C# implementation discussed earlier. The most important difference is that in F#, the state is automatically captured by the closure while in C# it was encapsulated in an explicitly written class. In some senses, the tuple of functions and the closure correspond to the receiver object from object-oriented programming. In fact, the F# compiler handles the closure by generating .NET code that is very similar to what we explicitly wrote in C#. The intermediate language used by .NET doesn't directly support closures, but it of course has classes for storing state.

Listing 8.9 completes the example, demonstrating how to modify the test using the `setMinimalIncome` function. The example assumes that the `testClient` function now uses the collection of tests declared in the previous listing.

Listing 8.9 Changing minimal income during testing (F# interactive)

```
> testClient(john);;
Client: John Doe, Offer a loan: YES

> setMinimalIncome(45000);;
val it : unit = ()

> testClient(john);;
Client: John Doe, Offer a loan: NO
```

Just as in the C# version, we first test the client using the initial tests (which the client passes) and then modify the income required by one of the tests. After this change, the client no longer fulfils the conditions and the result is negative.

Closures in C#

In the last section I used C# for writing object-oriented code and F# for writing functional code. This is because I wanted to demonstrate how the two concepts relate - that is how closures are similar to objects and in particular to the `Receiver` object in the Command design pattern.

However, closures are essential for lambda functions and the lambda expression syntax in C# 3.0 supports the creation of closures too. In fact, this was already present in C# 2 in the form of anonymous methods. The following example shows how to create a function which, when called several times, will return a sequence of numbers starting from zero:

```
Func<int> CreateCounter() {
   int num = 0;
   return () => { return num++; };
}
```

The variable `num` is captured by the closure and every call to the returned function increments its value. In C#, variables are mutable by default, so you should be extremely careful when you change the value of a captured variable like this. A common source of confusion is capturing the loop variable of a `for` loop. If you capture it during every iteration, at the end all of the closures will contain the same value, because we're working just with a single variable.

In this section, we talked about object oriented patterns and related functional techniques. In some cases, we used a function instead of an interface with a single method. In the next section, we'll look at an example showing what we can do when the behavior is still very simple, but cannot be described by just one function.

8.3 Working with composed behaviors

In this chapter, we're talking about applications or components that work with behaviors and allow new behaviors to be added later in the development cycle or even at run-time. The key design principle is to make sure that adding new behaviors is as easy as possible. After we implement the new functionality, we should be able to register the function (for example by adding it to a list) and use the application without any other changes in the code.

To simplify the implementation, it is better to minimize a number of functions that need to be implemented. Often, a single function is sufficient to represent the functionality, but in some cases it may not be enough; we may need to add some additional information or provide a few more functions. Of course, in a functional program another function *is* just 'additional information'. It is just information we can *run* to provide richer feedback.

An example of the first case may be a filter in a graphical editor. The filter itself is a function that works with pictures, but we could also provide a name of the filter (as a string). After all, the user of the editor would rather see a "friendly" name and description than whatever we happened to call our function, with all the inherent naming restrictions.

In the next section, we're going to look at the second case described above, where more functions are required. We'll improve our loan application, so that a test can report the details of why it is recommending against a loan, if the client "fails" the test. This will be implemented using a second function that does the reporting.

8.3.1 Records of functions

We've actually seen a way of working with multiple functions already. In the previous example, we returned a tuple of functions as a result. We could use the same technique to represent our application with the new reporting feature as well. Let's say that the reporting function takes the client, prints something to the screen and returns a unit as a result. Using this representation, the type of the list of behaviors would be:

```
((Client -> bool) * (Client -> unit)) list
```

This starts to look a bit scary. It's quite complicated and the functions don't have names, which makes the code less readable. In the previous example, it wasn't a big problem, because the function was used only locally, but this list is one of the key data structures of our application, so it should be as clear as possible. A simple solution that makes the code much more readable is to use a record type instead of a tuple. We can define it like this:

```
type ClientTest =
  { Check  : Client -> bool
    Report : Client -> unit }
```

This defines a record with two fields, both of which are functions. This is just another example of using functions in the same way as any other type. The declaration resembles a very simple object (or interface), but we'll talk about this similarity later. Before that, let's look at listing 8.10, which shows how we can create a list of tests represented using the record declared above.

Listing 8.10 Creating tests with reporting (F#)

```
let checkCriminal(cl) = cl.CriminalRecord = true                          #A
let reportCriminal(cl) =                                                  #A
   printfn "Checking 'criminal record' of '%s' failed!" cl.Name           #A

let checkIncome(cl) = cl.Income < 30000                                   #B
let reportIncome(cl) =                                                    #B
   printfn "Checking 'income' of '%s' failed (%s)!"                       #B
           cl.Name "less than 30000"                                      #B

let checkJobYears(cl) = cl.YearsInJob < 2                                 #C
let reportJobYears(cl) =                                                  #C
   printfn "Checking 'years in the job' of '%s' failed (%s)!"             #C
           cl.Name "less than 2"                                          #C

let testsWithReports =                                               #1
  [ { Check = checkCriminal; Report = reportCriminal };
    { Check = checkIncome;   Report = reportIncome };
    { Check = checkJobYears; Report = reportJobYears };
    (* more tests... *) ]
```

#A Checking and reporting for criminal record
#B Check requires minimal income
#C Check requires years in the current job
#1 Create a list of records

The listing is simply a series of let bindings. To make the code more readable, we haven't used lambda functions this time; instead we've define all the checks as ordinary F# functions. For each test, we've defined one function with the prefix "check" and one with the prefix "report". If you enter the code in F# interactive, you can see that the function types correspond to the types from the `ClientTest` record type. The last operation is creating a list of tests (#1). We just need to create a record for each test to store the two related functions, and create a list containing the record values.

We also need to update the function that tests a particular client. We'll first find those tests that fail (using the `Check` field) and then let them print the result (using the `Report` field). Listing 8.11 shows the modified function, as well as the output when we run it against our sample client.

Listing 8.11 Testing a client with reporting (F# interactive)

```
> let testClientWithReports(client) =
     let issues =                                                   #1
        testsWithReports                                            #1
        |> List.filter (fun tr -> tr.Check(client))                 #1
     let suitable = issues.Length <= 1                              #A
     for i in issues do                                             #2
        i.Report(client)                                            #2
     printfn "Offer loan: %s" (if (suitable) then "YES" else "NO")
  ;;
val testClientWithReports : Client -> unit

> testClientWithReports(john);;
Years in the job of 'John Doe' is less than 2!
Offer loan: YES
```

#1 Get a list of tests that failed
#A Calculate overall result
#2 Report all found issues

The `testClient` function has only changed slightly since listing 8.5. The first change is in the lines that select which tests have failed (#1). The list now a collection of records, so we have to test the client using a function stored in the `Check` field. The second change is that earlier, we were interested only in a number of failing tests. This time, we also need to print the detailed information about the failure (#2). This is implemented using an imperative `for` loop which invokes `Report` function of all the failing tests.

One problem in the current version of the code is that we had to write very similar functions when creating some tests. Let's fix that, reducing unnecessary code duplication.

8.3.1 Building composed behaviors

In listing 8.10 there is some obvious code duplication in the testing and reporting functions that verify the minimal income and minimal years in the current job. This is because the tests have a very similar structure: both of them test whether some property of the client is smaller than a minimal allowed value.

Identifying commonality is only the first step towards removing duplication. Then next one is to look at which parts of the `checkJobYears` and `checkIncome` functions (together with their reporting functions) are *different*:

38) They check different properties

39) They use different minimal values

40) They have slightly different messages

To write the code more succinctly, we can create a function that takes these three different parts as its arguments and returns a `ClientTest` record. When we create the list of tests, we'll simply call this new function twice with different arguments to create two similar tests. Listing 8.12 shows both the extra function (`lessThanTest`) and the new way of creating the list of tests.

Listing 8.12 Creating similar tests using a single function (F# interactive)

```
> let lessThanTest f min property =
     let report cl =                                                   #1
        printfn "Checking '%s' of '%s' failed (less than %d)!"         #1
                property cl.Name (f(cl)) min                           #1
     { Test = (fun cl -> f(cl) < min)                                  #2
       Report = report };;
val lessThanTest : (Client -> int) -> int -> string -> ClientTest      #3

> let tests =
     [ (lessThanTest (fun cl -> cl.Income) 30000 "income")             #A
       (lessThanTest (fun cl -> cl.YearsInJob) 2 "years in the job")   #A
       (* more tests... *) ];;
val tests : ClientTest list
```

#1 Nested reporting function
#2 Compare actual value with the minimal value
#3 Function signature
#A Creates two similar tests with reporting

As usual, the type signature (#3) tells us a lot about the function. The `lessThanTest` function returns a value of type `ClientTest`, which contains the testing and reporting functions. The test is built using several arguments. The first reads a numeric property of the client, and the second specifies a minimal required value (in our case representing either an income or a number of years). The final argument is a description of the property, used in the reporting test.

The code first declares a nested function called `report` (#1), which takes a `Client` as the argument and prints a reason why the test failed. Of course, the function uses the arguments of the `lessThanTest` function as well. This means that when `report` is later returned as a part of the result, all these arguments will be captured in a closure. When constructing a record value that will be returned (#1), we specify `report` as one of the function values and the second one is written inline using lambda function.

Working with tuples or records of functions is common in functional programming and it reflects the F# development style, but in C#, we'd probably use a different approach to implement this example. Let's look back at the development process and also think how we would implement the example in C# and improve the current F# version.

8.3.2 Further evolution of F# code

In the last section, we moved from a simple piece of F# that stored a list of functions to a more sophisticated version that uses a list of records. This is a part of the F# programming style that we talked about in chapter 1. I wrote that F# programs often start as very simple scripts and then evolve into robust code that follows standard .NET programming guidelines and benefits from the .NET object model.

We started with the most straightforward way to solve the problem using only what we knew at the beginning. When we later realized that we needed to add reporting using another function, we made some relatively small adjustments to the code (because this is quite easy to do in F#) resulting in a version with more features. However, the transition was not just in terms of features, but also in a sense of robustness and maintainability.

When extending the initial version, I mentioned that we could have used a list containing tuples of functions. Representations like this are more likely to be used in the initial prototype than in a finished application, and using F# record types clearly make the code more readable. Even though we went straight to a record type, it's worth bearing in mind that there's nothing wrong with using a simple representation when you start writing an application that should turn into a complex product. This kind of change is quite easy to make in F#, and when you develop an initial version, you usually want to get it running with useful features as soon as possible rather than writing it in a robust way.

Even though we have already made a few transitions on the road to the robust version of the application, there are still some improvements that are left to consider. Because F# is a language for .NET, we can use several object-oriented features to make the code more .NET-friendly. We'll return to this topic in the next chapter where we'll see how to turn our record into an F# *abstract object type*, which corresponds to C# interfaces.

COMPOSED BEHAVIORS IN C#

We started this chapter with an example of C# code that declared an interface with a single method representing the test, but then we used functions (and the Func delegate) as a more convenient way to write the code. If we wanted to implement a program that works with two functions, as we now have in F#, we'd probably turn back to interfaces. Using interfaces in C# is definitely more convenient and more reasonable than using a tuple or a class with functions as its members. Having said that, in C# we only have two options - for simple behaviors, we can use functions, but anything more complicated has to be written using an interface.

In F#, the transition between the various representations is easier. Most importantly, thanks to the type inference we don't have to change the types everywhere in the source code. Also, turning a lambda function into a class is a larger change then just adding another function. Even when using abstract object types in F#, there is an easy way to turn a lambda function into something you can think of as a "lambda object". The actual name for this feature is *object expression* and we'll talk about it in the next chapter.

In this chapter, we've primarily talked about behavior-centric applications, but in the introduction I explained that data-centric and behavior-centric approaches are often used together. We're going to see that in action now, combining functions with the discriminated union type which was so important for representing data in chapter 7.

8.4 Combining data and behaviors

Our original algorithm for testing whether a client is suitable for a loan offer used only the count of the tests that failed. This isn't very sophisticated: if the client has large income, we may not be interested in some other aspects, whereas for a client with a smaller income the bank may want to ask several additional questions. In this section, we're going to implement a simple but powerful algorithm using *decision trees* and we'll also look at the declaration of a more interesting F# data structure.

8.4.1 Decision trees

Decision trees are one of the most popular algorithms in machine learning. They can be used for making decisions based on some data or for classifying input into several categories. The algorithm works with a tree that specifies what properties of the data should be tested and what to do for each of the possible answers. The reaction to the answer may be another test or the final answer.

Machine learning theory provides sophisticated ways for building the tree automatically from the data, but for our example we'll create the tree by hand. The figure 8.3 shows a decision tree for our problem.

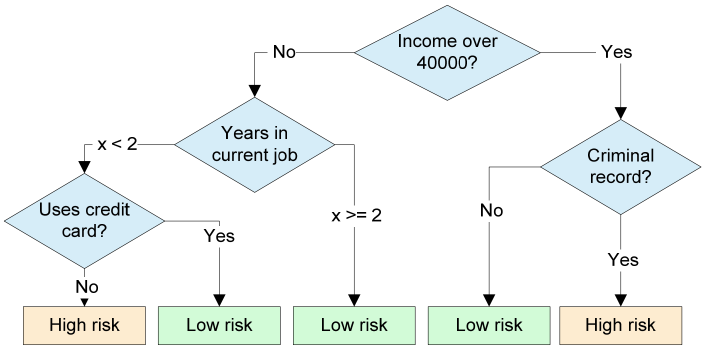


Figure 8.3 Decision tree for testing loan suitability; each diamond represents a question to ask and the links are possible answers that lead to another question or to a conclusion (shown in a rectangle).

We're going to start by implementing the F# version. In F#, it is usually very easy to write the code if we have an informal specification of the problem - in this case a data structure to work with. The specification for a decision tree could look like this:

DECISION TREE

A decision tree is defined by an initial query which forms the root of the tree. A query consists of the name of a test and a function that executes it and can return several possible answers. In our implementation, we'll limit the answer to just `true` or `false`. For each answer, the node also contains a link to the next query or the final decision for this path through the tree.

Equipped with this specification, we can now start writing the F# code. We'll look how to implement key parts of the problem in C# later: first I'd like to demonstrate how easy it is to rewrite a specification like this into F#.

8.4.1 Decision trees in F#

Let's now look what information we can gather from the specification. From the last sentence we can see that a link leads either to a query or to a final result. In F#, we can directly encode this using a discriminated union type with two options. The specification also talks about the query in more detail - it contains various fields, so we can represent it as an F# record type.

We'll define an F# record type (`QueryInfo`) with information about the query and a discriminated union (called `Decision`) which can be either another query or a final result. An interesting observation about these data types is that they both reference each other. In functional terminology, we'd say that the types are *mutually recursive*. Listing 8.13 shows what this means for the F# source code.

Listing 8.13 Mutually recursive types describing decision tree (F#)

```
type QueryInfo =                    #1
   { Title    : string
     Check    : Client -> bool      #2
     Positive : Decision            #3
     Negative : Decision }          #3

and Decision =                      #4
   | Result of string
   | Query  of QueryInfo            #B
```

#1 Declare first type using 'type' keyword
#2 Member representing the behavior
#3 References to the second type
#4 Make the declaration recursive using 'and' keyword
#B Reference to the first type

When writing type declarations in F#, we can only refer to the types declared earlier in the file (or in a file specified earlier in the compilation order or located higher in the Visual Studio solution). Obviously that's going to cause problems in this situation, where we want to define two types that reference each other. To get around this, F# includes the `and` keyword. The type declaration in the listing starts as usual with the `type` keyword (#1), but it continues with and (#4) which means that the two types are declared simultaneously and can see each other.

The QueryInfo declaration combines data and behavior in a single record. The name of the test is a simple data member, but the remaining members are more interesting. The Check member (#2) is a function (that is, a behavior). It can return a Boolean value which we'll use to choose one of the two branches to continue with (#3). These branches are composed values that may store a string or can recursively contain other QueryInfo values, so they can store both data and behavior. Note that we could return Decision value as a result from the function, but then we couldn't that easily report whether the checking failed or not - we'd only know what the next test to run is. In listing 8.14 we create a value representing the decision tree shown in figure 8.3.

Listing 8.14 Decision tree for testing clients (F#)

```
let rec tree =                                                  #1
   Query({ Title = "More than $40k"
           Check = (fun cl -> cl.Income > 40000)
           Positive = moreThan40; Negative = lessThan40 })
and moreThan40 =                                                #2
   Query({ Title = "Has criminal record"
           Check = (fun cl -> cl.CriminalRecord)
           Positive = Result("NO"); Negative = Result("YES") })
and lessThan40 =                                                #3
   Query({ Title = "Years in job"
           Check = (fun cl -> cl.YearsInJob > 1)
           Positive = Result("YES"); Negative = usesCredit })
and usesCredit =                                                #4
   Query({ Title = "Uses credit card"
           Check = (fun cl -> cl.UsesCreditCard)
           Positive = Result("YES"); Negative = Result("NO") })
```

#1 Root node on level 1
#2 First option on the level 2
#3 Second option on the level 2
#4 Additional question on level 3

There is one new thing about the listing 8.14 that we haven't seen before. When declaring values, we're using the rec keyword in conjunction with the new and keyword. This is not exactly the same use of the keyword as when we declared two *types* together in the previous listing, but the goal is similar. The and keyword allows us to declare several *values* (or functions) that reference each other. For example, this is how we can use the value moreThan40 (#2) in the declaration of tree (#1), even though it is declared later in the code.

The declaration order is the main reason for using let rec in this example, because we can start from the root node of the tree (#1), then create values for the two possible options on the second level (#2, #3) and finally declare one additional question for one case on the third level. We used let rec earlier for declaring recursive *functions*, which are functions that call themselves from their body (before they are declared). In general, F# also allows the declaration of recursive *values*, which can simplify many common tasks.

Initialization using recursive let bindings

We've already seen several examples of recursive functions, but what would recursive value look like? One example might be some code to create a user interface using Windows Forms. Using a simplified API, it could look like this:

```
let rec form = createForm "Main form" [ btn ]
and btn = createButton "Close" (fun () -> form.Close())
```

The first line creates a form and gives it a list of controls to be placed on the form as the last argument. This list contains a button, which is declared on the second line. The last argument to the `createButton` function is a lambda function that will be invoked when the user clicks on the button. It should close the application, so it needs to reference the `form` value, which is declared on the first line.

You may be wondering what's so difficult about this - after all, we could easily write code to do the same thing in C#, and we wouldn't think of it as being particularly recursive. However, in C# we'd be adding an event handler to the button after creating the form, or adding the button to the form after creating it - either way, we're mutating the objects. It's easy for two values to refer to each other via mutation, but the tricky bit comes when you want to make the values immutable.

Using recursive let bindings we can create values that reference other values and the whole sequence is declared at once. However, even recursion has its limitations. Consider the following code snippet:

```
let rec num1 = num2 + 1
and num2 = num1 + 1
```

In this case, we'd have to evaluate `num1` in order to get the value of `num2`, but to do this we'd need a value of `num1`. The difference that made the first example correct is that the value `form` was used inside a lambda function, so it wasn't needed immediately. Luckily, the F# compiler can detect code like this which can't *possibly* work, and generates a compilation error.

We've seen how to declare a record that mixes data with behaviors and how to create a value of this record type using lambda functions. In listing 8.15, we'll finish the example by implementing a function that tests the client using a decision tree.

Listing 8.15 Recursive processing of the decision tree (F# interactive)

```
> let rec testClientTree(client, tree) =                             #1
     match tree with
     | Result(msg) ->                                                #2
        printfn "  OFFER A LOAN: %s" msg
     | Query(qi) ->                                                  #3
        let s, case = if (qi.Check(client)) then "yes", qi.Positive  #A
                      else "no", qi.Negative
        printfn "  - %s? %s" qi.Title s
        testClientTree(client, case);;                                  #4
val testClientTree : (Client * Decision) -> unit

> testClientTree(john, tree);;                                          #B
  - More than $60k? no
  - Years in job? no
  - Uses credit card? yes
  OFFER A LOAN: YES
val it : unit = ()
```

#1 Recursive function declaration
#2 The case with the final result
#3 The case containing a query
#A A choice depending on the test result
#4 Recursive call on the selected sub-tree
#B Test the code interactively

The program is implemented as a recursive function (#1). The decision tree can be either a final result (#2) or another query (#3). In the first case, it just prints the result. The second case is more interesting. It first runs the test and chooses one of the two possible sub-trees to process later based on the result. It then reports the progress to the console, and finally calls itself recursively to process the sub-tree (#4). In the listing, we also immediately test the code and as you can see which path in the decision tree the algorithm followed for our sample client.

In this section, we’ve developed a pure functional decision tree in F#. As we’ve seen before, rewriting some functional constructs (particularly discriminated unions) in C# can be quite difficult, so in the next section we'll implement a similar solution by mixing object-oriented and functional style in C# 3.0.

8.4.2 Decision trees in C#

In chapter 5 we discussed the relationship between discriminated unions in F# and class hierarchies in C#. In this example, we'll use another class hierarchy to represent a node in a decision tree, deriving two extra classes to represent the two different cases (a final result and a query).

In the functional version, all the processing logic was implemented separately in the `testClientTree` function. Even though we can do this in object-oriented style too, for example using the Visitor pattern (as discussed in chapter 7), that isn't a particularly object-oriented solution. In this case, we don't need to implement functions for working with the decision tree separately, so we can use the more normal object-oriented technique of inheritance and virtual methods.

Listing 8.16 shows the base class (`Decision`) and the simpler of the two derived classes (`ResultDecision`) which represents the final result.

Listing 8.16 Object oriented decision tree (C#)

```
abstract class Decision {
   public abstract void Evaluate(Client client);                       #1
}
class ResultDecision : Decision {
   public bool Result { get; set; }
   public override void Evaluate(Client client) {                #2
      Console.WriteLine("OFFER A LOAN: {0}", Result ? "YES" : "NO");
   }
}
```

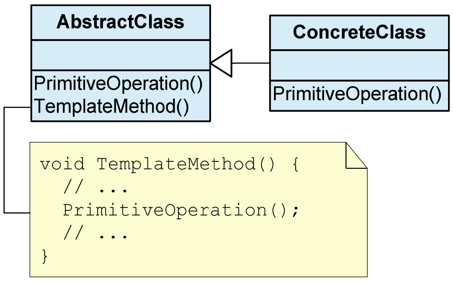
#1 Tests the given client
#2 Print the final result

This part of the code is quite simple. The base class contains only a single virtual method (#1), which will be implemented in the derived classes and which will test the client and print the result. Its implementation in the class representing the final result (#2) just prints the result to the console.

The more interesting part is the implementation of the class representing a query. The problem is that we need to provide different code for each of the concrete queries (testing the income, the number of years in the current job and so on). We *could* create a new derived class for each of the query with a very similar implementation of the `Evaluate` method–but that doesn't feel like a good solution, as it involves code duplication. A somewhat better way for implementing this is to use the *template method* design pattern.

THE TEMPLATE METHOD PATTERN

In general, the template method pattern allows us to define the skeleton of an algorithm or a class and fill in the missing pieces later, by implementing them in an inherited concrete class. The base class defines operations to be filled in later and uses them to implement more complicated operations. Figure 8.4 shows this in diagram form.



```
void TemplateMethod() {
  // ...
  PrimitiveOperation();
  // ...
}
```

Figure 8.4 The base class contains abstract method 'PrimitiveOperation', which is used in the implementation of 'TemplateMethod'. This missing piece is filled in by inherited class 'ConcreteClass'.

The abstract class from the Template method corresponds to our representation of the query (let's call the class `QueryDecision`). The primitive operation that needs to be supplied by the derived classes is the testing method, which would take a `Client` as an argument and return a Boolean value. The template method would be our `Evaluate` method, which would contain code to print the result to the console and recursively process the selected branch. However, we'd still have to implement a new concrete class for each of the specific query, which would make the code quite lengthy. Using functions, we can simplify the pattern and remove this need.

FUNCTIONAL IMPLEMENTATION

Instead of representing the primitive operation as a virtual method that can be filled in by deriving a class, we'll represent it as a property, where the type of the property is the function type `Func<Client, bool>`. The function is then supplied by the user of the class. Listing 8.17 shows an implementation of the `QueryDecision` class as well as an example of how we can create a simple decision tree.

Listing 8.17 Simplified implementation of Template method (C#)

```
class QueryDecision : Decision {
   public string Title { get; set; }
   public Decision Positive { get; set; }
   public Decision Negative { get; set; }
   public Func<Client, bool> Check { get; set; }                       #1

   public override void Evaluate(Client client) {
      bool res = Check(client);                                        #2
      Console.WriteLine("  - {0}? {1}", Title, res ? "yes" : "no");
      Decision next = res ? Positive : Negative;                       #3
      next.Evaluate();                                                 #3
    }
}

var tree =
   new QueryDecision {                                                #A
      Title = "More than $40k",
      Check = (client) => client.Income > 40000,                      #4
      Positive = new ResultDecision { Result = true },                #B
      Negative = new ResultDecision { Result = false } };             #B
```

#1 Primitive operation to be provided by the user
#2 Test a client using the primitive operation
#3 Select a branch to follow
#A The tree is constructed from a query
#4 Check is specified using lambda functions
#B Sub-trees can be 'ResultDecision' or 'QueryDecision'

The `QueryDecision` class represents a case where we want to perform another test regarding the client. If we had followed the template method pattern strictly then the test would be a virtual method, but we instead specified it as a property (#1). The type of the property is a function that takes a client and returns a Boolean value. This function is invoked when testing a client (#2) and depending on the result, the code follows one of the two possible branches (#3). When creating a decision tree, we don't have to write an extra class for every test, because we can simply provide the primitive testing functionality using lambda functions (#4).

This example demonstrates how we can very effectively mix object-oriented and functional concepts. In fact, the types we created could be easily made immutable, which would make the example even more functional. The only reason why we didn't do that is that using properties makes the code a bit more compact. We started with a standard object-oriented design pattern and simplified it using lambda functions that are now available in C# 3.0. The solution is somewhere between the traditional object-oriented solution and the functional version we implemented in F#.

8.5 Summary

In this chapter we have finished our exploration of the core functional concepts. After talking about basic principles such as functional values and higher order functions, we moved to a higher level perspective and discussed the architecture of functional applications. We divided applications (or components) into two groups: data-centric and behavior-centric.

In this chapter we discussed behavior-centric programs. We've seen how to develop an application where behaviors aren't hard-coded and new behavior can be added very easily later, either during development or at run-time, simply by using a list of functions. Later, we investigated several ways to extend the data structure to combine functions and other functional data types to develop a decision tree, which combines data and behaviors in a single data type.

We've also talked about design patterns that are related to behavior-centric programs. In particular we've seen how the strategy pattern corresponds to higher order functions and how the command pattern relates to closures in functional programming. Finally, we looked at how the template method pattern can be simplified using functions in C# 3.0.

In the next part of the book, we'll focus on language features that are specific to F# and on advanced functional concepts. The next chapter starts with F# features that allow us to take the next step of the iterative development style. We'll see how to turn conceptually simple data types such as tuples of functions or discriminated unions into real-world types. This means that the types follow standard F# and .NET development guidelines, are easy to document and could be distributed in a commercial F# or .NET library. This also means that the library will be easily accessible from a C# application.

9 Turning values into F# object types with members

When I introduced F# in the first chapter, I said it was a multi-paradigm language that takes the best elements of several worlds. Most importantly, it takes ideas from both functional and object-oriented languages. In this chapter, we're going to look at several features that are inspired by object-oriented programming or allow fluent integration with object-oriented ,NET languages like C# and VB.NET.

This chapter is particularly important for the later steps in the F# development process. As I've mentioned before, functional simplicity allows us to write a program very quickly and provides great flexibility. On the other hand, object-oriented programming in F# is valuable because it gives the code a solid structure, encapsulates related functionality, and allows painless integration with other systems or programming languages. In this chapter, we'll see how to take some F# code that we developed earlier in the book and evolve it to make it easier to use in a team or in a larger project.

In the previous two chapters, I've described two of the most common architectures of functional programs or components. Both of them can take advantage of some object-oriented concepts, so we'll start off by revisiting data-centric applications before discussing going back to behavior-centric applications. Finally, we'll talk about interoperability between C# and F#. You'll learn how to work with .NET classes from F# and how to write F# code that can easily be used from a C# project.

9.1Improving data-centric applications

Let's go over a few of the important elements of behavior-centric applications that we covered in the previous chapters. In chapter 7 we saw that the key aspect of data-centric application design is creating the data structures that will be used by the application. Functional languages give us very simple and conceptually clear constructs for data structures. We've seen all basic data types, namely tuples, discriminated unions and records. We've also seen how to declare generic types that can be reused by many applications and we talked about some of them that are available in the F# libraries such as the option type and functional list.

Unlike in object-oriented programming, so far we've implemented operations separately from the data types. This has several advantages. First of all, it allows us to easily add operations , especially when we use discriminated unions. The functional application design and architecture is suited for this approach, so it is more common than in object-oriented style. Also, writing the code in this way makes the syntax is very succinct, so the code can be written faster and we can easily prototype various solutions. Code written in the functional style can for example take the full advantage of type inference. On the other hand, if we use object types with members, we'll very often need to provide type annotations. The lightweight functional style also makes it easier to run the code interactively using the F# interactive shell.

DEVELOPING OPERATIONS INTERACTIVELY

The data structure changes less frequently, so once you define the data structure, you can create values of that type and keep them "alive" in F# interactive. Then you can write the first version of the function, test it using F# interactive, correct possible errors, improve it, and test it again on the same data. If we were updating the data structure together with all its operations, this process would be a lot more difficult.

On the other hand, there are many reasons *in favor of* keeping operations as part of the data structure too; you probably know most of them from experience with C#. Let's demonstrate this using an example. In chapter 7, we wrote a simple `Rect` type and two functions to work with it. You can see the code repeated in listing 9.1. The example uses some types from the `System.Drawing` namespace, so if you're creating a new project, you'll need to add a reference to the `System.Drawing.dll` assembly.

Listing 9.1 'Rect' type with processing functions (F#)

```
open System.Drawing

type Rect =                                                             #A
   { Left:  float32; Top:    float32
     Width: float32; Height: float32 }

let deflate(rc, wspace, hspace) =                                       #B
   { Left = rc.Left + wspace; Width  = rc.Width - (2.0f * wspace)
     Top  = rc.Top + hspace;  Height = rc.Height - (2.0f * hspace) }

let toRectangleF(rc) =                                                  #C
   RectangleF(rc.Left, rc.Top, rc.Width, rc.Height)
```
#A Type declaration

#B Shrinks the rectangle
#C Conversion to 'System.Drawing' representation

First we declare the type, and then define two operations for working with rectangle values. The operations are implemented independently as F# functions. However, if we implement them as methods instead, it is much easier to discover them when writing the code. Instead of remembering the function name, you just type dot after the value name and Visual Studio's IntelliSense pops up with a list of operations. The code is also better organized, because you know what operations belongs to which type. The obvious conundrum is how to get the best from both of the approaches in F#.

9.1.1 Adding members to F# types

This is where the F# iterative style of development comes very handy. The ability to debug and test the code interactively is of course more important during the early phase of the development. As the code becomes more polished and we start sharing the project with other developers, it is more important to provide the common operations as members that can be invoked using dot notation.

This means that in F#, encapsulation of data types with their operations is typically one of the last steps of the development process. This can be done using *members*, which can be added to any F# type and behave just like C# methods or properties. Listing 9.2 shows how to augment the `Rect` type with two operations using members.

Listing 9.2 'Rect' type with operations as members (F#)

```
type Rect =                                                       #1
   { Left   : float32
     Top    : float32
     Width  : float32
     Height : float32 }
                                                                  #2
   /// Creates a rectangle which is deflated by 'wspace' from the     #3
   /// left and right and by 'hspace' from the top and bottom        #3
   member x.Deflate(wspace, hspace) =                             #4
      { Left = x.Top + wspace
        Top = x.Left + hspace
        Width = x.Width - (2.0f * wspace)
        Height = x.Height - (2.0f * hspace) }

   /// Converts the rectangle to representation from 'System.Drawing'
   member x.ToRectangleF() =                                      #5
      RectangleF(x.Left, x.Top, x.Width, x.Height)
```

#1 Familiar declaration of F# record type
#2 Members have to be correctly indented!
#3 Documentation comment
#4 Member method with two arguments
#5 Member method with no arguments

To create an F# data type with members, you simply write the member declarations after the normal F# type declaration. As you can see in the example (#2) the member declarations have to be indented by the same number of spaces as the body of the type declaration. In our example, we started with a normal F# record type declaration (#1) and then added two different methods as members.

The member declaration starts with the keyword `member`. This is followed by the name of the member with a value name for the current instance. For example, `x.Deflate` means that we're declaring a method `Deflate` and that, inside the body, the value `x` will refer to the current instance of the record. This acts in a similar way to the C# `this` keyword - you can think of it as a way of being able to call `this` anything you like.

The first member (#4) takes a tuple as an argument and creates a rectangle, which is made smaller by subtracting the specified length from its vertical and horizontal sides. When creating types with members, F# developers usually follow the .NET style and declare a member's parameters as a tuple. However, if you specify the parameters without braces, you can use standard functional techniques such as partial function application with members.

Another thing to note in the example is that the comment preceding the member (#3) now starts with three slashes (`///`). This is a special kind of comment that specifies documentation for the member, analogous to C# XML comments. In F#, you can use similar XML-based syntax if you want, but if you just write plain non-XML text, the comment is automatically treated as a summary.

Now let's see how we can use the members we've declared. After you select the code and run it in F# interactive, you'll see an interface of the type, which also includes available members and their type signatures. Listing 9.3 demonstrates calling both members.

Listing 9.3 Working with types and members (F# interactive)

```
> let rc = { Left = 0.0f; Top = 0.0f                                #A
             Width = 100.0f; Height = 100.0f };;                    #A
val rc : Rect

> let small = rc.Deflate(10.0f, 30.0f);;                            #1
val small : Rect = { Left = 30.0f; Top = 10.0f
                     Width = 80.0f; Height = 40.0f }

> small.ToRectangleF();;                                            #2
val rcf : RectangleF = {X=30, Y=10, Width=80, Height=40} { ... }
```

#A Create a 'Rect' value
#1 The first member returns deflated rectangle
#2 The second member returns 'RectangleF' value

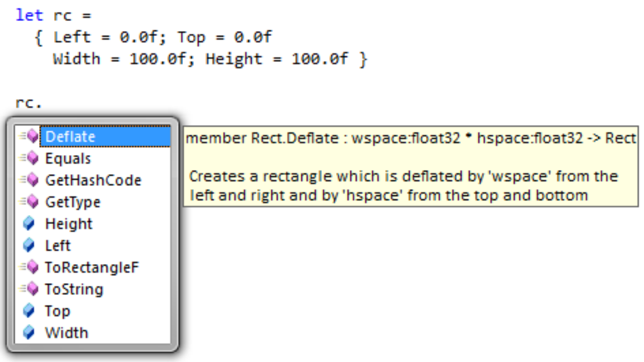
We start by creating a value of the `Rect` type. This hasn't changed; we still specify a value for each of the record type members. The next command (#1) invokes the `Deflate` member. As you can see, we can do this using standard object-oriented dot notation that we've already seen when working with .NET objects. In this case, the arguments are specified using a tuple, but if we specified them without braces in the declaration, the call would also use the F# function call syntax with parameters separated by a space. Finally, the last command converts the rectangle into a value of the `RectangleF` object from

`System.Drawing`. The example looks now very much like object-oriented code, but that doesn't mean that we're turning away from the functional programming style in any sense.

FUNCTIONAL 'DEFLATE' MEMBER

The code is still purely functional (as opposed to imperative), which means that there are no side-effects, despite its more object-oriented *organization*. If you implemented this in the imperative style, the `Deflate` method would probably modify the properties of the rectangle it was called on. However, our implementation doesn't do this. The `Rect` data type is still immutable: the property values can't be changed once the instance has been created. So, instead of modifying the value, the member returns a new `Rect` value with modified properties. This is the same behavior as the original `deflate` function had, but it is important to keep in mind that we can very nicely combine functional concepts (like immutability) with the object-oriented concepts (in this case, encapsulation). This isn't an alien concept in an imperative object-oriented world, of course - look at the `System.String` type, which takes exactly the same approach.

I've already mentioned that one of the benefits when using members instead of functions is that you can easily discover operations for working with the value using IntelliSense. In figure 9.1 you can see the Visual Studio editor working with `Rect` type.



```
let rc =
  { Left = 0.0f; Top = 0.0f
    Width = 100.0f; Height = 100.0f }

rc.
```

member Rect.Deflate : wspace:float32 * hspace:float32 -> Rect

Creates a rectangle which is deflated by 'wspace' from the left and right and by 'hspace' from the top and bottom

Figure 9.1 Hint showing members of the 'Rect' type when editing F# source code inside Visual Studio IDE.

Another important benefit is that types with members are naturally usable from other .NET languages like C#. For example, the `Deflate` member would look just like an ordinary method of the type if we were using it from C#, as we'll see later.

When we turned functions into members in listing 9.2, we converted functions declared using let bindings into members declared using the `member` keyword. This worked, but we had to make quite large changes in the source code. Fortunately, we can avoid this and make the transition from a simple F# style to a more idiomatic .NET style smoother.

9.1.2 Appending members using type extensions

In the previous section, I mentioned that you can add members to any F# data type. This time, we'll demonstrate it using a discriminated union. We'll use a technique that allows us to add members without making any changes to the original code. This means that we'll be able to leave the original type and original function declarations unmodified and add members to them later.

We'll extend an example from chapter 5 where we declared a schedule type. The type can represent events that occur once, repeatedly or never. As well as the data type itself, we created a function that tests whether an event occurs during the upcoming week. Listing 9.4 shows a slightly modified version of the code (to make it more compact). The original code was in listing 5.5, if you want to compare the two.

Listing 9.4 Schedule data type with a function (F#)

```
type Schedule =                                                 #1
   | Never
   | Once of DateTime
   | Repeatedly of DateTime * TimeSpan

let isNextWeek(dt) =                                            #2
   dt > DateTime.Now && dt < DateTime.Now.AddDays(7.0)

let occursNextWeek(schedule) =                                  #3
   match schedule with
   | Never -> false
   | Once(dt) -> isNextWeek(dt)
   | Repeatedly(dt, ts) ->
      let q = max ((DateTime.Now - dt).TotalSeconds/ts.TotalSeconds) 0.0
      isNextWeek(dt.AddSeconds(ts.TotalSeconds * (Math.Floor(q) + 1.0)))
```

#1 Original type declaration
#2 Utility function used by 'occursNextWeek'
#3 We want to expose this as a member

The most interesting change is that the utility function (#2) is now declared as an ordinary function instead of a nested function inside `occursNextWeek` (#3). I made this change just to demonstrate the choices available. The function is simple enough that it can reasonably be nested without impacting readability. In a more complicated project you may many large utility functions, where nesting would add too much complexity.

The point is that in a typical F# source file, we start with the type declaration (#1), then have a bunch of utility (private) functions, then finally a couple of functions that we want to expose as members (#3). If we wanted to turn the last function into a member using the technique from the last section, it would be quite difficult. The members have to be written as part of the type declaration, but we usually want to put several utility functions between the type and its members!

The solution is to use *intrinsic type extensions*, which allow us to add members to a type declared earlier in the file. Listing 9.5 shows how we can use extensions with our schedule type.

Listing 9.5 Adding members using intrinsic type extensions (F#)

```
type Schedule =
   | Never
   | Once of DateTime
   | Repeatedly of DateTime * TimeSpan

let isWeek(dt) =
   (...)
let occursNextWeek(schedule) =                        #1
   (...)

type Schedule with                                    #2
   member x.OccursNextWeek() = occursNextWeek x       #3
```

#1 Function exposed via a member
#2 Intrinsic type extension
#3 Member just calls the function

Most of the code hasn't changed since listing 9.4, so I've omitted it for brevity. We've just added the last two lines of code. The first one (#2) defines a type extension, which tells the F# compiler to add the following members to a type with the specified name. This is followed by the usual member declarations (#3). As we've already implemented the functionality as a function (#1), we can just call the function inside the member declaration.

If you come from C# 3.0, you can see similarities between type extensions and extension methods. They are in general quite similar, and you can use type extensions to augment existing types from other assemblies as well. However, the case in the previous listing was different, because we used *intrinsic type extension*. This is a special case when we declare both the original type and the extension in a single file. In that case, the F# compiler merges both parts of the type into a single class and also allows us to access private members of the type in the type extension.

Listing 9.6 demonstrates calling the members from in listing 9.5. Members added using type extensions behave in a same way as other members, so the listing shouldn't contain any surprised.

Listing 9.6 Working with schedule using members (F# interactive)

```
> let sch = Repeatedly(DateTime(2000, 9, 25), TimeSpan(365, 0, 0, 0));; #A
val sch : Schedule

> sch.OccursNextWeek();;                                                 #B
val it : bool = true

> let sch = Never;;                                                      #C
val sched : Schedule
```

```
> sch.OccursNextWeek();;                                                   #D
val it : bool = false
```

#A Create a value as usual
#B Invoke the member
#C Redefine the 'sch' value
#D Test the member again

Just like when we were working with records, we create the F# value in the usual way. For the discriminated union in our example this means using the `Repeatedly` or `Never` discriminator. (We could have used the `Once` discriminator as well, of course.) Once we have the value, we can invoke its members using the object-oriented dot notation.

As we've just seen, members are very useful when writing mature code, because they wrap the code into well-structured pieces and also it easier to use the types. In the F# development process, we don't usually *start* by writing code with members, but we add them later once the code is well tested and the API design is fixed. We've seen two ways of adding members:

41) When the type is simple enough, we can append members directly to the type declaration.

42) For more complex types, we can use intrinsic type extensions which require fewer changes to the code.

Type extensions have the additional benefit that we can also test the type and its processing functions in the F# interactive tool before augmenting it, because we don't have to declare the whole type in one go.

We've seen that members are very important for turning data-centric F# code into a real-world .NET application or component. Now we'll turn our attention to behavior-centric applications.

9.2 Improving behavior-centric applications

In the previous couple of chapters, I've shown that functional programming is based on several basic concepts which are then composed to get the desired result. We've seen this when discussing the ways to construct data types, with examples of tuples, functions, discriminated unions, and record types.

When creating behavior-centric applications, we used a function type to represent the behavior and we composed it with other types. For example, we used a record type to store two related functions in a single value.

9.2.1 Using records of functions

Using records that store functions is a common technique in OCaml and to some extent also in F#. Before looking at possible improvements, listing 9.7 provides a reminder of the original solution in chapter 8.

Listing 9.7 Testing clients using records of functions (F#)

```
type ClientTests =                                         #A
   { Check : Client -> bool
     Report : Client -> unit }

let testCriminal(cl) = cl.CriminalRecord = true            #B
let reportCriminal(cl) =                                   #B
   printfn "'%s' has a criminal record!" cl.Name           #B

let tests =
   [ { Check = testCriminal; Report = reportCriminal };    #C
     (* more tests... *) ]
```

#A Representation of the test
#B Testing and reporting function
#C Create a record value

The code first creates a record type that specifies types of functions that form the checking and reporting part of the test. It then creates two functions and composes them to form a value of the record type. Using records of functions is conceptually very simple and it's also easy to refactor code using individual functions into a design using records. However, if we want to evolve this code into a more traditional .NET version, we can take one more step.

I mentioned before that the function type is similar to an interface with a single method. It is not a surprise that a record consisting of two functions is quite similar to an interface with two methods. In C# you'd almost certainly implement this design using an interface, and F# lets us do the same thing.

Similarly to members, interfaces are more important when creating robust applications or reusable .NET libraries. First of all, if we use an interface, we don't say how exactly it should be implemented. This gives us a lot of flexibility how to write the application. We'll talk about various ways to implement an interface in F# later in this chapter. Interfaces are also useful when developing a .NET library that should be callable from C#. If we declare an interface in F#, the C# code will see it as an ordinary interface. On the other hand, an F# record type with functions as members looks like a class with properties of some hard-to-use type. Let's see how we can adapt our record type into an interface while still using it in a natural way from F#.

9.2.2 Using interface object types

Just like records and discriminated unions, interfaces type are declared using the `type` construct. Listing 9.8 shows our earlier test record type converted to an interface type.

Listing 9.8 Interface representing client test

```
type ClientTest =                        #A
   abstract Check : Client -> bool       #B
   abstract Report : Client -> unit      #C
```

#A Interface type declaration
#B Member that tests the client
#C Member that reports issues

This declaration says that any type implementing the `ClientTest` interface will need to provide two members. In the interface declaration, the members are written using the `abstract` keyword, which means that they don't yet have an implementation. The declaration specifies the names and type signatures of the members. Even though we didn't explicitly say that we're declaring an interface, the F# compiler is smart enough to deduce that. If we for example provided implementation of one of the members, the compiler would realize that we want to declare an abstract class. However, in the usual F# programming practice, we'll need abstract classes and implementation inheritance very rarely, so we'll focus on working with interfaces in this chapter.

However, if we wanted to create a test that checks for example the criminal record of the client in C#, we'd have to write a new class implementing the interface. F# supports classes as well, but provides another solution called *object expressions*. This is inspired by functional programming and is often more elegant, because we don't have to write any class declarations before creating useful values.

OBJECT EXPRESSIONS AND LAMBDA FUNCTIONS

The analogy between interface types and function types is very useful when explaining what an object expression is. The signature of a function types describes it in a very abstract sense. It specifies that the function takes some arguments and returns a result of some type. The concrete code of the function is provided when we're creating a function value. This can be done using a lambda function, which is an expression that returns a function, or a let-binding, which creates a named function.

Similarly, an interface is an abstract description of a value. It just specifies that the value should have some members and what their signatures are. Again, we provide the actual code for the members when creating a concrete value. One option is to write a named class that implements the interface, which is similar to creating a named function. On the other hand, object expressions are similar to lambda functions. They can be used anywhere in the code and create a value that implements the interface without specifying the name of the type providing the actual code.

In the following listing, we'll take a look at object expressions in practice. We'll create tests to check the client's criminal record and their income, and create a list of interface values just like our earlier lists of records.

Listing 9.9 Implementing interfaces using object expressions (F# interactive)

```
> let testCriminal =
     { new ClientTest with                                           #1
        member x.Check(cl) = cl.CriminalRecord = true                #2
        member x.Report(cl) =                                        #2
           printfn "'%s' has a criminal record!" cl.Name };;         #2
val testCriminal : ClientTest                                        #3
```

```
> let testIncome =
     { new ClientTest with
        member x.Check(cl) = cl.Income < 30000                      #A
        member x.Report(cl) =                                        #B
           printfn "Income of '%s' is less than 30000!" cl.Name };;
val testCriminal : ClientTest

> let tests = [ testCriminal; testIncome ];;                        #C
val tests : ClientTest list
```

#1 Object expression
#2 Provides code for the interface members
#3 A value of the interface object type
#A Implements testing of a client
#B Implements reporting
#C Create a list of interface values

The code creates two values implementing the `ClientTest` interface type using object expressions. Each object expression is enclosed in curly braces and starts with an initial header (#1) that specifies what interface we are implementing. This is followed by the `with` keyword and then by the member declarations (#2). Syntactically, this is quite similar to the type extensions that we discussed in the previous section. Member declarations give an implementation for the members specified by the interface, so the expressions in the previous listing implement members `Check` and `Report`.

The whole object expression fulfils the normal definition of an F# expression: it does a single thing, which is returning a value. If we look at the output from F# interactive (#3), we can see that it returns a value of type `ClientTest`. This is the interface type, so the object expression returns a concrete value implementing the interface, just like a lambda function returns a function value implementing an abstract function type.

SIMILARITY WITH ANONYMOUS TYPES

Technically, the F# compiler creates a class that implements the interface and object expression returns a new instance of this class. However, the declaration of the class is only internal, so we cannot access this class directly. The only thing we need to know about it is that it implements the specified interface.

This is similar to anonymous types in C# 3.0, where the compiler also creates a hidden class behind the scene that we cannot access directly. In C#, we know what the properties of the class are, but this is only available locally inside the method. On the other hand, in F# we know which interface the class implements, so we can work with it without any such limitations.

In this section, we've seen how to use interface types to make one final step in the iterative development of behavior-oriented applications in F#. The key benefit of using interfaces is that they give us an idiomatic .NET solution, but F# provides features to allow us to work with interfaces in a natural way which is consistent with its functional style.

Thanks to the object expressions, it is easier to implement the interface than to construct a record of functions.

Later in this chapter we'll also see that using interfaces makes it possible to call F# code comfortably from C#. We didn't talk about class declarations in F# yet, because ordinary classes aren't used that frequently in F#, but we'll look at them briefly later in section 9.4. Before that, let's have a look how we can take advantage of object expressions when using some common types from the .NET libraries.

9.3 Working with .NET interfaces

The .NET Framework is fully object-oriented, so we'll often work with interfaces when using .NET libraries from F#. In this section, we'll look at two examples. First we'll look how to implement an interface that can be used to customize equality of keys stored in the `Dictionary` object. in the second example, we'll work with the well-known interface for resource management: `IDisposable`.

9.3.1 Using .NET collections

So far, we've mostly used the built-in F# list type for storing collections of data. However, in some cases it's useful to work with other .NET types such as the `Dictionary` class from the `System.Collections.Generic` namespace. This type is particularly useful when we need very fast access based on some keys, because immutable types providing similar functionality (such as `Map` from the F# library) are less efficient.

Note that that the `Dictionary` type is a mutable type. This means that methods like `Add` change the state of the object instead of returning a new, modified copy. This means we have to be careful when working with it in scenarios where we want to keep our code purely functional.

Listing 9.10 shows how we can create a simple lookup table using `Dictionary` and how to specify custom way for comparing the keys by providing an implementation of the `IEqualityComparer<T>` interface.

Listing 9.10 Implementing 'IEqualityComparer<T>' interface (F# interactive)

```
> open System
  open System.Collections.Generic;;

> let equality =
     let replace(s:string) = s.Replace(" ", "")              #A
     { new IEqualityComparer<_> with                         #1
        member x.Equals(a, b) =                              #B
           String.Equals(replace(a), replace(b))             #B
        member x.GetHashCode(s) =                            #B
           replace(s).GetHashCode() };;                      #B

> let dict = new Dictionary<_, _>(equality)                  #2
  dict.Add("100", "Hundred")
  dict.Add("1 000", "thousand")
  dict.Add("1 000 000", "million");;

> dict.["10 00"];;
val it : string = "thousand"

> dict.["1000000"];;
val it : string = "million"
```

#A Removes spaces from a string
#1 Create custom comparison object
#B Compare strings, ignoring the spaces
#2 Create a dictionary using custom comparison

This example demonstrates that object expressions can be quite useful when we need to call a .NET API that accepts an interface as an argument. In this case, the constructor of the `Dictionary` type (#2) accepts an implementation of the `IEqualityComparer<T>` interface as an argument. The interface is then used to compare keys when accessing elements stored in the dictionary. We created a value called `equality`, which implements the interface in advance (#1). Our implementation compares strings and ignores any spaces in the string. We did that by creating a utility function that removes spaces from any given string and then comparing the trimmed strings. We also implemented a method that calculates the hash code of the string, which is used by the `Dictionary` type to perform the lookup efficiently.

It is also worth noting that F# type inference helped us again in this listing. We used an "_" instead of the actual type when writing the object expression (#1) as well as when creating an instance of the `Dictionary` class (#2). When the compiler sees the underscore, it uses other information to figure out what the actual type parameter is and in this particular example it had enough information from other parts of the code.

Another very familiar interface for a .NET programmer is `IDisposable`, which is used for explicit cleaning of resources. Let's have a look how we can use it from F#.

9.3.2 Cleaning resources using IDisposable

We've already worked with several types that implement `IDisposable`, like `Graphics` and `SolidBrush`. I wanted to make the code as easy to follow as possible, so when we finished using the object, we explicitly called the `Dispose` method.

C# contains syntactic sugar for this in the form of the `using` statement, which makes sure that `Dispose` is called even if an exception is thrown within the body of the statement. F# has a similar construct with the `use` keyword. Listing 9.11 shows a simple example that works with files.

Listing 9.11 Working with files and the 'use' keyword (F# interactive)

```
> open System.IO;;
> let writeHello() =
     use sw = new StreamWriter("C:\\test.txt")    #1
     sw.WriteLine("Hello world!")
     sw.WriteLine("Ahoj svete!")                   #2
val writeHello : unit -> unit
```

```
> writeHello();;
val it : unit = ()                                  #A
```

#1 Declare the 'sw' value using the 'use' keyword
#2 Dispose called after this line
#A A file is written to the disk

When creating a `StreamWriter` (which implements the `IDisposable` interface), we declare it using the `use` keyword (#1). Note that the syntax is very similar to the `let` keyword in a usual let binding. The difference is that the F# compiler automatically adds a call to the `Dispose` method at the end of the function (#2), so the `StreamWriter` is automatically disposed after we finish working with it. The compiler also inserts try-finally block to make sure that the cleanup is run even when an exception occurs.

An important difference between the `using` construct in C# and the `use` keyword in F# is that in C# we have to specify the scope explicitly using curly braces. On the other hand, in F#, the `Dispose` method is simply called at the end of the function. This is usually what we need, so it makes a lot of code snippets easy to write. Listing 9.12 shows the C# and F# version side by side.

Listing 9.12 Cleaning up resources in F# and C#

```
let test() =
   use sw = new StreamWriter(..)
   // code...
#A
   // some more code...
#A

#B
```

```
void Test() {
   using(sw = new StreamWriter())
{
      // code...                    #C
   }                                #D
   // some more code...
}
```

#A 'sw' is in scope here
#B Object disposed here
#C 'sw' is in scope here
#D Object disposed here

In both of languages, the object is disposed when the execution leaves the scope where the value `sw` is accessible. In F#, this happens at the end of the function by default, which is often what we need. However, when the function continues with some code that can run for a long time, it is better to make sure that the resource is disposed earlier. We can either refactor the code that uses the resource into a separate function, or we can specify the scope explicitly like this:

```
let test() =
   ( use sw = new StreamWriter(..)
     foo(sw) )
   // some more code
```

The syntax may be somewhat surprising, but it becomes clear once we realize that in F# every block of code is an expression. In the code above, we're just specifying the way in which the expression is constructed in the same way as when we write `(1 + 2) * 3` instead of the default `1 + (2 * 3)`. This way we can limit the scope of the `sw` value to the expression inside parentheses.

Even though the use keyword is primarily useful when working with .NET objects that keep some resources, it can be used for a wider range of scenarios. Let's look at an example.

PROGRAMMING WITH THE 'USE' KEYWORD

As we've seen, if we create a value using the use keyword the compiler will automatically insert a call to its Dispose method at the end of the function where it's declared. This is of course useful for resources, but there are other situations where we need to enclose a piece of code between two function calls.

For example, suppose we want to output some text to a console in a different color and then restore the original color. Traditionally, we'd have to store the original color, set the new one, send the output to the console and finally restore the original color.

However, the same thing can be done rather elegantly thanks to the use keyword. We can write a function that changes the color of the console and returns an IDisposable value. This value contains a Dispose method, which restores the original color when called and thanks to the use keyword, the method will be called automatically. Listing 9.13 shows the function and a demonstration of its use.

Listing 9.13 Setting console color using IDisposable (F# interactive)

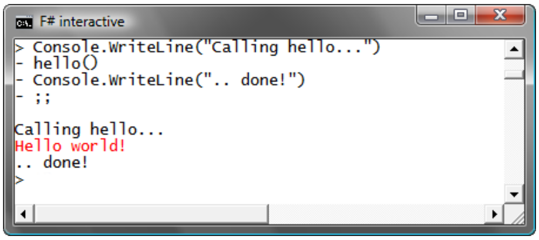
```
> open System;;

> let changeColor clr =
     let orig = Console.ForegroundColor              #1
     Console.ForegroundColor <- clr                  #2
     { new IDisposable with                          #3
        member x.Dispose() =
           Console.ForegroundColor <- orig };;       #4
val changeColor : ConsoleColor -> IDisposable

> let hello() =
     use clr = changeColor ConsoleColor.Red          #5
     Console.WriteLine("Hello world!")
     ;;                                              #6
val hello : unit -> unit
```

#1 Store the original color
#2 Set the new color immediately
#3 Create 'IDisposable' value
#4 Restore the original color inside 'Dispose'
#5 Color is changed to red
#6 Original color is restored

The most interesting part of the code is the changeColor function. We can imagine that it contains two pieces of code. The first part is executed immediately when the function is called and the second part is returned and executed at some later time. The first part of the code first stores the original color (#1) and then sets the new one (#2).

The second part needs to be returned as a result. We could return it as a function (possibly using lambda function syntax), but then the caller would have to call it explicitly. Instead, we create an IDisposable value using an object expression (#3) and place the code that restores the original color in the Dispose method (#4).

When the `changeColor` function is used the first part (which sets the new color) is executed immediately (#5). However, we store the result using the `use` keyword, so at the end of the function (#6) the `Dispose` method is called and the original color is restored. You can see a screenshot showing the result of running this code in the F# interactive console window in figure 9.2.



```
> Console.WriteLine("Calling hello...")
- hello()
- Console.WriteLine(".. done!")
- ;;

Calling hello...
Hello world!
.. done!
>
```

Figure 9.2 Changing the color of the console text using 'changeColor' function. You can see that the color is changed only inside the 'hello' function and then the original color is restored.

The same idea is useful in other contexts, such as temporarily changing the cursor in a GUI to an appropriate "please wait" indicator, or temporarily changing the current thread's culture to a specific value when unit testing culture-specific code. The clue here is the word "temporarily" which suggests the "change something, do some work, restore the original value" pattern - ideal for the `use` keyword!

In all the previous examples showing object-oriented features, we used the standard F# types, interfaces and object expressions. This is quite normal when using F# in a functional way, but the language supports other object-oriented features as well. As this book is primarily about functional programming we won't discuss all of them, but we'll look at a couple of the most important examples.

9.4 Concrete object types

The most important construct of object-oriented programming is a class declaration. In F#, this is very useful when writing a library that can be used from C#, because an F# class declaration looks just like a normal class when referenced from C#. However, classes are often used in F# for encapsulating data and behavior when we believe that the data structure or the behavior will change less frequently. Deciding whether this is the case is of course quite difficult, so classes are usually used in the later steps of the iterative development process.

Let's start with the simplest possible example. The listing 9.14 shows an implicit class declaration with a constructor, several properties and a method.

Listing 9.14 Class with client information (F# interactive)

```
> type ClientInfo(name, income, years) =             #1
      let q = income / 5000 * years                  #A
      do printfn "Creating client '%s'" name         #A

      member x.Name = name                           #2
      member x.Income = income                       #2
      member x.Years = years                         #2

      member x.Report() =                            #3
          printfn "Client: %s, q=%d" name q          #3
  ;;
type ClientInfo = (...)

> let john = new ClientInfo("John Doe", 40000, 2);;  #B
val john : ClientInfo
Creating client 'John Doe'

> john.Report();;                                    #C
Client: John Doe, q=16
val it : unit = ()
```

#1 Class declaration with constructor arguments
#A Code executed during construction
#2 Property declarations
#3 Method declaration
#B Create the class and run the constructor
#C Invoke method of the class

The declaration starts with the class name and constructor arguments (#1). The next couple of lines before the first member declaration are executed during construction. This part of code forms an implicit constructor. The arguments to the constructor (such as `name` and others) and values declared in the initialization code (like `q`) are accessible from anywhere inside the class. This is quite useful, because a C# constructor often just copies its arguments to private fields, so they can be accessed from other places. It is also worth noting that if you use the parameter only inside the code of the constructor, it isn't stored as a field, because the compiler knows that we won't need it.

Next, the class contains three member declarations that expose constructor arguments as properties of the client (#2) and a single method (#3). The "`x.`" prefix in the member declarations means that the current instance of the class can be accessed using the "`x`" value. For example we might use it to call another method or reading other properties.

CLASS DECLARATIONS IN F#

F# provides a richer set of features for declaring classes than what we've seen in this example. The goal of the F# language is to be a first-class .NET citizen, so nearly everything you can write in C# can be also translated to F#. However, in the usual F#

programming, we don't need advanced .NET object model features such as overloaded constructors and methods or for example publicly accessible fields.

The goal of this book is to introduce functional concepts and not to explain every F# feature, so we'll look only at the most useful object oriented constructs that F# provides and how they work with the functional style. You can find more information about class declarations on the book's web site and also in the [F# Documentation] and [F# Language Specification].

It is worth mentioning that the class from the previous example is still purely functional, in the sense that it doesn't have any mutable state. This demonstrates how object-oriented and functional paradigms can work very well together.

9.4.1 Functional and imperative classes

Just like the let bindings we've seen in other F# code, a let binding in a class or an argument to a class constructor is an immutable value. Also, a property declaration using the `member` keyword creates a read-only property (with just a getter). This means that if the class references only values of other immutable types, it will also become immutable.

Let's say that we want to allow changes of the client's income in the previous example. This can be done in two different ways:

43) In a purely functional style, the object will return a new instance with updated income and the original values of all other properties.

44) Using the imperative style the income will be a mutable field.

Listing 9.15 shows the functional version of the class (named `ClientF`) side by side together with the imperative class named `ClassI`.

Listing 9.15 Functional and imperative version of Client type (F#)

```
> type ClientF(name, inc) =             > type ClientI(name, inc) =
     member x.Name = name       #1           let mutable inc = inc     #4
     member x.Income = inc      #1
                                             member x.Name = name
     member x.WithIncome(ninc) =#2           member x.Income
        new ClientF(name, ninc)                 with get() = inc       #5
                                                and set(v) = inc <- v  #5
     member x.Report() =                     member x.Report() =
        printfn "%s %d" name inc                printfn "%s %d" name inc
type ClientF = (...)                    type ClientI = (...)

> let c = new ClientF("Joe", 30);;      > let c = new ClientI("Joe", 30);;
val c : ClientF                         val c : ClientI

> let c = c.WithIncome(40);;    #3      > c.Income <- 40;;              #6
val c : ClientF                         val it : unit = ()
```

```
> c.Income;;                              > c.Income;;
val it : int = 40                         val it : int = 40
```

#1, #2, #3 In the functional version all properties remain read-only (#1). In addition, the class contains a method called 'WithIncome' (#2) that creates a copy of the object with income set to the new value. When using the method, we store the returned customer using let binding (#3). We can use the same name for the new value, which hides the original value and it becomes inaccessible.
#4, #5, #6 The imperative version declares an updatable field for storing the income using the 'mutable' keyword (#4). The field has a name 'inc', so it hides the immutable constructor argument with the same name. The income can be updated using a read/write property (#5) that we added to the class. This creates a standard .NET property, which can be used in the usual way (#6).

Annotations below the code with bullets on the left (as in Ch. 03)

When declaring a mutable field in the imperative version (#4), I used the same name for both the value and the constructor parameter. The new value hides the original one, meaning that we can no longer access the original value. This may seem strange at first, but it prevents you from accidentally using initial value when you actually intend to use the current (possibly changed) one.

The next notable thing in the imperative version is the read/write property (#5). In F# syntax, the property is composed from a two members similar to method declaration. The `get` member doesn't have any parameters and returns the value, while the `set` member has a single parameter for the new value and should return `unit` as the result. Even though the syntax is slightly different from that of a C# property declaration, the principles are exactly the same.

Even though we're concentrating on functional programming, it is sometimes useful to know how to write a mutable class like this. If you need to expose a larger piece of F# code to a C# client, you'll probably wrap your code in at least one class, because this makes it easier to use from C#. At this point, you can choose which style to follow - an imperative one with some mutable types, or a purely functional one where everything is mutable. The second solution is cleaner from the F# point of view, but developers who aren't used to dealing with libraries composed entirely of immutable types may find it easier to use a wrapper with mutable state.

We're very nearly ready to show a complete example of calling F# code from C#, but we need to finish our tour of object-oriented F# features first.

9.4.2 Implementing interfaces and casting

We've already seen how to declare an interface in F# and how to create a value that implements the interface using object expressions. This is a very lightweight solution similar to lambda functions. However, just as lambda syntax isn't always the appropriate choice for creating functions, it sometimes makes sense to implement an interface in a named class.

We're going to work with the same example as earlier in the chapter. We'll look at implementing interfaces in both C# and F#, so let's briefly recap the declaration of the interface in both of the languages:

```
interface IClientTest {              type ClientTest =
      bool Test(Client client);             abstract Test : Client -> bool
      void Report(Client client);           abstract Report : Client -> unit
   }
```

The interface just has two methods: one that tests the client and one that prints a report to the screen. Now let's suppose we want to implement the interface using a coefficient calculated from several properties. Earlier we created similar tests using object expressions, but when the code becomes more complex it is better to move it into a separate class declaration or possibly into a separate source file.

One small point to note before we move on to look at the implementations: in F#, we didn't use the "I" prefix when declaring the interface; F# has various ways to declare a type and attempts to unify all of them. However, the C# version uses the standard .NET naming convention, so it implements an interface called `IClientTest`. If you're creating an F# library that is supposed to be used from other .NET languages, then it is of course a good idea to follow all the standard .NET coding conventions in all public API.

Listing 9.16 shows the C# implementation testing a client's income and how many years they've been in their current job using weightings and a threshold, all specified in the constructor. The class uses *explicit interface implementation*, which is slightly unusual–but we'll see why when we look at the F# implementation.

Listing 9.16 Client test using explicit interface implementation (C#)

```
class CoefficientTest : IClientTest {                                   #A
   readonly double income, years, min;
   public CoefficientTest(double income, double years, double min) {   #B
      this.income = income; this.years = years; this.min = min;        #B
   }                                                                    #B
   public void PrintInfo() {
    Console.WriteLine("income*{0}+years*{1} >= {2}", income, years, min);
   }
   bool IClientTest.Test(Client cl) {                                   #1
      return cl.Income*qIncome + cl.YearsInJob*qYrs < min;              #1
   }                                                                    #1
   void IClientTest.Report(Client cl) {                                 #1
      Console.WriteLine("Coefficient {0} is less than {1}.",            #1
         cl.Income*income + cl.YearsInJob*years, min);                  #1
   }                                                                    #1
}
```

#A Class implementing 'IClientTest'
#B Store arguments in a private field
#1 Private implementations of interface methods

To implementing an interface member using the explicit syntax in C#, we include the name of the interface when writing the method (#1) and remove the access modifier. This is just a minor change, but the more important difference is how the class can be used. The methods from the interface (in our case `Test` and `Report`) are not directly accessible when using the class. To call them, we first have to cast the class to the interface type. Let's look at an example:

```
var test = new CoefficientTest(0.001, 5.0, 50.0);
```

```
test.PrintInfo();                                    #A

var cltest = (IClientTest)test;                      #B
if (cltest.Test(john)) cltest.Report(john);
```
#A Call method of the class
#B Cast to the interface type

We cannot just write `test.Test(john)`, because `Test` is not directly available as a public method of the class. It is only usable a member of the interface, so we can access it using `cltest` value, which has a type `IClientTest`. We used an explicit cast and the `var` keyword in the code, because that will help understanding how interface implementations work in F#. Another option that we have is to declare the variable type as `IClientTest` and then just assign the `test` value to it, because the C# compiler would use implicit conversion to the interface type.

Other than using explicit interface implementation, the class is wholly unremarkable. We're really just using it as a point of comparison with the F# code. Speaking of which…

Implementing interfaces in F#

The reason listing 9.16 uses explicit interface implementation in C# is that this is the *only* style of interface implementation which F# allows. In the functional programming style, this is often adequate. If you *really* need to expose the functionality directly from the class, you can add an additional member that invokes the same code. Listing 9.17 shows an F# version of the previous example.

Listing 9.17 Implementing interface in a class (F#)

```
type CoefficientTest(income, years, min) =                          #1

   let coeff cl =                                                   #2
      ((float cl.Income)*income + (float cl.YearsInJob)*years)      #2
   let report cl =                                                  #2
      printfn "Coefficient %f is less than %f." (coeff cl) min      #2

   member x.PrintInfo() =                                           #A
      printfn "income*%f + years*%f > %f" income years min

   interface ClientTest with                                        #3
      member x.Report cl = report cl                                #3
      member x.Test cl = (coeff cl) < min                           #3
```
#1 Implicit class declaration
#2 Local helper functions
#A Standard public method of the class
#3 Interface implementation using helpers

The listing uses implicit class syntax, so it specifies arguments to the constructor directly in the declaration (#1). It takes three arguments specifying various coefficients for the calculation. Since we're referring to these parameters later in the members, the F# compiler will automatically store them in class fields.

Next, we defined two local helper functions using the standard let binding syntax (#2). These are not visible from outside of the class and we use them only for other members later in the code. When implementing an interface (#3), we group all members from a single interface together using the `interface ... with` syntax and implement them using usual members. If we also wanted to expose some of the same functionality as a public method, we could simply add another member to the class declaration and call the local helper function. Alternatively we could implement the functionality in the public member and call that member from the interface implementation.

Working with the class is very much like the previous C# version that was using explicit interface implementation. You can see the F# version of the code in listing 9.18.

Listing 9.18 Working with F# classes and interfaces (F# interactive)

```
> let test = new CoefficientTest(0.001, 5.0, 50.0)            #A
val test : CoefficientTest

> test.PrintInfo()                                            #B
income*0.001000 + years*5.000000 > 50.000000

> let cltest = (test :> ClientTest)                           #1
val cltest : ClientTest

> if (cltest.Test(john)) then cltest.Report(john)             #C
Coefficient 45.000000 is less than 50.000000.
```

#A Create an instance of the class
#B Use method of the class
#1 Cast to the interface type
#C Use methods of the interface

Most of the listing should be quite straightforward. The only exception is the code that casts the value to the interface type (#1), because we haven't yet talked about casts in F#. In F#, there are two kinds of casts. In this case, the compiler knows at the compilation that the cast will succeed, because it knows that the class (`CoefficientTest`) implements the interface (`ClientTest`). This is called an *upcast*. In the next section, we'll look at both of the casts in detail.

UPCASTS AND DOWNCASTS IN F#

When the conversion between the types cannot fail, it is called an upcast. We've seen that this is the case when converting a type to an interface implemented by that type. Another example is casting a derived class to its base class. In this case the compiler can also guarantee that the operation is correct and will not fail.

On the other hand, if we have a value of a base type and we want to cast it to an inherited class, then the operation can fail. That's because the value of the base class may or may not be a value of the target class. In this case, we have to use a second type of casting, which is called a *downcast*. Let's demonstrate this using an example. We'll use the standard `Random` class, which is (just like any other .NET class) derived from the `Object` class:

```
> open System;;
> let rnd = new Random();;
val rnd : Random
```

```
> let obj = (rnd :> Object);;                             #A
val obj : Object

> let rnd2 = (obj :> Random);;                            #B
stdin(4,12): error: Type constraint mismatch.             #B
The type 'Object' is not compatible with the type 'Random' #B

> let rnd2 = (obj :?> Random);;                           #C
val rnd2 : Random

> let err = (obj :?> String);;                            #D
val err : String
System.InvalidCastException: Specified cast is not valid.
```

#A Upcast - operation cannot fail
#B This can't be written using an upcast!
#C We have to use downcast
#D Downcast to a wrong type throws an exception!

As you can see, if we accidentally try to use an upcast inappropriately, the F# compiler reports this as an error. The error message says that `Object` isn't compatible with `Random`, which means that the compiler cannot guarantee that the value of type `Object` can be casted to the `Random` type. Finally, the listing shows that a downcast can fail and throw an exception if we try to cast an object to a wrong inherited class.

A good way to remember the F# syntax for upcasts (`:>`) and downcasts (`:?>`) is to realize that there is some uncertainty when using downcasts, because the operation can fail. This uncertainty is a reason why downcast operator contains the question mark symbol and upcast doesn't. The F# language also provides an equivalent to the `is` operator known from C# that returns Boolean value specifying whether an object instance can be casted to the specified type. To test whether `obj` can be casted to `String`, we'd write `obj :? String`.

It's worth thinking about the differences between F# and C# here. In C#, we didn't even *need* the cast in listing 9.16: when the compiler knows the conversion can succeed and it's not needed for disambiguation, you can just let it occur implicitly. F# doesn't convert implicitly conversions, so it makes sense for it to have a language construct just for this expressing conversions which are guaranteed to succeed. In C# it wouldn't make sense as you'd use it so rarely–it's simpler to use the same syntax for both kinds of conversion.

It would be impossible to review all the object-oriented features of F# in just a single (reasonably sized!) chapter, but we've seen that the ones that are most important in order to evolve functional applications into real-world .NET code.

I've said several times that these changes make our F# code more easily accessible from C#, and it's time to give proof of that, and show exactly how the interoperability hangs together.

9.5 Using F# libraries from C#

Like C#, F# is a statically typed language, which means that the compiler knows the type of every value as well as signatures of class methods and properties. This is very important for interoperability with C#, because the compiler can generate code that looks just like an ordinary .NET library.

Interoperability with other .NET languages

This is not the case for dynamically typed languages that have a .NET implementation like Python, Ruby or JavaScript. In these languages, the compiler doesn't know whether a method takes an argument of type `int` or for example `Customer`, so using code written in these languages is more difficult when using C# 3.0. Often you don't even know whether an object contains a method with a particular name, so the C# has to look like this:

```
obj.InvokeMethod("SayHello", new object[] { "Tomas" });
```

This example specifies the name of the method as a string and passes the arguments to the method in an array. This is of course an important problem for many languages, so C# 4.0 introduces the "dynamic" type that allows you to write something like:

```
obj.SayHello("Tomas");
obj.SaiHello("Tomas");
```

The syntax is same as for normal method calls, but there is an important difference. I intentionally added another method call, but with a misspelled method name. This will compile correctly, because the method name is internally represented as a string just as in the previous example. The problem only comes to light at run-time. The fact that F# is statically typed means we don't have to worry about this: we can rely on the compiler to spot the same kinds of errors it would when calling into other C# code.

We're going to start with a basic example. In the first section of this chapter, we saw how to add members to the `Rect` type that represents a rectangle. Now we're going to use the type from C#. First we need to create a new F# "Library" project and add a source file (for example "export.fs") containing the code from listing 9.19.

Listing 9.19 Compiling F# types into a library (F#)

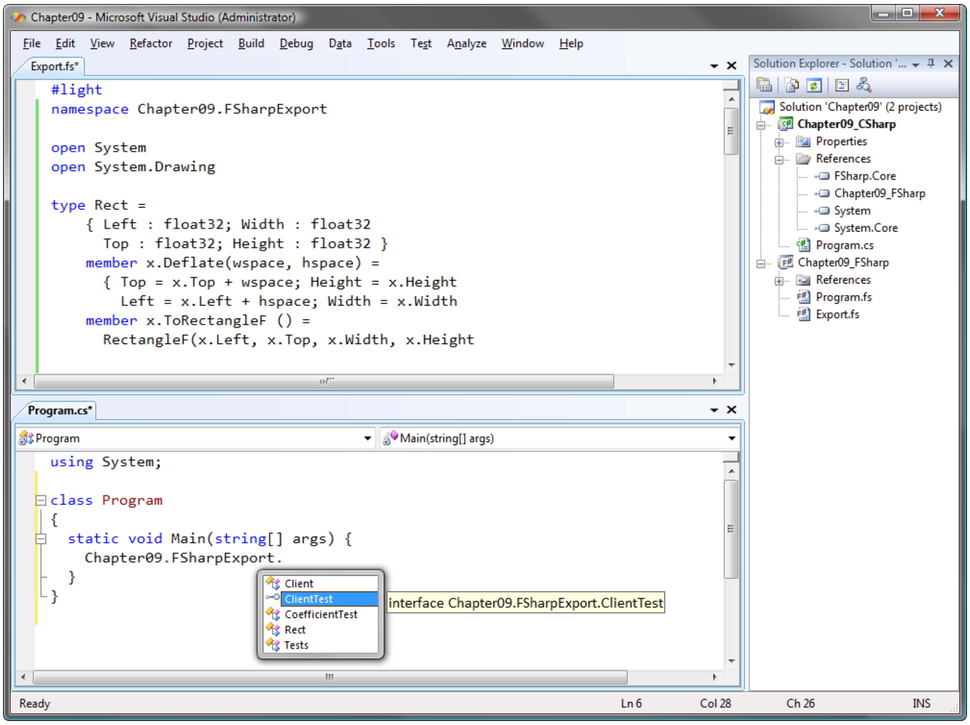
```
namespace Chapter09.FSharpExport                                   #1

open System
open System.Drawing

type Rect =
   { Left : float32; Width : float32                               #2
     Top : float32; Height : float32 }                             #2

   member x.Deflate(wspace, hspace) =                              #3
      { Top = x.Top + wspace; Height = x.Height - (2.0f * hspace)  #3
        Left = x.Left + hspace; Width = x.Width - (2.0f * wspace) }  #3
   member x.ToRectangleF () =                                      #3
```

```
          RectangleF(x.Left, x.Top, x.Width, x.Height)                #3
```

#1 Specifies namespace for the file
#2 Record fields are compiled as properties
#3 Methods of the 'Rect' class

As you can see, we've just added a single line to specify the .NET namespace (#1). This namespace will contain all the type declarations from the file (in our case, there is only a single type called `Rect`). This type will easy to use from C# because the fields of the record (#2) will become properties and members (#3) will appear as methods.

Next we're going to add a new C# project to the solution. Adding a reference to the F# project is done exactly as if you were referencing another C# class library, although you should also add a reference to the `FSharp.Core` assembly. This is an F# redistributable library that contains the F# core functions and types. After configuring the projects, you should see something similar to the figure 9.3. The figure also shows how other F# types from this chapter appear in IntelliSense from C#.



```
#light
namespace Chapter09.FSharpExport

open System
open System.Drawing

type Rect =
    { Left : float32; Width : float32
      Top : float32; Height : float32 }
    member x.Deflate(wspace, hspace) =
      { Top = x.Top + wspace; Height = x.Height
        Left = x.Left + hspace; Width = x.Width
    member x.ToRectangleF () =
      RectangleF(x.Left, x.Top, x.Width, x.Height)
```

```
using System;

class Program
{
  static void Main(string[] args) {
    Chapter09.FSharpExport.
  }
}
```

Figure 9.3 After adding a reference to the F# library, we can see types from the F# project in IntelliSense. The F# record type 'Rect' is compiled as an ordinary class.

If you experiment with IntelliSense, you'll see that the F# type is present in the namespace we specified in its source code. IntelliSense also shows what properties and methods the type has, so you'd be probably able to use it without any further help. Just for completeness though, listing 9.20 gives a short example.

Listing 9.20 Using types from the F# library (C#)

```
using System;
using Chapter09.FSharpExport;                                #A

class Program {
   static void Main(string[] args) {
      var rc1 = new Rect(0.0f, 100.0f, 0.0f, 50.0f);         #1
      var rc2 = rc1.Deflate(20.0f, 10.0f);                   #2
      Console.WriteLine("({0}, {1}) - ({2}, {3})",           #B
         rc2.Left, rc2.Top, rc2.Width, rc2.Height);          #B
   }
}
```

#A Reference namespace from the F# library
#1 Create an instance of the class
#2 Invoke a functional member of the class
#B Prints '(10, 20) - (60, 30)'

The code in the listing first creates an instance of the `Rect` type. It uses a constructor that was automatically generated by the F# compiler (#1) and corresponds to the F# code for creating a record. We have to specify values for all the fields of the record at construction time–we can't change them later, as the type is immutable. The next step is to invoke the `Deflate` method (#2). This is just a perfectly ordinary method, although we're dealing with a purely functional so the method returns a new `Rect` value instead of mutating the existing one. Finally, we print the information about the returned rectangle. This is also easy, because record fields are exposed as .NET properties.

USING F# LIBRARIES FROM F#

We've looked at referencing F# projects from C# because this is a common scenario and I wanted to explicitly show how nicely the two languages play together when the F# code uses object types. However, you can also reference F# libraries from F# applications. The steps to do this would be the same: specify a namespace for the F# library, add a reference in Visual Studio and add an appropriate `open` directive to your F# application. It is worth noting that when referencing an F# library from F#, the compiler will recognize that the library is authored in F# and all constructs (such as discriminated unions or functions) will be accessible in the normal F# way.

Using the `Rect` type from C# is quite simple, and figure 9.3 shows some other types from this chapter. For example an F# interface declaration (`ClientTest`) shows as an ordinary .NET interface, so the interoperability works very smoothly. However, what if we wanted to export a function or a value? What would these two constructs look like in C#?

9.5.1 Working with values and delegates

In this section, we're going to look at using two more typical F# constructs from C#. We'll see how to export a value and a high order function. The latter is quite tricky, because F# uses quite a sophisticated internal representation for functions.

For example if a function took `int -> int -> int` as an argument, a C# developer would see this as `FastFunc<int, FastFunc<int, int>>`. It is possible to work with this type, but it isn't very convenient; we'll use a different approach. If we're writing a higher order function that should be used from C# then we can use standard .NET delegates. This isn't as natural as using normal functions in F#, but the library will be much simpler to use from C#.

There's another problem which crops up when we want to export a value or function directly. Methods (and fields) don't appear on their own in .NET, or even as part of a namespace–they're always part of a type. The very idea of a method existing with no containing type to love and nurture it is enough to make a compassionate C# developer distraught. Help is at hand in the form of F# *modules*. Listing 9.21 shows how a value and a utility function can be exported so they can be used from C#, and also demonstrates the previous point about using delegates for higher order functions.

Listing 9.21 Exporting values and higher order functions (F#)

```
type Client =
  { Name : string; Income : int; YearsInJob : int
    UsesCreditCard : bool; CriminalRecord : bool }

module Tests =                                             #1
    let John =
      { Name = "John Doe"; Income = 25000; YearsInJob = 1
        UsesCreditCard = true; CriminalRecord = false }

    let WithIncome (f:Func<_, _>) client =                 #2
        { client with Income = f.Invoke(client.Income) }   #3
```

#1 Enclose values and functions in a module
#2 Function taking delegate as an argument
#3 Calls the delegate using 'Invoke' method

The module declaration (#1) tells that F# compiler to enclose the values and functions into a class with static methods (when compiling functions) and static properties (for values). I've chosen to follow the C# naming conventions here (using Pascal case) as the reason for creating the module in the first place is to expose the values to C#.

The next point to note is the `WithIncome` function. It's a higher order function, but instead of taking a normal F# function as an argument, it takes a .NET delegate `Func` with two generic arguments (#2). We're using an underscore so the F# compiler infers the actual types for us. When we need to invoke the delegate later in the code (#3), we use its `Invoke` method. This is somewhat inelegant compared with normal F# function calling, but it means the C# client can work with it in an idiomatic manner using lambda functions:

```
var client = Tests.John;                                   #A
client = Tests.WithIncome(income => income + 5000, client);   #B
```

```
Console.WriteLine("{0} - {1}", client.Name, client.Income);
```
#A Use a value from 'Tests' module
#B We can use a lambda function!

The module which we called `Tests` is compiled into a class, so the value `John` becomes a static property of this class and `WithIncome` becomes a method. As you can see, it takes an argument of type `Func<int, int>`, so anyone who knows C# 3.0 can use it even though the code is actually written in F#. In reality, we could of course make the `WithIncome` a member of the `Client` type and the C# user would call it using the familiar dot-notation. However, I wanted to demonstrate that even basic F# functions can be used from C# with no problems.

9.6 Summary

In the last few chapters we we've talked about functional programming and implemented several sample applications in the functional style. We started with simple functional ideas such as combining values into "multiple values" or "alternative values", then we discussed ways of working with functions. Finally in chapters 7 and 8 we talked about the design of functional programs. This was not a haphazard decision: the structure of the book corresponds to the iterative F# development style. We started with very simple concepts that allowed us to solve problems succinctly and quickly. Finally, in this chapter we took the final step of the iterative development process, exposing our code in familiar .NET terms.

We've seen members that allow us to encapsulate functionality related to a type with the type itself and intrinsic type extensions that can be used if we already have the code as ordinary functions. Next, we looked at abstract types (interfaces) that are quite useful when writing behavior-centric applications. We also talked about classes, which are particularly important in interoperability scenarios.

However, there are still many things that we haven't covered. In the next few chapters, we're going to turn our attention from architectural aspects back to the core functional programming techniques. In the upcoming chapter, we're going to revisit lists and simple recursive functions and you'll see some essential techniques for writing efficient functional code. This is an important aspect that we skipped earlier to make the introduction as simple as possible. However, you've already mastered all the basic functional ideas, so we're now ready to dive into some important advanced techniques.

10 Efficiency of data structures, tail recursion and continuations

In the first parts of the book, we began using functional techniques such as recursion and functional data structures like immutable lists. We wrote the code in the most straightforward way we could, using the basic F# collection type (a list) and expressing our intentions very directly. This works very well in many situations, but when we come to process large data sets, "obvious" code sometimes leads to performance problems. In this chapter, we'll look at several techniques for writing code that works regardless of the size of the input and at the ways to optimize the performance of functions working with data. We'll still strive to keep the code as readable as possible though.

If you've been developing for any significant length of time, you've almost certainly written a program that caused a stack overflow exception. In functional programming this error can easily be caused by a naively written recursive function, so we'll look at several ways for dealing with functions that can cause this error when processing large amounts of data. This will be our starting topic and we'll return to it at the end of the chapter.

In between these discussions on recursion, we'll talk about functional lists and arrays. When working with functional lists, it is important to understand how they work so you can use them efficiently. Finally, F# also supports arrays that can give us a better performance in some situations. Even though arrays are primarily imperative data types, we'll see that we can use them in a very functional way.

10.1 Optimizing functions

In the earlier chapters, we saw that recursion is the primary control flow mechanism for functions in F#. We first used it for writing simple functions that perform some calculation, such as adding up numbers in a specified range or a working out a factorial. Later we found it invaluable while working with recursive data structures - most importantly lists.

You may already be familiar with several limitations of recursion, the possibility of stack overflow being the most obvious one. As we'll see, some recursive computations can be very inefficient too. In imperative languages, you'd often use non-recursive function to avoid problems. However, functional languages have developed their own ways of dealing with these problems and can work with recursion very efficiently. First let's concentrate on correctness: it's no good being really efficient with up to 1K of data if an extra byte blows your stack…

10.1.1 Avoiding stack overflows with tail recursion

For every function call, the runtime allocates a *stack frame*. These frames are stored on a stack maintained by the system. A stack frame is removed when a call completes. If a function calls another function, then a new frame is added on top of the stack. The size of the stack is limited, so too many nested function calls leave no space for another stack frame, and the next function can't be called. When this happens in .NET, a `StackOverflowException` is raised. In .NET 2.0 and higher, this exception can't be caught and will bring down the whole process.

Recursion is based on nested function calls, so it isn't surprising that you'll encounter this error most often when writing complex recursive computations. (Well, that may not be true. The most common cause in C# is *probably* writing a property which accidentally refers to itself instead of its backing field. We'll ignore such typos though, and only consider *intentional* recursion.) Just to show the kind of situation we're talking about, let's use the list-summing code from chapter 3, but give it a really big list.

Listing 10.1 Summing list and stack overflow (F# interactive)

```
> let test1 = [ 1 .. 10000 ]                              #A
  let test2 = [ 1 .. 100000 ]                             #A
val test1 : int list
val test2 : int list

> let rec sumList(lst) =
     match lst with
     | [] -> 0                                            #1
     | hd::tl -> hd + sumList(tl)                         #2
val sumList : int list -> int

> sumList(test1)                                          #3
val it : int = 50005000

> sumList(test2)                                          #4
Process is terminated due to StackOverflowException.
```

#A Create lists for testing
#1 Branch that returns immediately
#2 Recursive branch
#3 Stack size is sufficient
#4 Too many nested function calls!

Just like every recursive function, `sumList` contains a case that terminates the recursion (#1) and a case where it recursively calls itself (#2). The function completes a certain amount of work before performing the recursive call (it performs pattern matching on the list and reads the tail), then it executes the recursive call (to sum the numbers in the tail). Finally, it performs a calculation with the result: it adds the value stored in the head with the total sum returned from the recursion. The details of the last step are particularly important as we'll see in a moment.

As we might have predicted, there is a point when the code stops working. If we give it a list with tens of thousands of elements (#3), it works fine. However, for a list with hundreds of thousands of elements, the recursion goes too deep and F# interactive reports an exception (#4). Figure 10.1 shows what's happening: the arrows above the diagram represent the first part of the execution, before and during the recursive call. The arrows below the diagram represent the recursion returning the result.

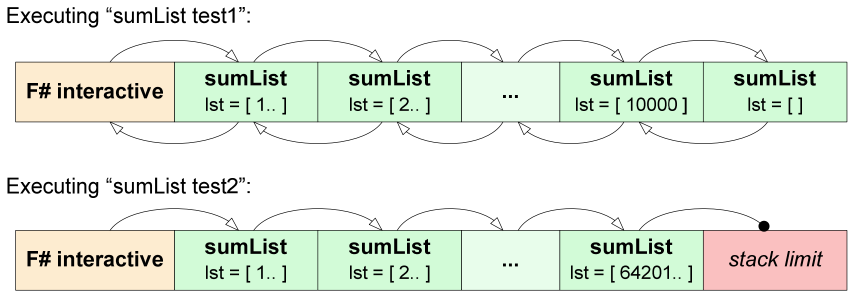


Figure 10.1 Stack frames when summing numbers in a list. In the first case, the stack frames fit within the limit, so the operation succeeds. In the second case, calculation reaches the limit and an exception is thrown.

I used a notation [1..] to denote a list containing series that begins with 1. In the first case, F# interactive starts executing `sumList` with a list from 1 to 10000 as its argument. The figure shows how a stack frame is added to the stack for each call. Every step in the process takes the tail of the list and uses it as an argument for a recursive call to `sumList`. In the first case, the stack is a sufficient size, so we eventually reach a case where the argument is an empty list. In the second case, however, we use up all of the space after roughly 64,000 calls. The runtime reaches the stack limits and raises `StackOverflowException`.

The figure shows how the calls proceed using arrows. Both arrows from the left to the right and backwards do some work. The first part of the operation is executed before the recursive call and decomposes a list into head and tail components. The second part, executed after the recursive call completes, adds the value from the head to the total.

Now we know why it's failing, what can we do about it? The essential idea is that we only need to keep the stack frame because we need to do some work after the recursive call completes. In our example, still need the value of the head element so we can add it to the result of the recursive call. If the function didn't have to do anything after the recursive call completed, it could jump from the last recursive call back, directly to the caller, without using anything from the stack frames in between. Let's demonstrate this with the following trivial function:

```
let foo(arg) =
   if (arg = 1000) then true
   else foo(arg + 1)
```

As you can see, the last operation that the `foo` function performs in the `else` branch is a recursive call. It doesn't need to do any processing with the result, it just returns it directly. This kind of recursive call is called *tail recursion*. Effectively, the result of the deepest level of recursion – which is a call to `foo(1000)`– can be directly returned to the caller, as shown in diagram 10.2.

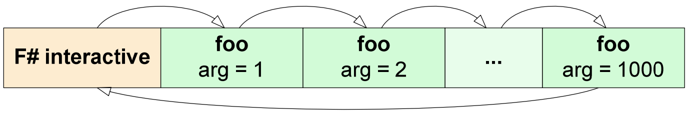


Figure 10.2 Recursive function 'foo' that doesn't do anything after the recursive call. The execution can jump directly to the caller (F# interactive) from the last recursive call, which is 'foo(1000)'.

From the diagram, you can see that the stack frames created during the computation (while jumping from the left to the right) are never used on the way back. This means that the stack frame is only needed before the recursive call, but when we recursively call `foo(2)` from `foo(1)`, we don't need the stack frame for `foo(1)`. The runtime can simply throw it away to save the space. Figure 10.3 shows the real-world execution of tail recursive function `foo`.

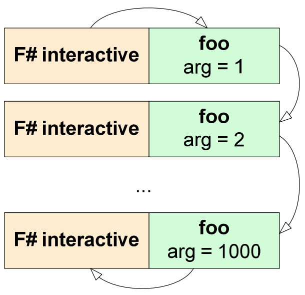


Figure 10.3 Execution of a tail-recursive function. The stack frame can be dropped during the recursive call, so only a single frame is actually needed at any point during the execution.

This diagram shows how F# executes tail recursive functions. When a function is tail recursive, we need only a single slot on the stack. This is makes the recursive version as efficient as an iterative solution.

TAIL RECURSION IN .NET LANGUAGES

The idea of tail recursion is directly supported by IL, so in theory the C# compiler could spot when it was applicable and make use of it. At the moment, it doesn't do so–but the F# compiler does, as tail recursion is such an important aspect of functional programming. C# developers normally try to design their code such that recursion to an arbitrary depth doesn't occur, precisely because of this problem–whereas it's a core part of a functional programmer's toolkit.

That's not to say that the runtime won't use tail call optimizations with code written in C#. Even if the IL doesn't contain explicit hints that it wants to use a tail call, the JIT may notice that it can do so safely and just go ahead. However, the rules for when this happens are complicated, and vary between the x86 and x64 JIT compilers. They're subject to change at any time. If your recursive code in C# happens to run when you'd expect it to blow up, that may be what's happening - but don't rely on it!

You may be wondering whether *every* recursive function can be rewritten to use tail recursion. Unfortunately the answer is no–but it *is* possible for many of them. As a simple rule thumb, if a function executes just a single recursive call in each branch, it should be possible to rewrite it using tail recursion.

USING AN ACCUMULATOR ARGUMENT

Let's think about how we'd make the `sumList` function tail recursive. It only performs the recursive call once in the branch where the argument is a cons cell (a non-empty list). Our rule of thumb suggests that we should be able to make it tail recursive–but at the moment it does more than just returning the result of the recursive call: it adds the value from the head to the total number.

To turn this into a tail recursive function, we can use a technique which supplies an *accumulator argument*. Instead of calculating the result as we jump from the right to the left (in the diagrams above, in other words as we're coming back towards the original function call), we can calculate the result as part of the operation that runs before the recursive call. We'll just need to add another parameter to the function to provide the current result. Listing 10.2 shows this technique in action.

Listing 10.2 Tail-recursive version of the 'sumList' function (F# interactive)

```
> let rnd = new System.Random()
```

```
      let test1 = List.init 10000 (fun _ -> rnd.Next(101) - 50)    #1
      let test2 = List.init 100000 (fun _ -> rnd.Next(101) - 50);; #1

    > let sumList(lst) =
         let rec sumListUtil(lst, total) =                        #2
            match lst with
            | [] -> total                                         #3
            | hd::tl ->
               let ntotal = hd + total                            #4
               sumListUtil(tl, ntotal)                            #A
          sumListUtil(lst, 0);;                                   #B
    val sumList : int list -> int

    > sumList(test1);;                                            #C
    val it : int = -2120                                          #C

    > sumList(test2);;                                            #C
    val it : int = 8736                                           #C
```

#1 Generate lists with random numbers
#2 Private function with the accumulator 'total'
#3 Return the accumulated value
#4 Add current value to accumulator
#A Recursive call
#B Calls the helper with total=0
#C Both calls compute the result now!

The listing starts by generating two lists containing random numbers (#1). We're using a function `List.init` that takes the required length of the list as the first argument and then calls the provided function to calculate value of the element at specified index. We're not using the index in the computation, so we used "_" to ignore it. The reason why we need better testing input is that if we added all numbers between 1 and 100000, we'd get incorrect results, because the result wouldn't fit into a 32-bit integer. We're generating random numbers between -50 and +50, so in principle the sum should be very close to zero.

More interesting part of the listing is the `sumList` function. When we use an accumulator argument, we need to write another function with an additional parameter. We don't usually want this to be visible to the caller, so we write it as a local function (#2). The accumulator argument (in our example named `total`) stores the current result. When we reach the end of the list, we already have the result, so we can just return it (#3). Otherwise, we add the value from the head to the result and perform a recursive call with the accumulator set to the new value (#4). Figure 10.4 shows how the new computation model works. Now if you look at the recursive call, we're returning the result immediately, so it can be executed using tail recursion.

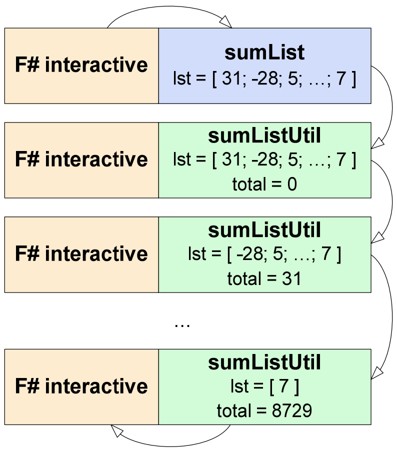


Figure 10.4 Execution of the tail-recursive sumList' function. The stack frames displayed in gray are dropped. You can see the result of summing all preceding elements in the accumulator value ('total').

The `sumList` example isn't very difficult, but it demonstrates the idea of using an accumulator. We just add another parameter to the function and use it to calculate a temporary result before making the recursive call. When you're trying to make a function tail recursive, look at the information you're currently using *after* the recursive call, and try to find a way to pass it *into* the recursive call instead.

We'll see some trickier examples when we talk about list processing, but we'll take a detour first, via another important optimization technique: *memoization*.

10.1.2 Caching results using memoization

Even though the name "memoization" may sound complicated, the technique is actually very simple. It can be simply described as caching the results of a function call. As I mentioned earlier, most functions in functional programming do not have side effects. This means that if we call a function with the same argument twice, we'll get the same result.

If we're going to get the same result we got last time, why would we want to go to all the trouble of executing the function again? Instead, we can just cache the results. If we store the result of the first call in some dictionary, we won't need to recompute the value for the second call. We can read the result from the dictionary and return it straight away. Listing 10.3 shows an example for function that adds two integers.

Listing 10.3 Adding numbers with memoization (F# interactive)

```
> open System.Collections.Generic;;

> let addSimple(a, b) =                              #1
    printfn "adding %d + %d" a b                     #A
```

```
     a + b;;
val addSimple : int * int -> int

> let add =                                        #2
     let cache = new Dictionary<_, _>()            #B
     (fun x ->                                     #3
        match cache.TryGetValue(x) with            #C
        | true, v -> v                             #C
        | _ -> let v = addSimple(x)                #C
               cache.Add(x, v)                     #C
               v);;                                #C
val add : (int * int -> int)

> add(2,3);;
adding 2 + 3                                       #D
val it : int = 5                                   #D

> add(2,3);;
val it : int = 5                                   #E
```

#1 Non-optimized addition
#A Prints info for debugging purposes
#2 Addition optimized using memoization
#B Initialize the cache
#3 Created function uses the private cache
#C Read the value from the cache or calculate it
#D Calls the 'addSimple' function
#E Value is obtained from the cache

The first part of the listing is just a normal addition function (#1) with the slight twist that it logs its execution to the console. Without this we wouldn't see any obvious differences between the original and memoized version, because the change in efficiency is really small for this example.

The function that implements addition with caching is called `add` (#2). It uses the.NET `Dictionary` type to store the cached results. The cache is declared as a local value and is used from the lambda expression (#3) that is assigned to the `add` value. We used a similar pattern in chapter 8 when we were talking about capturing mutable state using closures. Here, the cache value is also mutable (because `Dictionary` is a mutable hash table) and is also captured by a closure. The point is that we need to use the same `cache` value for all calls to the `add` function, so we have to declare it before the function, but we don't want to make it a global value.

The last part of the function is the lambda itself. It only uses the `addSimple` function when the result isn't cached already. As you can see from the F# interactive session, the function that does the actual calculation is executed only for the first time.

This technique is more widely applicable than tail recursion. It can be applied to any function which doesn't have any side effects[§§]. This means that we can use it successfully from C# 3.0 as well. In the next subsection, we're going to use C# 3.0 to write a bit more generic version of the code.

REUSABLE MEMOIZATION IN C# AND F#

If you look at the code that builds the `add` value from listing 10.3, you can see that it is doesn't really know about addition. It happens to use the `addSimple` function, but it could as well work with any other function. To make the code more general, we can turn this function into a parameter.

We're going to write a function (or method in C#) that takes a function as an argument and returns a memoized version of this function. The argument is the function that does the actual work and the returned function is augmented with caching capability. You can see the C# version of the code in listing 10.4.

Listing 10.4 Generic memoization method (C#)

```
Func<T, R> Memoize<T, R>(Func<T, R> func) {                   #1
   var cache = new Dictionary<T, R>();                        #A
   return arg => {
      R val;
      if (cache.TryGetValue(arg, out val)) return val;        #B
      else {
         val = func(arg);                                     #C
         cache.Add(arg, val);                                 #C
          return val;                                         #C
      } };
}
```

#1 Returns memoized version of the 'func' function
#A Cache captured by the closure
#B Return cached value
#C Calculate the value and add it to the cache

The code is very similar to the addition-specific function in listing 10.3. Again, we first create a cache and then return a lambda function that captures the cache in the closure. This means that there will be exactly one cache for each returned function, which is just what we want.

The method signature (#1) indicates that it takes a function `Func<T, R>` and returns a function of the same type. This means that it doesn't change the structure of the function;

[§§] This may sound slightly confusing, because the function in the previous listing had a side effect (printing to the screen). However, this is just a "soft side effect" that we can safely ignore. The core requirement is that the result should depend only on the arguments passed to the function.

it just wraps it into another function that does the caching. The signature is also generic, so it can be used with any function which takes a single argument. We can overcome this limitation with tuples. The following code shows the C# version of the memoized function for adding two numbers:

```
var addMem = Memoize((Tuple<int, int> arg) => {
   Console.Write("adding {0} + {1}; ", arg.First, arg.Second);
   return arg.First + arg.Second; });

Console.Write("{0}; ", addMem(Tuple.New(19, 23)));                    #A
Console.Write("{0}; ", addMem(Tuple.New(19, 23)));                    #A
Console.Write("{0}; ", addMem(Tuple.New(18, 24)));                    #A
```

#A Prints "adding 19 + 23; 42; 42; adding 18 + 24; 42;"

As we can see, the code that adds 19 and 23 is executed only once. This works because when the cache compares two tuple values, it will find a match when their elements are equal. This wouldn't work with our first implementation of `Tuple` because it didn't have any implementation of value equality, but the `Tuple` type in the source code for this chapter overrides `Equals` method to compare the component values. This behavior is called structural comparison and we'll talk about it in the next chapter. Another option to make the `Memoize` method work with functions with multiple parameters would be to overload it for `Func<T1, T2, R>`, `Func<T1, T2, T3, R>` and so on.

Implementing the same functionality in F# is easy now that we've seen the C# version. Listing 10.5 shows the code, which is pretty much a direct translation of the C# method.

Listing 10.5 Generic memoization function (F# interactive)

```
> let memoize(f) =
     let cache = new Dictionary<_, _>()       #A
     (fun x ->
        match cache.TryGetValue(x) with
        | true, v -> v
        | _ -> let v = f(x)
               cache.Add(x, v)
               v);;
val memoize : ('a -> 'b) -> ('a -> 'b)         #1
```

#A Initialize cache captured by the closure
#1 Inferred type signature

The only difference is that in the F# version, the type signature is inferred (#1), so we don't have to make the function generic by hand. The F# compiler uses generalization to do this for us; the inferred signature corresponds to the explicit one in the C# code.

This time, we'll use a more interesting example to demonstrate how effective memorization can be. We'll go back to the world's favorite recursion example: the factorial function. Listing 10.6 attempts to memorize this, but it doesn't quite go according to plan…

Listing 10.6 Difficulties with memoizing recursive function (F# interactive)

```
> let rec factorial(x) =                                      #1
     printf "factorial(%d); " x
     if (x <= 0) then 1 else x * factorial(x - 1));;    #2
val factorial : int -> int

> let factorialMem = memoize factorial                 #3
val factorial : (int -> int)

> factorialMem(2);;
factorial(2); factorial(1); factorial(0);              #A
val it : int = 1

> factorialMem(2);;
val it : int = 1                                       #B

> factorialMem(3);;
factorial(3); factorial(2); factorial(1); factorial(0)  #4
val it : int = 2
```

#1 Standard recursive factorial
#2 Recursive call
#3 Memoize it using 'memoize' function
#A Calculate 2! for the first time
#B Use the cached value
#4 Why is the value of 2! being recalculated??

At the first glance, the code seems correct. It first implements the factorial computation as a straightforward recursive function (#1) and then creates a version optimized using the `memoize` function (#3). When we test it later by running the same call twice, it still seems to work. The result is cached after the first call and it can be reused.

However, the last call (#4) doesn't work correctly– or more precisely, it doesn't do what we'd like it to. The problem is that the memoization covers only the first call, which is `factorialMem(3)`. The subsequent calls made by the `factorial` function during the recursive calculation call the original function directly instead of calling the memoized version. To correct this, we'll need to change the line that does the recursive call (#4) to use the memoized version (`factorialMem`). This function is declared later in the code, so we can use the "`let rec... and...`" syntax to declare two mutually recursive functions.

A simpler option is to use lambda functions and only expose the memoized version as a reusable function. Listing 10.7 shows how we can do this with just a few lines of code.

Listing 10.7 Correctly memoized factorial function (F# interactive)

```
> let rec factorial = memoize(fun x ->
     printfn "Calculating factorial(%d)" x
     if (x <= 0) then 1 else x * factorial(x - 1));;           #1
warning FS0040: This and other recursive references to the       #2
object(s) being defined will be checked for initialization-      #2
soundness at runtime through the use of a delayed reference...   #2

val factorial : (int -> int)

> factorial(2);;
factorial(2); factorial(1); factorial(0);                        #A
val it : int = 2
```

```
> factorial(4);;
factorial(4); factorial(3);                                         #3
val it : int = 6
```

#1 Recursive reference to the 'factorial' value
#2 There will be a runtime check
#A Compute first few values
#3 Compute only the missing value

The `factorial` symbol in this example refers actually to a value. It is not syntactically defined as a function with arguments and instead it is a value (which happens to be a function) returned by the `memoize` function. This means that we're not declaring a recursive *function*, but a recursive *value*. We used `let rec` to declare recursive values in chapter 8 when creating the decision tree, but we only used it for writing nodes in a more natural order – there weren't any recursive calls within the code.

This time, we're creating a truly recursive value, because the `factorial` value is used within its own declaration (#1). The difficulty with recursive values is that if we're not careful, we can write code that refers to some value during the initialization of that value, which is an invalid operation. An example of incorrect initialization looks like this:

```
let initialize(f) = f()
let rec num = initialize (fun _ -> num + 1)
```

Here, the reference to the value `num` occurs in a lambda function which is invoked during the initialization when the `initialize` function is called. If we run this code, we'll get a run-time error at the point where `num` is declared. On the other hand, when using recursive functions, the function will always be defined at the time when we'll perform a recursive call. The code may keep looping forever, but that's a different problem.

However, in our declaration of `factorial`, the reference to the `factorial` value occurs in a lambda function, which is *not* called during initialization, so it's a valid declaration. The F# compiler can't distinguish these two cases at compile time, so it emits a warning (#2) and adds some run-time time checks. Don't be too scared by this! Just make sure that the lambda function containing the reference will not be evaluated during the initialization.

Since the declaration of `factorial` uses the memoized version when it makes the recursive call, it can now read values from the cache for any step of the calculation. For example, when we calculate factorial of 4 (#3) after we've already calculated the factorial of 2, we only need to compute the two remaining values.

TAIL RECURSION, MEMOIZATION, AND ITERATIVE F# DEVELOPMENT

So far we've seen two optimization techniques used in functional programming. Using tail recursion we can avoid stack overflows and write better recursive functions. Memoization can be used for optimizing any functions without side effects.

Both of these techniques fit perfectly with the iterative development style that I consider an important aspect of F# programming. We can start with a straightforward implementation–often a function, possibly recursive, with no side effects. Later on in the process, we can identify areas of code that need to be optimized. Just as we saw how it's easy to evolve the structure of the code earlier, the changes required for optimization are reasonably straightforward to do. The iterative process helps us to pay the small additional price in complexity only in places where the benefit is actually significant.

So far we've seen some general-purpose tricks for writing efficient functions. There's one type of data structure which lends itself to very specific optimizations, however: collections. In the next section we'll talk about functional lists and also look at how we can use .NET arrays in a functional way.

10.2 Working with large collections

I mentioned that we'd come back to tail recursion and show some slightly more complicated situations involving lists. Hopefully by now any recursion-induced headaches will have worn off, and after a fresh cup of coffee you should be ready for the upcoming examples.

As well as just making sure our programs don't blow up with stack overflow exceptions, we tend to want them to run in a reasonable amount of time, too. (What is it with employers making such unrealistic demands?) Functional lists are fabulously useful and *can* be used very efficiently, but if you use them in the *wrong* way you can end up with painfully slow code. I'll show you how to avoid these problems.

10.2.1 Avoiding stack overflows with tail recursion (again!)

Our naïve list processing functions in chapter 6 weren't tail recursive. If we passed them very large lists, they would fail with a stack overflow. We'll rewrite two of them (`map` and `filter`) to use tail recursion, which will remove the problem. Just for reference, I've included the original implementations in listing 10.8. To avoid name clashes, I've renamed them to `mapN` and `filterN`.

Listing 10.8 Naïve list processing functions (F#)

```
let rec mapN f ls =                   let rec filterN f ls =
   match ls with                         match ls with
   | [] -> []                            | [] -> []
   | x::xs ->                            | x::xs ->
      let xs = (mapN f xs)   #1             let xs = (filterN f xs)     #1
      f(x) :: xs             #2             if f(x) then x::xs else xs  #2
```

Both of the functions contain a single recursive call (#1), which isn't tail recursive. In each case the recursive call is followed by an additional operation (#2). The general scheme is that the function first decomposes the list into a head and a tail. Then it recursively processes the tail and performs some action with the head. More precisely, `mapN` applies the `f` function to the head value and `filterN` decides whether the head value should be included in the resulting list or not. The last operation is appending the new head value (or no value in case of filtering) to the recursively processed tail, which has to be done after the recursive call.

To turn these into tail recursive functions, we use the same *accumulator argument technique we saw earlier*. We collect the elements (either filtered or mapped) as we iterate over the list and store them in the accumulator. Once we reach the end, we can return elements that we've collected. Listing 10.9 shows the tail recursive implementations for both mapping and filtering.

Listing 10.9 Tail recursive list processing functions (F#)

```
let map f ls =
   let rec map' f ls acc =
      match ls with
      | [] -> List.rev(acc)
#1
      | x::xs ->
         let acc = f(x)::acc
#2
         map' f xs acc
#3
   map' f ls []
```

```
let filter f ls =
   let rec filter' f ls acc =
      match ls with
      | [] -> List.rev(acc)       #1
      | x::xs ->
         let acc =
            if f(x) then x::acc   #2
            else acc              #2
         filter' f xs acc         #3
   filter' f ls []
```

Let's start by looking at the branch that terminates the recursion (#1). I said that we just return the collected elements, but we're actually reversing their order first by calling `List.rev`. This is because we're collecting the elements in the "wrong" order. We always add to the accumulator list by prepending an element as the new head, so the *first* element we process ends up as the *last* element in the accumulator. The call to the `List.rev` function reverses the list, so we end up returning the results in the right order.

The branch that processes a cons cell is now tail recursive. It processes the element from the head and updates the accumulator as a first step (#2). It then makes the recursive call (#3) and returns the result immediately. The F# compiler can tell that the recursive call is the last step, and optimize it using tail recursion.

We can easily spot the difference between the two versions if we paste them into F# interactive and try to process a large list. For these functions, the depth of the recursion is the same as the length of the list, so we run into problems if we use the naïve version:

```
> let large = [ 1 .. 100000 ]
val large : int list = [ 1; 2; 3; 4; 5; ...]

> large |> map (fun n -> n*n);;                       #A
val it : int list = [1; 4; 9; 16; 25; ...]

> large |> mapN (fun n -> n*n);;                      #B
Process is terminated due to StackOverflowException.
```

#A Tail recursive function works fine
#B Non-tail recursive function causes stack overflow

As you can see, tail recursion is an important technique for recursive processing functions. Of course, the F# libraries contain tail-recursive functions for working with lists, so you don't really have to write your own map and filter implementations like we have here.

However, in chapters 7 and 8 we saw that designing our own data structures and writing functions that work with them is the key of functional programming.

Many of the data structures that you'll create will be reasonably small, but when working with a large amount of data, tail recursion is an essential technique. Using tail recursion we can write code that works correctly on large data sets. Of course, just because a function won't overflow the stack doesn't mean it will finish in a reasonable amount of time–which is why we need to consider how to handle lists efficiently, too.

10.2.2 Processing lists efficiently

Tail recursive functions usually improve efficiency slightly to start with, but usually the choice of algorithm is much more important than micro-optimization of its implementation. Let's demonstrate this with example where we want to add elements to an existing list.

ADDING ELEMENTS TO A LIST

So far we've seen how to append elements to the front of an existing (functional) list. However, what if we wanted to append elements at the end of the list? This sounds like a reasonable requirement, so let's try to implement it. Listing 10.10 shows the difference in performance between inserting at the front of a list and a naïve attempt to insert at the end.

Listing 10.10 Adding elements to a list (F# interactive)

```
> let appendFront el list = el::list                          #1
val appendFront : 'a -> 'a list -> 'a list

> let rec appendEnd el list =                                 #2
     match list with
     | []    -> [el]                                          #A
     | x::xs -> x::(appendEnd el xs)                          #B
val appendEnd : 'a -> 'a list -> 'a list

> #time;;                                                     #3
> let l = [ 1 .. 30000 ];;
val l : int list

> for i = 1 to 100 do ignore(appendFront 1 l);;               #4
Real: 00:00:00.000, CPU: 00:00:00.000

> for i = 1 to 100 do ignore(appendEnd 1 l);;                 #5
Real: 00:00:00.434, CPU: 00:00:00.421
```

#1 Appends simply using cons operator
#2 Appends to the end using a recursive function
#A Append to an empty list
#B Recursive call append to the tail
#3 Turns on time measuring in F# interactive
#4 Executing 'appendFront' 100x takes almost no time
#5 100x 'appendEnd' takes much longer

The implementation of `appendFront` is trivial (#1), because we can just simply construct a new list cell using the cons operator (`::`). On the other hand, appending an element to the end of the list requires writing a recursive function (#2). This follows the normal pattern for recursive list processing, with one case for an empty list and another for a cons cell.

Next, we enter a very useful F# interactive command `#time`, which turns on timing (#3). In this mode, F# will automatically print the time taken to execute the commands that we enter. In this mode, we can see that appending element at the end of large list is much slower. We run this hundred times in a for-loop and the time needed for append to the front is still reported as zero (#4), but appending elements to the end takes a significant amount of time (#5). Any "simple" operation which takes half a second just for a hundred iterations is a concern.

Our appending function isn't tail recursive, but that's not really a problem here. Tail recursion helps us to avoid stack overflow, but it only affects performance slightly. The problem is that functional lists are simply not suitable for the operation that we're trying to execute. Figure 10.5 shows why this operation simply can't be implemented efficiently for functional lists.

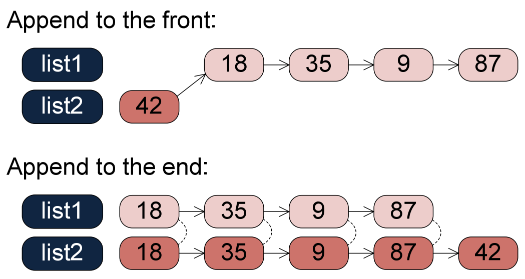


Figure 10.5 When appending element to the front, we just create a new cons cell and reference the original list. However, to append element to the end, we need to iterate over and clone the whole list.

The diagram shows that appending element to the front is easy. Because a list is an immutable data structure, we can create just a single cell and reference the original list. Immutability guarantees that nobody can mutate the original list later, changing the contents of the "new" list behind our back. Compare that with appending an element to the end, which requires changing the last element. Previously the last element "knew" it came last, whereas we need it to have the new element following it. The list is immutable so we can't actually change the information stored in the last element. Instead, we have to clone the last element, which also means cloning the previous element too (so it knows that it's followed by the cloned last element) and so on.

Of course, there are various different data structures and each of them has different operations that can be executed very efficiently. There's always a tradeoff and that's why it is important to choose the right data structure for your problem.

Complexity of algorithms

Computer scientists use very precise mathematical terms to talk about complexity of algorithms, but the concepts behind these terms are important even when we use them very informally. In general, the complexity of an operation tells us how the number of "primitive" steps the algorithm requires depends on the size of the input. It doesn't predict the exact number of steps–just its relationship to the size of the input.

Let's analyze our previous example. Appending an element to the front of the list always involves a single step: creating a new list cons cell. In the formal notation this is written as O(1), which means that the number of steps is constant, no matter how large the list is. Adding an element to the start of a list with a million elements is as cheap as adding an element to the start of a list with just one element!

Appending an element to the end of the list is trickier. If the list has N elements at the beginning, we'll need to process and duplicate N cons cells. This would be written as O(N), which means that the number of steps is roughly proportional to the size of the list: adding an element to the end of a list of size 1000 is roughly twice as expensive as adding an element to the end of a list of size 500.

If we wanted to append for example M new elements to the list, the complexity would be multiplied by M. This means that appending to the front would require O(M) steps, because 1 * M = M. Using similar reasoning, appending to the end would require O(N*M) steps, which could be bigger by an order of magnitude.

So far we've talked about functional lists, the most important collections in functional programming. Let's now take a big leap and look at the collection which exists in almost all imperative programming languages: the humble array. F# is a .NET language, so it can use normal .NET arrays too.

10.2.3 Working with arrays

Arrays correspond very closely to a simple model of computer memory–essentially a sequence of numbered boxes, where you can read or change the value in any box cheaply if you know the number. Arrays form continuous blocks of memory, so the overheads are small and they are very useful for storing larger data sets. However, arrays are allocated in advance: once they are created, the size is fixed. This means we can't add a new value to an existing array, for example.

Arrays are mutable data structures, so we can easily update them. This is sometimes useful, but for a functional programmer, this means that we're losing many guarantees about the program state. First let's look at the basic F# syntax for arrays, as shown in listing 10.11.

Listing 10.11 Creating and using arrays (F# interactive)

```
> let arr = [| 1 .. 5 |];;                    #1
val arr : int array = [|1; 2; 3; 4; 5|]
```

```
> arr.[2] <- 30;;                          #2
val it : unit = ()

> arr;;
val it : int array = [|1; 2; 30; 4; 5|]

> let mutable sum = 0                      #3
  for i in 0 .. 4 do                       #3
     sum <- arr.[i] + sum;;                #3
val mutable sum : int = 42
```

#1 Initialize an array with 5 elements
#2 Change the value at the specified index
#3 Imperative code to sum the elements

Arrays in F# support all basic operations that we'd expect from an array. We start by initializing `arr` using syntax very similar to list initialization (#1). Next, we use the assignment operator to mutate the array and set the value at specified index (#2). Note that when accessing an element in F#, we have to write "." before the square braces that specify the index. The next couple of lines show how we can process an array in an imperative style (#3). It uses a for loop to iterate over all the elements and a mutable value to store sum of them.

Don't worry if you feel slightly dirty looking at listing 10.11–so do I. It just means you're becoming accustomed to the functional style. I wouldn't normally write code like this of course–it's just for the sake of demonstrating the syntax.

Even though arrays are typically used in imperative programming, we can work with them in a very functional style. Aside from the basic operations we've just seen, F# also provides several higher order functions similar to those for working with lists. Let's see how we can use arrays *without* feeling dirty.

USING ARRAYS IN A FUNCTIONAL WAY

We'll start by looking at an F# example that shows couple of useful higher order functions for working with arrays from the F# library and then implement the same functionality in C#. Listing 10.12 shows a script that first initializes an array with random numbers and then calculates their squares.

Listing 10.12 Functional way of working with arrays (F# interactive)

```
> let rnd = new System.Random()
val rnd : System.Random

> let numbers = Array.init 5 (fun _ -> rnd.Next(10))       #1
val numbers : int[]

> let squares = numbers |> Array.map (fun n -> (n, n*n))   #2
val squares : (int * int)[]

> for sq in squares do                                     #A
     printf "%A " sq                                       #A
```

```
(1, 1) (0, 0) (7, 49) (2, 4) (2, 4)
```

#1 Initialize array using the given function
#2 Calculate new element from the original
#A Print tuples from the resulting array

The first higher order function that we're working with is `Array.init` (#1). It takes the length of the array we want to create and a function as its arguments. The supplied function is called to calculate a value at every index of the array and it is given the index as an argument. In our example, we don't need the index, so we use the underscore pattern. In the body of the function, we just generate a new random number.

The second function is `Array.map` (#2), which does exactly the same thing as the `List.map` function has become so familiar. In this example we use it to create an array of tuples where each element of the result contains the original integer and its square.

The interesting thing about this example is that we don't use the assignment operator anywhere in the code. The first operation simply constructs a new array. The second one doesn't modify it, but instead returns another newly-created array. Even though arrays are mutable, we can work with them using high order functions which never actually mutate them in our code. This example would have worked in a very similar fashion if we had used functional lists.

CHOOSING BETWEEN ARRAYS AND LISTS

We've seen that arrays and lists can be used in a similar ways, so you need to know when to pick which option. The first point to consider is whether the type is mutable or not. Functional programming puts a strong emphasis on immutable data types and we'll see practical examples showing why this is valuable in the next chapter and in chapter 14. We can work with arrays in a very functional way, but lists give us much stronger guarantees about correctness of our programs.

Another point is that some operations are easier or more efficient with one data type than the other. For example, appending an element to the front of a list is much easier than copying the contents of one array into a slightly bigger one–but on the other hand, arrays are much better for random access. Finally, operations that process arrays are often somewhat faster. We can see this with a simple example using the `#time` directive:

```
let l = [  1 .. 100000  ]
let a = [| 1 .. 100000 |];;
for i in 1 .. 100 do ignore(l |> List.map  (fun n -> n))  #A
for i in 1 .. 100 do ignore(a |> Array.map (fun n -> n))  #B
```

#A Operation takes 885ms
#B Operation takes 109ms

In general, arrays are useful if you need to work efficiently with large data sets. However, in most situations you should use aim for clear and simple code first, and functional lists usually lead to greater readability.

The previous examples have shown us how to use some of the basic operations that are available for arrays. However we'll often need to write some similar operations ourselves. Listing 10.14 shows a function that works with arrays in a functional style: it takes one array as an argument and returns a new one calculated from the inputs. The function is used to "smooth" or "blur" an array of values, so that each value in the new array is based on the corresponding value in the original *and* the values either side of it.

Listing 10.14 Functional implementation of blur for arrays (F#)

```
let blurArray (arr:int[]) =                                           #A
   let res = Array.create arr.Length 0                                #B
   res.[0] <- (arr.[0] + arr.[1]) / 2                                 #C
   res.[arr.Length-1] <- (arr.[arr.Length-2] + arr.[arr.Length-1]) / 2 #C
   for i in 1 .. arr.Length - 2 do
      res.[i] <- (arr.[i-1] + arr.[i] + arr.[i+1]) / 3                #2
   res
val blurArray : int[] -> int[]
```

#A Type annotation, so we can use "Length" member
#B Initialize empty result
#1 Calculate value at borders
#2 Calculate average over 3 elements

The function starts by creating an array for storing the result, which has the same size as the input. It then calculates the values for the first and the last element (#1) of the new array, which are average value over two elements. These are calculated separately from the rest of the array because they're edge cases which don't quite fit the rest of the pattern. Finally it iterates over the elements in the middle of the array, taking the average of three values and writing the results to the new array (#2).

The function uses mutation internally. It creates an array filled with zeros at the beginning and later writes the calculated values to this array. However, this mutation is not visible from outside: by the time the caller is able to use the array, we've finished mutating it. When we use this function, we can safely use all the normal functional techniques:

```
> let ar = Array.init 10 (fun _ -> rnd.Next(20));;              #A
val ar : int [] = [|14; 14; 4; 16; 1; 15; 5; 14; 7; 13|]

> ar |> blurArray;;                                             #B
val it : int [] = [|14; 10; 11; 7; 10; 7; 11; 8; 11; 10|]

> ar |> blurArray |> blurArray |> blurArray;;                   #C
val it : int [] = [|7; 8; 9; 9; 9; 9; 9; 9; 8; 8|]
```

#A Initialize random array
#B Blur the array once
#C Blur three times using pipelining

The `blurArray` function has type `int[] -> int[]`, which makes it very compositional. In the second command, we use the pipeline operator to send a randomly generated array to this function as an input and the F# interactive console automatically prints the result. The final command shows that we can also call the function several times in a sequence in the same way we would use map or filter operations on a list.

You can probably imagine extending this example to process images, turning our `blurArray` function into a real blur filter working with bitmaps. If you want to try this out, you'll need to use the `Array2` module which has functions for working with 2D arrays, and the .NET `Bitmap` class with functions such as `GetPixel` and `SetPixel` for reading and writing graphical data. We'll get back to this problem later in chapter 14 where we'll also discuss how to use parallelism to perform the operation more efficiently.

Having seen how we can use arrays neatly in F#, we'll turn our attention back to C#. Of course all C# programmers already know the basics of how to use arrays–what we're interested in is how we can write C# code which uses arrays in a functional style.

USING ARRAYS IN A FUNCTIONAL WAY IN C#

You can already use many functional constructs with arrays in C# 3.0 thanks to LINQ to Objects. However, most LINQ operators don't return arrays: if you call `Enumerable.Select` on an array, it will return the result as `IEnumerable<T>`. In some situations we'd prefer to keep the results in an array, and we may wish to avoid the overhead of calling `Enumerable.ToArray` to copy the result sequence back into an array. Fortunately, we can implement our own `Select` method easily enough. Listing 10.13 shows a C# implementation of the functions we used in the earlier F# example.

Listing 10.13 Methods for functional array processing (C#)

```
static class ArrayUtils {
   public static T[] Create<T>(int length, Func<int, T> init) {     #1
      T[] arr = new T[length];
      for (int i = 0; i < length; i++) arr[i] = init(i);
      return arr;
   }
   public static R[] Select<T, R>(this T[] arr, Func<T, R> map) {   #2
      R[] res = new R[arr.Length];
      for (int i = 0; i < arr.Length; i++) res[i] = map(arr[i]);
      return res;
   }
}
```

#1 Initializes array using the given function
#2 Extension method that returns an array

The `Create` method is a normal static method (#1). It takes a function `fInit` as an argument and uses it to initialize the elements of the array. The `Select` method is an extension method that applies a mapping function to each element in the original array, and returns the result as a new array. It hides the standard `Select` operation provided by LINQ. We can use these methods in a similar way to the earlier corresponding F# functions:

```
var rnd = new Random();
var numbers = ArrayUtils.Create(5, n => rnd.Next(20));                #A
var squares = numbers.Select(n => new { Number = n, Square = n*n });  #B

foreach (var sq in squares)
   Console.Write("({0}, {1}) ", sq.Number, sq.Square);
```

#A Fill array with random numbers
#B Store the results in an anonymous type

Just like in the F# version, we don't modify the array once it is created. From a high-level perspective, it is a purely functional code working with an immutable data structure. Of course we are actually performing mutations–but only within the `ArrayUtils` class, and only on collections that haven't been exposed to any other code yet. The mutation isn't observable to the outside world. This way of writing code is even more valuable in C#, where functional lists are harder to use than they are in F#.

Our final topic in the chapter deals with *continuations*. These can be somewhat hard to wrap your head around, but once you understand them there are some amazing possibilities. The good news is that if you've ever written any asynchronous code in .NET, you've already been using continuations in some sense - but F# makes them a lot easier. We'll look at them in more detail in chapter 13, but using continuations is an interesting optimization technique for recursive functions, which is the aspect we'll concentrate on here.

10.3 Introducing continuations

We started this chapter with a discussion about recursive calls. We've seen an important technique called tail recursion which allows us to perform a recursive call without allocating any space on the stack. Thanks to tail recursion, we can write functional list processing functions that can handle very large data sets without breaking into a sweat.

Tail recursion isn't a silver bullet though; I mentioned earlier that not every function can be rewritten to use it. If a function needs to perform two recursive calls then it clearly cannot be written in this way. (They can't *both* be the very last thing to be executed before returning, after all.)

10.3.1 What makes tree processing tricky?

Let's take a simple example working with trees. Listing 10.14 declares a type representing a tree of integers, and shows a recursive function that sums all the values in the tree.

Listing 10.14 Tree data structure and summing elements (F# interactive)

```
> type IntTree =                                    #1
     | Leaf of int
     | Node of IntTree * IntTree
type IntTree = (...)

> let rec sumTree(tree) =                           #2
     match tree with
     | Leaf(n)    -> n                              #A
     | Node(l, r) -> sumTree(l) + sumTree(r)        #B
val sumTree : IntTree -> int
```

#1 Tree is a leaf with value or a node containing sub-trees
#2 Recursive function calculating sum of elements
#A Sum of a leaf is its value
#B Recursively sum values in the sub-trees

The `IntTree` type (#1) used for representing the tree is a discriminated union with two options. Note that this is actually quite similar to the list type! A tree value can represent either a leaf that contains an integer or a node. A node doesn't contain numeric value, but it has two sub-trees of type `IntTree`. The recursive function for calculating sum (#2) uses pattern matching to distinguish between these two cases. For a leaf, it simply returns the numeric value; for a node, it needs to recursively sum the elements of both the left and right sub trees and then add the two values together.

If we look at the `sumTree` function, we can see that it isn't tail recursive. It performs a recursive call to `sumTree` to sum the elements of the left sub-tree and then needs to perform some additional operations. More specifically, it still needs to sum the elements of the right sub-tree and finally it has to add these two numbers. A function like this cannot be written in a tail recursive way, because it has two recursive calls to perform. The last of these two calls could be made tail-recursive with some effort (using some sort of accumulator argument), but we'd still have to do one ordinary recursive call! This is annoying, because for some kinds of large trees, this implementation will fail with a stack overflow.

We need to think of a different approach. First let's think about what trees might actually look like. Figure 10.6 shows two different examples.

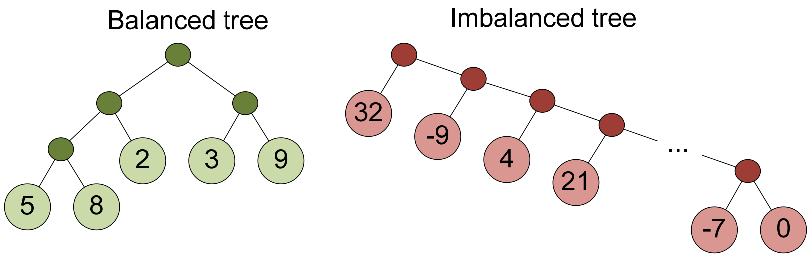


Figure 10.6 Example of balanced and imbalanced trees. Dark circles correspond to the "Node" case and light circles containing values correspond to the "Leaf" case.

The first tree in figure 10.6 is a fairly typical case where the elements of the tree are reasonably divided between the left and the right sub-trees. This isn't too bad, as we never end up recursing particularly deeply. (With our current algorithm, the maximum recursion depth is the longer path which exists between the root of the tree and a leaf.) The second example is much nastier. It has many Node elements on the right side, so when we process it recursively we'll have to make around 100 recursive calls. The difference between the handling of these two trees is shown in this code snippet:

```
> let tree = Node(Node(Node(Leaf(5), Leaf(8)), Leaf(2)),
                 Node(Leaf(2), Leaf(9)));;
  sumTree(tree);;
val it : int = 26

> let imbalancedTree =
      test2 |> List.fold_left(fun st v ->
         Node(Leaf(v), st)) (Leaf(0))                              #A
   sumTree(imabalncedTree);;
Process is terminated due to StackOverflowException.               #B
```

#A Add node with previous tree on the right
#B Deep recursion causes stack overflow!

The first command creates a very simple tree and sums the leaf values. The second command uses the `fold_left` function to create a tree similar to the imbalanced example in figure 10.6, but bigger. It starts with a leaf containing zero and in each step appends a new node with a leaf on the left and the original tree on the right. It takes the numbers from the list that we've created earlier in listing 10.2 and that contains 100000 random numbers between -50 and +50. As a result, we'll get a tree with a height of 100000 nodes. When we try to sum leaves of this tree we get a stack overflow. Of course, this isn't a particularly typical situation, but we can still encounter it in our tree processing code. Luckily, continuations give us a way to write functions that work correctly even on trees like this one.

10.3.2 Writing code using continuations|

The problem is that we want to make a tail recursive call, but we still have some code that we want to execute after the tail recursive call completes. This looks like a very tricky problem, but there is an interesting solution. We'll take all the code that we want to execute after the recursive call completes and provide it as an argument to the recursive call. This means that function that we're writing will contain just a single recursive call.

Don't worry if this all sounds a bit wacky. Think of it as just another sort of accumulator argument - instead of just accumulating values, we're accumulating "more code to run later". Now, how can we take the remaining code and use it as an argument to a function? This is of course possible thanks to first class functions and this last argument is called a *continuation* because it specifies how the execution should continue.

This will all become much clearer after looking at some practical examples. Listing 10.15 shows a simple function implemented first in the normal style and then using continuations. We're using C# here so that there's only one new concept to understand, but bear in mind that C# doesn't support tail recursion: this technique can't be used as an optimization for recursion in C#. (Continuations are still useful in C#, just not for recursion.)

Listing 10.15 Writing code using continuations (C#)

```
int StrLength(string s) {   #1     void StrLengthC(string s,
   return s.Length;                   Action<int> cont) {             #2
}                                    cont(s.Length);                  #3
                                   }
void TestLength() {         #4     void TestLengthC() {
   int x1 = StrLength("One");        StrLengthC("One", x1 =>          #5
   int x2 = StrLength("Two");           StrLengthC("Two", x2 =>       #6
   Console.WriteLine(x1 + x2);             Console.WriteLine(x1 + x2)
}                                    ));
                                   }
```

In both versions, we first declare a function that calculates length of the string. In the usual programming style (#1) it gives the result as a return value. When using continuations (#2), we add a function (continuation) as the last argument. To return the result, the `StrLengthC` function invokes this continuation (#3).

The next function called `TestLength` (#4) first calculates length of two strings; adds these values and finally prints the result. In the version using continuations, it includes only a single top-level call to the `StrLengthC` function (#5). The first argument to this call is a string and the second one is a continuation. The top-level call is the last thing that the function does, so in F# it would be executed using a tail call and it wouldn't occupy any stack space.

The continuation receives the length of the first string as an argument. Inside it, we call `StrLengthC` for the second string. Again, we give it a continuation as a last argument and once it is called, we can sum the two lengths and print the result. In F#, the call inside the continuation (#6) would be again a tail call, because it is the last thing that the code in the lambda function does. Let's now look how we can use this style of programming to optimize our previous function for summing elements of a tree.

TREE PROCESSING USING CONTINUATIONS

To change our previous implementation of the `sumTree` function into a version that uses continuations, we'll first add additional argument (continuation) to the function. We'll also need to update the way how the function returns the result. Instead of simply returning the value, we'll call the continuation given as the argument. The final version of the code is shown in the listing 10.16.

Listing 10.16 Sum elements of a tree using continuations (F# interactive)

```
> let rec sumTreeCont tree cont =
     match tree with
     | Leaf(n)    -> cont(n)                          #1
     | Node(l, r) -> sumTreeCont l (fun n ->          #2
        sumTreeCont r (fun m ->                       #3
           cont(n + m)))                              #4
val sumTreeCont : IntTree -> (int -> 'a) -> 'a        #5
```

#1 Return value by calling the continuation
#2 Recursively sum left sub-tree
#3 Then recursively sum right sub-tree
#4 Finally, call the continuation with the result
#5 Inferred type signature

Modifying the branch for the leaf case is quite easy, because it previously just returned the value from the leaf. The second case is far more interesting. We're using similar pattern to the one in the previous C# example. We call the function to sum the elements of the left sub-tree (#2) (note that this is a tail recursion!) and give it a lambda function as the second argument. Inside the lambda we do a similar thing for the right sub-tree (#3) (note that this is again a tail recursive call!). Once we have sums of the both sub-trees, we invoke the continuation that we originally got as the argument (#4).

Another interesting thing about the function that we've just written is its type signature (#5). As usual, we didn't write any types explicitly and F# inferred the types for us. The function takes the tree as the first argument and the continuation as the second one. However, the continuation now has a type `int -> 'a` and the overall result of the function is `'a`. In other words, the return type of the whole function is the same as return type of the continuation.

Earlier in the discussion, I highlighted that all recursive calls in the code are now tail-recursive, so we can try this function on the imbalanced tree that failed in the previous version:

```
> sumTreeCont imbalancedTree (fun r ->
      printfn "Result is: %d" r)                            #A
Result is: 8736                                             #B
val it : unit = ()

> sumTreeCont imbalancedTree (fun a -> a)                   #C
val it : int = 8736
```

#A Print the result inside the continuation
#B This version works fine on large trees!
#C Returning sum from the continuation

As you can see, the code now works on very large trees without any trouble. In the first example, we print the result directly in the continuation and the continuation doesn't return any value, so overall result of the expression is `unit`. In the second case we give it an *identity function* (a function which just returns its argument) as the continuation. The return type of the continuation is `int` and the value returned from the call to `sumTreeCont` is the sum of all the elements in the list.

10.4 Summary

In this chapter, we explored various topics related to the efficiency of functional programs and we discussed how to deal with large amounts of data in a functional way. Since most of the functional programs are implemented using recursion, a large part of the chapter was dedicated to this topic.

We saw that when using recursion we have to write our code carefully to avoid errors caused by the stack overflowing if the recursion level becomes too deep. In the beginning of the chapter we looked at a technique called tail recursion that allowed us to rewrite some familiar list processing functions (such as `map` and `filter`) in a way that makes them resistant to stack overflow. However, tail recursion alone cannot help us in every situation, so we also looked at continuations and used them to write a robust version of a simple tree processing function.

As well as avoiding stack overflow, we also looked at techniques for optimizing the performance of processing functions. In particular, we looked at memoization, which allows us to cache results of functions without side-effects. Effective optimization relies on complexity analysis, so we looked at functional data structures and their performance characteristics. We have to be careful when choosing algorithms and operations, as some differences which look small–such as whether we add elements to the head or tail of functional lists–can have a very significant impact on performance. We also talked about arrays, which are not primarily functional data structures, but can be used functionally if we’re careful.

In the next chapter, we'll continue our exploration of common tricks for implementing algorithms in a functional language. Many of the topics from the following chapter are related to the use of immutable data types and mathematical clarity of functional programming.

11
Refactoring and testing functional programs

One of the themes of this book is the claim that functional programming makes it easier to understand code just by reading it. This is particularly important when you need to modify an unfamiliar program or implement some behavior by composing existing functions. Function programming makes refactoring easier due to both clarity and modularity: you can make improvements to the code and be confident that the change doesn't break other parts of the program.

Inspiration from mathematics

As with many things in functional programming, the idea of modifying code without changing its meaning is closely related to math, because operations that don't change the meaning of an expression are basis of many mathematical tasks. For example, we can take a complex equation and simplify it to get an equation that is easier to read but means the same thing. Let's take the following equation: $y = 2x + 3(5 - x)$. If we multiply the expression in parentheses by 3, we can write it as: $y = 2x + 15 - 3x$, which in turn can be simplified to: $y = 15 - x$.

Another technique we can learn from math is substitution. For example if we have two equations $y = x/2$ and $x = 2z$, we can substitute the right-hand side of the second one into the first one and we'll get (after simplification) $y = z$. The important point is that by substituting correct equation into another one, the substituted equation cannot suddenly become incorrect. This technique appears in functional programming as composition.

Functional programming is closely related to mathematics, so it's unsurprising that some of the techniques used in algebra can be applied to functional programs too. In the programming world, the simplification of equations corresponds to *refactoring*, which is the centerpiece of this chapter. In particular, we'll look at reducing code duplication and discuss code dependencies. We'll see that immutability is the key that makes it possible to refactor code without changing its meaning.

Substitution is also a form of refactoring, but we'll see that it has other important practical benefits, particularly when unit testing. Substitution allows us to focus on testing primitive functions and spend much less time testing functions that are composed from simple building blocks, because the composition can't break already tested components.

Finally, we'll take a look at a topic that is very closely related to refactoring. When a program lacks side-effects, we should get the same result regardless of the order in which the individual parts are executed. A value can be calculated as soon as it is described, or we can delay execution until the value is really needed. This technique is called *laziness (or lazy evaluation)*, and we'll see some of the practical benefits when we explore potentially-infinite data structures.

11.1 Refactoring functional programs

Refactoring is an integral part of many modern development methodologies. In some languages, this technique is also supported by IDEs such as the C# editor in Visual Studio. Most of the refactoring techniques have been developed for the object-oriented paradigm, but we'll be looking at it from a functional point of view.

WHAT IS REFACTORING?

Refactoring is the process of modifying source code to improve its design without changing its meaning. The goal of refactoring is to make the code more readable, easier to modify or extend in the future, or to improve its structure. A very simple example of refactoring is renaming a method to make the name more descriptive; another is turning a block of code into a method and reusing it to avoid code duplication.

Refactoring allows us to write code that works first, and then make it "clean". Performing these two tasks separately simplifies testing because refactoring shouldn't affect the behavior of the application. While some changes such as renaming are fairly simple (particularly with help from tools), others can involve more thoughtful consideration.

If you switch the order of two statements, will the code behave the same way afterwards? With imperative code using side-effects, you'd have to look carefully at the two statements. Functional programming makes reasoning about the code easier, so refactoring becomes easier too. We'll take a look at several examples in this section, but let's start with a very common functional refactoring that removes code duplication.

11.1.1 Reusing common code blocks

One of the best programming practices is to avoid duplicating the same code in multiple places. If you have two routines that look very similar it is worth considering how they could be merged into one. The new routine would take a new argument that specifies what code path to follow in the part that was originally different.

In functional programming, we have one very powerful weapon - the ability to use function values as arguments. This makes it much easier to parameterize a function or method. We'll demonstrate it using an example that works with location data. We've used this data structure already in previous chapters, so the following snippet is just for a reminder:

```
let loadPlaces() =
  [ ("Seattle", 594210);   ("Prague", 1188126)
    ("New York", 7180000); ("Grantchester", 552)
    ("Cambridge", 117900) ]
```

The data structure is simple, but it's close to what we could use in a real-world application. Instead of using tuples for storing the name and the population, we'd probably use records or object types, and we might load the data from a database.

Now that we have the data, we may want to run some kind of report on it. Listing 11.1 shows two functions: one prints a list of cities with more than 200,000 inhabitants, and the other prints all the locations in alphabetical order. In a real application this might generate an HTML report, but we'll keep things simple and print it to the console as plain text.

Listing 11.1 Printing information about places (F# interactive)

```
> let printBigCities() =
     let places = loadPlaces()
     printfn "== Big cities ==" #1
     let sel =
        List.filter (fun (_, p) ->
           p > 100000) places   #2
     for n, p in sel do
        printfn " - %s (%d)" n p;;
val printBigCities : unit -> unit

> printBigCities();;
== Big cities ==
 - Seattle (594210)
 - Prague (1188126)
 - New York (7180000)
 - Cambridge (117900)
```

```
> let printAllByName() =
     let places = loadPlaces()
     printfn "== All by name =="#3
     let sel = List.sort_by     #4
                  fst places    #4
     for n, p in sel do
        printfn " - %s (%d)" n p;;
val printAllByName : unit -> unit

> printAllByName();;
== All by name ==
 - Cambridge (117900)
 - Grantchester (552)
 - New York (7180000)
 - Prague (1188126)
 - Seattle (594210)
```

The two functions have very similar structure, but there are some differences. The most important difference is that they select the list of places to print in different ways. The `printBigCities` function (#2) filters places using `List.filter`, while `printAllNames` (#4) uses `List.sort_by` to reorder them. They also differ in terms of the report title that's printed (#1) (#3).

On the other hand, there are many common aspects. Both functions first call `loadPlaces` to obtain the collection of places, then process this collection in some way and finally print the result to the screen.

When refactoring the code, we want to write a single function that can be used for both tasks. We also want to make the code more extensible. It should be possible to use the printing function with a completely different strategy. For example, if we were creating a crossword, we might look for cities with the specified length starting with a particular letter. This means that we should be able to provide almost any strategy as an argument. Functional programming gives us a great way to do this kind of parameterization using functions.

Listing 11.2 shows a higher order function `printPlaces`, and we'll soon see that we can use it to replace both of the functions from the previous listing.

Listing 11.2 Reusable function for printing information (F# interactive)

```
> let printPlaces title select =
     let places = loadPlaces()
     printfn "== %s ==" title                               #1
     let sel = select(places)                               #2
     for name, pop in sel do
        printfn " - %s (%d)" name pop
  ;;
val printPlaces : string ->                                 #3
    ((string * int) list -> #seq<string * int>) -> unit     #3
```

#1 Title is passed as an argument
#2 Strategy for processing places is a function
#3 Type signature inferred by F#

Our new function has two parameters. These specify what to do in places where the original two functions were different from each other. The first is the report title (#1) and the second is a function which selects the places to be printed (#2). We can learn more about this function by looking at its type in the printed type signature (#3).

The argument of the function is a list of tuples, each of which contains a string and an integer. This is our data structure for representing places. We would expect the return type of the function to be the same, because the function returns collection of places in the same data format, but the type inferred by F# is `#seq<string * int>`. The difference is that instead of `list` it inferred the `#seq` type.

This choice is interesting for two reasons. First of all, `seq<'a>` is a common interface implemented by all collections and is an alias for the standard .NET `IEnumerable<T>` type. This means that the function *can* return a list, but could equally return an array, because the only thing we need is the ability to iterate over all the elements in the collection. We'll go into more detail about sequences in the next chapter, but if you're familiar with LINQ to Objects this should be familiar territory: most of the common operators work with (and return) `IEnumerable<T>`.

The second interesting point is the hash symbol. It means that the returned collection doesn't have to be up-cast to the `seq<'a>` type explicitly. This means we can provide a function which is actually typed to return a `list<'a>`, for example. In the strictly typed sense, this is a different type of function, but the hash symbol adds some very useful flexibility. Most of the time you don't need to worry about this very much - it just means that the compiler inferred that the code can be more generic.

Now that we have the function, we need to show that it can really be used in place of the two functions that we started with. The following example shows arguments we can supply to get the same behavior as the original functions:

Listing 11.3 Working with 'printPlaces' function (F#)

```
printPlaces "Big cities" (fun places ->                           #1
   List.filter (fun (_, s) -> s > 200000) places)

printPlaces "Big cities" (List.filter (fun (_, s) -> s > 200000))   #2
printPlaces "Sorted by name" (List.sort_by fst)                     #3
```

#1 Printing only big cities
#2 The same call using partial function application
#3 Sorting using partial function application

The only interesting aspect of the first example (#1) is the lambda function that we use as the second parameter. It takes the data set as an argument and filters it using `List.filter` to select only cities with more than 200 thousands inhabitants. The next example (#2) shows that we can write the call more succinctly using partial function application. Finally, in the last example (#3) we use `List.sort_by` to sort the collection.

As you can see in the last listing, using the function that we created during refactoring is quite easy. It could be used to print different lists just by specifying another function as the second argument.

The refactoring we performed in this section relied on the ability to use functions as arguments. C# has the same ability, so the same kind of refactoring can be applied very effectively there, using delegates. We could specify the transformation argument either as a lambda expression or by creating the delegate from another method with an appropriate signature.

Another functional principle that is very valuable when refactoring code is the use of immutable data. The impact here is slightly more subtle than just being able to express differences in behavior using functions, but it's no less important.

11.1.2 Tracking dependencies and side-effects

One of the many benefits of immutability is the clarity it provides. If a function takes a list as an argument and returns a number, you can safely assume that it calculates the result based on the list content, but does *not* modify the list. We don't have to look at any code to reach that conclusion; we don't have to examine the implementation *or* any other functions that it calls. Let's start by looking at an example that demonstrates how easy it is to introduce errors when using mutable objects.

Using mutable data structures

In listing 11.4 you can see two functions that work with a collection storing names of places from the previous example. This time, we're using C# and storing the names in the standard `List<T>` type, which is mutable.

Listing 11.4 Working with places stored in List<T> (C#)

```
List<string> LoadPlaces() {                                          #1
   return new List<string> { "Seattle", "Prague",
      "New York", "Grantchester", "Cambridge" };
}
void PrintLongest(List<string> names) {                              #2
   var longest = names[0];                                           #A
   for(int i = 1; i < names.Count; i++)
      if (names[i].Length > longest.Length) longest = names[i];      #B
   Console.WriteLine(longest);
}
void PrintMultiWord(List<string> names) {                            #3
   names.RemoveAll(s => !s.Contains(" "));                           #C
   Console.WriteLine("With space: {0}", names.Count);
}
```

#1 Returns list containing names
#2 Prints place with the longest name
#A Start with the first place
#B A name is longer than the longest so far
#3 Print the count of multi-word names
#C Remove all single-word names

The code first shows a function that loads some sample data (#1). It's like our `loadPlaces` function from earlier, but without the population values. Next, we implement two processing functions. The first one (#2) finds the place with the longest name; the second (#3) determines how many names contain more than one word by removing any name that doesn't contain a space. Even though the method uses lambda function syntax, it's definitely not functional: the `RemoveAll` method modifies the `names` collection. If we wanted to use these functions later in our program, we could write the following code:

```
PrintMultiWord(LoadPlaces());   #A
PrintLongest(LoadPlaces());     #B
```

#A Prints '1'
#B Prints 'Grantchester'

This gives the correct results. However we're calling the `LoadPlaces` function twice, which seems to be unnecessary. If the function actually loaded data from a database, it would be better to retrieve the data only once for performance reasons. A simple refactoring is to call the function once and store the places in a local variable:

```
var places = LoadPlaces();
PrintMultiWord(places);      #A
PrintLongest(places);        #B
```

#A Prints '1'
#B Prints 'New York'

However, after this simple change we get incorrect results! If you've been following the source code carefully, you've probably already spotted the problem, but it's still subtle enough to potentially cause confusion and head-scratching. The problem is that `List<T>` is a mutable data structure and the function `PrintMultiWord` accidentally mutates it when it calls `RemoveAll`. When we call `PrintLongest` later in the code, the collection places contains only a single item, which is "New York". Now let's look why we couldn't make similar mistake if we used immutable data structures.

USING IMMUTABLE DATA STRUCTURES

We implemented immutable lists for C# in chapter 3 and called the type `FuncList<T>`. We didn't implement all the standard .NET patterns, so the source code for this chapter (available on the book's web site) extends the type by implementing `IEnumerable<T>` and standard some LINQ methods including `Where` and `Select`. The implementation is relatively simple, so we won't look at it in detail.

Using this improved version of the immutable list, we can rewrite the imperative code we've just seen. The `LoadPlaces` and `PrintLongest` methods don't change very much, so we've omitted them here. However, the `PrintMultiWord` method is more interesting: we can't use our previous strategy of using `RemoveAll`, because the `FuncList` type is immutable. Earlier we used this method to remove all single-word names from the collection. This side-effect made the method harder to reason about. Using immutable types, we can't introduce any side-effects in the same way, so if we want the same kind of results we have to be more explicit about it, as shown in listing 11.5.

Listing 11.5 Implementation of 'PrintMultiWord' using immutable list (C#)

```
FuncList<string> PrintMultiWord(FuncList<string> names) {
   var namesSpace = names.Where(s => s.Contains(" "));            #1
   Console.WriteLine("With space: {0}", namesSpace.Count());
   return namesSpace;                                             #2
}
```

#1 Create list containing names with spaces
#2 Return the list

We can't modifying a collection when we're working with immutable data structures, so the method first creates a new collection that contains only multi-word names (#1). We've also made the side-effect from the previous implementation explicit, so the method now returns the new collection. Of course, it isn't really a side-effect at all now - it's just a return value. However, it achieves the same result of making the multi-word names list available to the caller if they want it.

Our first example was searching for the longest name from all the names and our second example (which printed "New York") returned the longest name containing a space. Listing 11.6 shows how both can be implemented both of these examples using our new function.

Listing 11.6 Printing the longest and the longest multi-word name (C#)

```
FuncList<string> pl =                     FuncList<string> pl1 =
   LoadImmutablePlaces();                    LoadImmutablePlaces();
```

```
PrintMultiWord(pl);                  var pl2 = PrintMultiWord(pl1); #2
PrintLongest(pl);       #1           PrintLongest(pl2);             #3
```

#1 Prints 'Grantchester'
#2 Returns filtered list
#3 Prints 'New York'

This example demonstrates that using immutable data types makes it more obvious how constructs depend on each other. If we know that none of the functions can alter the data structure, it's easier to reason about the program and we can say what kinds of refactorings are valid. In the example on the left side, we could change the order of `PrintMultiWord` and `PrintLongest` and they would still print the same results (just in the opposite order). On the other hand, we can't change the order of the calls in the right side of listing 11.6, because the value `pl2` is result of the first call (#2).

This means that when refactoring functional code, we can track dependencies of the computations more easily. We can see that a function depends on other calls if it takes a result of these calls as an argument. Because this is explicitly visible in the code, we can't make accidental refactoring errors because incorrectly modified code will not compile. This is also very useful when testing the code using unit tests.

11.2 Testing functional code

Neither functional programming nor any other paradigm can eliminate bugs entirely or prevent us from introducing bugs when making changes to existing code. This is one reason behind the widespread adoption of unit testing. The good news is that most of the unit testing techniques that you already use when testing C# code can be applied to F# programs as well. Additionally, functional programming and F# make testing easier in many ways.

Choosing a unit testing framework for F#

As we saw in chapter 9, we can write standard classes in F#, so any of the unit testing frameworks for .NET work as normal. On the other hand, why should we write unit tests in F# as members of a class rather than simply using functions declared with let bindings? Classes certainly have some benefits, such as enabling sophisticated setup and teardown code. However, most of the unit tests that we'll write benefit from using the simplest possible syntax.

In this chapter we'll use the **xUnit.net** framework. This works with standard F# functions as well as F# classes. Under the hood, F# functions written using let bindings are compiled into static methods of a class. When we wrap the code inside a module, that module is used to contain the function. Otherwise, F# generates a class based on the name of the file. The **xUnit.net** framework supports unit tests that are implemented as static methods without applying a special attribute (such as `TestFixture`) to the class, which makes it friendlier to F# programmers. If you don't have **xUnit.net** installed, you can get the latest version from http://www.codeplex.com/xunit

When I've mentioned testing so far, I've usually talked about checking whether the code works immediately after writing it in the F# interactive shell. If you're a veteran of unit testing you may well have been mentally screaming that a test which can't be reproduced later on is hardly worth running. Well, let's see how this kind of test can evolve into a unit test.

11.2.1 From the interactive shell to unit tests

Testing code interactively is valuable when you're writing the initial implementation, but we'd also like to make sure that the code keeps giving the same results later even if change it. This can be done very easily by turning the one-off interactive test code into a solid unit test that we keep alongside our production code and run repeatedly. You may be surprised at how small a change is required to achieve this.

Testing programs in F# interactive

Let's demonstrate the whole process from the beginning. We'll use two functions similar to `PrintLongest` and `PrintMultiWord` from the previous section, but this time we'll implement them in F#. As you can see in listing 11.7, we'll use the interactive shell slightly differently.

Listing 11.7 Testing code interactively using xUnit.net (F# interactive)

```
> #if INTERACTIVE                                                   #1
  #r @"C:\Programs\Development\xUnit\xunit.dll"                     #1
  #endif                                                            #1
  open Xunit;;

> let getLongest(names:list<string>) =                              #2
     names |> List.max_by (fun name -> name.Length);;               #2
val getLongest : list<string> -> string

> let test = [ "Aaa"; "Bbbbb"; "Cccc" ];;
val test : string list = ["Aaa"; "Bbbbb"; "Cccc"]

> Assert.Equal("Bbbbb", getLongest(test));;                         #3
val it : unit = ()
```

#1 Reference the xUnit.net library
#2 Returns the longest name
#3 Test the function using xUnit.net

First of all, we need to place the code into a file with an extension of `.fs` such as `Program.fs` (as opposed to `fsx` files that represent interactive scripts) because we want to compile the program into a .NET assembly. Also, we need to add a reference to the xUnit.net core library and use some of its features interactively. This is simply a matter of using the normal "Add Reference" dialog box in Visual Studio. However, we also want to run the code interactively, so we need to load the library in F# interactive. We'd usually do that using `#r` directive, but this directive is allowed only in F# scripts (FSX files). Fortunately, F# supports conditional compilation and defines the `INTERACTIVE` symbol when running the code from the command shell, which mean the initial part of the listing (#1) will work whether we're running it interactively or not.

Next we implement the function for finding the longest name from a given list (#2). The code is quite simple because it uses a higher order function from the F# library. This function selects the element for which the given function returns the largest value. Once we have the function, we test it in the next two lines. The most interesting line is the last one (#3) where we use the `Assert.Equals` method. This is imported from the `Xunit` namespace and verifies that the actual value (given as the second argument) matches the expected value (the first argument). The method throws an exception if that's not the case–the fact that it just returned `unit` as the result means the test passed.

WRITING UNIT TESTS IN F#

If we write our immediate testing code in this manner, it's very easy to turn it into a unit test and make it part of a larger project. We'll see how to do that using xUnit.net soon, but first let's try to write another call that should be definitely covered by the unit tests: calling the `getLongest` function with a `null` value as the argument:

```
> getLongest(null);;
Program.fs(24,12): error FS0043: The type 'string list'
does not have 'null' as a proper value
```

We haven't tried that before, and as you can see F #interactive reports a compile-time error rather than an exception. This means that we can't even *write* code like that, which means that if we're only using the function from F# we don't need to test that possibility. Values of types declared in F# (including discriminated unions, records but also class declarations) simply aren't allowed to be `null`. They always have to be initialized to a valid value. As we've seen in chapter 5, the right way to represent missing value in F# is to use the `option` type.

UNCHECKED 'NULL' IN F#

Unfortunately, other languages such as C# don't understand the restriction to disallow `null` as a value for an F# type. This means that an F# function such as `getLongest` still can receive `null` as an argument if it's called from C#. We can handle this case by using the generic `Unchecked.defaultof<T>` value, which gives us an *unsafe* way to create a `null` value of any reference type in F# or to get the default value of a value type. (In other words, it's the equivalent of `default(T)` in C#.)

We only intend to use our simple function from F#, so we don't have to handle the case when a C# user calls it with a null argument. Listing 11.8 shows several other tests that we can add. Note that a large part of the listing is just a slightly modified version of the code that we wrote in listing 11.7 when testing the function interactively. The most notable differences are that we've wrapped the testing code inside functions and added an attribute that marks it as an xUnit.net test.

Listing 11.8 Function with unit tests to verify its behavior (F#)

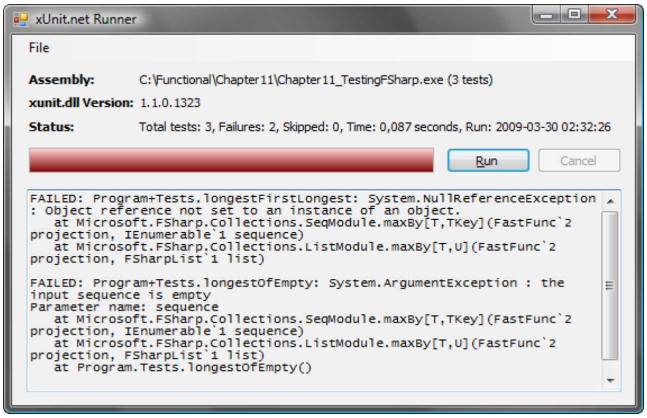
```
#if INTERACTIVE
#r @"C:\Programs\Development\xUnit\xunit.dll"
#endif
open Xunit

let getLongest(names:list<string>) =
   names |> List.max_by (fun name -> name.Length)

module LongestTests =
   [<Fact>]                                                #1
   let longestOfNonEmpty() =
      let test = [ "Aaa"; "Bbbbb"; "Cccc" ]                #A
      Assert.Equal("Bbbbb", getLongest(test))              #A

   [<Fact>]
   let longestFirstLongest() =
      let test = [ "Aaa"; "Bbb" ]                          #2
      Assert.Equal("Aaa", getLongest(test))                #2

   [<Fact>]
   let longestOfEmpty() =
      let test = []                                        #3
      Assert.Equal("", getLongest(test))                   #3
```

#1 Mark tests using 'Fact' attribute
#A Adjusted interactive test
#2 Should return empty string for an empty list
#3 Should return first of the longest elements

In addition to wrapping every test into a function, we've also created a module to keep all the unit tests together in a single class. This isn't technically necessary, but it's a good idea to keep the tests separated from the main part of the program. Depending on your preferences, you can of course move the tests to the end of the file, to a separate file in a single project, or even to a separate project.

The xUnit.net framework uses an attribute called `Fact` to mark methods that represent unit tests (#1). We can apply this to F# functions declared with let bindings, as they're compiled as methods. The first test in the module is just an adjusted version of the code we wrote when testing the code interactively, but we've also added two new tests.

The second test (#2) verifies that the `getLongest` function returns the first of the elements that have the maximal length when there are several of them. The `max_by` function from the F# libraries follows this rule, but it isn't documented so it may depend on the implementation; testing it explicitly is a good idea. The last test (#3) checks that the function returns an empty string when we pass it an empty list. This is one of the corner cases that are worth considering. Returning an empty string may be the desired behavior when you're displaying the result in a user interface, for example. As you've probably guessed, our original implementation doesn't follow this rule. If you run the xUnit.net GUI on the compiled assembly, you'll get a result similar to the one in figure 11.1.



Assembly: C:\Functional\Chapter11\Chapter11_TestingFSharp.exe (3 tests)
xunit.dll Version: 1.1.0.1323
Status: Total tests: 3, Failures: 2, Skipped: 0, Time: 0,087 seconds, Run: 2009-03-30 02:32:26

```
FAILED: Program+Tests.longestFirstLongest: System.NullReferenceException
: Object reference not set to an instance of an object.
   at Microsoft.FSharp.Collections.SeqModule.maxBy[T,TKey](FastFunc`2
projection, IEnumerable`1 sequence)
   at Microsoft.FSharp.Collections.ListModule.maxBy[T,U](FastFunc`2
projection, FSharpList`1 list)

FAILED: Program+Tests.longestOfEmpty: System.ArgumentException : the
input sequence is empty
Parameter name: sequence
   at Microsoft.FSharp.Collections.SeqModule.maxBy[T,TKey](FastFunc`2
projection, IEnumerable`1 sequence)
   at Microsoft.FSharp.Collections.ListModule.maxBy[T,U](FastFunc`2
projection, FSharpList`1 list)
   at Program.Tests.longestOfEmpty()
```

Figure 11.1 Instead of returning an empty string, the tested function throws an exception when given an empty list as its argument.

Now that we've clarified the required behavior of the `getLongest` function, we can fix it very easily by adding a pattern to match the empty list:

```
let getLongest(names:list<string>) =
   match names with
   | [] -> ""
   | _ -> names |> List.maxBy (fun name -> name.Length)
```

All three unit tests pass after this change. So far, the tests have been quite simple and we've only had to check whether the returned string matched the expected one. Often unit tests are more involved than this. Let's look at how we might test a more complicated function and in particular how to compare an actual value with an expected one when a function returns a list.

11.2.2 Writing tests using structural equality

Testing for equality with complicated data structures can be tricky in C#. If we simply construct a new object with the same properties and compare the two using the "==" operator, the result is likely be `false`, because we'd be comparing two distinct *instances*.

The "==" operator can be overloaded in C#, and `Object.Equals` can be overridden, but both of these should usually only be done for immutable types. When you compare two different instances of mutable types, it is important to distinguish between them, because the data can change later on. On the other hand, if we have two immutable types storing the same values, we can treat them as equal. The data can't change in the future, so the two objects will always be equal.

STRUCTURAL EQUALITY AND COMPARISON

As most of the types that we can declare in F# are immutable, the F# compiler automatically implements the `IComparable<T>` interface and overrides the `Equals` method if we don't provide an explicit implementation. It does this using a comparison for *structural* *equality*. This isn't done automatically for F# classes; just for simple functional types like records, discriminated unions and also tuples, which don't have to be declared explicitly.

Values of types that use this comparison are considered equal if they are either equal primitive types such as integers or strings or if they are composed from the same values, using structural equality recursively. Listing 11.9 demonstrates structural equality with records containing tuples and primitive values.

Listing 11.9 Comparing records with structural equality (F# interactive)

```
> type WeatherItem =
     { Temperature : int * int;
       Text : string }
  let winter1 = { Temperature = -10, -2; Text = "Winter" }          #A
  let winter2 = { Temperature = -10, -2; Text = "Winter" };;        #A
(...)

> System.Object.ReferenceEquals(winter1, winter2);;                 #1
val it : bool = false

> winter1 = winter2;;                                               #2
val it : bool = true
```

#A Create records containing the same values
#1 Values are represented by different instances
#2 …but are considered as equal

First we declare an F# record type, which contains two fields. The first field type is a tuple of two integers, and the second is a string. We create two values of the record type using exactly the same value for each corresponding field.

We can see that we genuinely have two instances - a test for reference equality (#1) returns false. However, if we use the standard F# operator for testing equality (#2), the runtime will use structural equality and it will report that the values are equal. First the two tuple values are compared for structural equality, and then the two strings are compared.

As I said earlier, this technique works for records, tuples and discriminated unions. Since immutable F# lists are declared as discriminated unions, they receive the same treatment. In a moment we'll use this feature when writing a unit test expectation, but first let's look at one more feature of the automatically generated comparisons. We've seen how to use structural equality to test whether values are equal, but F# also provides *structural comparisons for ordering*:

```
> let summer = { Temperature = 10, 20; Text = "Summer" };;
(...)

> summer = winter1;;
val it : bool = false

> summer > winter1;;
val it : bool = true
```

This snippet creates a new value of the record type declared in listing 11.9 and compares it with the value from the previous listing. The first result isn't surprising: the two values are different. However the second one deserves an explanation. Why should the `summer` value be considered to be *larger than* the `winter1` value? The reason is that the F# compiler also generates a default comparison for the `WeatherItem` type. The comparison uses the values of the fields in the order in which they are declared: a tuple value `(10, 0)` is larger than a tuple `(9, 100), for example`. This default behavior can be useful, particularly if you take it into consideration when you design your type, but for the rest of this chapter we'll be focusing on structural equality.

WRITING TESTS FOR LISTS

The function we're going to test is a generalized version of the one that printed names consisting of multiple words. The difference is that instead of printing the names, the function will return them as a result. In fact, the result will be a tuple of two lists: one containing the multi-word names, and one containing the single-word names. In functional terminology this operation is called *partitioning* and we can easily implement our function using the `List.partition` function from the standard F# library:

```
> let partitionMultiWord(names:list<string>) =
     names |> List.partition (fun name -> name.Contains(" "));;
val partitionMultiWord : string list -> string list * string list
```

The `partition` function takes a predicate as an argument and divides the input list into two lists. The first list contains elements for which the predicate returned true and the second contains the remaining elements. Listing 11.10 shows two unit tests for the function declared above.

Listing 11.10 Unit tests for the partitioning operation (F#)

```
module PartitionTests =
   [<Fact>]
   let partitionKeepLength() =
      let test = ["A"; "A B"; "A B C"; "B" ]
      let multi, single = partitionMultiWord(test)
      Assert.True(multi.Length + single.Length = test.Length)    #1

   [<Fact>]
   let partitionNonEmpty() =
      let test = ["Seattle"; "New York"; "Reading"]
      let expected = ["New York"], ["Seattle"; "Reading"]
      Assert.Equal(expected, partitionMultiWord(test))           #2
```

#1 Verify length of the returned lists
#2 Test the result using structural equality

The listing shows two unit tests implemented as functions marked with the `Fact` attribute. The first test (#1) checks that the lengths of the two lists returned as results add up to the same number as the length of the original input. This is a very simple way to partially verify that no elements are lost by the partitioning. We're using only a single input (the value `test`) in the listing, but we could simply extend the test to use multiple input lists.

The second test is more interesting, because it uses the structural equality feature we discussed earlier. It declares a value with the test input and a value that represents the expected output of the tests. The `expected` value is a tuple of two lists. The first list contains a single element, which is the only name composed from multiple words. The second list contains single-word names in the same order in which they occur in the input list. If you run the test, the assertion (#2) succeeds, because the runtime uses structural equality to compare the tuples and also the lists contained in the tuple. This means that it compares all the individual strings in the lists.

If we wrote similar code using for example the standard `List<T>` type, the test would fail, because the test would compare reference equality. As I mentioned earlier, using reference equality is important when working with mutable types, because the contained values can change, but immutable types like functional lists of immutable values will never change so we can treat them as equal.

As we've seen, structural equality is a simple but useful feature which streamlines unit testing, even though it's not a fundamental aspect of functional programming. A more important and more inherently functional technique which aids testing is function composition.

11.2.3 Testing composed functionality

In the section about tracking dependencies in code, we used C# methods similar to the two F# functions from the last two examples to demonstrate how functional programming makes it easier to recognize what a function does and what data it accesses. This is useful when writing the code, but it's also extremely valuable when testing it.

In section 11.1, we wrote an imperative method for printing names consisting of multiple words, but with the side-effect of removing elements from the mutable list passed to it as argument. This didn't cause any problems as long as we were not using the same list later. Any unit tests for that method which just checked the printed output would have succeeded.

What made the method tricky was that if we used it in conjunction with another method that was also correct, we could get unexpected results. This makes it hard to test imperative code thoroughly. In principle we should test that every method does exactly what it is supposed to do, and nothing more. Unfortunately, the "and nothing more" part it is really hard to test, because any piece of code can access and modify any part of the shared state.

In functional programming we- shouldn't modify any shared state, so we only need to verify that the function returns correct results for all the given inputs. This also means that when we're using two tested functions together, we only have to test that the combination has the appropriate effect: we don't need to verify that the functions don't tread on each other's data in subtle ways. Listing 11.11 shows the kind of test which looks completely pointless... but imagine what it might show if we were working with `List<T>` instead of immutable F# lists.

Listing 11.11 Testing calls to two side-effect free functions (F#)

```
[<Fact>]
let partitionThanLongest() =
   let test = ["Seattle"; "New York"; "Grantchester"]
   let expected = ["New York"], ["Seattle"; "Grantchester"]
   let actualPartition = partitionMultiWord(test)             #1
   let actualLongest = getLongest(test)                       #1
   Assert.Equal(expected, actualPartition)                    #2
   Assert.Equal("Grantchester", actualLongest)                #2
```

#1 Run the functions in sequence
#2 Verify the results

As we can see, the unit test runs the two functions in sequence (#1), but only uses the results in the section where we verify whether the results match our expectations (#2). This means that the two function calls are independent and if they don't contain any side-effects we can reorder the calls freely. In a functional world, this unit test isn't needed at all: we've already written unit tests for the individual functions and this test doesn't verify any additional behavior.

However, if we'd written similar code using the mutable `List<T>` type, this test could catch the error we found in section 11.1. If the `partitionMultiWord` function modified the list referenced by the value `test`, removing all single-word names, the result of the second call wouldn't be "Grantchester" as expected by the test. This is a very important general observation about functional code - if we test all the primitive pieces properly and test the code that composes them, we don't need to test whether the primitive pieces will still behave correctly in the new configuration.

So far we've talked about refactoring and testing functional programs. We've seen that first-class functions allow us to reduce code duplication and immutable data structures help us to understand what the code does as well as reducing the need to test how two pieces of code might interfere with each other.

The remainder of this chapter discusses when (and if) code is executed, and how we can take advantage of this to make our code more efficient. First we need to get a clear idea of when there are opportunities available, and how F# and C# decide when to execute code.

11.3 Evaluation order

We've looked at how to track dependencies between functions in code that uses immutable data structures. Once we know what the dependencies are, we can sometimes reorder operations to make the program more efficient. Listing 11.11 shows a simple example of this kind of optimization.

Listing 11.11 Reordering calculations in a program (C#)

```
var a = Calculate1(10);          #1      var b = TestCondition();
var b = TestCondition();                 if (b == true) {
if (b == true)                              var a = Calculate1(10);  #2
   return Calculate2(a);                    return Calculate2(a);
else return 0;                           } else return 0;
```

#1 If 'b' evaluates to true, we won't need this!
#2 Run 'Calculate1' only if we really have to

In the first version, we call the `Calculate1` function at the beginning of the program (#1). However, the result of this call is used only if `TestCondition` returns `true`. In the second version, we moved this computation inside the `if` condition, so it will be calculated only if the result will be really needed.

This was a very simple modification and you'd probably have written the more efficient version without even thinking about it. As a program grows larger, however, optimizations like this become more difficult to spot. Listing 11.12 shows a slightly trickier example.

Listing 11.12 Passing a computed result to a function (C#)

```
int TestAndCalculate(int arg) {
   var b = TestCondition();            #1
   if (b == true)
      return Calculate2(arg);          #2
   else return 0;
}
// Used later in the program
TestAndCalculate(Calculate1(10));      #3
```

#1 Evaluate test condition
#2 The argument 'a' is needed only when 'TestCondition' returns false!
#3 Calling 'Calculate' may be unnecessary here

The function in this example takes a value `arg` as an argument–but this value may not be needed by the function at all. If the condition (#1) evaluates to false, then the function returns `0` and the value of `arg` is not relevant. When calling this function (#3), the function `Calculate1` is executed even if we later find out that we don't need its result.

In Haskell (another popular functional language) this code wouldn't call `Calculate` if it didn't need the result, because Haskell uses a different *evaluation strategy*. Let's take a quick detour by looking at a few options before we return to optimizing listing 11.12.

11.3.1 Different evaluation strategies

Haskell is a purely functional language. One of its interesting aspect is that it doesn't allow any side-effects. There are techniques for printing to a screen or working with file systems, but they are implemented in a way that they don't actually look like side-effects to the programmer. In a language like that, it possible to reorder expressions when evaluating them, so Haskell doesn't evaluate a function unless it really needs the result. This doesn't affect the program’s result because the function can't have side-effects.

On the other hand, both C# and F# functions can have side effects. They are discouraged in F# and the language supports many ways to avoid them, but they can still be present in the program. Both languages specify the order in which the expressions will run, as otherwise we couldn’t tell which side-effect would occur first, making reliable programming impossible!

EAGER EVALUATION IN C# AND F#

In almost all mainstream languages the rule that specifies evaluation order is quite simple: to make a function call, the program first evaluates all the arguments and then executes the function. Let me demonstrate this using our previous example:

```
TestAndCalculate(Calculate(10));
```

In all mainstream languages, the program will execute `Calculate(10)` and then pass the result as the argument to `TestAndCalculate`. As we've seen in the previous example, this is unfortunate if the function `TestAndCalculate` doesn't really need the value of the argument. In that case we just wasted some CPU cycles for no good reason! This is called an *eager evaluation strategy*.

The benefit of eager evaluation is that it is very easy to understand how the program executes. In C# and F# this is clearly very important, because we need to know the order in which side-effects (such as I/O and user interface manipulation) will run. On the other side, in Haskell this is controlled by arguments and the return values of functions, so we don't need to know that much about the order.

LAZY EVALUATION STRATEGY IN HASKELL

In a lazy evaluation strategy, arguments to a function are not evaluated when the function is called, but later when the value is actually needed. Let's again take a look at the previous example:

```
TestAndCalculate(Calculate(10));
```

In this example, Haskell jumps directly into the body of `TestAndCalculate`. The name of the argument is `arg`, so Haskell remembers that if it needs the value of `arg` later, it should run `Calculate(10)` to get it. Then it continues to execute by getting the result of `TestCondition`. If this function returns `true`, it needs the value of `arg` and executes `Calculate(10).` If `TestCondition` returns `false`, the `Calculate` function is never called.

11.3.2 Evaluation strategies side-by-side

We can demonstrate different evaluation strategies using the computation by calculation technique described in chapter 2. This shows how the program runs step by step, so you can clearly see the difference between lazy and eager evaluation. Listing 11.13 shows evaluation of an expression that uses two functions: `PlusTen(a)` returns `a + 10` and `TimesTwo(a)` returns `a * 2`.

Listing 11.13 Lazy evaluation (left) and eager evaluation (right)

```
[CA] Start with a nested call:              [CA] Start with a nested call:
  PlusTen(TimesTwo(4))                        PlusTen(TimesTwo(4))

[CA] Start calculating PlusTen:             [CA] Calculate result of TimesTwo:
  TimesTwo(4) + 10                 #1         PlusTen(8)                      #3

[CA] Calculate TimesTwo if needed:          [CA] Calculate PlusTen next:
  8 + 10                           #2         8 + 10                          #4
```

```
[CA] Calculate the result:              [CA] Calculate the result:
  18                                      18
```

#1, #2 The lazy evaluation strategy starts by evaluating PlusTen and doesn't evaluate the argument first. In the next step it will need to add 10 to the argument (#1), but the argument hasn't been evaluated yet. Since the value of the argument really is needed to make further progress, the call to 'TimesTwo' is executed (#2) and we get the final result.
#3, #4 The eager evaluation strategy starts by evaluating the argument, so in the first step it evaluates 'TimesTwo(4)' to obtain the value 8 (#3). All arguments to the function 'PlusTen' have now been evaluated, so it can continue by evaluating this function (#4) and calculating the result.

Annotations below the code with bullets on the left

So far we have only looked at one motivation for using a lazy evaluation strategy, but it seems useful already. You may be wondering why I've brought it up at all if it only exists in Haskell; fortunately, similar effects can be achieved in F# and C# 3.0.

11.4 Evaluating values lazily

The evaluation order in F# is eager: expressions used as arguments to a function are evaluated before the function itself starts to execute. In both C# and F#, we can simulate lazy evaluation using function values and F# even supports lazy evaluation via a special keyword. But first, there is actually one exception from the eager evaluation rule. You definitely know about and use it frequently, but it's so common that you may not realize that it's actually doing something special.

11.4.1 When are arguments evaluated lazily?

When you write a condition that uses logical "or" (||) or logical "and" (&&) operator we sometimes don't need the expression on the right-hand side during the evaluation. If the left-hand side expression evaluates to `false` then the "and" operator ignores the right-hand side all together and just returns `false`, because it already knows that this is the overall result. Similar logic holds for the "or" operator, but if the left-hand side expression evaluates to `true`. The listing 11.14 demonstrates the difference between built-in lazy operators and a custom method that we can write.

Listing 11.14 Comparing built-in or operator with a custom 'Or' method (C#)

```
bool Foo(int n) {
   Console.WriteLine("Foo({0})", n);                                  #1
   return n <= 10;
}
bool Or(bool first, bool second) {
   if (first) return true;                                            #2
   else if (second) return true;
   else return false;
}

// If written using "||" operator and "Or" method:
if (Foo(5) || Foo(7))                                            #3
   Console.WriteLine("True");

if (Or(Foo(5), Foo(7)))                                          #4
   Console.WriteLine("True");
```

#1 Prints information about the call as a side-effect
#2 If 'a' is 'true' then the value of 'b' isn't needed!
#3 Prints 'Foo(5)' and 'True' only
#4 Prints 'Foo(5)', 'Foo(7)' and 'True'

We're demonstrating the problem using a Foo method that writes to the screen (#1) so we can track how it's being called. First, let's take a look at the built-in "or" operator (#3). You can see that both expressions would evaluate to true, but if you run the code, only Foo(5) is printed. If we write the same thing using our Or method (#4) then both of the arguments are evaluated first and so the output contains both Foo(5) and Foo(7).

The Or method is written in a way that makes it obvious that the value of second isn't needed if first is true (#2), because it is used only inside else branch. So, how can we change the code so that the expression used as an argument will not be evaluated unless the result is really needed?

The first option is to use function values. Instead of having a method with a parameter of type bool, we'll make it take Func<bool>. When we need the value later in the code, we can just execute the function, which will in turn evaluate the argument. You can see how to write the "or" operator (now called LazyOr) using this trick in listing 11.15.

Listing 11.15 Lazy "or" operator using functions (C#)

```
   bool LazyOr(Func<bool> first, Func<bool> second) {      #A
      if (first()) return true;                            #B
      else if (second()) return true;                      #C
      else return false;
   }

   if (LazyOr(() => Foo(5), () => Foo(7)))                 #D
       Console.WriteLine("True");
```

#A Takes functions instead of values
#B Force evaluation of the first argument
#C Force evaluation of the second argument
#D Prints 'Foo(5)' and 'True' only

The arguments to the LazyOr method are now wrapped inside lambda functions. When the method is called, its arguments are eagerly evaluated, but the value of the argument is just a function. The expression *inside* the lambda function isn't evaluated until the function is called.

Let's now take a look at the LazyOr method. Its structure stays the same, but in the places where we originally accessed the value of an argument, we now call the function provided by that argument. This will, in turn, evaluate the expression used as an argument and in our example call the Foo method. If the function first returns true, then the LazyOr method immediately returns true and the second function never gets called. This means that the code behaves just like the built-in logical "or" operator.

However, suppose we needed to access the argument value more than once. Should we invoke the function multiple times? That doesn't sound like a very efficient solution, so we'd probably want to store the result locally. In F#, this is made simpler using a feature called *lazy values*. First we'll look at some F# code and then we'll implement the same behavior in C#. After that, we'll look at a sample application which may give you some ideas for places to use this technique in your own code.

11.4.2 Lazy values in F#

A lazy value in F# is a way to represent delayed computation. This means a computation that is evaluated only when the value is actually needed. In the previous section, we implemented similar thing using functions in C#, but lazy values automatically calculate the value only once and then remember the result.

The best way to explore this feature is to play with it inside F# interactive. You can see a script that demonstrates how to use it in listing 11.16.

Listing 11.16 (F# interactive)

```
> let foo(n) =                            #1
      printfn "foo(%d)" n
      n <= 10;;
val foo : int -> bool

> let n = lazy foo(10);;                  #2
val n : Lazy<bool>                        #2

> n.Force();;
foo(10)                                   #3
val it : bool = false                     #3

> n.Force();;
val it : bool = false                     #4
```

#1 Equivalent of the 'Foo' method
#2 Create delayed computation using 'lazy' keyword
#3 'foo' gets called and the result is returned
#4 Result is returned immediately!

We start by writing a function (#1) similar to our C# `Foo` method. This lets us track when the computation is actually evaluated by writing to the console. The second command uses the F# `lazy` keyword (#2). If you mark an expression with `lazy`, the expression will not be evaluated immediately and will be wrapped in a lazy value. As you can see from the output, the `foo` function hasn't been yet called and the created value has type `Lazy<bool>`. This represents a lazy value that can evaluate to a Boolean value.

On the next line, you can see that lazy values have a member called `Force` (#3). This member evaluates the delayed computation. In our case, this means calling the `foo` function. The last command shows that calling `Force` again (#4) doesn't re-evaluate the computation. If you look at the output, you can see that the `foo` function wasn't called.

Working with lazy values using the `lazy` keyword and the `Force` member isn't as explicit as using functions, so the following sidebar presents a slightly different look that should give you more insight.

Operations for working with lazy values

When we looked at values and types in chapter 5 and 6, we saw that the easiest way to understand a type is often to look at what we can do with it: which functions operate on it. Even though these are not available for lazy values in the core F# library, we can easily write them and then look at the type signature. Let's start with a function that creates lazy value:

```
> let create f = lazy f();;
val create : (unit -> 'a) -> Lazy<'a>
```

Our `create` function takes a function as an argument. This is the only way to delay a computation without using the `lazy` keyword. The function used as an argument doesn't have any arguments and when it's called it just returns a value of a type `'a`. The code marked using `lazy` contains a call to this function. However, this is executed lazily when the value is needed, so the `f` function will not be immediately called here. It will just be wrapped in a lazy value, which has a type `Lazy<'a>`.

Now, let's take a look at the second important operation for working with lazy values:

```
> let force(v:Lazy<_>) = v.Force();;
val force : Lazy<'a> -> 'a
```

This function simply invokes the `Force` member of the lazy value. The function signature shows us that it takes a lazy value and returns the actual value. Lazy values contain a mutable state, which is updated when the value is evaluated for the first time. This allows the lazy value to cache the result.

Our motivation when we started talking about lazy values was that we couldn't write our own implementation of the logical "or" operator that would only evaluate the argument on the right-hand side if and when it needed to. Let’s try again now, armed with our new knowledge of lazy values.

Implementing or and lazy or

Since we're implementing an operator, we're going to define it as a true operator rather than just as a normal function. As we've seen in chapter 6, we can introduce our own operators in F#, so listing 11.17 shows two different variations of the "or" operator.

Listing 11.17 Comparing eager and lazy "or" operator (F# interactive)

```
> let (||!) a b =                       > let (||?) (a:Lazy<_>) (b:Lazy<_>) =
    if a then true          #1              if a.Force() then true         #2
    elif b then true                        elif b.Force() then true
    else false                              else false
  ;;                                      ;;
```

```
val ( ||! ) :                        val ( ||? ) :
  bool -> bool -> bool                 Lazy<bool> -> Lazy<bool> -> bool

> if (foo(5) ||! foo(7))   #3        > if lazy foo(5) ||? lazy foo(7)   #4
    then printfn "True";;                then printfn "True";;
foo(5)                               foo(5)
foo(7)                               True
True
```

#1, #2 Arguments of the eager version of the operator (#1) are Boolean values, so we can use them directly in the if-condition. The lazy version (#2) takes lazy values wrapping a computation that returns a Boolean value. To read the value, we use the 'Force' member.
#3, #4 When using the eager operator (#3), we specify the arguments as normal. As the output shows, both of the arguments are evaluated. When using the lazy version (#4) we add additional 'lazy' keywords to delay both arguments. The result is that only one expression is evaluated.

Annotations below the code with bullets on the left

In many ways this example was only a curiosity, although it shows an important fact about F# - it is surprisingly flexible and extensible. By combining the `lazy` keyword with a custom operator we could write a construct that can't be written in most of the common languages. Next we'll implement lazy values as a type we can use from C#. It's not quite as syntactically compact, but even in this form it can be very useful.

11.4.3 Implementing lazy values for C#

In section 11.4.1 we represented delayed computation in C# using functions. The `Lazy<'a>` type in F# adds the ability to cache the value when its value is calculated, and we can achieve the same effect in C# by wrapping the function inside a class. Listing 11.18 shows a simple implementation of `Lazy<T>`.

Listing 11.18 Class for representing lazy values (C#)

```
public class Lazy<T> {
   Func<T> func;
   Option<T> value = Option.None<T>();                  #A

   public Lazy(Func<T> func) {                          #1
      this.func = func;
   }
   public T Force() {                                   #2
      T result;
      if (!value.MatchSome(out result)) {
         result = func();                               #B
         value = Option.Some(result);                   #B
      }
      return result;
   }
}
public class Lazy {                                     #C
    public static Lazy<T> Create<T>(Func<T> func) {
       return new Lazy<T>(func);
    }
}
```

#A Cache that can contain the evaluated value
#1 Creates lazy value from a function
#2 Evaluate the lazy value
#B Compute value and modify the cache
#C Helper class that enables type inference when creating values

The first important part of the class is a constructor (#1) that takes a function and wraps it. The function doesn't take any arguments, but evaluates the value when it's called, so we're using the `Func<T>` delegate. There's also a static method in a non-generic type to make it easier to use C#'s type inference when we create lazy values.

The lazy value uses the functional `Option<T>` type from chapter 5. This is an elegant way to express the fact that initially we don't have the computed value, but later we do. Note that we're using generics, so we can't easily represent this using the `null` value, and even if we added a restriction to force `T` to be a reference type, we need to allow for the possibility that the function could return `null` as the computed value.

Most of the code which uses the cached value is in the `Force` method. From the user's perspective this is the second important part of the class. First it tests whether we've already evaluated the function. If we have, we can just use the value we computed earlier. If not, it calls the function and stores the result using `Option.Some`.

Let's take a look at a simple code snippet shows how we can work with this type:

```
var lazy = Lazy.Create(() => Foo(10));
Console.WriteLine(lazy.Force());            #A
Console.WriteLine(lazy.Force());            #B
```

#A Prints "Foo(10)" and "True"
#B Prints only "True"

If you try this code, you should see exactly the same behavior as in the F# version. When creating the lazy value, we give it a function: the `Foo` method will not be called at this point. The first call to `Force` evaluates the function and calls `Foo`; any subsequent call to `Force` uses the cached value computed earlier, so the last line just prints the result.

Lazy values are most useful when we have a set of computations that can take a long time and we need to calculate the value (or values) on demand. We'll conclude this chapter by looking at an application which uses this programming pattern.

11.5 Caching results using lazy values

We're going to write an application that finds all the photos in a specified folder and displays a list of them. When the user selects a photo, the application resizes it and shows it in a window. (For simplicity, we're not going to allow the user to resize the window.) When we draw the photo, we'll need to resize it to fit the screen and then show the resized image.

Obviously, we don't want to resize all photos when the application starts: it could take an enormous amount of time for a large set of photos. On the other hand, we don't want to resize the photo every time we draw it because we'd have to resize the same photo again and again. From the description it's fairly obvious that lazy values can help us . We'll start by writing the F# version of the application and then look at how we can use the same idea in C#.

11.5.1 Browsing photos in F#

The most interesting part of the application is the code that is executed when the application starts. It finds all the files in the specified directory and creates an array with information about each file. This information contains the name of the file and the lazy value that will evaluate to the resized preview. Listing 11.19 shows how we can create this data structure.

Listing 11.19 Creating collection of photo information (F#)

```
#light
open System.IO
open System.Drawing

type ImageInfo = { Name : string; Preview : Lazy<Bitmap> }              #1

let dir = @"C:\My Photos"                                               #A
let files =                                                             #2
  Directory.GetFiles(dir, "*.jpg") |> Array.map (fun file ->
    let lazyPrev =                                                      #3
        lazy(let bmp = Bitmap.FromFile(file)
             let resized = new Bitmap(400, 300)
             use gr = Graphics.FromImage(resized)                       #B
             let dst = Rectangle(0, 0, 400, 300)                        #C
             let src = Rectangle(0, 0, bmp.Width, bmp.Height)           #C
             gr.InterpolationMode <- Drawing2D.InterpolationMode.High   #C
             gr.DrawImage(bmp, dst, src, GraphicsUnit.Pixel)            #C
             resized)
    { Name = Path.GetFileName(file)                                     #4
      Preview = lazyPrev })                                             #4
```

#1 Stores photo name and lazily created preview
#A Specify directory with your photos
#2 Array of ImageInfo value for each photo
#3 Preview has type Lazy<Bitmap>
#B 'use' will dispose the object automatically
#C Draw resized bitmap to the target
#4 Return record with name and preview

We start by declaring a record type (#1) that represents information about the photo. As you can see, the type of the preview is `Lazy<Bitmap>`, which is a delayed computation that will give us a `Bitmap` when we'll need it. Next, we create the data structure that contains information about photos (#2). We obtain an array of files using a normal .NET method call and use the `Array.map` function to create an `ImageInfo` value for every photo.

Inside the lambda function, we first create the lazy value to represent the preview (#3) and then return a record value containing the name and the preview (#4). To draw the preview lazily we can simply wrap the whole code using the `lazy` keyword. One interesting property of the program is that we could delete all uses of the `lazy` keyword, change all types from Lazy<A> to A and delete all uses of the Force member, and the code would still work correctly, except everything will be evaluated eagerly.

Now that we have all the data we need about the photos, we can add a simple GUI using Windows Forms. In the listing 11.20, we create a couple of controls to show the data and code that shows the selected photo.

Listing 11.20 Adding user interface for the photo browser (F#)

```
open System
open System.Windows.Forms

let main = new Form(Text="Photos", ClientSize=Size(600,300))
let pict = new PictureBox(Dock=DockStyle.Fill)
let list = new ListBox(Dock=DockStyle.Left, Width=200,
                       DataSource=files, DisplayMember = "Name")    #1
list.SelectedIndexChanged.Add(fun _ ->                              #A
   let info = files.[list.SelectedIndex]
   pict.Image <- info.Preview.Force())                              #2
main.Controls.Add(pict)
main.Controls.Add(list)

[<STAThread>]
do Application.Run(main)                                            #3
```

#1 Configure to display 'Name' property from 'files' array
#A Called when the selection changes
#2 Evaluate the lazy value
#3 Runs the application

To show the list of photos in the ListBox control, we use data binding (#1), which is a feature used in many .NET controls. We simply specify that the DataSource for the control is our array of files. To specify what should be displayed, we set the DataMember property to the name of the record member that we want to display ("Name").

Next, we register a lambda function as a handler for the SelectedIndexChanged event of the ListBox. When this is triggered, we choose the selected ImageInfo value and use the Force member to get the resized bitmap. If this is the first time that particular bitmap has been shown it will be resized at that point; if we've seen it before we can immediately use the cached result. The listing above shows the code as a stand-alone application, which means that we run it using the Application.Run method (#3). In F# interactive, you would use main.Show to display the form instead. You can see how the application looks in the figure 11.2.

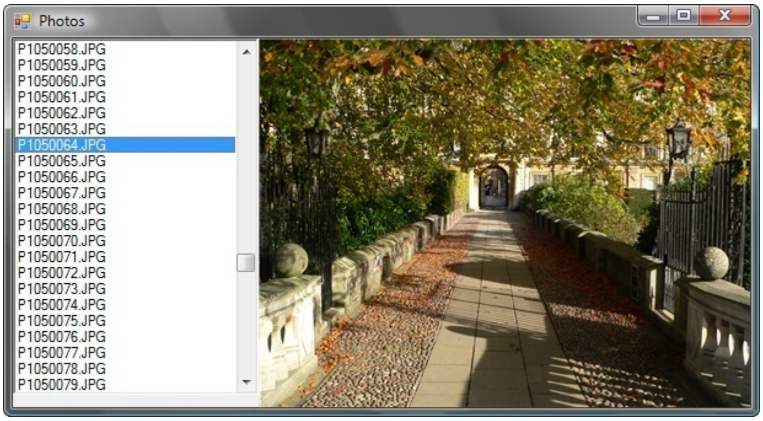


Figure 11.2 Photos can be selected from the list on the left side. The resized version is cached automatically thanks to the use of lazy values.

If you run the application using a folder containing large photos, it's obvious the difference made by lazy values. Selecting a "new" photo can take some time, but if you revisit a photo you've already seen, it will be rendered immediately.

In the next section, we'll show how to implement the most important parts of the application in C# using our `Lazy<T>` class. One of the interesting things that we'll see is that you can use C# 3.0 anonymous types for representing the information about photo.

11.5.2 Browsing photos in C#

First we need to create the user interface in the Windows Forms designer. We'll add the same controls as in the previous F# version: a `ListBox` called `list` and a `PictureBox` called `pict`.

The remaining source code for the application is shown in listing 11.21. The structure of the code is exactly the same as in the F# version. We add all the code to an event handler for the form's `Load` event. We use an anonymous type for the image information, which works because all our code is in a single method.

Listing 11.21 Implementing photo browser using Lazy<T> (C#)

```
private void MainForm_Load(object sender1, EventArgs e1) {
   var dir = @"C:\My Photos";                                   #A

   // Load photos into a collection of lazy values
   var locations = Directory.GetFiles(dir, "*.jpg");
   var files = locations.Select(file => {                       #1
      var resizedLazy = Lazy.Create(() => {                     #2
         var bmp = Bitmap.FromFile(file);
         // Omitted: drawing the bitmap                         #B
```

```
            return resized;
         });
         return new {                                                #3
            Name = Path.GetFileName(file),                           #C
            Preview = resizedLazy };                                 #C
      });

      // Display the list of photos and register event handler
      var filesArray = files.ToArray();
      list.SelectedIndexChanged += (sender2, e2) => {
         pict.Image = filesArray[list.SelectedIndex].Preview.Force();   #D
      };
      list.DataSource = filesArray;                                  #E
      list.DisplayMember = "Name";                                   #E
   }
```

#A Specify directory with your photos here!
#1 Use 'Select' to return information for each photo
#2 Create lazy value from lambda function
#B Drawing code is the same as in F#
#3 Anonymous type
#C Return name and lazy value with the resized image
#D Force evaluation and show the bitmap
#E Setup the data binding

This example uses a large number of functional ideas. The first is that we use the `Select` method to transform an array of file names into an array of anonymous types storing information about the photos (#1). Inside the `Select` method, we construct the lazy value (#2) from a lambda function. The drawing code is exactly the same as in the F# version, so I haven't repeated it in the listing.

After creating the lazy value, we return the photo information for the specified file. We use an anonymous type with two properties - `Name` is a `string` and `Preview` is a `Lazy<Bitmap>`.

Thanks to type inference and the `var` keyword, we can work with these values anywhere inside the `MainForm_Load` method. When we use the values later in the event handler that is triggered when selection changes, we can refer to the `Preview` property and call the `Force` method of our lazy value to get the bitmap.

The use of anonymous types in this example follows the programming style where we start with the simplest possible solution and later turn it into a more sophisticated version if we need to. Writing the code with anonymous types is very simple and it works well in this example. If we wanted to use the data structure elsewhere in the application, we'd have to declare a class similar to the `ImageInfo` record we used in F#. However, this would only require minimal changes to the existing code.

WHY NOT USE MULTIPLE THREADS?

You may be wondering if we could improve this application using multiple threads. There are actually two areas where using multiple threads could help here. First of all, we could start the computation when user selects a file without blocking the user interface. Currently, the application is frozen until the image is resized. To this end, we could use asynchronous programming techniques such as the F# asynchronous workflows discussed in chapter 13.

Another possibility is that the application could pre-compute the resized bitmaps in the background. Instead of doing nothing, it could resize some images in advance so that the user wouldn't have to wait when clicking on the photo. In chapter 14, we'll see that this is quite easy - we'll look at the `Future<T>` type, which is like `Lazy<T>`, except that it isn't as lazy and computes the value on a background thread.

So far, we've been using lazy values in a very straightforward way. In the next section, we're going to briefly introduce a slightly different use for them - implementing infinite data structures. However, we will not go into great detail, because F# provides a more convenient way of solving the same problem, which we'll talk about it in the next chapter. The following section will serve as an introduction to the functional way of looking at the problem.

11.6 Introducing infinite data structures

The title of this section may sound a little odd (or insane), so let me give a word of explanation. One of the data structures that we've used quite a lot is the functional list. However, we might also want to represent logically infinite lists, such as a list of all prime numbers. Of course, in reality we wouldn't *use* all the numbers, but we can work with a data structure like this without worrying about the length. If the list is infinite, we know that we'll be able to access as many numbers as we actually need. This is a very mathematical problem, so it probably won't surprise you to hear that infinite lists are very common in Haskell.

Aside from mathematical challenges, the same concept can be useful in more mainstream programming too. When we drew a pie chart in chapter 4 we used random colors, but we could instead use an *infinite* list of colors generated in a way that makes the chart look clear. We'll see all these examples in the next chapter, but now we'll see how the idea can be represented using lazy values.

11.6.1 Creating lazy lists

Storing an infinite list of numbers in memory seems like a tricky problem. Obviously we can't store the whole data structure, so we need to store just part of it and represent the rest as a delayed computation. As we've seen, lazy values are a great way for representing the delayed computation part.

We can represent a simple infinite list in a similar way to an ordinary list. It is a cell that contains a value and the rest of the list. The only difference is that the rest of the list will be a delayed computation that gives us another cell when we actually execute it. We can represent this kind of list in F# using a discriminated union, as shown in listing 11.22.

Listing 11.22 Infinite list of integers (F#)

```
type InfiniteInts =
   | LazyCell of int *                      #1
                  Lazy<InfiniteInts>        #2
```

#1 Value in the current cell
#2 Next cell is a delayed computation

This discriminated union has only a single discriminator, which means it's similar to a record. We could have written the code using a record instead, but discriminated unions are more convenient for this example. The only discriminator is called `LazyCell` and it stores the value stored in the current cell (#1) and a reference to the "tail". The tail is a lazy value, so it will be evaluated on demand. This way, we'll be able to evaluate the list cell by cell and when a cell is evaluated, the result will be cached.

Lazy lists in F# and Haskell

As mentioned earlier, Haskell uses lazy evaluation everywhere. This means that a standard list type in Haskell is automatically lazy. The tail is not evaluated until the value is really accessed somewhere from the code.

In F#, lazy lists are not used very frequently. We'll see a more elegant way of writing infinite collections in F# and also in C# 3.0 in the next chapter. However, F# provides an implementation of lazy lists similar to the type we've implemented in this section. You can find it in the `FSharp.PowerPack.dll` library as `LazyList<'a>`.

Now let's that we've got our type, let's use it to create a simple infinite list that stores integers 0, 1, 2, 3, … as well as how to access values from the list.

Listing 11.23 Creating a list containing 0, 1, 2, 3, 4, … (F# interactive)

```
> let rec numbers(num) =                                         #1
     LazyCell(num,                                               #A
              lazy numbers(num + 1));;                           #B
val numbers : int -> InfiniteInts

> numbers(0);;                                                   #2
val nums : InfiniteInts = LazyCell(0, Lazy`1[...] { ... } )

> let next(LazyCell(hd, tl)) =                                   #3
     tl.Force();;                                                #C
val next : InfiniteInts -> InfiniteInts

> numbers(0) |> next |> next |> next |> next |> next;;           #D
val nums : InfiniteInts = LazyCell(5, Lazy`1[...] { ... } )
```

#1 Create an infinite list starting from 'num'
#A Store first value in the cell
#B The next cell is a delayed computation
#2 Get all numbers from zero
#3 Returns the next cell of the list
#C Evaluate the lazy value representing the next cell
#D Access sixth value from the list

We start by writing a recursive function numbers (#1) that returns an infinite list of integers starting with the number given as an argument and continuing to infinity. It returns a cell that contains the first value and a tail. The tail is a lazy value that (when evaluated) recursively calls numbers to get the next cell.

If we call the function with 0 as an argument, we'll get an infinite list starting from 0. The output from the F# interactive isn't particularly readable, but you can spot that the first value is 0 and that the tail is a value of type Lazy<InfiniteInts>. The subsequent command declares a function next, which gives us the next cell of the list (#3). We use pattern matching in the declaration to decompose the only argument. This looks a bit unusual, because you don't typically use discriminated unions with only a single discriminator, but it is the same principle as decomposing a tuple into its components. In the body of the function, we call the Force member, which evaluates the next cell. Finally, the last line uses the next function several times to read the sixth value from the list.

Of course, there are many more things that we could do with lazy lists, but I won't go into them here as we'll see a more idiomatic F# technique in the next chapter.

Writing functions for working with infinite lists

When working with the standard list type, we can use functions like List.map and List.filter. We can implement the same functions for infinite lists as well, but of course, not all of them. For example, List.fold and List.sort need to read all the elements, which isn't possible for our lazy list. As an example of what *is* possible, here's an implementation of the map function:

```
> let rec map f (LazyCell(hd, tl)) =
    LazyCell(f hd, lazy map f (tl.Force()));;
```

The structure is very similar to the normal map function. It applies the given function to the first value in the cell and then recursively processes the rest of the list. The processing of the tail is delayed using the lazy keyword. Other common list processing functions would look similar.

In this introduction to infinite data structures we've focused more on the functional style without even showing a C# example. Of course it would be possible to write the same type in C# now that we know how to write a lazy value in C#, but in the next chapter we'll see a more natural way for representing infinite structures or streams of values in C#.

11.7 Summary

We started this chapter with a discussion about refactoring functional programs. However, as we thought about what refactoring really means we saw how closely functional programs are related to mathematics, and the important benefits of this relationship.

In particular, we've seen that functions give us an easy way for representing small units of code. We used them to avoid code duplication, because we were able to represent only the smallest part of the code that varies between two uses using a function. Next, we looked at immutability and we've seen that immutability gives us a way to see how parts of our program depend on each other.

Then we focused at unit testing of functional programs using xUnit.net, which supports writing tests as functions in F# in a simple way. We saw how easy it was to combine unit testing and interactive testing using F# interactive. More importantly, I demonstrated how immutability makes it easier to test code, because we only need to test that a function gives the expected result: we don't need to worry about side-effects.

Next we turned our attention to laziness. We started by looking at built-in logical operators that behave lazily and explored ways to implement the same behavior using the features we already knew about, before examining F#'s built-in support for laziness.

In the last section, we briefly looked at using lazy values to create infinite data structures. However, this was really only a teaser for the next chapter. We'll talk about C# iterators and F# sequence expressions, both of which allow us to express a sequence of values in a much more natural way. This is just one example of a bigger idea, so we'll also look at how we can change or extend the meaning of code in general.

12 Sequence expressions and alternative computation models

Before we can start talking about sequence expressions, we have to explain 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 introducing enumerators is that they give us a unified way to work with collections of data such as arrays, mutable .NET lists and immutable F# lists. However, 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. We'll see that the infinite sequences, which sound somewhat academic, can still be useful in real applications.

We'll start this chapter by looking at various 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*.

However, 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*. These can be used for writing code that looks like ordinary F#, but behaves differently in some way. In the case of sequence expressions, a sequence of results is generated instead of just one value, but we'll look at various other examples. We'll see how to use computation expressions for logging, and how they can make option values easier to work with too.

NON-STANDARD COMPUTATIONS IN F#

Computation expressions can be used for customizing the meaning of the code in many ways, but there are some limits. 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 cannot be used to manipulate with the code and execute it in a different environment, in the way that LINQ to SQL does for example. This is possible in F# as well, but we'd have to combine ideas from this chapter with a feature called *F# quotations*, which isn't discussed in this book. You'll find additional resources about quotations on the book's web site.

We'll start by talking about sequences and once you'll 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 them, we need to know how to create them.

12.1 Generating sequences

There are several techniques to generate sequences, so let's look at some options we have. They all boil down to the same model: we have 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 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.

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 just look at one example. As we'll see later, both C# and F# give us an easier way to do 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_left` function, which takes a collection and "folds" it into a single value. On the other hand, `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(num.ToString(), 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. The first one is a value that will be returned in the sequence (in our case, it is the number converted to string); the second value 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# abbreviation for the IEnumerable<string> type. It's just 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, 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. However, the programming style used in C# has been evolving, and iterators are now used together with other functional constructs for a wide variety of data processing operations.

Iterators can be used for any type of sequence, and we'll look at 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 num = 0, factorial = 1;                                 #1
   while (factorial < 1000000) {
      num = num + 1;                                           #2
      factorial = factorial * num;                             #2
      yield return String.Format("{0}! = {1}", num, factorial);   #A
   }
}
```

#1 Local mutable state
#A Return the next string
#2 Modify the state of the iterator

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 the code above is how it works with the local state. First we declare two variables (#1) that represent the state of the iterator. The rest of the code is a loop that is executed every time we want to pull another value from the iterator. The loop body updates the local state of the iterator (#2) and then yields the newly calculated value.

The code is very imperative, because it heavily relies on mutation, but from the outside iterators look like functional data types, because the mutable state is hidden. Now let's look at *sequence expressions*, which represent a more general idea, but can be used for generating sequences in F#.

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 non-standard. This new statement is indicated with `yield return` (or `yield break` to terminate the sequence) and the non-standard behavior is to return a value as the next element of a lazily generated sequence. Sequence expressions in F# are similar: they use an operator which is equivalent to `yield return`, and one other additional feature.

WRITING SEQUENCE EXPRESSIONS

In C#, we can use iterators to implement methods that return `IEnumerable<T>`, `IEnumerator<T>` or their non-generic equivalents. On the other hand, F# sequence expressions are marked explicitly using the `seq` keyword, and don't have to be used as the body of a method or function. As the name suggests, sequence expressions are just 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                        #1
           yield n + 1                       #2
           printfn "second.."                #3
           yield n + 2 }
val nums : seq<int>                          #4
```

#1 Expression wrapped inside 'seq'
#2 Return element of the sequence
#3 Side-effect inside sequence expression
#4 Value is a sequence of numbers

When writing sequence expressions, we enclose the whole F# expression that generates the sequence in a `seq` block (#1). The block is written using curly braces and the `seq` keyword at the beginning denotes that the compiler should interpret the body of the block as a sequence expression. We'll later see that there are other keywords for similar features. This block turns the whole expression into a sequence. You can see this by looking at the inferred type of the value (#4).

The body of the sequence expression can contain statements with a special meaning. Similarly to C#, there is a statement for returning a single element of the sequence. In F# this is written using the `yield` keyword (#2). Of course, the body can also contain other standard F# constructs such as value bindings, and even calls that perform side-effects (#3).

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.of_seq` function, which converts the sequence to an immutable F# list:

```
> nums |> List.of_seq;;
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. However, if we take only a single element from the sequence, the sequence expression will only evaluate until the first `yield` call, so the string will not be printed:

```
> nums |> Seq.take 1 |> List.of_seq;;
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. For example, 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.

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. Composability is a more important aspect in functional programming, so F# allows us to compose sequence expressions and yield the whole sequence from a sequence expression using a special language construct: `yield!` (usually pronounced yield-bang). Listing 12.3 demonstrates this, creating a sequence of cities in three different ways.

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

```
> let capitals = [ "Paris"; "Prague" ]                                #1
  let withNew(name) =                                                 #2
     seq { yield name
           yield "New " + name };;
val capitals : list<string>
val withNew : string -> seq<string>

> let allCities =
     seq { yield "Oslo"                                               #3
           yield! capitals                                            #4
           yield! withNew("York") };;                                 #5
val allCities : seq<string>

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

#1 List of capital cities
#2 Return a name and a name with a prefix
#3 Return a single value
#4 Compose with another sequence
#5 Return all cities generated by the function
#A All data composed together

The listing starts by creating two different data sources. The first one is an F# list that contains two capital cities (#1). 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 (#2) 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 (#3). This shows that you can mix both ways of yielding elements inside a single sequence expression. Next, we use the `yield!` construct to return all the elements from the F# list. Finally, we call the function `withNew` (#5), which returns a sequence, and return all the elements from that sequence.

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 (#5), 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, stop executing and wait until the caller requests another element.

We've learned almost everything about the syntax of sequence expressions, but they can sound quite awkward until you actually start using them. There are several patterns which 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#. However, we haven't yet seen 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) =                          #1
      seq { if (factorial < 1000000) then
                yield String.Format("{0}! = {1}", num, factorial)   #2
                let num = num + 1
                yield! factorialsUtil(num, factorial * num) }       #3
val factorialsUtil : int * int -> seq<string>

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

#1 Recursive utility function
#2 Return single result
#3 Recursively generate remaining factorials
#4 Get sequence starting from the first factorial

The listing starts with a utility function that takes a number and its factorial as an argument (#1). When 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 (#4). 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 will not yield any additional numbers. If the condition is true, we first yield a single result (#2), which indicates the next factorial formatted as a string. Next, we increment the number and perform a recursive call (#3). 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 would be inefficient due to a large number of nested loops. In F#, the `yield!` construct is implemented in an optimized way so it doesn't add any significant inefficiency.

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, 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.

List and array expressions

So far, we've seen sequence expressions enclosed in curly braces and denoted by the `seq` keyword. This kind of expression generates a lazy sequence of type `seq<'a>` which correspond to the standard .NET `IEnumerable<T>` type. However, 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 =                            > let cities =
     [ yield "Cambridge"                       [| yield "Barcelona"
       yield! capitals ];;                        yield! capitals |];;
val cities : list<string>                 val cities : array<string>

> List.hd(cities);;                       > cities.[2];;
val it : string = "Cambridge"             val it : string = "Barcelona"
```

As you can see, we can also enclose the body of the sequence expression in square braces just as we normally do to construct F# lists, and in square braces followed by the bar "|" to construct arrays. F# treats the body as an ordinary sequence expression and converts the result to a list or an array respectively. The example above shows that we can then use the resulting collection in a normal way.

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

12.2.2 Using infinite sequences

In the previous chapter, I 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 element was delayed: the element's value was only calculated when we actually 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 useful and give us a great way to separate different parts of an algorithm and make the code more readable.

In the last chapter, I briefly mentioned that when drawing charts, we could represent the colors used by the chart as an infinite sequence of colors. Listing 12.5 shows how we can implement a sequence generating random colors in both C# and F#.

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

```
IEnumerable<Color> ColorsRnd() {
   Random rnd = new Random();
   while(true) {                 #1
      int r = rnd.Next(256),
          g = rnd.Next(256),
          b = rnd.Next(256);
      yield return Color         #2
         .FromArgb(r, g, b);     #2
   }
}
```

```
let rnd = new Random()
let rec colorsRnd = seq {       #3
   let r, g, b =
      rnd.Next(256),
      rnd.Next(256),
      rnd.Next(256)
   yield Color.FromArgb(r,g,b)  #4
   yield! colorsRnd }           #5
```

#1, #5 Both implementations contain an infinite loop that generates colors. In C#, the loop is achieved using 'while(true)' (#1). The functional way to create infinite loops is to use recursion (#5).
#2, #4 In the body of the infinite loop, we yield a single randomly generated color value. In F# this used the 'yield' construct (#4) and in C# we use 'yield return' (#2).

Annotations below the code with cueballs on the left

If you compile the F# version of the code, you'll get a warning on the line with the recursive call (#5). The warning says that the recursive reference will be checked at run-time. We've already seen 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 more compact way of enclosing F# code in a `seq` block (#3). Instead of starting on a new line and indenting the whole body, we added the `seq` keyword and the opening curly brace to the end of the line. I'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 extremely similar) is available online.

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

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

let numbers = [ 490; 485; 450; 425; 365; 340; 290; 230; 130; 90; 70; ]
let clrData = Seq.zip numbers colorsRnd                                #1

let frm = new Form(ClientSize = Size(500, 350))
frm.Paint.Add(fun e ->
   e.Graphics.FillRectangle(Brushes.White, 0, 0, 500, 350)
   clrData |> Seq.iteri(fun i (num, clr) ->                            #2
      use br = new SolidBrush(clr)
      e.Graphics.FillRectangle(br, 0, i * 32, num, 28) )               #A
   )
frm.Show()
```

#1 Combine data with colors into one sequence
#2 Iterate over data and colors with index
#A Calculate location of the bar using index

In order 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 (#1). 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. Of course we could generate let's say 100 of 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.

In the C# version, we'll need to implement an alternative to the `zip` function if we want to run the code under .NET 3.5. (There's an implementation built into .NET 4.0.) This is an interesting example of sequence processing, so we'll look at the C# implementation later.

Once we have the combined sequence, we just need to iterate over its elements and draw the bars. We're using `Seq.iteri` (#2), 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 naively, the color would be computed in the drawing function (#2). 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#)

```
IEnumerable<Color> ColorsGrBl() {
   while(true) {                      #1
      for(int g=0; g<255; g+=25) {
         int r = g / 2, b = g / 3;
         yield return Color.
            FromArgb(r,g,b);
      }
   }
}
```

```
let rec colorsGrBl = seq {
   for g in 0 .. 25 .. 255 do    #2
      let r, b = g / 2, g / 3
      yield Color.FromArgb(r,g,b)
   yield! colorsGrBl }           #3

let clrData =
   Seq.zip numbers colorsGrBl
```

The code in the listing again contains an infinite loop, implemented either using a `while` loop (#1) or using recursion (#3). 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 then 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 (#2). The value of g will start off as 0 and then increment by 25 for each iteration until the value is larger than 250. Figure 12.1 shows the end result.

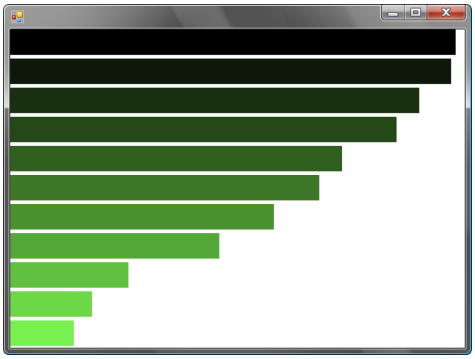


Figure 12.1 A chart painted using color gradient generated as a sequence of colors.

As you can see, infinite sequences can be useful in real world programming, becaus 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 are often used to express numerical calculations.

Infinite lists in Haskell and sequence caching in F#

As I mentioned in the previous chapter, 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 allow us to emulate some other Haskell constructs in the same way. 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 is 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 braces 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 then 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. However, the F# version is horribly inefficient, because in each recursive call, it starts enumerating the sequence from the first element. In Haskell, the calculated values are cached, so it gives better results. Fortunately, the F# libraries give us an easy way to cache values too:

```
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.

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

12.3 Processing sequences

The basic 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. In C#, we can use LINQ methods such as `Where` and `Select` that work with any sequence (represented using the `IEnumerable<T>` type).

However, higher order functions aren't the only option we have. When we need to implement some more complex lower-level behavior in C# (for example if we wanted reimplement LINQ methods such as Where) we can use iterators. On the other hand, for writing higher-level processing code, we can use the C# 3.0 query syntax. The F# language doesn't explicitly support any query syntax, but we'll see that sequence expressions to some point unify the ideas behind iterators and queries.

12.3.1 Transforming sequences in C#

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'd use exactly the 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 is a more interesting problem, because we cannot use foreach to simultaneously take elements from two source sequences. As you can see in the listing 12.8, the only option that 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())                   #1
   using (var secondEn = second.GetEnumerator()) {               #1
      while (firstEn.MoveNext() && secondEn.MoveNext()) {        #2
         yield return Tuple.New(firstEn.Current, secondEn.Current);     #3
      }
   }
}
```

#1 Get enumerators for both of the sequences
#2 Loop until one sequence ends
#3 Return elements from both sequences in a tuple

If we look 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 our C# tuple implementation from chapter 3, 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 (#1). We repeatedly call the MoveNext method on each enumerator to get the next element from both of the sequences (#2). If neither sequence has ended, we yield a tuple containing the current element of each enumerator (#3).

This example shows that sometimes processing methods need to use the IEnumerator<T> interface explicitly, because not everything can be implemented using the foreach loop. If we wanted to implement Seq.zip in F#, we'd have to use exactly the same technique. We could either use a while loop inside a sequence expression or a recursive sequence expression. However, most of the processing functions we'll need are already available in the .NET and F# libraries. We'll use these where we can, either explicitly or 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've already 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 exactly the same as with List, and we've already seen how to use LINQ extension methods in C#. However, there is one notable difference between working with lists and sequences: sequences are lazy. The processing code isn't executed until we actually 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:

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

When we run the code above, it will not process any elements. It just 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.

Anyway, you're probably already familiar with using higher order processing functions after our extensive discussion in chapter 6 and many examples in various places of the book. In this chapter, we'll instead look at other ways to express unusual computation models.

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 just 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 which just 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 the example above using a query in C# and a sequence expression in F#.

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

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

In C#, query expressions and iterators are quite different, but the sequence expression in F# shows 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` statement, 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 are not easily expressible using F# sequence expressions. This means that the C# version is more powerful, but the F# implementation is more uniform.

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 just projection and filtering. These operators are mostly present for this single purpose and there is no special syntax for them in F#. However, all these standard operators are available as regular higher order functions operating on sequences. For instance, take ordering data:

```
var q =                             let q =
   from c in customers                 customers
   orderby c.Name                      |> Seq.order_by (fun c -> c.City)
   select c;
```

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

```
var q =                             let q =
   from c in customers                 customers
   group c by c.City;                  |> Seq.group_by (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 the examples above, 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 can be easier to understand.

It's interesting to look at 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`, `Join` and `GroupBy` using lambda expressions. These are typically extension methods, but 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" lies behind the ability to use the same syntax for both in-process queries with LINQ to Objects and out-of-process queries using LINQ to SQL, LINQ to Entities and similar providers.

On the other hand, sequence expressions can be used to express more complicated and general-purpose constructs. For example 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 would require a separate method–you couldn't express the transformation "inline".

The actual implementation of sequence expressions in F# may be optimized by the compiler, but it could be implemented using a flattening projection, which we'll discuss in the next section. This is an interesting point, because F# allows us to define our own non-standard computations, so it is useful to know how they work behind the scenes.

12.3.3 Flattening projections

A *flattening projection* allows us to generate a sequence of elements for each element from the source collection and then merges all the returned sequences.

TERMINOLOGY IN LINQ AND F#

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 is 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.map_concat`. 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 discussing the F# function, because its signature is slightly simpler than the one available in LINQ.

USING FLATTENING PROJECTIONS IN F#

As usual, the first step in understanding how the function works is to look at its type signature. Figure 12.2 compares the signatures of `Seq.map` and `Seq.map_concat`.

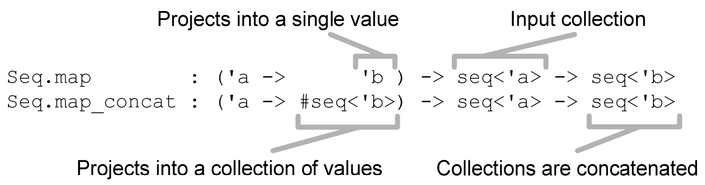

Projects into a single value
Input collection
Seq.map : ('a -> 'b) -> seq<'a> -> seq<'b>
Seq.map_concat : ('a -> #seq<'b>) -> seq<'a> -> seq<'b>
Projects into a collection of values
Collections are concatenated

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

Just as a reminder, the "#" symbol in the part of the type signature describing the projection function passed to `map_concat` means that the actual return type of the function doesn't have to be exactly the `seq<'b>` type. It can be any type implementing the `seq<'b>` interface. This means that we can return an F# list, an array or even our own collection type.

Now let's look at an example. Suppose we have a list of tuples, each of which contains a city's name and the country it's in, and we also have a list of cities selected by a user. We can represent some 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 perform a projection (using the `Seq.map` function) 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 UK and USA, so we need to be able to return multiple records for a single city. This can be done using a flattening projection with `Seq.map_concat` because the function that we specify as an argument can return a collection of elements. Listing 12.10 shows the code to implement this.

Listing 12.10 Searching for country of entered cities (F# interactive)

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

#1 Find all cities and format the output
#A Both countries returned for 'Cambridge'

The code above calls the flattening projection and gives it list of cities entered by the user as input. 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. This is implemented using a nested `map_concat` call. The lambda function we use in here

(#1) returns a list containing a single element if the name of the city matches or an empty list if the name is different.

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. In the example above, we could also implement the nested call with `Seq.filter` (to find cities with the same name) and `Seq.map` (to format the output), but I intentionally used a nested `map_concat`, because it will help us understand how sequence expressions work. Listing 12.11 shows how we can use them to solve the same problem.

Listing 12.11 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                                        #3
              yield sprintf "%s from %s" n c };;                      #3
val it : seq<string> = seq [ "London from UK"; "Cambridge from UK";
                             "Cambridge from USA" ]
```

#1 Return a collection for each entered city
#2 Iterate over all known cities
#3 Yield zero or one element for each combination

This example is definitely easier to read and is the preferred way for working with sequences in F#. However, thanks to our previous discussion about the flattening projection, you can better understand how sequence expressions work. As you can see, we're using two `for` loops: one to iterate over the entered names (#1) and one to iterate over the list of known cities (#2). These two loops correspond to the two nested `map_concat` calls in the previous implementation and implement the cross join of two collections. The code that is nested in these two loops (#3) uses the `yield` statement to produce a single item if the names are the same or doesn't yield any items, which corresponds to the case that returned an empty list in the previous implementation.

I 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 look at the same operation in C#.

USING FLATTENING PROJECTIONS IN C#

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

In C#, it will be easier to see the code using query syntax first and then see how it is 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 country contains instances of a class `CityInfo` with two properties and the list of entered names contains just strings. The listing 12.12 shows a LINQ query that we can write to find countries of the entered cities.

Listing 12.12 Searching for country of entered cities using a query (C#)

```
var q =
   from e in entered                                                  #1
   from known in cities                                               #2
   where known.City == e                                              #3
   select string.Format("{0} ({1})", known.City, known.Country);     #3
```

#1 Iterate over the entered names
#2 Search all known cities
#3 Filter matching cities and format the output

The query expresses exactly the same idea as we did in the previous implementations. It iterates over both of the data sources ((#1) and (#2)), 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 (#3).

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 I said before, 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 12.13 shows the translation as it is performed by the C# compiler.

Listing 12.13 Query translated to explicit operator calls (C#)

```
var q = entered
   .SelectMany(
      (e => cities),                                                  #1
      (e, known) => new { e, known })                                 #2
   .Where(tmp => tmp.known.City == tmp.e)                             #A
   .Select(tmp => String.Format("{0} ({1})",                          #A
      tmp.known.City, tmp.known.Country));                            #A
```

#1 Foreach entered city, iterate over known cities
#2 Create temporary value storing the results of join
#A Filter and format the 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 (#1) 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# `map_concat` function. In the query above, we just return all the known cities, so the operation performs only joining, without any filtering or further processing. The second parameter (#2) 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 just 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. On the other hand, 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 in the query together with the last `select` clause are translated into a call to the `Select` method. However, the remaining `from` clauses are compiled to flattening projections using the `SelectMany` method. Multiple flattening projections are nested in the same way as in F#.

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

12.4 Introducing computation expressions (monads)

You may well have heard of monads before. They have an unfortunate reputation for being brain-bustingly difficult - but don't worry, I promise I'll introduce them gently. You may be surprised to know we've already been using them in this chapter. In fact you've probably used them before even picking up this book: LINQ is based on a monad too.

In section 6.7 we looked at the `bind` function for both option values and lists. The equivalent function for sequences is `Seq.map_concat`; we've just seen its importance in LINQ queries and F# sequence expressions. Let me just briefly remind you the type signatures of the three operations:

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

The function provided as the argument specifies what to do with each actual 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. 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. However, nothing limits us to use the query syntax only for working with collections.

12.4.1 Customizing query expressions

In general, 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 above. Now let's consider what the meaning of a query applied to option types would be. The listing 12.14 shows two queries side by side. The first one works with lists and the second 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 12.14 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 exactly the same with the exception that they work with different types of data, so they use different query operator implementations. Both of them read two different inputs and then return multiples of the entered integers. Table 12.1 gives some examples of inputs to show what the results would be.

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

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

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. If the second input is also a value then the result is again `Some` value containing the result of the multiplication. On the other hand, if the second input is `None` then we don't have values to multiply, so the query returns `None`. Finally, when the first input is `None`, then we already know the result without even needing the second input. The whole query is executed lazily, so we don't even 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 12.14 is definitely easier to write (and read) than the equivalent code we used in chapter 6, where we used higher order functions explicitly. We'll see how to implement all the necessary query operators later in the chapter, but let's first look at some 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 just a single special case that is implemented in the F# core.

We've seen that computation expressions can contain standard language constructs such as `for` loops, but also additional non-standard constructs like `yield`. The identifier that precedes the block gives the meaning to these constructs in a same sense as query operators (e.g. `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 12.15. The version on the left side uses syntax similar to sequence expressions and the version on the right is a more natural way of writing the same thing.

Listing 12.15 Computation expressions for working with option values (F#)

```
option {                                option {
   for n in tryReadInt() do                let! n = tryReadInt()
      for m in tryReadInt() do             let! m = tryReadInt()
         yield n * m                       return n * m
}                                       }
```

In the version on the left side, each `for` loop can be executed at most once. When the option value contains an actual value, it will be bound to the symbol `n` or `m` respectively and the body of the loop will execute. However, 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 value, we'll use the syntax on the right.

The right side uses two more non-standard primitives. The first one is `let!` (read let-bang), which represents a customized value binding. In the example above, the type of values `n` and `m` is `int`. The non-standard value binding un-wraps 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 non-standard primitive in the expression is `return`. It specifies how to construct an option value from the actual value. In the example above, 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. In the following sidebar we'll discuss the core concepts of the idea and I'll also mention how it relates to Haskell monads, which are the origin of F# computation expressions.

Understanding computation expressions and monads

Computation expressions in F# are an implementation of an idea that has been proven useful in Haskell, called *monads*. The name *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 type 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 to return an undefined value (`None`). In fact, 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. The first one is *bind* that allows us to create and compose computations that work with values of the monadic type. In listing 12.15, the bind operation was used whenever we used the `let!` primitive. The second operation is *return*, which is used to construct a value of the monadic type. In the example above, this is the F# `return` keyword.

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 from the previous sidebar 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 non-standard computation is to think about the type that which represent 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 `TheValue<T>` and it will simply store the value of the generic type parameter T. It will not 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#.

The listing 12.16 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 12.16 Value of the computation in C# and F#

```
class TheValue<T> {                          type TheValue<'a> =
   public T Value { get; set; }  #1             | Value of 'a       #2
}
```

The C# type is very simple and just stores the value of type T. If we wanted to implement the code more properly, we'd make it immutable (so the property (#1) would be read only). We don't actually mutate any values after initialization, but I'm trying to keep the example as concise as possible so we can concentrate on the new ideas.

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 very 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 look at 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 non-standard way (by returning some monadic value) as if they returned an ordinary value. The computation type we're using in this section is `TheValue<T>`, so primitive functions will return values of type `TheValue<T>` instead of just 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 very 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 `TheValue<T>` is very easy. The following code shows how to implement a primitive method in C# that reads a number from the console and wraps it inside this computation type:

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

The method just reads a number from the console and wraps it inside the `TheValue<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 just need to know that the type supports all the primitives (most importantly bind and return) that are needed to write a query or computation expression. Listing 12.19 shows a snippet that reads two integers using the primitive above and performs a calculation with then.

Listing 12.19 Calculating with computation values in C# and F#

```
var v =                                       value {
   from n in ReadInt()           #1              let! n = readInt()            #2
   from m in ReadInt()           #1              let! m = readInt()            #2
   let add = n + m                               let add = n + m
   let sub = n - m                               let sub = n - m
   select add * sub;             #3              return add * sub }            #4
```

#1, #2 In C# we're using the 'from' clause to access the actual values (#1). In F#, we can use the customized value binding (#2) to unwrap the value

#3, #4 Once the calculation is done, we again wrap the actual value inside the computation type. In C#, we're using a 'select' clause (#3) and in F# we use the 'return' primitive (#4)

Annotations below the code with bullets on the left side

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 non-standard 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, I'll focus more on the F# version, because it will make it easier 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 non-standard computations. However, we'll get back to C# soon once we've implemented the F# computation expression.

You can see that the example above is using just two primitives. The bind primitive is used when we call the computation primitives (#2) and the return primitive is used to wrap the result in the `TheValue<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 the computation operators

The identifier value that precedes the computation expression block is actually an object that implements the required operations. Various 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 12.19, 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 is translated to a call to the `Bind` member. This is because the `readInt` function actually returns a value of type `TheValue<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 actual value from the computation type and call the function that represents the rest of the computation with this value as an argument.

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 cannot 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.

The example above also shows that multiple `let!` constructs are translated into a nested calls to the `Bind` member. This is because the function given as the last argument to this member represents everything in the rest of the computation. The example above 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<'a> * ('a -> M<'b>) -> M<'b>
Return : 'a -> M<'a>
```

In our previous example, the type `M` is of course the `TheValue` type. In general, the bind operation needs to know how to get the actual 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<'a>`) with the information extracted from the result of the function call (of type `M<'b>`) and return them as part of the overall result. The return operation is much simpler, because it simply constructs a computation from the primitive value. If the computation carries some additional information, the return operation sets this argument to some value that represents empty information.

The object named `value` in the previous example which is used when constructing the computation is called a *builder* in F#. Now that we know what members it has and what their type signatures are, we can start implementing them.

IMPLEMENTING A COMPUTATION BUILDER IN F#

Listing 12.20 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 12.20 Implementing computation builder for values (F#)

```
type TheValueBuilder() =
   member x.Bind(Value(v), f) = f(v)                #1
   member x.Return(v) = Value(v)                    #2
let value = new TheValueBuilder()                   #A
```

#1 Invoke the rest of the computation
#2 Wrap value inside the computation type
#A Create instance of the builder

The `Bind` member (#1) first needs to unwrap the actual value from the `TheValue<'a>` 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 actual 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 just wraps the actual value inside the computation type.

Using the `value` declared in this listing, we can now run the computation expression from listing 12.19. F# also lets us use computation expressions to implement the `readInt` function as well. We just need to wrap the result in instance of `TheValue<int>` type, which can be done using the `return` primitive:

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

This function doesn't need the bind operation, because it doesn't use any values of type `TheValue<'a>`. The whole function is enclosed in the computation expression block, which causes the return type of the function to be `TheValue<int>` instead of just `int`. If we didn't know anything about the `TheValue<'a>` type, the only way to use the function would be to call it using the `let!` primitive from another computation expression. This is important because it shows that the bind operation gives us a way to compose computation expressions.

This isn't possible in C#, because the query cannot begin with a `let` clause (which corresponds to a standard F# let binding). However, even queries like the one we've seen earlier can be useful. Now that we've got the F# part of listing 12.19 compiling, let's implement the C# part.

IMPLEMENTING QUERY OPERATORS IN C#

We've seen how the C# queries are translated to method calls in listing 12.13 when we were talking about sequences and when we analyzed the `SelectMany` operation. I said that the query with a single `from` clause can be translated to a call to the `Select` method, but for multiple `from` clauses, we'll also need the `SelectMany` operation. When writing computations using queries, we use the `from` clause in a similar way to the F# `let!` construct to represent a non-standard value binding, so we'll use it quite often. This means that we'll need to implement both `Select` and the `SelectMany`.

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 12.21 shows the implementation of both of the primitives.

Listing 12.21 Implementing query operators (C#)

```
static class TheValueExtensions {
   public static TheValue<R> Select<S, R>
         (this TheValue<S> source, Func<S, R> sel) {
      return new TheValue<R> { Value = sel(source.Value) };         #1
   }
   public static TheValue<R> SelectMany<S, C, R> (this TheValue<S> source,
         Func<S, TheValue<C>> sel, Func<S, C, R> selRes) {
      var newVal = sel(source.Value);                               #2
      var resVal = selRes(source.Value, newVal.Value);              #3
      return new TheValue<R> { Value = resVal };                    #4
```

```
        }
    }
```

#1 Projection for 'TheValue' type
#2 Unwrap the value and run the function
#3 Combine values to build the result
#4 Wrap and return the result

Both of the methods are implemented as extension methods. This means that C# will be able to find them when working with values of type `TheValue<T>` using the standard dot notation which is used during the translation from the query syntax. The `Select` operator (#1) implements projection using the given function, so it only needs to access the wrapped value, run the given function and then wrap the result again.

The `SelectMany` operator is confusing at first, but it's useful to look at the types of the various 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 (#2). However, we also need to run function pass in as the final argument to combine the source value with the value returned by the first function call. We call the second function (#3), and wrap the result into the computation type (#4) and use it to return from the method.

PRIMITIVES IN C# AND F#

We had to implement different primitive operators for F# and C#, but they're closely related. In particular, if we had methods representing the bind and return operators in C#, we could implement both `Select` and `SelectMany` with just those operators. This doesn't work the other way round, because there's no way to implement the return operator using just the LINQ primitives. This explains why we could write `readInt` using computation expressions in F#, but we can't do the same thing in C# using a query.

After implementing the operators above, the query expression in listing 12.19 will compile and run. The computation type that we created in this section is quite simple, because it doesn't augment the computation with any additional information. However, 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

I used option values as an example in section 12.4 when I introduced the idea of creating non-standard 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 expression.

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.

In order 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 12.22 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 show what a typical bind operation does.

Listing 12.22 Computation builder for option type (F#)

```
type OptionBuilder() =
   member x.Bind(v, f) =
      match v with                         #1
      | Some(value) -> f(value)            #2
      | _ -> None                          #3
   member x.Return(v) = Some(v)            #A

let option = new OptionBuilder()
```

#1 Unwrap the option value
#2 Run the rest of the computation
#3 The result is undefined
#A Wrap actual value

The Bind member starts by extracting the actual value from the option given as the first argument. This is similar to the Bind we implemented earlier for the TheValue<'a> type. Again we're using pattern matching (#1), but in this case, the value may be undefined so we're using the match construct. If the value is defined, we call the specified function (#2). 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 (#3).

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.

In order 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 12.23, implements 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 12.23 Query operators for option type (C#)

```
static class OptionExtensions {
   public static Option<R> Select<S, R>
         (this Option<S> source, Func<S, R> sel) {
      return source.Map(sel);                                      #1
   }
```

```
      public static Option<R> SelectMany<S, C, R>(this Option<S> source,
            Func<S, Option<C>> sel, Func<S, C, R> selRes) {
         return source.Bind(s =>                                       #2
            sel(s).Map(c => selRes(s, c)));                            #3
      }
   }
```

#1 Same operation is called 'Map'
#2 Use 'Bind', which we have already
#3 Format the result

The Select method should apply the given function to the value carried by the given option value if it contains an actual value and then again wrap the result into an option type. We've already implemented this functionality under a different name. 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 just calls Map (#1).

SelectMany is more complicated. It is 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 (#2). To call the formatting function selRes, we need two arguments - the original value carried by the option and the value produced by the binding function (named sel). 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 (#3). This is because we also need to access the original value from the source. Inside the lambda function, the original value is in scope (variable named s), so we can use it together with the new value, which is assigned to the variable c.

This implementation is a bit tricky, but it shows that many things in functional programming can be just composed from what we already have. If you try 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. However, you'd see what types are incompatible and if you looked at what functions are available, you'd quickly discover what needs to be added in order to get the correct types. As I wanted to highlight several times in this book, 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. However, we still need to implement the TryReadInt method (and the similar F# function). These are really simple, because they just 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 called the i*dentity* 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 called the *maybe* monad, because `Maybe` is the Haskell type name that corresponds to the `option<'a>` type in F#.

The first example was mostly just a toy example to demonstrate what we need to do when implementing computations, but the second one can be useful when writing code that is 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 non-standard.

So far the examples have been somewhat abstract. Our next section is a lot more concrete. It allows us to add automatic logging into our code.

12.7 Augmenting computations with logging

Logging can be usually implemented using global mutable state. If we want to use different logging mechanisms in different parts of the program, it's best to avoid having global loggers. However, implementing that would be quite difficult, because we'd have to pass a state of the logger as an additional argument to every function we call.

However, we can create a non-standard computation that enables logging and hides the state of the logger inside the computation type. This example relies on the fact that we can surround any piece of standard F# code with the computation expression block. As such, it's not really 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 in addition, it 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<'a> =
   | Log of 'a * list<string>
```

This type is quite similar to the `TheValue<'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 allows us to write computations that don't return any value. We'll see how that's used later on.

The implementation of the builder is shown in listing 12.24. 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 12.24 Computation builder that adds logging support (F#)

```
type LoggingBuilder() =
   member x.Bind(Log(v, logs1), f) =        #1
      let (Log(nv, logs2)) = f(v)           #2
      Log(nv, logs1 @ logs2)                #3
   member x.Return(v) =
      Log(v, [])                            #A
   member x.Zero() =
      Log((), [])                           #B

let log = new LoggingBuilder()
```

#1 Unwrap the value and log buffer
#2 Run the rest of the computation
#3 Wrap the value and merge log buffers
#A Augment value with an empty log
#B No value with an empty log

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 `TheValue` types:

1. In the first step, 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 (#1).
2. The second step is to call the rest of the computation if we have a value to do that. In the example above, we always have the value, so we can run the given function (#2). However, 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 in the last step 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 (#3).

In the next chapter, we'll see a useful non-standard computation where the whole computation is delayed. In that case, the steps above will be a bit different, because the `Bind` member returns a result that captures the rest of the computation and can run it later, but in general, most of the bind operations look like the one we've just seen.

The `Return` and `Zero` members are simple. `Return` just needs to wrap the actual value into the `Logging<'a>` type and the `Zero` represents a computation that doesn't carry any value (meaning that it returns a unit). In both of the 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. However, if you look at the code we've got so far, there is no way we could create a non-empty 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 logWrite(s) =
     Log((), [s])
```

```
    val logWrite : 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 some code that uses the newly created logging computation.

12.7.2 Using the logging computation

Listing 12.25 starts off by implementing two helper functions for reading from and writing to the console. Both of them will also write a message to the log, so they will be enclosed in the `log` computation block. We then use these two functions in a third function, to show how we can compose non-standard computations. In the previous examples, we used the `let!` primitive, but listing 12.25 introduces `do!` as well.

Listing 12.25 Logging using computation expressions (F# interactive)

```
> let write(s) = log {                                  #1
      do! logWrite("writing: " + s)
      Console.Write(s) }
val write : string -> Logging<unit>

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

> let testIt() = log {
      do! logWrite("starting")          #2
      do! write("Enter name: ")         #3
      let! name = read()                #4
      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 the log
#2 Call the primitive logging function
#3 Call function written using computation expressions
#4 Using customized value binding

We use the new `do!` primitive in several places, to call functions that return `Logging<unit>`. In this case, we want to write a non-standard binding that executes the `Bind` member, because we want to concatenate logging messages. However, we can ignore the actual value, because it is `unit`. That's the exact behavior of the `do!` primitive. In fact, when we write `do! f()`, it is just a shorthand for writing `let! _ = f()`, which uses the customized binding and ignores the returned value.

When implementing the computation builder, we added a member called `Zero`. This is used behind the scenes in the example above. When we write a computation that doesn't return anything (#1), the F# compiler automatically uses the result of Zero as the overall result. If we didn't provide this member, we'd have to explicitly write `return ()`. This would call the `Return` member with a unit value as an argument.

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<'a>`), which is same as the result type of the `logWrite` function that we implemented earlier. This demonstrates that we have two ways of writing functions of a non-standard computation type. We can build the computation type directly (as we did in the `logWrite` function) or use the computation expression. The first case is useful mostly for writing primitives and 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 (#2). Writing to the screen (and to the log) is implemented using a computation expression, but we call it in exactly the same way (#3). Finally, 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 (#4). In general, the code we wrote looks just like an ordinary F# code with a couple of added "`!`" symbols.

Refactoring using computation expressions

In the previous chapter, we saw various 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.

There is 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<'a>`. You can try implementing the computation on your own: the only difficult part is writing the `Bind` member. We won't talk about this any more here, but you can find additional information on the book's web site.

The interesting thing is how easy it is to turn standard F# code into code that has the non-standard behavior. We have to enclose the code into a computation expression block and then add calls to the primitives provided for the computation expression, such as the `logWrite` 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, and then refactor it into an advanced version using computation expressions.

Perhaps the most difficult thing about using computation expressions is to identify when it is beneficial to design and implement them. After summing up the examples we've just seen, we'll look at one very useful real-world example in the next chapter.

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 `map_concat` function in F# and the `SelectMany` method in LINQ.

However, 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've looked at two practical examples, implementing computation expressions for working with option types and to store log messages during execution. The same idea can be implemented in C# to some extent, with query expressions being used in the place of computation expressions. However, the F# language features are more general, while C# query expressions are really tailored to queries.

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# allows us to interactively process and visualize data, which is becoming an important task in the today's increasingly-connected world.

13 Asynchronous data retrieval and processing

Let me start this chapter with a quote from a recent interview with Bill Gates. He talks about the type of programming tasks that he's interested in and describes the typical scenario when writing the application:

> *"Grab data from the web, don't just think of it as text, bring structure into it and then […] try out different ways of presenting it, but very interactively. […] Write a little bit of code that may have your specific algorithm for processing that data." [Gates, 2008]*

This brief quote exactly describes what we're going to do in this chapter and as you'll see, the F# language and its interactive shell are excellent tools for solving this kind of task. We'll call this approach *explorative programming*, because the goal is to explore a massive amount of data and find a way to gather useful information from it. We'll spend most of the chapter working with F# interactive, because it gives us a great way to "write a little bit of code" with "our specific algorithm for processing the data" and immediately execute it to see the results.

The F# language and libraries support this type of programming in many ways and we'll look at all the important technologies involved. To obtain the data, we can use asynchronous workflows based on the computation expression syntax that we introduced in the previous chapter. Then we'll look at "bringing a structure" to the data using F# types. We'll also use units of measure that allow us to specify that a certain value isn't just a floating point number, but that it has a unit such as square kilometers.

Finally, we'll look at "trying out different ways of presenting the data". In particular, we'll see how to export the structured data to Excel using its .NET API, and programmatically visualize the data as a chart.

13.1 Asynchronous workflows

In one sense, *asynchronous workflows* are just another example of F# computation expressions–but they are one of the most important implementations of the idea, allowing us to write asynchronous code in a readable and efficient way. Let's start off by looking at a simple but practical example.

13.1.1 Downloading web pages asynchronously

There are many areas where we can use asynchronous operations. When working with disk or connecting to the database, asynchronous workflows can give us a notable performance benefit. However, the best example is downloading content from the web. The network connection is often slow or unreliable, so if we used synchronous operations from the main UI thread, our application could be unresponsive for a significant period of time.

Before we can use asynchronous workflows to fetch web content, we'll need to reference the `FSharp.PowerPack.dll` library that contains asynchronous versions of many .NET methods. When developing a standalone application, you would use the "Add Reference" command. In this chapter we're using the interactive development style, so we'll create a new F# Script File and use the `#reference` directive. You can see the content of the script file in the listing 13.1.

Listing 13.1 Writing code using asynchronous workflows (F# interactive)

```
> #reference "FSharp.PowerPack.dll";;

> open System.IO
  open System.Net
  open Microsoft.FSharp.Control;;                    #A

> let downloadUrl(url:string) = async {              #1
      let req = HttpWebRequest.Create(url)
      let! resp = req.AsyncGetResponse()             #2
      let stream = resp.GetResponseStream()
      use reader = new StreamReader(stream)          #3
      return! reader.AsyncReadToEnd() }              #4
val downloadUrl : string -> Async<string>            #5
```

#A Namespace containing all the asynchronous functionality
#1 Using the 'async' computation builder
#2 Run operation asynchronously
#3 Dispose 'StreamReader' when completed
#4 Run asynchronously and then return the result
#5 Returns a non-standard computation type

After opening all the required namespaces, we define a function that is implemented using the asynchronous workflow. It uses the `async` value as a computation builder (#1). If you type "." immediately after the value in the Visual Studio, IntelliSense shows that it contains all the usual computation builder members such as `Bind` and `Return`, and also a couple of additional primitives that we'll need later. The printed type signature (#5) shows that the type of the computation is `Async<'a>`. We're going to look at this type in more detail in a moment.

The code in the listing above uses the `let!` construct once when executing a primitive asynchronous operation `AsyncGetResponse` (#2) provided by the F# library. The return type of this method is `Async<WebResponse>`, so the `let!` construct composes the two asynchronous operations and it binds the actual `WebResponse` value to the `resp` symbol. This means that we can work with the value once the asynchronous operation completes.

Next step we get the response stream and create a `StreamReader` object. The `use` primitive will make sure that the object will be disposed automatically when the asynchronous workflow completes.

On the last line, we're using a primitive that we haven't seen before: `return!` (#4). This allows us to run another asynchronous operation (just like the `let!` primitive) but returns the result of the operation when it completes rather than assigning it to some symbol. Just like the `do!` primitive, this is simply syntactic sugar. In particular, the computation builder doesn't have to implement any additional members; the compiler treats the code as if it were written like this:

```
let! text = reader.AsyncReadToEnd()
return text
```

Now that we have the `downloadUrl` function that creates the asynchronous computation, we should also look how we can use it to actually download the content of a web page. As you can see in the listing 13.2, we can use functions from the `Async` module to execute the workflow.

Listing 13.2 Executing asynchronous computations (F# interactive)

```
> let downloadTask = downloadUrl("http://www.manning.com");;     #1
val downloadTask : Async<string>

> Async.Run(downloadTask);;                                      #2
val it : string = ""

> let tasks =                                                    #3
     [ downloadUrl("http://www.tomasp.net");
       downloadUrl("http://www.manning.com") ]
val tasks : list<Async<string>>

> let all = Async.Parallel(tasks)                                #4
  Async.Run(all);;                                               #A
val all : Async<string[]>
val it : string[] = [ ""; "" ]
```

#1 Build the asynchronous workflow
#2 Run the workflow and wait for the result
#3 Create collection of workflows
#4 Join all workflows into one
#A Run the joined workflow

Code written using asynchronous workflows is delayed, which means that when we execute the `downloadUrl` function on the first line, it doesn't actually start downloading the web page yet (#1). The returned value (of type `Async<string>`) represents the computation that we want to run just like a function value represents some code that we can later execute. The `Async` module provides various ways of actually running the workflow, some of which are described in table 13.1.

Primitive	The type of the primitive and description
Run	`Async<'a> -> 'a` Runs the given workflow in the background. This operation blocks the caller thread and waits for the result of the workflow.
Spawn	`Async<unit> -> unit` Starts the given workflow in the background and returns immediately. The workflow executes in the parallel with the subsequent code of the caller. As indicated in the signature, the workflow cannot return a value.
SpawnFuture	`Async<'a> -> AsyncFuture<'a>` Starts executing the given workflow in the background and returns immediately. The result is an object with a property named `Value` that can be used at any later time to get the result of the computation. If the computation hasn't completed when the `Value` property is accessed, the calling thread will block.
Parallel	`seq<Async<'a>> -> Async<array<'a>>` Takes a collection of asynchronous workflows and returns a single workflow that executes all of the arguments in parallel. The returned workflow waits for all the operations to complete and then returns their results in a single array.

Table 13.1 Selected primitives for working with asynchronous workflows that are available in the 'Async' module in the standard F# library.

In listing 13.2 we initially use `Async.Run` (#2), which blocks the caller thread. This is useful for testing the workflow interactively. In the next step, we create a list of workflow values (#3). Again, nothing starts executing at this point. Once we have the collection, we can use the `Async.Parallel` function to build a single workflow that will execute all workflows in the list in parallel. This still doesn't execute any of the original workflows. To do that, we need to use the `Async.Run` primitive, which will start the composed workflow and wait for its result. This waits for the results of all the workflows in the list.

The code still waits for the overall result, but it runs very efficiently. It uses the .NET thread pool to balance the maximal number of running threads, so if we created hundreds of tasks, it wouldn't create hundreds of threads, because that would be inefficient, but instead use a smaller number (which may be around 20 threads). However, the number of tasks running in parallel can be significantly larger than 20. When the workflow reaches a primitive asynchronous operation called using the `let!` construct, it registers a callback in the system and releases the thread. This means that the thread can be reused for starting another asynchronous workflow.

In this chapter, we need to obtain the data interactively, so we're interested in running workflows in parallel rather than in developing responsive GUI applications. The latter class of applications (also called *reactive* applications) is important, and chapter 16 will focus solely on this topic.

Now we've seen what code *using* asynchronous workflows looks like, let's see how they're implemented.

13.1.2 Looking under the cover

In the previous chapter we saw that F# code written using a computation expression is translated into an expression that uses the primitives provided by the appropriate computation builder. For asynchronous workflows, this means that the `let!` construct is translated into a call to `async.Bind`, and `return` is translated into `async.Return`. In addition, asynchronous workflows are delayed. This means that the computation itself needs to be wrapped in an additional primitive to make sure that the whole code will be enclosed in a function. The function can then be executed later when we start the workflow. Listing 13.3 shows the translated version of listing 13.2.

Listing 13.3 Asynchronous workflow constructed explicitly (F#)

```
async.Delay(fun () ->                                        #1
   let req = HttpWebRequest.Create(url)                      #2
   async.Bind(req.AsyncGetResponse(), fun resp ->            #3
      let stream = resp.GetResponseStream()
      // ...                                                 #A
   )
)
```

#1 Create delayed workflow
#2 Standard value binding
#3 Customized asynchronous value binding
#A The rest of the code is omitted

The `Delay` member (#1) is one of the computation builder members that we can provide when implementing a computation expressions. In the example above, it takes a function that returns the asynchronous workflow (the type is `unit -> Async<'a>`) and returns a workflow value (`Async<'a>`) that wraps this function. Thanks to this primitive, the whole computation is enclosed inside a function and it isn't executed when we create the `Async<'a>` value. This is an important difference from the examples in the previous chapter such as the `option<'a>` type. An option represents a value, so the computation expression runs immediately, performing the computation and returning a new option value. On the other hand, the `Async<'a>` type represents a computation that can be executed later, so evaluating the `async` block just creates a workflow without executing it. It will become clearer what this means when we look at the `Async<'a>` type in detail.

The other primitive that occurs in the listing is the `Bind` member. As we learned in the previous chapter, this is crucial for all computation expressions. In asynchronous workflows, Bind allows us to start an operation without blocking the caller thread. The following list summarizes the steps that happen when we execute the workflow above using a primitive such as Async.Run:

- The function given as an argument to the Delay primitive (#1) starts executing. It synchronously creates the object that represents the HTTP request for the given URL (#2).
- AsyncGetResponse is called. The result is a primitive asynchronous workflow that knows how to start the request and call a specified function when the operation completes.
- We execute the Bind member and give it the workflow from step 2 as the first argument and a function that takes the HTTP response as an argument and should be executed when the workflow completes. This function is called a *continuation*, which is a term we've seen already in chapter 10.
- The Bind member runs the workflow created by AsyncGetResponse, passing it the specified continuation. The primitive workflow then calls the .NET BeginGetResponse method that instructs the system to start downloading the response and call the given continuation when the operation completes. At this point, the Bind member returns and the thread that was executing the operation is returned to the thread pool.
- When the response is ready, the system will call the continuation. The workflow gets the actual response object using the EndGetResponse .NET method and then executes the continuation given to the Bind member, which represents the rest of the computation. Note that the system again picks a thread from the tread pool, so the rest of the computation may be executed on a different thread each time we use the let! primitive.

The key point is that when we execute an asynchronous workflow, we don't wait for the result. Instead, we give it a continuation as an argument; this continuation will be executed when the corresponding step in the workflow has completed. The great thing about asynchronous workflows is that we don't have to write the code using continuations explicitly. The compiler translates let! primitives into the calls to the Bind member, creating the continuation automatically.

Investigating the asynchronous workflow type

You can use asynchronous workflows without understanding all the details, but you may be interested in a little bit of information about how they're implemented. We've seen that asynchronous workflows are similar to functions in that they represent a computation that we can execute later. If you look under the covers, you can see that the type is actually represented as a function in the F# library. The actual type is a bit more sophisticated, but the simplest asynchronous computation could be represented using the following:

```
type Async<'a> = (('a -> unit) * (exn -> unit)) -> unit
```

This is a function that takes two arguments as a tuple and returns a unit value. The two arguments are important, because they are continuations - functions that can be called when the asynchronous workflow completes. The first one is of type `'a -> unit`, which means that it takes the result of the workflow. This continuation will be called when the workflow completes. It can then run another workflow or any other code. The second continuation takes an `exn` value as an argument, which is the F# abbreviation for the .NET `Exception` type. and as you can guess, it is used when the operation that the workflow executes fails.

Even though the precise implementation details of asynchronous workflows aren't important, it's useful to be able to create your own primitive workflows - the equivalent of the `AsyncGetReponse` method used in listing 13.3. You can then use the rest of the building blocks to run your code asynchronously with the minimum of fuss.

13.1.3 Creating primitive workflows

The F# PowerPack library contains asynchronous versions for many of the important I/O operations, but it can't include all of them. It also provides methods for building your own primitive workflows. If the operation you want to run inside the workflow uses a standard .NET pattern and provides `BeginSomeOperation` and `EndSomeOperation` methods, then you can use `Async.BuildPrimitive` method. If you give it these two methods as an argument, it'll return an asynchronous workflow.

However, there are other operations that can be executed without blocking the thread. For example, we may want to wait for a particular event to occur and continue executing the workflow when it's triggered. Listing 13.4 creates a primitive that waits for the specified number of milliseconds using a timer and then resumes the workflow.

Listing 13.4 Implementing asynchronous waiting (F# interacitve)

```
> module Async =
     let Sleep(time) =                                                #1
        Async.Primitive(fun (cont, econt) ->
           let tmr = new System.Timers.Timer(time, AutoReset = false)  #A
           tmr.Elapsed.Add(fun _ -> cont())                            #2
           tmr.Start()
        );;
(...)

> Async.Run(async {
     printfn "Starting..."
     do! Async.Sleep(1000.0)                                          #3
     printfn "Finished!"
  });;
Starting...
```

```
Finished!
val it : unit = ()
```

#1 Primitive that delays the workflow
#A Initialize timer
#2 Run the rest of the computation
#3 Non-blocking waiting using a timer

This may look like a toy example, but we'll need the `Sleep` function later in the chapter. Of course, we could block the workflow using the .NET `Thread.Sleep` method, but there is an important difference. This method would block the thread, while our function creates a timer and returns the thread to the .NET thread pool. This means that when we use our primitive, the .NET runtime can execute workflows in parallel without any limitations.

Let's now look at the implementation. The `Sleep` function (#1) takes the number of milliseconds for which we want to delay processing and uses the `Async.Primitive` function to construct the workflow. It reflects the internal structure of the workflow quite closely. The argument is a lambda function that will be executed when the workflow starts. The lambda takes a tuple of two continuations as an argument. The first function should be called when the operation completes successfully, and the second should be called when the operation throws an exception. In the body of the lambda, we create a timer and specify the handler for its `Elapsed` event. The handler simply runs the success continuation (#2).

Having created our new primitive, the listing shows a simple snippet that uses it. Because it returns a unit value, we're using the `do!` primitive rather than `let!` (#3). When the code is executed, it constructs the timer with the handler and starts it. When the specified time elapses, the system takes an available thread from the thread pool and runs the event handler, which in turn executes the rest of the computation (in our case, printing to the screen).

Asynchronous workflows in C#

There have been various attempts to simplify asynchronous programming in C#, but none of the available libraries works quite as neatly as the asynchronous workflow syntax. The F# syntax is extremely simple from the end-user point of view (just wrap the code in an `async` block) which is quite difficult to achieve in C#.

We've seen that LINQ queries roughly correspond to F# computation expressions, so you might be tempted to implement `Select` and `SelectMany` operations. In principle, it would be possible to write asynchronous operations using query expressions, but the syntax we can use inside queries is very limited. Interestingly, C# iterators can be also used for this purpose and you can find more information about this in my article Asynchronous programming in C# using iterators [Petricek, 2007].

A more sophisticated technique, which is also based on C# iterators is available thanks to the Concurrency and Coordination Runtime (CCR). This library was developed as part of Microsoft Robotics studio, where responsiveness and asynchronous processing is essential for any application. You can find more information about this library in Jeffery Richter's Concurrency Affairs article [Richter, 2006].

That's all we need to know about asynchronous workflows for now–it's time to start using them for more practical purposes. In the next section, we'll look at the data services provided by the World Bank, and see how to obtain it with asynchronous workflows.

13.2 Connecting to the World Bank

It's no accident that the discussion about asynchronous workflows is located in a chapter about explorative programming. Many of the interesting data sources you'll work with today are available online in the form of a web service or other web based application. As we've seen, asynchronous workflows are the essential F# feature for obtaining the data.

However, downloading the data efficiently isn't our only problem. The data sources usually return the data in an untyped format (such as a plain text or XML without a precisely-defined schema), so we first need to understand the structure. Also, remote data sources can be unreliable, so we have to be able to recover from failure. This means that even before we write the code to obtain the data, we need to explore the data source. As we'll see, the F# interactive tools give us a great way for doing that.

13.2.1 Accessing the World Bank data

The data source we'll use in this chapter is the service provided by the World Bank. The World Bank is an international organization that provides funding and knowledge to the developing countries. As part of its job, the organization need to identify what type of support is the most efficient, where is it needed and evaluate whether it had an impact on the economy, quality of life or the environment of the developing country. The World Bank has a data set called "World Development Indicators" with information about many countries, and it makes the data available online. In this chapter, we'll work with information about the environment and more specifically about the area covered by forests and agricultural land. The data provided by the World Bank is available for free, but you need to register on the bank's web site first.

REGISTERING WITH THE WORLD BANK

Registration is performed on the http://developer.worldbank.org web site. Once you fill in the form and get the confirmation email, you can return to the web site and obtain an API key, which is used when sending requests to the World Bank services. The web site also contains documentation and a brief tutorial about the service. You can take a look at it there, but I'll explain everything we use in this chapter. One interesting feature on the web page is Query Generator, which allows you to run and configure queries interactively and shows the URL that we can use to request the data programatically.

The World Bank exposes the data using a simple HTTP based service, so we can use the `downloadUrl` function we created earlier. If you look at the documentation or experiment with the Query Generator for some time, you'll quickly learn the structure of the request URLs. The address always refers to the same page on the server and all the additional properties are specified in the URL as key-value pairs. In listing 13.5, we'll start by creating a function that constructs the request URL from an F# list containing the key-value pairs, so that we can access the data more easily.

Listing 13.5 Building the request URL (F#)

```
let worldBankKey = "xxxxxxxxxx"                                       #A

let worldBankUrl(props) =
   seq { yield "http://open.worldbank.org/rest.php?per_page=100"     #1
         yield "api_key=" + worldBankKey                              #1
         for key, value in props do
            yield key + "=" + value }                                 #2
   |> String.concat "&"
```

#A Specify your World Bank key here
#1 Same for all requests
#2 Additional properties specified by the user

The function body contains a sequence expression that generates a collection of strings. This collection is then concatenated using the "&" symbol as a separator. In the sequence expression, we first return the base part of the URL and the API key (#1), which is the part shared by all the requests we'll need in this chapter. We then iterate over all the key/value pairs specified as the `props` argument and return a `"key=value"` string (#2) that forms the part of the resulting URL.

In this chapter, we're creating an F# script file rather than a traditional application, so the next step is to write a couple of F# interactive commands that we can execute immediately to see whether the function we just wrote works correctly. This "test request" is also useful to see the data format used by the bank, so we know what we need to do later to parse the data.

The statistics provided by the World Bank are available for individual countries, but they can also be grouped based on region or income. These aggregated statistics make it easier to see overall trends. The first thing we need to do is to get the information about all the available groups. You can try this on the web site using the Query Generator. First select the "Countries" option in the "Country Calls" tab and enter your API key. To get a list of aggregated country groups, you can choose "Aggregates" from the "Region" list and then run the request. Listing 13.6 shows how to run the same request using F# interactive.

Listing 13.6 Testing the World Bank data service (F# interactive)

```
> let url = worldBankUrl ["method", "wb.countries.get";               #1
                          "region", "NA" ];;                          #1
val url : string =
   "http://open.worldbank.org/rest.php?per_page=100&
    api_key=xxxxxxxxxx&method=wb.countries.get&region=NA"
```

```
> Async.Run(downloadUrl(url));;                                          #2
val it : string = "<?xml version=\"1.0\" encoding=\"utf-8\" (...)"
```

#1 Build request URL with the specified properties
#2 Download the page as a string

We start by creating the URL using the function we just implemented (#1). We give it two additional parameters. The "method" parameter specifies which of the World Bank data sources we want to access and the "region" parameter specifies what types of countries we want to list. The "NA" value specifies that we're interested in the aggregated country information. As we're using F# interactive, we immediately see the composed URL. It contains all the specified parameters, the World Bank key and also a flag specifying that we want to return up to 100 of records per page. We'll talk about paging of the output later when we need to obtain larger number of indicators.

Once we have the URL, we can copy it into a web browser to see what data the World Bank returns. To download the page programmatically, we can use our `downloadUrl` function. As with any network operation, the download may fail for various reasons. This doesn't matter if we're running the request manually, but when we're executing a bulk operation to download data from various URLs in parallel, we need to write the code in a way it can recover from non-fatal failures.

13.2.2 Recovering from failures

The World Bank service only allows us to a limited number of requests each day for a single user key, and it also limits the frequency of requests. This means that if we a run large number of requests at once, some of them may return an error. The workaround is to catch the exception and retry the request later.

Listing 13.7 implements a loop that executes a request repeatedly until either it succeeds or we've tried 20 times. The failure is reported using exceptions and we're using the F# `try` … `with` construct to catch the exception.

Listing 13.7 Running the web request repeatedly (F# interactive)

```
> let worldBankDownload(props) =
     let url = worldBankUrl(props)                                      #A
     let rec loop(n) = async {                                          #1
        try
           return! downloadUrl(url)                                     #2
        with e when n < 20 ->                                           #3
           printfn "Failed, retrying (%d): %A" n props
           do! Async.Sleep(500.0)                                       #4
           return! loop(n+1) }                                          #5
   loop(1);;                                                            #B
val worldBankDownload : seq<string * string> -> Async<string>

> let props = ["method", "wb.countries.get"; "region", "NA"];;          #C
val props : list<string * string>

> Async.Run(worldBankDownload(props))
Failed, retrying (1): [("method", "wb.countries.get"); ("region", "NA")]
val it : string = "<?xml version=\"1.0\" encoding=\"utf-8\" (...)"
```

#A Construct the request URL
#1 Recursive asynchronous function
#2 Run the actual download asynchronously
#3 Catch exception when we want to retry
#4 Delay the workflow without blocking
#5 Recursively retry the request
#B Return the recursive workflow
#C Try the function interactively

The normal functional way to create a loop is to write a recursive function that takes the iteration number as an argument and increments this number on each iteration. Listing 13.7 this pattern with a twist. The `loop` function (#1) is implemented using an asynchronous workflow, so we're creating a recursive asynchronous workflow. The recursive call is in the exception handler (#4) and it uses the `return!` primitive to run the next iteration of the asynchronous loop. The body of the workflow attempts to download the page (#2), but it does this in a `try` … `with` block that catches possible exceptions.

The `try` … `with` block in F# is similar to the `try` … `catch` in C#, but it has some additional features. It allows us to distinguish between exceptions using pattern matching in a similar way we can use the `match` construct. In the example above, we're simply catching all exceptions, but in addition we've added a `when` clause (#3). This means that the exception will be caught only when the number of attempts is less than 20. Finally, it is worth noting that we're handling exceptions inside the asynchronous workflow in the same way you can handle exceptions in normal F# code. This is possible thanks to an additional primitive that the asynchronous workflow provides under the hood which tells F# how to deal with exceptions that occur during asynchronous operations.

On the last few lines of the listing, you can see how to use the function to get data from the World Bank. You can simulate a failure in the connection by disconnecting your computer from the network for a short time and you'll see that the code is able to recover from the failure. Now that we have a reliable function for downloading data, we can move forward and download all data we want to work with.

13.3 Exploring and obtaining the data

As we've seen in the last couple of examples, the World Bank data service returns the data as XML documents, so before we can write any code to process the data in a meaningful way, we'll need to convert it an F# type. In chapter 7, we converted between XML and our own custom record type, but in this case we're just going to use tuples and sequences. This is because the data structure will be quite simple and when we working with data interactively we need to modify the code quite frequently, either to tweak how we're using the existing values or to download different information. Tuples are more flexible for this task - we won't end up constantly renaming values.

We'll use LINQ to XML again, just as we did in chapter 7, but this time, we won't use the whole file. Instead we'll just pick out the nodes that are relevant. First we need a few helper functions.

13.3.1 Implementing XML helper functions

LINQ to XML is primarily designed for C# and VB, and working with it from F# can be a bit cumbersome. For example, F# doesn't support implicit type conversions (because it would complicate type inference), so every time we specify an element name, we have to use `XName.Get` instead of simply using a string.

However, we can easily implement a couple of F# functions to wrap the most commonly used parts of LINQ to XML and give us a very "F#-friendly" way to work with the data. As you can see in listing 13.8, most of the functions are very straightforward. The listing is created using F# interactive, so you can use the inferred type signatures to understand what a function does. One notable aspect is that each function takes the input element as its last argument, which means that we'll be able to compose the functions using the pipelining operator.

Listing 13.8 Helper functions for reading XML (F# interactive)

```
> #reference "System.Xml.dll"
  #reference "System.Xml.Linq.dll";;

> open System.Xml.Linq;;

> let xattr s (el:XElement) =                                 #A
     el.Attribute(XName.Get(s)).Value
  let xelem s (el:XContainer) =                               #B
     el.Element(XName.Get(s))
  let xvalue (el:XElement) =                                  #C
     el.Value
  let xelems s (el:XContainer) =                              #D
     el.Elements(XName.Get(s));;
val xattr : string -> XElement -> string
val xelem : string -> XContainer -> XElement
val xvalue : XElement -> string
val xelems : string -> XContainer -> seq<XElement>

> let xpath path (el:XContainer) =                            #1
     let res = path |> Seq.fold (fun xn s ->
        xn |> xelem s :> XContainer) el                       #E
     res :?> XElement
val xpath : seq<string> -> XContainer -> XElement
```

#A Returns value of the specified attribute
#B Returns child node with the specified name
#C Returns the text inside the node
#D Returns child elements with the specified name
#1 Return child node specified by a path
#E Move to the child element

The listing first references the necessary assemblies for LINQ to XML and opens the namespace containing classes such as `XElement`. The first group of functions are used to access child nodes, attributes or the value of any given element. It is worth noting that the `xelem` function takes `XContainer` as an argument, which means that we can use it for both ordinary elements, but also with an object that represents the whole document.

The last function (`xpath`) and is slightly more sophisticated (#1). It takes a sequence of names as an argument and follows this path to find a deeply nested element. It is implemented using `Seq.fold` and uses the input element as the initial state. The lambda function is executed for each name in the path. It finds a child of the current element with the specified name and returns it as a new child element. We want the type of the input to be `XContainer`, so the folding operation uses this type to represent the current state. As a result, we need to upcast the returned element to `XContainer` inside the lambda function and downcast the final result to `XElement`.

Equipped with these helper functions, we can easily extract all the information we want from the downloaded XML documents. If you're unsure about what any of the new functions do, don't worry: everything will become clearer once we start using them with real data.

13.3.2 Extracting region codes

The result of our download function is a string, so we need to parse this as an XML document. We'll need this operation frequently, so we'll write a simple wrapper function that downloads the data using `worldBankDownload` and returns the result as an `XDocument` object. The download executes asynchronously, so we'll implement the function using asynchronous workflows:

```
let worldBankRequest(props) = async {
   let! text = worldBankDownload(props)
   return XDocument.Parse(text) }
```

The code first invokes the asynchronous download using `let!` and when it completes, it parses the XML data and returns the `XDocument` object. Once we execute the download using `Async.Run`, we can query the returned XML document using the helper functions from the previous section. Listing 13.9 shows an example of this, downloading the aggregated information about countries and then accessing some values we'll need later.

Listing 13.9 Exploring the region information (F# interactive)

```
> let doc = Async.Run(worldBankRequest ["method", "wb.countries.get";
                                        "region", "NA"] );;
val doc : XDocument = ...

> let c = doc |> xpath [ "rsp"; "countries"; "country" ];;           #1
val c : XElement

> c |> xattr "id";;                                                    #2
val it : string = "EAP"

> c |> xelem "name" |> xvalue;;                                        #3
val it : string = "East Asia & Pacific"
```

#1 Select the first country element
#2 Read the value of the 'id' attribute
#3 Get the value of the 'name' child element

We start by accessing the first "country" element in the returned document. This element is a child element of the "countries" element, which is a child element of the root element named "rsp". To walk down the XML tree, we use the `xpath` function (#1) and specify the path to the element we want to select.

Now we can look at the content of the element to see what information we want to extract. We'll need the ID of the region, because this is used to identify it elsewhere. This is stored in the "`id`" attribute, so we can read it using the `xattr` function (#2). Finally, we'll also need the name of the region, so that we can display the data in a user-friendly format. This is the value of the "name" element (#3).

Now that we've explored the structure and made sure we know how to access all the region information we need for a single region, we can loop over all the regions. Listing 13.10 uses the same functions, but in a sequence computation.

Listing 13.10 Creating sequence with region information (F# interactive)

```
> let regions =
      seq { let countries = doc |> xpath [ "rsp"; "countries" ]
            for r in countries |> xelems "country" do                  #1
               yield r |> xattr "id",                                  #2
                     r |> xelem "name" |> xvalue }                     #2
val regions : seq<string * string> = seq
   [ ("EAP", "East Asia & Pacific"); ("ECA", "Europe & Central Asia");
     ("EMU", "European Monetary Union");
     ("HPC", "Heavily indebted poor countries (HIPC)"); ...]
```

#1 Read all child nodes
#2 Yield information about the region

The only important change from the previous listing is that we're now processing all the "country" nodes in the data. We access these elements as a sequence using the `xelems` function (#1), and then iterate over them using a `for` loop. As we're using a sequence expression, we can generate result elements using the `yield` keyword. We use the code that we tried in the previous listing to get the ID and the user-friendly name of the country, and return them as a tuple containing two strings (#2).

In this section, we've seen how to get a list of regions that we want to further study. The important aspect isn't the exact code we've used, but the general process. We created some helper functions to make data access easy, checked that we understood the document structure by fetching some information interactively, and then we wrapped the code inside a function. As a next step, we'll download the indicators that we want to show such as the area occupied by forests.

13.3.3 Obtaining the indicators

To obtain the data about countries or regions, we'll use a different method of the World Bank service. The method name is "wb.data.get" and you can find it in the Query Generator under the "Data Calls" tab. This allows us to request indicator data about a specific country for a given time period. Instead of downloading the data individually for each region that we're interested in, we'll fetch the information for all countries at once and then process them in memory. Even though we'll download more data in this way, we'll use a smaller number of requests, because we won't have to create request for every region.

We'll follow the same pattern as before, starting off by downloading a sample portion of the data and then examining it using our XML querying functions. Listing 13.11 shows how to download indicators specifying the proportion of a country covered by forests, as a percentage. The key for this indicator is "AG.LND.FRST.ZS" which is best discovered by simulating the query in the Query Generator. We'll download the data for 1990, requesting the first page of the data set.

Listing 13.11 Obtaining area covered by forests (F# interactive)

```
> let ind = "AG.LND.FRST.ZS"                                              #A
  let date = "1990"                                                       #A
  let page = 1                                                            #A
  let props =
     [ "method", "wb.data.get"; "date", date;                             #B
       "indicator", ind; "page", string(page) ];;                         #B
 (...)

> let doc = Async.Run(worldBankRequest(props))                            #1
  printfn "%s..." (doc.ToString().Substring(0, 285));;                    #1
val doc : XDocument
<data page="1" pages="3" per_page="100" total="227">
   <dataPoint>
      <country id="ABW">Aruba</country>
      <indicator id="AG.LND.FRST.ZS">Forest area (% ...</indicator>
      <date>1990</date>
      <value />
   </dataPoint>
   <dataPoint>...

> doc |> xpath [ "rsp"; "data" ] |> xattr "pages" |> int;;                #2
val it : int = 3

> doc |> xpath [ "rsp"; "data"; "dataPoint"; "country" ] |> xattr "id";;#3
val it : string = "Aruba"
```

#A First page of forest area data from 1990
#B Build arguments for the request
#1 Get the data and print a preview
#2 Read the total number of pages
#3 Read the ID of the first country

The listing first defines a couple of properties that we need to specify in order to create the request and creates a list with the properties that we need for the `worldBankRequest` function. After downloading the document, we want to explore its structure, so we convert it back into string and print out the first few lines (#1). The output shows us that the total data set has three pages. Information for each country is nested in "dataPoint" elements which contain the country name and ID, information about the data and the actual value. However, for the first country the value is missing, so we'll have to be careful and handle this case when parsing the data.

Next we'll write two simple expressions that we'll need very soon. First we need to read the number of pages (#2) so that we can download all the data. The next expression (#3) reads the ID attribute of the first country. This will be needed later, because we'll want to match it with the region ID that we collected in the previous section.

Now we have a pretty good idea about the structure of the data, we can write a function to download everything we need. Listing 13.12 shows an asynchronous which runs in a loop until it gets all the pages. We're not downloading pages in parallel, because that would be slightly harder to write, but we're going to run the same function in parallel for different indicators and years, so there will be enough parallelism in the end.

Listing 13.12 Downloading all indicator data asynchronously (F#)

```
let rec getIndicatorData(date, ind, page) = async {
   let! doc = worldBankRequest [ "method","wb.data.get"; "date",date;
                                 "indicator", ind; "page", string(page) ]
   let pages = doc |> xpath [ "rsp"; "data" ] |> xattr "pages" |> int   #1
   if (pages = page) then
      return [doc]                                                     #2
   else
      let! rest = getIndicatorData(date, ind, page + 1)                #3
      return doc::rest }
```

#1 Get the number of pages
#2 Data from the last page
#3 Download the remaining pages

The function takes the date, indicator and the required page number as parameters. We use them to build the list of arguments for the `worldBankRequest` function. When we receive the XML, we read the attribute that specifies the total number of pages of the data set (#1). If the page we're currently processing is the last one, we return a list containing only the current page (#2) as a single-element list. Otherwise, we need to download the remaining pages. Note that the function is declared with `let rec`, so we can invoke it recursively to get the remaining pages (#3). This is done using `let!` because we're inside an asynchronous workflow. Once we get the list of remaining pages, we just append the page we just downloaded and return all the pages as the result.

Before moving on, you can verify that this function works correctly using F# interactive. Make a request for indicator "AG.LND.FRST.ZS", year 1990 and page number 1. When you run the workflow using `Async.Run`, you should get three pages containing data about all the countries and regions.

Now let's introduce some parallelism, downloading all the indicators for all the years that we're interested in. We'll be using the `Async.Parallel` primitive, so we need to create a sequence of asynchronous workflows. The code in listing 13.13 does this using a simple sequence expression that calls the `getIndicatorData` function for all the combinations of parameters. Don't forget that just calling `getIndicatorData` doesn't actually perform the fetch - it just returns a workflow which *can* perform the fetch.

Listing 13.13 Downloading multiple indicators for multiple years in parallel (F#)

```
let downloadAll = seq {                                      #1
   for ind in [ "AG.SRF.TOTL.K2"; "AG.LND.FRST.ZS" ] do
```

```
          for year in [ "1990"; "2000"; "2005" ] do
            yield getIndicatorData(year, ind, 1) }

    let data = Async.Run(Async.Parallel(downloadAll))              #2
```

#1 Return workflow for each indicator and a year
#2 Run all workflows in parallel

The sequence expression first iterates over two indicators (#1). The first represents the total surface of the country or region in square kilometers and the second is the percentage of forest area, as we've already seen. If you look at the data on the web site, you can see that the forest area indicator is only available for three different years, so the nested loop iterates over these. For each combination of these parameters, we create (and yield) a workflow that runs the download starting from the first page.

This means that we'll get in total 6 tasks, each of which may download multiple pages. We combine the tasks into a workflow that returns an array of these 6 results and run the combined workflow using `Async.Run` (#2). The download can take some time and you may see that some of the requests failed and were restarted as we discussed earlier. The type of the `data` value that we get as a result is `array<list<XDocument>>`. The array contains a list of pages that were returned for each of the indicator-year combination.

Since we're writing an F# script, we don't have to worry about putting the settings such as years and indicators into a configuration file. We're writing the code only for a single purpose at the moment. Of course we can modify it later to be generally useful, but that would happen later in development. Now that we've retrieved the data, we need to do something useful with it.

13.4 Gathering information from the data

The amount of data that we can download from the internet is enormous, but the difficult part is gathering useful information from it. So far in this chapter, we have downloaded a list of regions and converted it into a sequence containing the name and ID of each region using normal F# types. Then we downloaded a bunch of XML documents that contain information about all regions and countries. In this section, we'll take this untyped XML data and convert it into a typed data structure that contains information we can easily display to the user.

13.4.1 Reading values

The first thing we need to do is to extract the data we're interested from the XML. We're going to write a function that takes a list of `XDocument` objects (one for each page of the data set) and returns a sequence where each element contains the value of the indicator, the ID of the region and the year in which the value was measured.

Listing 13.14 shows this in the form of the `readValues` function, as well as a helper function to data from an XML node representing a single record. Each function has a parameter named `format`, which is a function used to parse the actual value. We'll soon see the reason behind this parameter.

Listing 13.14 Reading values from the XML data (F#)

```
let readValue format node =
   let value = node |> xelem "value" |> xvalue                       #A
   let country = node |> xelem "country" |> xattr "id"               #A
   let year = node |> xelem "date" |> xvalue |> int                  #A
   if (value = "") then []                                           #1
   else [ (year, country), format(value) ]                           #1

let readValues format data = seq {
   for page in data do
      let root = page |> xpath [ "rsp"; "data" ]
      for node in root |> xelems "dataPoint" do                      #B
         yield! node |> readValue format }                           #2
```

#A Get the value, year and the country ID
#1 Value missing - return an empty list
#B Find all 'dataPoint' elements for all pages
#2 Parse the element and yield returned tuples

We start by writing the utility function that takes the formatting function and an XML node that contains a single data point. It reads values from child nodes and attributes, converting the year to an integer. If you look at the data we downloaded, you can see that the "value" element is sometimes empty. We handle this by returning an empty list if the value is missing and a list containing single element otherwise. Note that we could have used an option type instead, but a list makes the second function more elegant: we don't have to distinguish between the two cases; we simply return "all" the elements (either none or one) using the `yield!` primitive (#2).

The second function takes the entire input data as a sequence of `XDocument` objects. It finds all the XML elements containing data entries, formats them and returns a sequence. The type of the element in the returned sequence is `(int * string) * 'a`. The first tuple contains the year and the country ID. We'll use this as a key later on when searching for the data, which is why we're using a nested tuple. The second element is the value formatted using the `format` function, so the type will be the same as whatever the function returns.

As usual, we can try the function immediately. The key input for the function is the data source, which is written as the last argument so we can use the pipelining operator. The simplest formatter we can use (for test purposes) is one that just returns whatever string it's given, without really processing it at all. The following short snippet shows how to process the first data set, which contains the total surface area of all the countries in the year 1990.

```
> data.[0] |> readValues (fun s -> s)
val it : seq<(int * string) * string> =
   seq [ ((1990, "ABW"), "180"); ((1990, "ADO"), "470");
         ((1990, "AFG"), "652090"); ((1990, "AGO"), "1246700"); ...]
```

You can see we're getting closer to what we need: we can now read the data directly from the sequence. The only remaining irritation is that the values are clearly numbers, but we're treating them just as strings. Fortunately this is easy to fix…

13.4.2 Formatting data using units of measure

When reading the values of many of the indicators from the XML data, we *could* just convert them to `float` values. That would work, because both the surface area and forestation percentages are numbers, but it wouldn't tell us much about the data. The purpose of converting the data from untyped XML into a typed F# data structure is to annotate it with types that help us understand the meaning of the values. To make the type more specific, we can use *units of measure*, which we mentioned briefly in chapter 2. Using this feature we can specify that surface is measured in square kilometers and the area covered by forests in percentage of the total area. Let's start by looking at a couple of examples that introduce units of measure.

USING UNITS OF MEASURE

Working with units of measure in F# is very easy, which is why I've introduced them just as a brief digression in this chapter. We can declare a measure using the `type` keyword with a special attribute. Strictly speaking, a measure isn't a type, but we can use it as part of another type. Let's start by defining two simple measures to represent hours and kilometers:

```
[<Measure>] type km
[<Measure>] type h
```

As you can see, we're using the `Measure` attribute to specify that the type is a measure. This is a special attribute which the F# compiler understands. Instead of defining units ourselves, we could also use the standard set in the FSharp.PowerPack.dll library, but for now we'll use our own declarations. Now that we have units `km` and `h`, we can create values that represent kilometers or hours. Listing 13.15 shows how to create values with units and how to write a function that calculates with them.

Listing 13.15 Writing calculations using units of measure (F#)

```
> let length = 9.0<km>;;                                         #1
val length : float<km> = 9.0

> length * length                                                #2
val it : float<km^2> = 81.0

> let distanceInTwoHours(speed:float<km/h>) =                    #3
     speed * 2.0<h>;;
val distanceInTwoHours : float<km/h> -> float<km>                #4

> distanceInTwoHours(30.0<km/h>);;
val dist : float<km> = 60.0
```

#1 A constant representing distance
#2 We get an area by multiplying distances
#3 Parameter type uses a unit
#4 Shows the inferred unit of the return type

F# allows us to specify units for floating point values. This is done by appending a unit in angle brackets to the value (#1). We started by defining a value that represents a length in kilometers. If we write a calculation using value with units, F# automatically infers the units of the result, so we can see that multiplying two distances gives us an area in square kilometers (#2). When specifying units, we can use the conventional notation, so "^" represents a power, "/" is used for division and multiplication is written just by placing juxtaposing the units.

The next example shows that we can write functions with parameters that include information about the units. Our sample function takes a speed and returns the distance traveled in two hours (#3). We want the parameter to be specified in kilometers per hour, so we add a type annotation that contains the unit. This is written by placing the unit in angle braces in a same way as when specifying type arguments of a type such as `list<int>`. The F# compiler infers the return type for us (#4) just as it does when working with ordinary types. As usual, this provides a useful clue to understanding what a function does when you're reading it. It's also a valuable quick check to avoid making simple mistakes when writing the function–if we were trying to calculate a distance, but ended up with a return type using a unit of time, we'd know something was wrong.

In our World Bank data, we'll use the unit `km^2` to represent the total area of a country. So far, so good - but the second indicator that we obtained is provided as a percentage. How can we specify the unit of a percentage? Even though units of measure are primarily used to represent physical units, we can use them to represent percentages as well:

```
[<Measure>] type percent
let coef = 33.0<percent>
```

This code creates a unit for specifying that a number represents a percentage and then defines a constant `coef` which has a value 33%. Strictly speaking, the value in percents doesn't have a unit, because it is just a coefficient, but defining it as a unit is quite useful. To demonstrate this, let's try to calculate 33% of a 50 kilometer distance. Since `coef` represents a coefficient, we can simply multiply the two values:

```
> 50.0<km> * coef;;
val it : float<km percent> = 1650.0
```

This is obviously wrong. We want the result to be in kilometers, but if you look at the actual inferred type, you can see that the result is in kilometers multiplied by our new `percent` unit. Since we're running the code interactively, we can also see that the number is too high, but the great thing about units of measure is that we can spot the error during the type-checking without actually running the program. So what went wrong? The problem is that a percentage value represents a coefficient multiplied by 100. To write the calculation correctly, we need to divide the value by 100 percents:

```
> 50.0<km> * q / 100.0<percent>;;
val it : float<km> = 16.5
```

As you can see, this is much better. We divided the result by 100%, which means that we don't have the `percent` unit in the result. F# automatically simplifies the units and it knows that `km percent/percent` is the same thing as `km`. This example demonstrates a significant reason for using units of measure: just like other types, they help us to catch a large number of errors as early as possible.

UNITS OF MEASURE IN DETAIL

There are many other interesting aspects of units of measure that we haven't covered in this brief introduction. For example, you can define derived units such as `N` (representing a force in newtons), which is in fact just `kg m/s^2`. It is also possible to use units as generic type parameters in functions or types. For more information about units of measure, consult the F# online documentation and the blog of the feature's architect, Andrew Kennedy (http://blogs.msdn.com/andrewkennedy).

Let's get back to our main example and convert the data that we downloaded into a typed form that includes information about units. We'll use the `km` unit for representing the area and the `percent` unit for representing the portion of the area covered by forests.

Formatting the Word Bank data

When we declared the `readValues` function to read the values from XML documents, we included a formatting function as the final parameter. This is used to convert each data point into a value of the appropriate type. The array we downloaded contains three data sets of surface areas in square kilometers and three data sets of the forest area percentages. Listing 13.16 shows how we can turn the raw documents into a data structure from which we can easily extract the important information.

Listing 13.16 Converting raw data into a typed data structure (F#)

```
let floatInv(s) =                                                  #1
   let inv = System.Globalization.CultureInfo.InvariantCulture
   System.Double.Parse(s, inv)

let areas =
   Seq.concat(data.[0..2])                                         #2
      |> readValues (fun a -> floatInv(a) * 1.0<km^2>)             #3
      |> Map.of_seq                                                #4
let forests =
   Seq.concat(data.[3..5])                                         #5
      |> readValues (fun a -> floatInv(a) * 1.0<percent>)          #5
      |> Map.of_seq                                                #5
```

#1 Parse a float using the invariant culture
#2 Concatenate data for the first indicator
#3 Convert to square kilometers
#4 Return the data as a hashtable
#5 Create a hashtable storing the forested area in percents

The listing starts with a simple `floatInv` function (#1) that converts a string to the F# `float` type. This is similar to the built-in F# `float` function, with the difference that our function uses the invariant culture. The format used by the World Bank uses dot as a delimiter, so the number is for example "1.0", however the `float` function parses the data using the culture of the current thread. The code is more robust when we specify the culture we know to be appropriate for our data.

The main part of the data processing is written using pipelining. It uses a new feature that we haven't yet introduced to get the first three elements from the data set. This is called *slicing* and the syntax `data.[0..2]` gives us a sequence containing the array items with indices 0 to 2 (#2). We concatenate the returned sequence using `Seq.concat`, so we'll get a single sequence containing data for all the years. The next step in the pipeline is to read the values and convert them to the appropriate type using units of measure (#3). This turns out to be the easiest bit - just a simple lambda expression!

Finally, we use the `Map.of_seq` function to build an F# hashtable type from the data (#4). This function takes a sequence containing tuples and uses the first element as a key and the second element as the value. In the example above, the key has a type `int * string` and contains the year and the region ID. The value in the first case has a type `float<km^2>` and in the second case (#5) `float<percent>`. We've converted the data into a hashtable so that we can easily lookup the indicators for different years and regions.

13.4.3 Gathering statistics about regions

Our goal is to show how the forested area has changed in different regions since 1990. We'll need to iterate over all the regions that we have, test whether the data is available and find the value of the indicators we downloaded. This can be done quite easily using the hashtables we created, because they have the year and the region ID as the key.

We have to be slightly careful because some data may be missing, so we'll filter out any region for which we don't have data for all the years we're interested in. Also, we want to display the total area of forests rather than the percentage, so we need a simple calculation before returning the data. Even though it may sound difficult, the code isn't very complicated. The listing 13.18 shows the final few commands that we need to enter to the F# interactive to get the data we wanted to gather.

Listing 13.18 Calculating information about forested area (F# interactive)

```
> let calculateForests(area:float<km^2>, forest:float<percent>) =       #1
     area * forest / 100.0<percent>
val calculateForests : float<km ^ 2> * float<percent> -> float<km ^ 2>

> let years = [ 1990; 2000; 2005 ]
  let dataAvailable(key) =                                               #2
     years |> Seq.for_all (fun y ->
        (Map.mem (y, key) areas) && (Map.mem (y, key) forests))
val years : int list
val dataAvailable : string -> bool

> let getForestData(key) =                                               #3
     [| for y in years do
          yield calculateForests(areas.[y, key], forests.[y, key]) |]
val getForestData : string -> float<km ^ 2> array

> let stats = seq {                                                      #4
     for key, title in regions do
```

```
          if dataAvailable(key) then                                    #A
            yield title, getForestData(key) }                           #A
val stats : seq<string * float<km ^ 2> array>
```

#1 Calculate the total forest area
#2 Is the value available for the specified key?
#3 Get the forested area for each monitored year
#4 Iterate over all regions
#A Return title and the data if available

The listing defines a couple of helper functions that work with the data we downloaded and then defines a single value named `stats` that contains the final results. Thanks to units of measure, you can easily see what the first function (#1) does. It calculates the total area of forests in square kilometers from the total area of the region and the forested area in percentage.

The second function (#2) tests whether the data we need is available for the specified region ID for all the three years that we're interested in. It uses a function `Map.mem`, which tests whether an F# hashtable (specified as the second argument) contains the key given as the first argument. The last utility function (#3) looks similar to the second one. It assumes that the data is available and extracts them from the hashtables using the year and the region ID as the key for all the monitored years. It then calculates the forest area from the raw data using the first function.

Equipped with the last two functions, we can finally collect statistics for all the regions (#4). The returned value is a sequence of tuples containing the title of the region as the first element and an array as the second element. The array will always have three elements with the values for the three years that we're monitoring.

Once we get the data into F# interactive we can make observations about it, but it's difficult to see any patterns just by printing the data in the interactive window. To get the most from the data we gathered, we have to visualize them in a more user friendly way, such as using Microsoft Excel.

13.6 Visualizing data using Excel

F# gives us an almost unlimited number of ways to visualize the data. We can use the standard .NET libraries such as Windows Forms or WPF to create the visualization ourselves, we can implement a sophisticated visualization using DirectX or we could use one of the many third-party libraries available for .NET. In this chapter, we'll use a slightly different approach, presenting the data using Excel. As you'll see this is relatively easy to do, because Excel can be accessed using a .NET API. There are also many benefits of using Excel. Some operations are easier to do using a graphical user interface, so once we obtain the data, we can do the final processing in Excel. Also, Excel is widely used across the world, which makes it a useful distribution format.

The Excel API for .NET is exposed via the *Primary Interop Assemblies* (PIA) that are installed with Visual Studio 2008. They can be also obtained as a separate download, so if you run into any issues with them, you can find a link on the book's web site. Let's take our first steps into the world of the Office API…

13.6.1 Writing data to Excel

The Excel interop assemblies are standard .NET assemblies that we can reference from F# interactive using the `#reference` directive. Once we do this, we can use the classes to run Excel as a standalone (visible or invisible) application and script it. Listing 13.19 shows how to start Excel, create a new workbook with a single worksheet and then write some data to the worksheet.

Listing 13.19 Starting Excel and creating worksheet (F#)

```
#reference "office.dll"                                              #A
#reference "Microsoft.Office.Interop.Excel.dll"                      #A
open System
open Microsoft.Office.Interop.Excel

let app = new ApplicationClass(Visible = true)                       #1
let workbook = app.Workbooks.Add(XlWBATemplate.xlWBATWorksheet)      #2
let worksheet = (workbook.Worksheets.[1] :?> _Worksheet)             #3

worksheet.Range("C2").Value2 <- "1990"                               #4
worksheet.Range("C2", "E2").Value2 <- [| "1990"; "2000"; "2005" |]   #4
```

#A Reference the Excel interop assemblies
#1 Run Excel as a visible application
#2 Create new file using the default template
#3 Get the first worksheet
#4 Write values to the worksheet

After referencing the libraries that contain the Office and Excel .NET API and opening the necessary namespace, we create a new instance of the `ApplicationClass` (#1). This type comes from the Excel namespace and represents the application. After you run this line, a new Excel window should appear. The next line (#2) creates a workbook, so after running it, you should see the usual Excel grid. Next we fetch an object that represents the first sheet from the workbook (sheets are displayed at the bottom left of the application). As you can see, we need to cast the object to a `_Worksheet` class (#3), because the Excel API is weakly typed in many places. Once we get the worksheet, we can start writing data to the grid. This can be done using the `Range` indexer and the `Value2` property (#4). The type of this property is object, so we can use it in various ways. The first example writes a single string value to a single column and the second one fills a range (a single row containing three columns) with values from a .NET array.

So far we have created headers for the table we want to display, so the next step is to fill in all the remaining information and the most importantly, the matrix containing the forested area in different years. Listing 13.20 converts the data into a two dimensional array, which is also a valid data source for the `Value2` property.

Listing 13.20 Exporting data to Excel worksheet (F#)

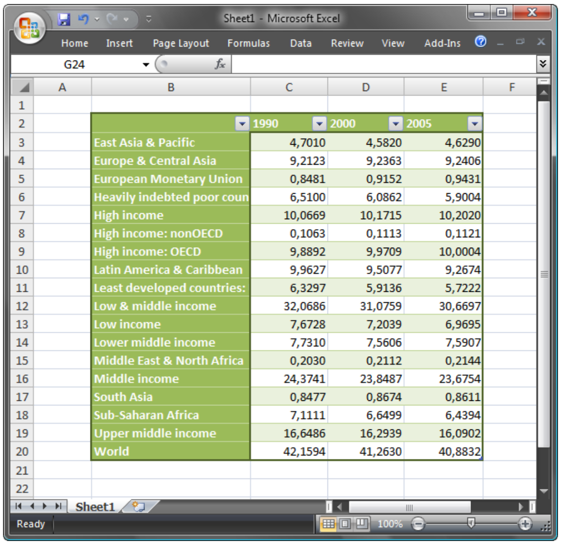
```
let names = statsArr |> Array.map fst
let namesVert = Array2.init names.Length 1 (fun i _ -> names.[i])    #1

let statsArr = stats |> Array.of_seq
let tableArr = Array2.init statsArr.Length 3 (fun i y ->             #2
   let _, values = statsArr.[i]                                      #A
   values.[y] / 1000000.0 )                                          #A

let slen = string(statsArr.Length + 1)
worksheet.Range("B3", "B" + slen).Value2 <- namesVert                #3
worksheet.Range("C3", "E" + slen).Value2 <- tableArr                 #3
```

#1 Get names of regions as 2D array
#2 Initialize 2D array with the data
#A Read value for a year 'y' from the i-th region
#3 Write the data to the worksheet

When writing data to Excel worksheets, we can use a primitive value, an array or a two-dimensional array. One dimensional arrays can be used for writing rows of data, as we saw in the first example, but if we want to fill a matrix or a column with data, we have to use a 2D array. In this listing, we start by creating a 2D array that stores the names of the regions vertically. To do this, we create a simple array containing the names and then use the `Array2.init` function (#1) to convert it to a 2D array. The resulting array contains only a single column, so we can ignore the second coordinate in the initialization.

The next step is to generate a 2D array with the data about the regions. We convert the input sequence into an array, so that we can index it when generating the 2D array using the `Array2.init` function (#2). In the lambda function, which is executed for every array cell, we first get the information about the region and then find the value for the specified year. Finally, we calculate the right ranges in the Excel worksheet (depending on the number of regions) and set the data using the same approach as in the previous example.

After running the code, the data should appear in the opened Excel. We can work with it at the same time as we execute our F# script, so if you tweak the design of the table we just generated, you could see something similar to the screenshot displayed in figure 13.1.



	1990	2000	2005
East Asia & Pacific	4,7010	4,5820	4,6290
Europe & Central Asia	9,2123	9,2363	9,2406
European Monetary Union	0,8481	0,9152	0,9431
Heavily indebted poor coun	6,5100	6,0862	5,9004
High income	10,0669	10,1715	10,2020
High income: nonOECD	0,1063	0,1113	0,1121
High income: OECD	9,8892	9,9709	10,0004
Latin America & Caribbean	9,9627	9,5077	9,2674
Least developed countries:	6,3297	5,9136	5,7222
Low & middle income	32,0686	31,0759	30,6697
Low income	7,6728	7,2039	6,9695
Lower middle income	7,7310	7,5606	7,5907
Middle East & North Africa	0,2030	0,2112	0,2144
Middle income	24,3741	23,8487	23,6754
South Asia	0,8477	0,8674	0,8611
Sub-Saharan Africa	7,1111	6,6499	6,4394
Upper middle income	16,6486	16,2939	16,0902
World	42,1594	41,2630	40,8832

Figure 13.1 Excel table generated by F# script showing the changes in the area covered by forests in various regions all over the world during the last 20 years.

Understanding the data is much easier now that we have it in Excel. However, we can take one additional step and create a chart with the data. Of course, you could do this by hand, but generating a complete Excel file that includes a chart is quite easy in F#.

13.6.2 Displaying data in an Excel chart

To create a chart, we need to specify quite a lot of properties. Fortunately the Excel API provides the `ChartWizard` method to make it easier. This method takes all the important attributes of the chart as optional parameters, so we can specify only those we actually need. The F# language supports optional parameters, so the code in listing 13.21 that creates the chart is very straightforward.

Listing 13.21 Generating Excel chart (F#)

```
let chartobjects = (worksheet.ChartObjects() :?> ChartObjects)        #1
let chartobject = chartobjects.Add(400.0, 20.0, 550.0, 350.0)         #1

chartobject.Chart.ChartWizard                                         #2
   (Title = "Area covered by forests",
    Source = worksheet.Range("B2", "E" + slen),
    Gallery = XlChartType.xl3DColumn, PlotBy = XlRowCol.xlColumns,
    SeriesLabels = 1, CategoryLabels = 1,
    CategoryTitle = "", ValueTitle = "Forests (mil km^2)")
chartobject.Chart.ChartStyle <- 5                                     #3
```

#1 Add new item to the charts collection
#2 Configure the chart using the wizard

#3 Set graphical style of the chart

First we need to create a new chart in the worksheet. This is done by adding a new element to the collection of charts. Again the weakly typed API means we have to cast it to the appropriate type (`ChartObjects`) before we can call the `Add` method (#1). This method gives us a new chart that we can configure using the `ChartWizard` method (#2). I mentioned that the method takes optional parameters, so the code uses the F# syntax to specify them. For each parameter that we want to set, we include the name of the parameter and the value. Most of the parameter names are self-explanatory, but it's worth noting that we specify the range including the text labels and then set `SeriesLabels` and `CategoryLabels` to 1, which tells Excel that the first row and column contain data labels.

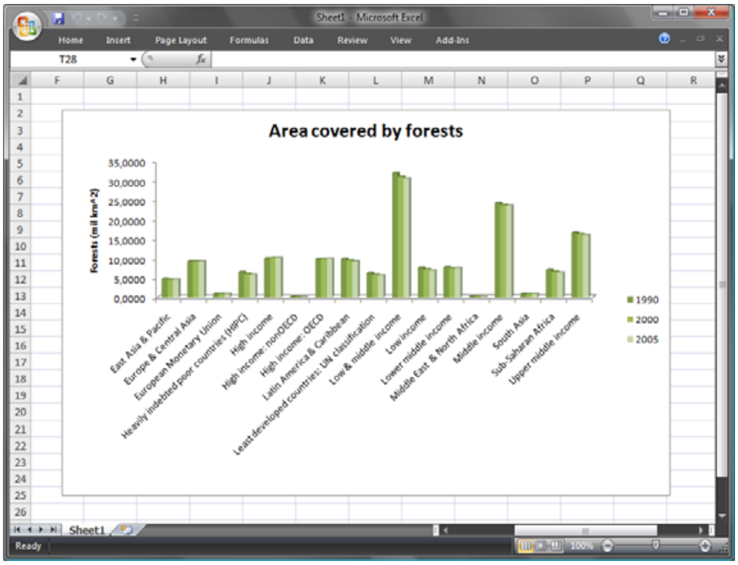


Figure 13.2 A chart generated from F# showing the changes in the forested area

The last line sets a `ChartStyle` property to specify the graphical style of the chart. Note that this property is available only in Office 2007, so if you're using older version of Excel, you'll have to remove this line. After you run the code, you should see a chart like the one displayed in figure 13.2.

The chart in Excel gives us a perfect way to understand and examine the data that we obtained from the World Bank. If you look at the chart carefully, you can see that the area covered by forests is very slightly increasing in Europe Central Asia and high income countries, but decreasing more significantly almost everywhere else in the world.

13.7 Summary

The "big picture" of this chapter was to demonstrate a typical lifecycle of *explorative programming* in F#. However, we've also introduced a couple of F# language and library features that are very important in other development processes.

We started by obtaining data from the web. To do this, we used asynchronous workflows, an F# computation expression implemented in the standard F# library. Asynchronous workflows can be used for efficiently implementing I/O and other time-consuming operations without blocking the caller thread and wasting resources. Once we downloaded the data, we used the LINQ to XML library to explore its structure before parsing it and converting it into a typed F# representation. All of this was done in an interactive fashion, often alternating between writing a couple of lines of code to try something with one piece of data, and then writing a function to apply the same logic to all the information we'd downloaded.

We used many advanced features such as sequence expressions when processing collections and we also used units of measure to specify the precise nature of the data. Finally, we looked at how to control Excel from the F# interactive shell. This shows a general principle that can be used when working with any Office application or with other applications that expose COM interfaces.

We'll return to F# asynchronous workflows when we talk about reactive programming in chapter 16. The next chapter is on closely related topic, and for many people it's the most convincing reason for looking at functional programming. We're going to look at parallelizing functional programs to get the most out of multi-core processors.

14
Writing parallel functional programs

We've already seen many arguments in favor of functional programming. One reason that is becoming increasing important in these days is parallelism. Writing code that scales to a large number of cores is much easier in the functional style than using the usual imperative approach.

The two concepts from the functional world that are essential for parallel computing are the declarative programming style and working with immutable data structures. These two are closely related. The code becomes more declarative when using immutable data, because the code is more concerned with the expected result of the computation than with the details of copying and changing data. However, both concepts are important in different ways when it comes to parallelization.

The declarative style allows most code that works with collections to be parallelized very simply, because the declarative style doesn't specify how the code actually runs. This means that we can replace the sequential implementation by a parallel implementation with a minimal effort. Immutable data structures are important for more fine grained parallelism. We'll see that we can parallelize any code that recursively processes immutable data structures using task-based parallelism. Finally, both C# and F# allow you to use mutable state. In chapter 10, we've seen that we can hide this mutable state and make the overall program functional. These hidden imperative islands are also good candidates for parallelization, for example when processing an array.

As you can see, there's a lot to explore. I'll start this chapter with a brief overview to demonstrate all of these techniques and explain when each of them is useful. After this introduction, we'll look at two more complex sample applications that show how parallel functional programming works on a larger scale. There isn't room to show all the code for two complete real-world examples in a single chapter, so we'll omit some of the less interesting details in the book. We'll focus particularly on the architectural aspects and areas directly related to parallelism. You can obtain the complete source code, which fills in the missing pieces, from the book's web site.

14.1 Understanding different parallelization techniques

In this section, we're going to briefly look at three different techniques and I'll use a simple example to demonstrate each of them. We're going to use Parallel Extensions to .NET, which is a library for parallel programming. It's part of the standard .NET 4.0 framework.

Parallel Extensions to .NET

The library consists of two key parts that we're going to use in this chapter. The first one is the Task Parallel Library (TPL). It includes underlying constructs that can execute *tasks* (primitive units of work) in parallel. Another component of TPL allows creating of tasks for common computations such as `for` loop. The second key part is Parallel LINQ (PLINQ), which can be used for writing *data parallel* code. This is code that processes a large amount of data using the same algorithm.

The underlying technology used to execute tasks in parallel is implemented in fully managed code and uses advanced techniques originating from Microsoft Research. It uses dynamic work distribution, which means that tasks are divided between worker threads depending on the availability of the threads. Once a thread completes all its own assigned tasks, it can start "stealing" tasks from other threads, so the work will be evenly distributed between all the available processors or cores. The tasks are stored in queues for each worker thread, which also minimizes the needed synchronization and locking when working with shared memory.

Let's start with a very specific technique I mentioned in the introduction: parallelizing imperative code that works with arrays. This isn't relevant for pure functional languages that don't allow any side effects, but as we saw in chapter 10, working with arrays in a functional style is a useful technique in C# and F#.

14.1.1 Parallelizing islands of imperative code

The most common construct in imperative programming that can easily be parallelized is the `for` loop. When the iterations of the loop are independent, we can execute them on separate threads. By "independent", I mean that no iteration can rely on a value computed by any earlier iterations.

For example, when summing the elements in an array, we need the sum of all the previous elements to calculate the next one. (This can be still parallelized, but not quite so simply.) However, consider the function for "blurring" an array, which we implemented in chapter 10. This is a good candidate for parallelization: even though each iteration uses multiple elements from the *input* array, it doesn't rely on anything in the *output* array. Listing 14.1 shows a simple for loop based on the earlier example, in both C# and F#.

Listing 14.1 For loop for calculating blurred array (C# and F#)

```
for(int i=1; i<inp.Length-1; i++){       for i in 1 .. inp.Length - 2 do
    var sum = inp[i-1] +                     let sum = inp.[i-1] +
        inp[i] + inp[i+1];                             inp.[i] + inp.[i+1]
    res[i] = sum / 3;                        res.[i] <- sum / 3
}
```

The code is almost trivial. The only thing that I'd like to remind you of is that even though this is imperative code, it can still be part of a pure functional program. The `inp` array is an input that isn't modified anywhere in our code and `res` is the output array, which shouldn't be modified after it is calculated by the loop.

To parallelize the loop, we can use `Parallel.For` method (where the `Parallel` class lives in the `System.Threading` namespace). This method takes an `Action<int>` delegate argument, which we'll supply using a lambda function.

Let's rewrite the code from listing 14.1 using this method. When we use `Parallel.For` from F#, we have to explicitly construct a delegate with a lambda function, whereas in C# lambda functions are automatically converted to delegates (or expression trees). This would make the F# version quite ugly, so we'll define a simple `pfor` function first:

```
let pfor nfrom nto f =
    Parallel.For(nfrom, nto + 1, Action<_>(f))
```

This simply wraps the function `f` (which has a type `int -> unit`) in a delegate type and runs the parallel for loop. Note that we also add 1 to the upper bound. This is because in `Parallel.For`, the upper bound is not included in the iteration, whereas in an ordinary F# loop it is. Listing 14.2 shows the parallelized versions of the previous example.

Listing 14.2 Parallelized for loop (C# and F#)

```
Parallel.For(1,inp.Length-1,i => {       pfor 1 (inp.Length-2) (fun i ->
    var sum = inp[i-1] +                     let sum = inp.[i-1] +
      inp[i] + inp[i+1];                               inp.[i] + inp.[i+1]
    res[i] = sum / 3;                        res.[i] <- sum / 3
});                                        )
```

As you can see, this is nearly as simple as the original sequential version. Again, this shows the power of functional constructs: thanks to lambda functions, the only thing you have to do when you want to convert a sequential `for` loop into a parallel one is to use the `Parallel.For` method (or `pfor` function in F#) instead of the built-in language construct.

THE PARALLEL CLASS

Aside from the `For` method, the `Parallel` class also contains `ForEach`, which can be used to parallelize the `foreach` construct in C# or the `for … in … do` construct in F#. Both of these methods have overloads available to let you customize the iteration. For example, there are overloads allowing you to change the step used to increment the index in the `For` method, or stop the parallel execution (similar to `break` in a C# loop). If you ever feel you need a little more control, consult the documentation to see if one of these overloads can help you.

The `Parallel.For` method is particularly useful when working with arrays. We'll use it in one of the larger sample applications later in this chapter, where we'll once again work with arrays in a functional way. First, let's finish our brief overview. The other two techniques we'll look at are purely functional.

14.1.2 Declarative data parallelism

The key idea behind the declarative style of programming is that the code doesn't specify how exactly it should be executed. Execution is provided by a minimal number of primitives such as `select` and `where` in LINQ or `map` and `filter` in F#, and these primitives can behave in a sophisticated way.

In the first chapter, I demonstrated how you can change an ordinary LINQ query into a query that runs in parallel using PLINQ. I showed this using C# query expressions, but to understand how it actually works, it's better to examine the translated version using method calls and lambda functions. I'll use a trivial example here, but we'll look at something more complicated later. Listing 14.3 counts the number of primes between 1 and 2 million. It shows the C# code using method calls, and also an F# version.

Listing 14.3 Counting the number of primes (C# and F#)

```
bool IsPrime(int n) {           #1      let isPrime(n) =                #2
  int max = (int)Math.Sqrt(n);            let ns = int(sqrt(float(n)))
  for (int i = 2; i <= max; i++)          let rec isPrimeUtil(m) =
    if (n % i == 0) return false;           (m >= ns || (n % m <> 0 &&
  return true;                                (isPrimeUtil(m+1))))
}                                         isPrimeUtil(2)

// Count the primes                     // Count the primes
var nums = Enumerable.Range             let nums = [1000000 .. 2000000]
  (1000000, 2000000);
var primeCount =                        let primeCount =
  nums.Where(IsPrime)           #3        nums |> List.filter isPrime   #4
      .Count();                 #3             |> List.length           #4
```

#1, #2 A number is prime if it can be divided without remainder only by 1 and itself. We test divisibility only by numbers from 2 to square root of the given number, because this is sufficient. In C#, the code is implemented using an imperative for-loop. In F#, we use a recursive function; thanks to tail-recursion, this is an efficient implementation.
#3, #4 To count the number of primes in the given range, we first select only numbers that are primes (using Where and filter respectively) and then count the returned numbers.

Code annotations below the code with bullets on the left

The listing starts with typical imperative and functional solutions for testing whether a number is a prime. I implemented them differently in order to use the most idiomatic code for each language.

The second part of the listing is more interesting. In C# (#3), we generate a range of integers (`nums`) of type `IEnumerable<int>`. LINQ provides us with extension methods `Where` and `Count` for this type, so we use these to calculate the result. In F#, we specify the functions explicitly. We're working with a list, so we implemented the code using functions from the `List` module.

Now let's modify the code to run in parallel. In C# this just means adding a call to the `AsParallel` extension method. In F#, we'll use functions that wrap calls to the .NET PLINQ classes similarly to the `pfor` function from the previous section. These functions are available in a module called `Parallel`.

Getting the Parallel module

The `Parallel` module contains just a collection of very simple wrappers. A module like this may eventually become part of the F# library, so I'm not going to show how to implement it. For now, you can download functions that we'll need from the book's web site: www.functional-programming.net/files/parallel.fs. To reference the file from the F# script, you can use the `#load` directive and specify the path of the "fs" file.

Listing 14.4 shows the parallelized queries in both C# and F#. The "prime testing" function hasn't been repeated, as it doesn't need to change.

Listing 14.4 Counting primes in parallel (C# and F#)

```
var primeCount =                       let primeCount =
  nums.AsParallel()        #1            nums |> Parallel.of_seq         #2
      .Where(IsPrime)      #3                 |> Parallel.filter isPrime  #4
      .Count();            #3                 |> Parallel.length          #4
```

#1, #3 In the C# version, we convert the IEnumerable<int> into a data structure that can be processed in parallel and then use parallel implementations of 'Where' and 'Count'.
#2, #4 In F#, we use an 'of_seq' function to create parallel sequence and then use parallel versions of the processing functions from the 'Parallel' module.

Code annotations below the code with bullets on the left

The F# sample is consistent with everything we've seen already. Functions for parallel data processing follow the same pattern as functions for working with lists and arrays. This means that we first have to convert the data to a parallel data structure using `Parallel.of_seq` (which is just like `Array.of_seq`) and then we can use various processing functions. The parallel data structure is just another type of sequence, so if we needed to, we could convert it back to a functional list using the `List.of_seq` function.

The C# version requires more careful examination–ironically, because it's changed less than the F# version. In chapter 12 we saw how to implement custom LINQ query operators and PLINQ uses a similar technique. The return type of the `AsParallel` method is `IParallelEnumerable<T>`. When the C# compiler searches for an appropriate `Where` method to call, it finds an extension method called `Where` that takes `IParallelEnumerable<T>` as its first argument, and it prefers this one to the more general method that takes `IEnumerable<T>`. However, the parallel `Where` method returns `IParallelEnumerable<T>` again, so the whole chain uses the methods provided by PLINQ.

Measuring the speedup in F# interactive

In chapter 10 we measured performance when discussing various functions for working with lists. To quickly compare the parallel and sequential version of the samples above, we can use F# interactive and the `#time` directive. Once we turn timing on, we can just select one of the versions and run it by hitting Alt+Enter:

```
> #time;;
> (...)                                     #A
Real: 00:00:01.606, CPU: 00:00:01.606
val it : int = 70501

> (...)                                     #B
val it : int = 70501
Real: 00:00:00.875, CPU: 00:00:01.700
```

#A Version using 'List' module functions
#B Version using 'Parallel' module functions

The "Real" time is the elapsed time of the operation, and as you can see, running the operation in parallel gives us a speedup of about 180%-185% on a dual-core machine. This is impressive when you bear in mind that the maximum theoretical speedup is 200%. The "CPU" time shows the total time spent executing the operation on all cores, which is why it's larger than the actual time in the second case.

Unfortunately, measuring the performance in C# isn't as easy, because we can't use any interactive tools. However, we'll write some utility functions to measure performance of the compiled code later in this chapter.

The last topic we'll look at in this introduction to declarative data parallelism is how to simplify the F# syntax. In chapter 12, we learned how to write sequence expressions to perform computations with numeric collections. Creating a computation expression to work with sequences in parallel is the natural next step.

PARALLEL SEQUENCE EXPRESSIONS IN F#

The nice thing about the C# version of the code was that switching between the sequential and parallel versions was just a matter of adding or removing the `ToParallel` call. In the

F# example, we explicitly used functions like `List.xyz` or `Parallel.xyz`, so the transition was less smooth.

However, if we rewrite the code using sequence expressions, then we can turn it into a parallel version by touching the keyboard exactly once. You can see both of the versions in listing 14.5.

Listing 14.5 Parallelizing sequence expressions (F#)

```
seq {                                #1      pseq {                               #2
  for n in 1000000 .. 2000000 do               for n in 1000000 .. 2000000 do
    if (isPrime n) then                          if (isPrime n) then
          yield n } |> Seq.length                      yield n } |>
                                             Parallel.length
```

#1 Sequence expression
#2 Parallel sequence expression

The parallel sequence expression is denoted by the `pseq` value, which is available in the `Parallel` module. It changes the meaning of the `for` operation inside the expression from a sequential version to a parallel one. The syntax is more flexible than in C#, because you can return multiple values using the `yield` and `yield!` keywords. On the other hand, the performance may be slightly lower when compared to a version that uses `Parallel.filter` and `Parallel.map` explicitly. Parallel sequence expressions are implemented using computation expressions that we discussed in chapter 12 and it is actually not very difficult to implement them once you can use the PLINQ library.

Parallelism using LINQ and computation expressions

In chapter 12 we learned how to implement our own set of LINQ operators and how to write computation expressions in F#. These two concepts are based on the same principles: we implemented a set of basic operators and the LINQ query or F# computation expression is then executed using these operators.

The PLINQ library implements virtually all operators supported by the C# query syntax including `Select`, `SelectMany`, `Where`, `OrderBy` and many others. So, what members have to be implemented in the `pseq` expression?

We saw most of the primitive operators in chapter 12. The expression uses `yield`, so we'll need a `Yield` member. It will simply return a sequence containing the single element that it gets as an argument. Since you can have multiple yields in the expression, we'll also need the `Combine` member, which will take two sequences and concatenate them into one. Finally, the `Zero` member (which allows us to write an `if` condition without an `else` branch) will return an empty sequence.

We also need to support the `for` construct. To allow this, we need to implement a `For` member, which takes an input sequence as the argument and a function that returns a sequence of values for every element from the input sequence. This is very similar to the

`Bind` member; however, the for-loop syntax is more readable than using the `let!` construct, so this can be written independently.

If you're interested in detailed information about the F# implementation of the "pseq" computation expression, you can read more on the book's web site.

Parallelizing declarative code that works with large amounts of data is perhaps the most appealing aspect of functional programming, because it's very easy and gives great results for large data sets. However, often we need to parallelize more complicated computations. In functional programming, these would be often written using immutable data structures and recursion, so we'll look at a more general technique in the next section.

14.1.2 Task-based parallelism

In chapter 11 we saw that you can very easily track dependencies between function calls in a functional program. The only thing that a function or a block of code can do is to take some values as arguments and produce a result. If we want to find out whether one call depends on some other call, we can just check whether it uses the output of the first call as part of its input. This is possible thanks to the use of immutable data structures. If the first call could modify shared state and the second call relied on this change, then we couldn't change the order of these calls, even though this wouldn't be obvious in the calling code. The fact that we can see dependencies between blocks of code is vital for *task-based parallelism*. We've seen data-based parallelism which performs the same task on different inputs in parallel; task-based parallelism performs *different* tasks concurrently.

Listing 14.6 shows an F# script which recursively processes an immutable data structure. Again, we'll look at a simple example here but show a more complicated scenario later. The code uses the binary tree type we designed in chapter 10, and implements a function to count the prime numbers in the tree.

Listing 14.6 Counting primes in a binary tree (F# interactive)

```
> type IntTree =                                          #A
      | Leaf of int                                       #B
      | Node of IntTree * IntTree                         #B
type IntTree = (...)

> let rnd = new Random()
  let rec tree(depth) =                                   #1
     if depth = 0 then Leaf(rnd.Next())
     else Node(tree(depth-1), tree(depth-1))              #C
val rnd : Random
val tree : int -> IntTree

> let rec count(tree) =                                   #2
    match tree with
    | Leaf(n) when isPrime(n) -> 1                        #D
    | Leaf(_) -> 0                                        #D
    | Node(l, r) -> count(l) + count(r)                   #3
val count : IntTree -> int
```

#A Binary tree type as in chapter 10
#B Either a leaf with value or a node containing sub-trees
#1 Generates random tree with the specified depth
#C Recursively generate sub-trees
#2 Count prime numbers in the tree
#D Return 1 when number is prime and 0 otherwise
#3 First process the left sub-tree, then the right one

The listing starts by declaring a binary tree data structure that can store values of type `int`. Then we implement a function (#1) that generates a tree containing randomly generated numbers. The function is recursive and takes the required depth of the tree as an argument. When generating a node that is composed of two sub-trees it recursively generates sub-trees with a depth decremented by 1.

Finally, we implement a `count` function (#2). It uses pattern matching to handle three different cases - when the tree is a leaf node with a prime value, it returns 1; when it's a leaf node with a non-prime value it returns 0; when it's a node with two sub-trees it recursively counts the primes in these sub-trees (#3). The important point to note is that the tasks of counting primes in the left and the right sub-tree are independent. In the next section, we'll see how to run these two calls in parallel.

TASK-BASED PARALLELISM IN F#

In the previous section, we were using the PLINQ component from the Parallel Extensions to .NET. To implement task-based parallelism, we'll use classes from the Task Parallel Library (TPL). This is a lower-level library that allows us to create tasks that will be executed in parallel by the .NET runtime. In this section, we'll work with a generic class `Future<T>`. Before we see how to use it, we need to extend it a little to make it easier to work with from F#. The class has a static member `Create` that takes a `Func<T>` argument, so we'll add a static member that takes an F# function, wraps it in a delegate and constructs the `Future<T>`:

```
open System.Threading.Tasks
module FutureExtensions =
   type System.Threading.Tasks.Future with          #A
      static member Create f =                      #A
         Future.Create(Func<_>(f))                  #A
```

#A Extend the 'Future' type with an F#-friendly 'Create' method

The syntax we're using here is the F# way of creating extension members. I say "extension members" rather than "extension methods" because F# allows you to extend an existing type with all kinds of members and not just instance methods. The syntax is similar to type augmentation, which we saw in chapter 9. However, there is an important difference. Extension members must be declared in a separate module. Also, the use of an extension member is translated into a static method call, just like in C#.

Now that we have extended the `Future<T>` type to make it easily useable from F#, we can finally parallelize our previous example. The most interesting part of listing 14.7 is the case when a tree is a node with two sub-trees that can be processed recursively:

Listing 14.7 Parallel processing of a binary tree (F#)

```
let pcount(tree) =
    let rec pcountDepth(tree, depth) =         #1
       match tree with
       | _ when depth >= 5 -> count(tree)      #2
       | Leaf(n) when isPrime(n) -> 1
       | Leaf(_) -> 0
       | Node(l, r) ->
          let cl = Future.Create(fun() ->      #3
             pcountDepth(l, depth+1))          #3
          let cr = pcountDepth(r, depth+1)     #4
          cl.Value + cr                        #5
   pcountDepth(tree, 0)
```

#1 Implementation function that also counts the depth
#2 Use sequential version for small sub-problems
#3 Create 'Future' to process the left sub-tree
#4 Process the right sub-tree
#5 Wait for both of the results and add them

We need to store an additional argument during the recursion, so we've create a local function called `pcountDepth` (#1). The additional argument (named `depth`) specifies the depth within the tree that we're currently processing. This allows us to use the non-parallel version of the function (`count`) after we've created a number of tasks that run in parallel. If we created a separate task for every tree node, then the overhead of creating new tasks would exceed the benefit we get from running the computations in parallel. Creating thousands of tasks on a dual-core machine doesn't look like a good idea. The overhead isn't as bad as creating an extra thread for each task, but it's still non-zero.

The `depth` argument is increased in every recursive call and once it exceeds a threshold, we simply process the rest of the tree using the sequential algorithm. In our example we test this with pattern matching (#2) and the threshold is set to 5 (which means that we'll create roughly 31 tasks).

When we process a non-leaf tree node, we create a value of type `Future<int>` and give it a function that processes the left sub-tree (#3). The `Future` type represents a computation that will start executing in parallel when it is created and will give us a result when we need it at some point in the future. It is worth noting that we don't create a future value for the other sub-tree (#4). If we did that, the caller thread would just have to wait to collect both results and wouldn't do any useful work. Instead we immediately start recursively processing the second sub-tree. Once we finish the recursive call, we need to sum the values from both sub-trees. To get the value computed by the future, we can use the `Value` property (#5). If the future hasn't completed yet, the call will block until the value is available. The execution pattern can be tricky to understand, but figure 14.1 shows it in a graphical way.

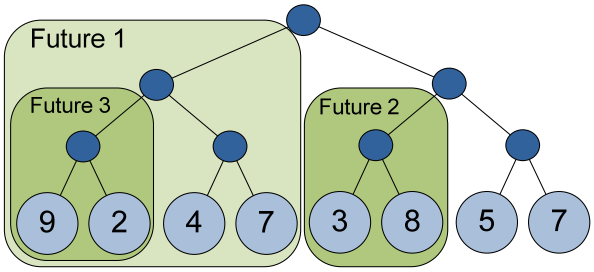


Figure 14.1 In the root node, we create "Future 1" to process the left sub-tree and immediately start processing the right sub-tree. This is repeated for both sub-trees and 2 more futures are created.

Just like with the data parallelization example, we're interested in the performance gains we each from task parallelization. Again we can measure the speedup easily using `#time` in F# interactive:

```
> let t = tree(15);;
> count(t);;
Real: 00:00:00.900, CPU: 00:00:00.889

> pcount(t);;
Real: 00:00:00.492, CPU: 00:00:00.920
```

As you can see, the statistics look very good. Like our previous example, the speedup is between 180% and 185%. One of the reasons we get such good results is that the tree was balanced; it had the same depth for all leaf node. If we didn't know in advance whether or not that was the case, it would have been wise to generate more tasks to make sure that the work would be evenly distributed among processors. In our example, we'd do that by increasing the threshold.

So far I've only shown code in F# for task-based parallelism, because implementing the binary tree is easier in F#, but of course it's feasible in C# too. Rather than showing all of the code here, we'll just look at the key parts of the C# version. The full code is available on the book's web site.

TASK BASED PARALLELISM IN C#

In C#, we'll first need to implement classes that represent the binary tree. I've implemented an `IntTree` class with two methods which allow us to test whether the tree is a leaf or a node:

```
bool TryLeaf(out int value);
bool TryNode(out IntTree left, out IntTree right);
```

These methods return true if the tree is a leaf or a node respectively. In that case, the method also returns details about the leaf or the node using out parameters. Listing 14.8 shows how to implement sequential and parallel versions of the tree processing code in C#.

Listing 14.8 Sequential and parallel tree processing using futures (C#)

```
static void Count(IntTree t) {
   int v;
   IntTree l, r;
   if (t.TryNode(out l, out r)
      return Count(l)+Count(r);#1
   else if (t.TryLeaf(out v))
      return IsPrime(v)?1:0;   #2
   throw new Exception();
}
```

```
void CountP(IntTree t, int d) {  #3
   int v;
   IntTree l, r;
   if (d > 4) return Count(t);   #4
   if (t.TryNode(out l, out r)) {
      var cl = Future.Create     #5
         (() => CountP(l, d+1)); #5
      var cr = CountP(r, d+1);   #6
      return cl.Value + cr;      #7
   } else if (t.TryLeaf(out v))
      return IsPrime(val)?1:0;
   throw new Exception();
}
```

#1, #2 For a node, the sequential version recursively processes the left and the right sub-tree (#1). When processing leaf, it tests whether the number is prime and returns 1 or 0.
#3, #4 In the parallel version, we have additional argument that represents the depth (#3). When the depth exceeds threshold, we calculate the result using sequential 'Count' method (#4)
#5, #6, #7 When processing the node in parallel, we create future to process the left sub-tree (#5) and process the right sub-tree immediately (#6). The program waits for both operations to finish and adds the results (#7).

Annotations below the code with bullets on the left

This is almost a literal translation of the F# code. The `Future<T>` type from the `System.Threading.Tasks` namespace can be used from both F# and C# in a similar fashion. The only important thing is that the computation that is performed by the future shouldn't have (or rely on) any side-effects. The `Future<T>` type is surprisingly similar to the `Lazy<T>` type that we implemented in chapter 11.

Future and lazy values

When talking about lazy values, I highlighted the fact that we can use them when we don't need to specify when exactly should be the value executed. This is exactly the case for future values as well. Both of them evaluate the function exactly once. A lazy value evaluates the result when it is needed for the first time where a future value performs the computation when a worker thread becomes available.

Another way to see the similarity between Future<T> and Lazy<T> is to look at the operations that we can do with them. When constructing a future or lazy value, we create them from a function that calculates the value. The F# type signature for this would be: `(unit -> 'a) -> T<'a>`, where T is either `Lazy` or `Future`. The second operation is to access the value. This simply takes a lazy or future value and gives us the result of the computation, so the type signature is `T<'a> -> 'a`.

In this section, we looked at the last of the three techniques for parallelizing functional programs that we're discussing in this chapter. Task-based parallelism is particularly useful when we're recursively processing large immutable data structures. This kind of computation is very common in functional programming, so task-based parallelism is a great addition to our toolset together with declarative data processing.

Now we're going to return to the first topic in more depth, parallelizing imperative code that is hidden from the outside world to keep the program functional. We'll demonstrate this using a larger application which applies graphical filters to images.

14.2 Running graphical effects in parallel

To demonstrate the first technique, we'll develop an application that needs to process large arrays in parallel. One of the simplest examples of large arrays is image data represented as a 2-dimensional array of colors. We used the same example in chapter 8 when we discussed behavior-centric applications, but this time we'll obviously be focusing on different aspects.

The user will be able to open an image, select one of the filters from a list and apply it to the image. First we'll develop a few filters, and then work out how to run a single effect on different parts of the image in parallel.

14.2.1 Calculating with colors in F#

In order to implement graphical effects such as blurring or grayscaling, we need perform calculations with colors. This could be done by working with the standard `Color` type in `System.Drawing`, but we'd have to treat the red, green and blue components separately, which isn't always convenient.

There's a more natural way to perform these calculations in both F# and C#. We can use operator overloading and implement our own color type. When we blur the image later, we'll be able to simply add colors together and divide the resulting color by the number of pixels. You probably already know how to do this in C#, but you can find an implementation in the downloadable source code. Listing 14.9 shows the F# version.

Listing 14.9 Implementing color type with operators (F#)

```
[<Struct>]                                                      #1
type SimpleColor(r, g, b) =
   member x.R = r
   member x.G = g
   member x.B = b
   member x.Validate() =                                        #A
      let check c = min 255 (max 0 c)
      SimpleColor(check r, check g, check b)
   static member (+) (c1:SimpleColor, c2:SimpleColor) =         #B
      SimpleColor(c1.R + c2.R, c1.G + c2.G, c1.B + c2.B)
   static member (*) (c1:SimpleColor, n) =                      #C
      SimpleColor(c1.R * n, c1.G * n, c1.B * n)
   static member DivideByInt (c1:SimpleColor, n) =              #D
      SimpleColor(c1.R / n, c1.G / n, c1.B / n)
   static member Zero = SimpleColor(0, 0, 0)                    #E
```

#1 Compile the type as value type
#A Create color with components in range 0-255
#B Component-wise addition of two colors

#C Multiply color components by an integer
#D Divide color components by an integer
#E Color initialized with zeros

The type is annotated using a .NET attribute named `Struct` (#1). This is a special attribute that instructs the F# compiler to compile the type as a value type; it corresponds to the C# `struct` keyword. In this example, it is very important to use value type, because we'll create an array of these values and allocating a new object on the heap for every pixel would be extremely inefficient.

Just like in C#, overloaded operators are implemented as static members of the type. We've already seen another way to implement operators in F# in chapter 6, where we declared them like functions using let bindings. Overloaded operators are more suitable if the operator is an intrinsic part of the type. For example the pipelining operator (|>) doesn't logically belong to any type, whereas our operators really are specific to `SimpleColor`. Also, some F# library functions can work with any types that provide some basic operators and members. That's also the reason why we called member that performs division by an integer `DivideByInt` and why we added member `Zero` that returns a black color. When a type has the plus operator and `Zero` member, it should be true that `clr = clr + T.Zero` for any `clr`. We can see that this is true for our type.

Another important aspect of the type is that it's immutable. None of the operations modify the existing value; instead they return a new color (even the instance member `Validate`). Even if you're not programming in a functional style, this is good practice when you write your own value types. Mutable value types can cause headaches in all kinds of subtle ways.

Now that we have a type to represent colors, let's look how to represent graphical filters and how to run them. We won't parallelize the operation yet - it's generally worth writing code which works correctly when run sequentially before trying to parallelize it, while bearing parallelization in mind, of course.

14.2.2 Implementing and running color filters

First we'll look at one special type of effect: color filters. Later, we'll extend the application to work with any effect, implementing blurring as an example. A color filter just changes the coloration of the image, so it's simpler. The filter simply calculates a new color for each pixel without accessing other parts of the image. As we saw in chapter 8, this is a behavior which is naturally represented as a function.

Filters for adjusting colors can be represented as a function that takes the original color and returns a new color. The F# type signature would be `SimpleColor -> SimpleColor`. In C# we can represent the same thing using `Func` delegate. The code that runs the filter will simply apply this function to every pixel of the image. When we process the bitmap we'll represent it as a two-dimensional array.

CONVERTING BITMAPS TO ARRAYS

The .NET representation for images is the Bitmap class from the System.Drawing namespace. This class allows us to access pixels using GetPixel and SetPixel, but these methods are very inefficient when you need to access lots of pixels–they're the graphical equivalent of reopening a file each time you want to read a byte of data. That's why we're going to represent the bitmap as a two-dimensional array instead.

However, we still need to convert the bitmap to an array and back. This can be done efficiently using LockBits method. This gives us a location in unmanaged memory that we can address directly. Writing and reading to the memory can then be done using the .NET Marshal class. In our application, we need two functions to do the conversion. These functions are implemented in the BitmapUtils module and are called ToArray2D and ToBitmap. While they're of some interest in themselves, they're not directly relevant to the topic of parallelization. You can find the full implementation in the online source code at www.functional-programming.net.

The implementation of the filters themselves will be similar in C# and F#, but the code to execute the filter sequentially will be different. The F# library has built-in support for using higher order functions with two-dimensional arrays, but .NET doesn't. Let's start off by implementing a couple of filters in C# though.

CREATING AND APPLYING COLOR FILTERS IN C#

Even though we're going to represent color filters using the Func delegate, however we'll implement them as ordinary methods that we can convert to delegates when we need to, such as to store them in a collection of filters. Listing 14.10 shows two simple color filters. The first converts the color to grayscale and the second lightens the image.

Listing 14.10 Grayscale and lighten filters (C#)

```
class Filters {
   public static SimpleColor Grayscale(SimpleColor clr) {
      var c = (clr.R*11 + clr.G*59 + clr.B*30) / 100;         #1
      return new SimpleColor(c, c, c);                        #1
   }
   public static SimpleColor Lighten(SimpleColor clr) {
      return (clr * 2).Validate();                            #2
   }
}
```

#1 Calculates weighted average from the color components
#2 Lighten the color an make sure it is valid

To calculate the grayscale color, we use a weighted average (#1) because the human eye is more sensitive to green light than to red or blue. The implementation of the second filter is even simpler, but this time it uses the overloaded operators of the SimpleColor type. It uses component-wise multiplication to multiply the color by two. This may create colors with components outside the normal range of 0-255, so we use the Validate method (#2) to limit each component appropriately.

Now that we've got our filter methods, let's apply them to the two-dimensional array representation of an image. Listing 14.11 does this by implementing an extension method on the array type itself. At the moment we're still performing all the computation in a single thread.

Listing 14.11 Sequential method for applying filters (C#)

```
public static SimpleColor[,] RunFilter
      (this SimpleColor[,] arr, Func<SimpleColor, SimpleColor> f) {
   int wid = arr.GetLength(0), hgt = arr.GetLength(1);
   var res = new SimpleColor[wid, hgt];                            #A
   for(int x = 0; x < wid; x++)                                    #B
      for(int y = 0; y < hgt; y++)                                 #B
         res[x, y] = f(arr[x, y]);                                 #B
   return res;
}
```

#A Create a new array as the result
#B Calculate new color for every pixel

The `RunFilter` method first creates a new array that will be returned as a result. We're writing the application in a functional way, so the method will not modify the array given as the input. In the body of the method, we imperatively iterate over all the pixels in the array and apply the color filter function to every pixel.

Given our earlier experience with `Parallel.For` you can probably already see how to parallelize this code. Before we get onto that though, we'll just finish up the single-threaded version by looking at the F# code.

CREATING AND APPLYING COLOR FILTERS IN F#

In chapter 10, when we wanted to apply a function to all elements of an array and collect the results in a new array, we used the `Array.map` function. This is exactly what our method `RunFilter` from the previous listing did, with the exception that it worked on two-dimensional arrays. It may not surprise you that F# library contains a module `Array2` for working with 2D arrays, which is very similar to the one-dimensional `Array` module. This module also contains a `map` function, which makes the F# implementation of `runFilter` trivial. You can see it together with the two color filters in listing 14.12.

Listing 14.11 Applying filters and two simple filters (F#)

```
let runFilter f arr = Array2.map f arr                 #1

module ColorFilters =                                  #2
   let Grayscale(clr:SimpleColor) =
      let c = (clr.R*11 + clr.G*59 + clr.B*30) / 100   #A
      SimpleColor(c, c, c)
   let Lighten(clr:SimpleColor) =
      (clr * 2).Validate()                             #B
```

#1 Apply 'f' to all elements of the 2D array
#2 Filters are encapsulated in a module
#A Grayscale using weighted average
#B Calculate lighter color

The `runFilter` function simply calls `Array2.map` to do the work (#1) and in fact we could just use `Array2.map` in our later code. However, wrapping it into another function makes the code more readable and self-explanatory. Also, if we eventually decided to change the representation of the image, we could just update the `runFilter` function without touching the code that uses it.

We also use F# modules to organize the code in a more structured fashion. All the graphical filters are encapsulated in a module called `ColorFilters`. The implementation of our two sample filters is almost the same as in C#, but we'll see later that F# allows us to do a little more with custom types that provide standard overloaded operators.

Before we look at how to parallelize the application, we need to wrap all the code we've written so far into an application that we can actually run. This will allow us to test our filters and also measure the performance. We'll do this in the next section. I'll only show you the C# version, and not in very much detail, but the full source code is available online. We'll focus on the interesting bits.

14.2.3 Designing the main application

So far, we've only created color filters, but we want our final application to cater for more general graphical effects. A color filter such as grayscaling or lightening applies a function to each pixel based only on that pixel's value. Other effects may be much more general - they could do anything with the image, such as geometrical transformations or blurring. We'll use blurring as an example later on, just to show that it's possible. We will take this goal into account as we build the application.

The application allows you to open an image file, select an effect from a list, and apply it to the image. You have the option to apply the filter in a sequential or parallel fashion, and it automatically displays how long the filter took to apply. You can see a screenshot of the finished application in figure 14.2.

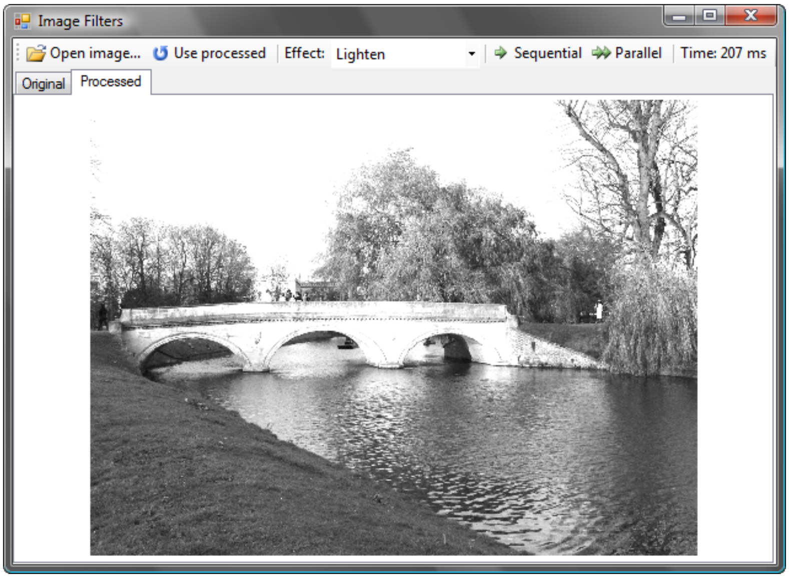


Figure 14.2 C# version of the image processing application after first running the grayscale and then the lighten filter on a sample image.

In C#, we can create the user interface for the application using the Windows Forms designer. The application uses the `ToolStrip` control to create the toolbar with the necessary commands, and a `ToolStripComboBox` control for the list of available effects. A `PictureBox` control wrapped in a `TabControl` shows the image, so we can easily switch between the original image and the processed version.

Creating windows applications in F#

Unfortunately the F# support in Visual Studio doesn't include a Windows Forms designer. We've seen how it's easy to create simple GUIs by hand in F#, but for this kind of application a designer would be useful. Fortunately F# can easily reference C# libraries and vice versa, so there are several options available to us.

If you only need to create forms, you can create a C# class library project that contains the graphical elements such as forms and user controls, then reference the library from your F# application and use the GUI components from F#. This is the approach I used to create the F# version of this application, so you can see exactly how it works if you download the source code.

An alternative approach is to implement the user interaction in C# and reference an F# library that contains all the data processing code. If we wanted to use this approach, we'd wrap the graphical effects in an F# object type (as discussed in chapter 9) and compile the F# code as a library. The C# application would then use the types in the library to run the graphical effects.

Once we've create the GUI in the designer, we can use the filtered we've already implemented. As I said earlier, the application will be flexible enough to work with general effects beyond just filters, so let's look at how we want to represent these effects in code.

REPRESENTING EFFECTS

A color filter is a function that took a color and returned the new color. Effects can be represented as functions too, but as they can do anything with the image the functions need to take an image as input and return the new image. We'll provide an argument to say whether or not the effect should run in parallel, so that we can measure the performance. Different effects may be parallelized in different ways. In the C# GUI application, we also need to store the name of the effect. Listing 14.12 shows all of this information wrapped up into an `EffectInfo` type.

Listing 14.12 Representation of graphical effect (C#)

```
class EffectInfo {
   public Func<SimpleColor[,], bool, SimpleColor[,]>     #1
      Effect { get; set; }                               #1
   public string Name { get; set; }                      #A
}
```

#1 Function representing the effect
#A Name of the effect

The class is very simple, with just two properties. We've created it in the most straightforward way possible, with mutable properties. We're only going to use this type within the GUI itself, so while that may leave us feeling a little uncomfortable, we won't worry about it too much. The first property of the class is a function that runs the effect (#1) and the second is a name. This is very similar to an F# record containing a function and a string; that's the design we'll use in the F# version of the application. Next we'll look at how we can create `EffectInfo` instances to represent the color filters we implemented earlier.

14.2.4 Creating and running effects

In the earlier section, we implemented a couple of graphical filters, but our application contains a list of more general graphical effects. It appears that we still have a lot of work to do, but actually, we already have everything we need to create graphical effect from a simple color filter. Everything should start making sense in the next section where we'll look how to create an effect from a color filter.

CREATING EFFECTS FROM COLOR FILTERS IN C#

To create a general effect based on a color filter, we can simply apply the filter to all the pixels in the image. We've already implemented the sequential form of this as the `RunFilter` method. For the moment, we'll also assume that we've already implemented the parallel version in a method called `RunFilterParallel`. We'll look at the implementation of that in a minute.

Listing 14.13 shows how to create an effect from a graphical filter. Since an effect is represented as a function, we use a lambda function to return a delegate from the method.

Listing 14.13 Creating graphical effect from a color filter (C#)

```
Func<SimpleColor[,], bool, SimpleColor[,]> MakeEffect
      (Func<SimpleColor, SimpleColor> clrFunc) {                  #1
   return (arr, parallel) => {                                     #2
      if (!parallel) return Filters.RunFilter(arr, clrFunc);       #3
      else return Filters.RunFilterParallel(arr, clrFunc);         #3
   };
}
```

#1 Takes color filter as an argument
#2 Return lambda function
#3 Apply color filter to each pixel

The method has only a single argument which is the color filter that we want to convert into an effect (#1). The effect should apply this filter to each pixel of an image, how can we do this when we don't have the image yet? The answer is that we'll get the image later as an argument for the function that represents the effect, so the body of the method just returns the effect via a lambda function (#2).

The lambda function takes two arguments - the image to process and a flag specifying whether the code should run in parallel. Once we have this information, we can call `RunFilter` or `RunFilterParallel` depending on the Boolean argument. If you remember our discussion about closures in chapter 8, you can see that the `clrFunc` argument to the method will be captured by a closure that is associated with the returned function.

It's important to note that that the return type is exactly the same as the type of the function stored in `EffectInfo`, so we can use it immediately when we're building our drop-down list of effects for the toolbar. Here's an example of how to create effects from our two existing color filters and then add them to the `listFilter` control:

```
var effects =
   new List<EffectInfo> {
      new EffectInfo { Name = "Grayscale",
                       Effect = MakeEffect(Filters.Grayscale) }, #A
      new EffectInfo { Name = "Lighten",
                       Effect = MakeEffect(Filters.Lighten) }    #B
   };
listFilters.ComboBox.DataSource = effects;                      #C
listFilters.ComboBox.DisplayMember = "Name";                    #C
```

#A Convert grayscale to effect
#B Create effect from lighten filter
#C Show the list in a drop-down list

We're using a C# 3.0 collection initializer to create a `List<EffectInfo>` containing information about the two color filters that we created so far. When we call `MakeEffect`, we give it a method group from the `Filters` class as an argument. The method group is automatically converted into a `Func` delegate by the C# compiler. The last two lines set the list as the data source for the drop-down control and use the `DisplayMember` property to specify that the text in the drop-down should be the name of the effect.

The corresponding code in the F# version of the application is quite interesting, so even though we won't look at the full source code, we'll discuss this part.

USING PARTIAL FUNCTION APPLICATION IN F#

In F#, we don't have to write this conversion as a method (or a function) that explicitly returns another function, because we can use partial function application. The first function in listing 14.14 takes all the arguments it needs to actually apply the color filter, so it doesn't make sense to call it `MakeEffect`. However, if we specify only the first argument it will return a function that represents the effect.

Listing 14.14 Creating effects using partial function application

```
> let runColorFilter clrFunc (arr, isParallel) =                 #1
      if (not isParallel) then arr |> runFilter clrFunc           #A
      else arr |> runFilterParallel clrFunc                       #A
val runColorFilter : (simpleColor -> SimpleColor) ->
                     (SimpleColor[,] * bool) -> SimpleColor[,]    #2

> let effect = runColorFilter ColorFilters.Grayscale              #3
val effect : (SimpleColor[,] * bool) -> SimpleColor[,]
```

#1 All arguments needed for applying the filter
#A Apply the filter
#2 Representation of graphical effect
#3 Create effect using partial application

It's not obvious at first, but this really is the same function as the `MakeEffect` method, but with the goal of making it useable with partial function application. To achieve this, we specify all the arguments in the function declaration (#1). The second parameter specifies arguments that will be specified later when we're asked to run the actual effect; as the two pieces of information will be always provided together, we've used a tuple. The body of the function is easier to write than it is in C#, because we don't have to explicitly create and return a function.

If we take a look at the type signature, we can see that the first argument (on the first line) is the color filter and when we look at the second line (#2) we can see the representation used for graphical effects. This means that we can create effect just by specifying the first argument, which is exactly what we at the end of the listing (#3).

Now let's get back to the user interface, and look at what the event handlers for the "Parallel" and "Sequential" buttons have to do.

EXECUTING GRAPHICAL EFFECTS

When we apply the effect, we need to measure the time it takes. We could remember the time before running the effect, then run the effect and finally subtract the original time from the current time. However, this mixes the calling aspect with the timing aspect. If we wanted to measure the time in various different places, we'd have to copy and paste the code, which isn't a good practice. Functional programming gives us a better way to approach the problem.

We can implement time measurement as a higher order function, taking another function as an argument and measuring the time taken to run it. The return value is a tuple containing the result of the function and the elapsed time in milliseconds. Listing 14.15 shows this implemented in both F# and C#.

Listing 14.15 Measuring the time in F# and C#

```
open System.Diagnostics;                       using System.Diagnostics;

let measureTime(f) =                           Tuple<T, long> MeasureTime<T>
   let st = new Stopwatch()                          (Func<T> f) {
   st.Start()                                     var st = new.Stopwatch();
   let res = f()                                  st.Start();
   let t = st.ElapsedMilliseconds                 var res = f();
   (res, t)                                       var t = st.ElapsedMilliseconds;
                                                  return Tuple.New(res, t);
                                               }
```

The function first initializes `Stopwatch` class to measure the time and then runs the specified function. We don't want to throw away the result, so we store it locally and then count the elapsed time. Since we need to return multiple values from the function, we use a tuple value. The first element of the tuple is the result of the function we passed in, which can be any type depending on the function. The second element will contain the time taken in milliseconds.

Listing 14.16 uses this new method in the event handler for the Click event of the "Parallel" and "Sequential" buttons.

Listing 14.16 Applying the selected effect to a bitmap (C#)

```
var filter = ((EffectInfo)listFilters.SelectedItem).Filter;         #A
var arr = loadedBitmap.ToArray2D();
var res = MeasureTime(() =>
   filter(arr, sender == btnParallel));                             #B

pictProcessed.Image = res.First.ToBitmap();                         #C
lblTime.Text = string.Format("Time: {0} ms", res.Second);           #C
```

#A Get the selected filter
#B Run filter and measure performance
#C Display result and time taken

The available effects are in the drop-down list, stored as `EffectInfo` instances, so we start by accessing the selected item from the list. Once we have the effect, we can perform the bitmap processing. We first convert the bitmap to a 2D array and then apply the filter. The first argument to the filter is the array and the second one is a Boolean specifying whether or not the effect should run in parallel, based on which button was clicked. The operation is wrapped in a call to the `MeasureTime` method, so the type of `res` is

`Tuple<SimpleColor[,], long>`. We first convert the returned array into a bitmap, display it, and then show the time taken to apply the effect.

We're currently concerned only with the performance of the effect itself, but it would be possible to parallelize the conversion between a bitmap and an array as well. I'll leave that as an exercise if you're interested, but for the moment let's get on with parallelizing the effect.

14.2.5 Parallelizing the application

As this chapter is really about parallelization, this is the most interesting part of the application. We're going to discuss the code in both languages, starting off by implementing the C# version in the simplest way possible.

Running filters in parallel in C#

To implement the C# version, we'll simply take the `RunFilter` method from listing 14.11 and replace for loop with a call to the `Parallel.For` method. You can see the modified version in listing 14.17. Thanks to lambda functions in C# 3.0, this is just a syntactic transformation.

Listing 14.17 Applying color filter in parallel (C#)

```
public static SimpleColor[,] RunFilterParallel
      (this SimpleColor[,] arr, Func<SimpleColor, SimpleColor> f) {
   int wid = arr.GetLength(0), hgt = arr.GetLength(1);
   var res = new SimpleColor[wid, hgt];
   Parallel.For(0, wid, x => {                                   #1
      for(int y = 0; y < hgt; y++)                               #2
         res[x, y] = f(arr[x, y]);
   });                                                           #A
   return res;
}
```

#1 Parallelize the outer loop
#2 Leave inner loop sequential
#A End of lambda function and method call

The original code contained two nested for loops, but we're only parallelizing the outer loop. For most images this will give the underlying library enough flexibility to parallelize the code efficiently, without creating an unnecessarily large number of tasks. Making the filter run in parallel only involved changing two lines of code. However, changing for-loops to `Parallel.For` method calls isn't always as simple as it looks. You always have to look carefully at the code and consider whether parallelization could introduce any problems.

For instance, we have to be careful if the loop modifies any mutable state, or when state is shared by several iterations of the for loop executing in parallel. In the previous example, we avoided this problem by using only local mutable state. The `res` array cannot be accessed from outside of this function, which makes the overall method functional. Also, each iteration only uses a separate part of the array (a single vertical line).

Additionally, many .NET types aren't thread-safe, which means that when you start accessing a single instance from several threads, their behavior may be undefined. In section 14.3, we'll see that this is a problem even for simple types such as `Random` and we'll also see how to solve this problem by using locks. First, let's look at the F# version of the previous code.

Parallel array processing in F#

The source code for the F# version will be almost a direct translation of what we've seen in the previous C# listing–but at the same time, it'll be a much more general function. If you reimplemented the previous C# listing in F#, one of the changes you'd probably make would be to delete all the unnecessary type annotations. After doing that, you'd see that the code doesn't explicitly mention the `SimpleColor` type anywhere and it doesn't really need to know that it is working with colors. If you hover over the function translated from C# in Visual Studio, you'd see the following inferred type:

```
(('a -> 'b) -> 'a [,] -> 'b [,])
```

Just by deleting type annotations, we've made the function more generic. The type of the function is actually the same as the type of `Array2.map`, which we used earlier in this chapter. The change in type signature also suggests that the name should be generalized too - after all, we're really performing a mapping operation, just in parallel. The result of these changes is shown in listing 14.18.

Listing 14.18 Parallel 'map' function for 2D array (F#)

```
module Array2 =
   module Parallel =                                                 #1
      let map f (arr:_ [,]) =                                        #A
         let wid, hgt = arr.GetLength(0), arr.GetLength(1)
         let res = Array2.create wid hgt SimpleColor.Zero            #B
         pfor 0 (wid-1) (fun x ->                                    #2
            for y = 0 to hgt - 1 do
               res.[x, y] <- f(arr.[x, y]))
         res

let runFilterParallel f arr  = Array2.Parallel.map f arr            #3
```

#1 Declare function in a module
#A Apply 'f' to all elements in parallel
#B Create new array as a result
#2 Parallelized outer loop
#3 Expose function with a domain-specific alias

The fact that the simple act of translation has revealed a deeper aspect of our original code is quite a strange phenomenon. The new function does exactly the same thing as `Array2.map`, but executed in parallel, so we've named it function `map` and placed it inside a module `Array2.Parallel` (#1) to make it more reusable. To implement the parallelization, we're using our utility function `pfor` from earlier section.

After we've noticed this generalization in F#, we could change the C# version to match it, changing the method declaration to something like this:

```
public static TResult[,] RunFilterParallel<TSource, TResult>
```

```
        (this TSource[,] arr, Func<TSource, TResult> f) {
```

We'd then have to propagate the type parameters appropriately through the code, substituting `SimpleColor` for either `TSource` or `TResult` depending on the context. Type inference would then take care of providing the type arguments where we call the method.

The final line of listing 14.18 creates alias for the parallel `map` function (#3). Our original goal was to write a function to run a graphical filter in parallel, and this alias makes the code more readable, because the name provides a better clue as to how the function can be used.

Now that we've parallelized simple color filters, we're just going to implement a single more general effect: blurring the image. This will wrap up our coverage of the application, although of course you may want to experiment with some more effects yourself.

14.2.6 Implementing a blur effect

Our final effect won't be just a color filter. The process of blurring an image relies on computing a new pixel value based on *multiple* original pixels. We can still perform a pixel-by-pixel transformation, but we'll have to pass the transformation the whole image, as well as the coordinates of the pixel we want to transform.

I've left the implementation of `RunEffect` and `RunEffectParallel` as an exercise for you, but it's fairly straightforward; really it's just a matter of changing the details of the loop, and giving the transformation function more information. Converting the sequential form into a parallel form is exactly the same for this effect as for color filter effects. If you get stuck, you can always look at the full source code from the web site.

The blurring transformation itself is quite interesting though, as shown in listing 14.19. It's not particularly difficult, but it does provide a nice demonstration of declarative programming.

Listing 14.19 Implementing 'Blur' effect (F#)

```
let Blur(arr:SimpleColor[,], x, y) =                                   #1
   let wid, hgt = arr.GetLength(0)-1, arr.GetLength(1)-1
   let checkW x = max 0 (min wid x)                                    #A
   let checkH y = max 0 (min hgt y)                                    #A
   [ for dx in -2 .. 2 do                                              #B
        for dy in -2 .. 2 do                                           #B
           yield arr.[checkW(x + dx), checkH(y + dy)] ]                #B
   |> List.average                                                     #2
```

#1 Gets the image and required X, Y coordinates
#A Check that index is in range
#B Collect all close pixel colors
#2 Calculate average color

If you were implementing blur in an imperative style, you'd create a mutable variable, initialize it to zero and then add the colors of all the nearby pixels. Finally, you'd divide the result by the number of pixels to get the average color. In F#, we can use a more declarative approach and we can just write that we want to calculate the average color (#2). We first use a sequence expression to create a list containing colors of all the nearby pixels and then calculate the average value from this list.

EXPLORING LIST.AVERAGE

To calculate the average value, the `List.average` function needs to know how to do three things. It needs to know what a "zero" value is for the particular type, and how to add values together. This is enough to sum the list. Then it needs to divide the result by the number of elements in the list. In our effect, we're working with values of the `SimpleColor` type and this type implements overloaded plus operator. We also added a special members `Zero` and `DivideByInt`. The `average` function uses exactly these members. It's a generic function, but it requires the type to implement the appropriate members. You can find more information about implementing functions like this online on the book website.

In this example, we've looked at key parts of a larger application. You can get the complete source code from the book's web site and see how the parts we implemented in this chapter are connected together. Even though the most important parts of the application use mutable arrays, we've designed the whole application in a functional way, including using the arrays in the functional style as shown in chapter 11. This approach allowed us to parallelize the core algorithms easily and safely.

This has been an example of a behavior-centric application. Our main concern was how to parallelize individual behaviors. Another way to parallelize a behavior-centric application is to run different behaviors in parallel. For example, we might want to process a series of images in a batch, applying multiple effects to each image. In the next section, we're going to turn our attention to data-centric applications.

14.3 Creating a parallel simulation

Our next sample application is going to be a simulation of a world containing animals and their predators. The goal of predators is to move close to animals to hunt them and the aim of animals is to run away within the area of the world. Just like our image processing example, we'll only show the most interesting aspects here, but the full source code is available online.

This is a data-centric application, so the first task is to identify the primary data structure involved. In this case, it's the representation of the "current" state of the world. The world effectively has a single operation: make time "tick", moving all the animals and predators. When we parallelize a data-centric application, our aim is to parallelize the operations that can be performed with the data structure, so we're going to perform a single step of the world in parallel. To do this, we'll use the normal techniques involved in data-centric functional programs, with a combination of declarative and task-based parallelism.

Before we look at the world representation, we'll digress slightly and discuss ways of accessing objects that are not thread safe. This can be problem when we're dealing with objects with mutable state, which is often the case with .NET types.

14.3.1 Accessing shared objects safely

The `Random` class is a commonly used .NET class which is not thread-safe. In our application, we're going to need to generate random locations to choose where the animal or predator should move to, this can be called from several threads simultaneously. However, `Random` needs to be initialized once and then used again and again. (If you create a new `Random` instance for each call, you'll often get repeated numbers as the initial "seed" for the random number generator is taken from the system time.) If you call the `Next` method on the same instance from multiple threads, it will eventually start returning zero, because we're responsible for making sure that only a single thread will access the object at a time.

To avoid this problem we can use locking, which blocks other threads from executing code guarded by the same lock until the operation completes. However, this makes the code less efficient. Listing 14.20 provides a solution which is safe but allows us to be efficient as well.

Listing 14.20 Safe way for generating random numbers (F# and C#)

```
module SafeRandom =                      static class SafeRandom {
   let private rnd =           #1           static Random rnd =          #1
      new Random()                             new Random();
                                            public static Random New() { #2
   let New() =                 #2              lock (rnd)                #3
      lock rnd (fun () ->      #3                 return new             #3
         new Random(rnd.Next())#3                    Random(rnd.Next()); #3
      )                        #3           }
                                         }
```

#1, #3 Local random number generator (#1) is accessed safely from multiple threads using locks (#3)
#2 A method or function returns a new random number generator initialized with a random seed that should be accessed only from a single thread

Annotations below the code with cueballs on the left

We created a module in F# and a static class in C#, both of which serve the same purpose: they can be used for generating random number generators. These generators are created using a random seed that is obtained from a single global random number generator (#1) that is safely accessed within a lock (#3). Thanks to this approach, we don't have to use lock every time we'll need to create a random number. We only need to create a new generator every time we execute an operation that can be executed in parallel with other tasks. Within a single thread, we can reuse the same instance safely, knowing that no other thread will have access to it.

The F# equivalent of C#'s `lock` keyword F# is the `lock` function that takes a simple function as its argument. It acquires the lock using the `Monitor` class, runs the specified function and then releases the lock.

We'll make good use of `SafeRandom` when we determine the locations of animals and predators, but first let's look at the representation of the world.

14.3.2 Representing the simulated world

Our simulated world is quite simple. It only contains animals and predators, so we can represent it using two lists. In principle, we should also include the width and height of the world area, but we'll used a fixed size to make things simpler. You can get a better picture about the world we're trying to represent by looking at figure 14.3, which shows a screenshot of the running simulation.

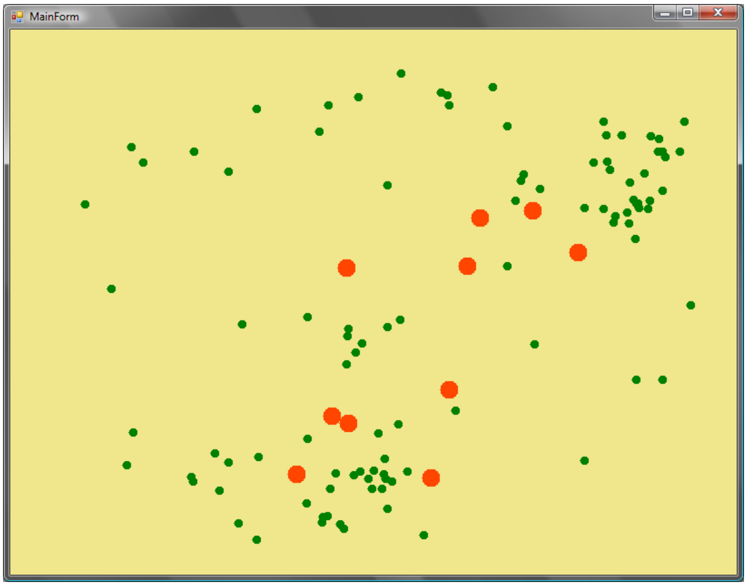


Figure 14.3. Running simulation with 10 predators (larger circles) hunting 100 animals (small circles)

We'll look at the interesting elements of the simulation in both languages, just like we did for the image application. We'll start with the F# version to show a typical functional approach.

REPRESENTING THE SIMULATION STATE IN F#

As I've already mentioned, the state of the simulation will be just two lists with the locations of animals and predators. To make it easier to work with locations, we'll also implement our own Location type that has a couple of overloaded operators for adding and subtracting locations as well as for multiplying both X and Y coordinates by a floating point number.

We'll implement location as a simple immutable value type and the state of the simulation will be an F# record type with two fields. You can see the data structure declaration in the listing 14.21.

Listing 14.21 Representing the state of the world (F#)

```
[<Struct>]                                                          #1
type Location(x:float, y:float) =
   member t.X = x
   member t.Y = y
   static member (+) (l1:Location, l2:Location) =
      Location(l1.X + l2.X, l1.Y + l2.Y)                            #A
   static member (*) (l:Location, f) =
      Location(l.X * f, l.Y * f)                                    #B
   static member (-) (l1:Location, l2:Location) =
      l1 + (l2 * -1.0)                                              #C

type Simulation =                                                   #2
 { Animals : list<Location>
   Predators : list<Location> }
```

#1 Value type representing location
#A Add X and Y coordinates
#B Multiply by coefficient of type float
#C Implemented using + and *
#2 Represents the simulation state

The location is a simple object marked using the Struct attribute (#1). It only contains the X and Y coordinates as immutable properties, set in the constructor. All the operators return new values, as you'd expect.

The type that represents the simulation is also straightforward (#2). This type is immutable too, so in order to work with it, we'll need to construct a new Simulation value for each step of the simulation.

Now let's look at our C# representation. We're mostly going to use standard .NET types, but we'll work with them in a functional way.

REPRESENTING SIMULATION STATE IN C#

In C#, the simplest approach is to represent some of the state using mutable types, because that's what the C# language and the standard .NET libraries provide the most support for. In particular, .NET doesn't provide a functional list type. We could have used our FuncList<T> type from earlier chapters, which would have made the two representations very similar. However, functional programming is a style and not a technology, so we can write functional code even with the classes that we already have; we'll just have to be more careful to do it correctly.

Listing 14.22 shows the class we're going to use to represent the simulation in C#. I've omitted the implementation of the Location type because it's just a simple immutable struct with overloaded operators, exactly the same as the F# version.

Listing 14.22 Representing the state of the world (C#)

```
public class Simulation {
   public Simulation(List<Location> animals, List<Location> predators) {#1
      Animals = animals;
      Predators = predators;
   }
   private readonly IEnumerable<Location> animals;
   private readonly IEnumerable<Location> predators;

   public IEnumerable<Location> Animals { get { return animals; } }     #2
   public IEnumerable<Location> Predators { get { return predators; } } #2
}
```

#1 Create a new simulation state
#2 Expose properties as an immutable sequence

We use two different collection types here; one for the constructor arguments (#1) when we're creating the simulation state and a different one for the properties (#2). In the constructor, we use List<T>, to make sure that we get a fully-evaluated collection that contains all the locations. Since IEnumerable<T> is a lazy sequence, we wouldn't know if the locations were evaluated already or whether they'll be evaluated later when we'll actually need them somewhere later in the code.

On the other hand, we don't want to expose the state as List<T>, because that's a mutable type and someone could modify it. Instead, we use IEnumerable<T> so client code can iterate over the animals and predators, but can't modify the existing state. As you can see, we use properties with private setter (#2). This is not exactly what we mean - the field where the value is stored should be marked as readonly, which means that it can only be set in the constructor. However, we used private setter for simplicity and we'll just remember for now that the field shouldn't be set anywhere else in the class.

Now that we have the data structures to represent the state, we should also look at what we can do with it. In a typical functional design, that's always the next thing to do.

14.3.3 Designing simulation operations

In this section we'll think about the operations that we need to implement for the simulation. We won't implement all the difficult operations now, because we just want to design the structure of the application. Our first goal is to get the application running with minimal effort and then we can get back to the interesting parts, such as the algorithms describing the movements of animals and predators.

In a typical functional fashion, we'll start with some initial state and in each step we'll create a new state based on the previous one. This means that we'll need an operation to create an initial state, and another to run a single step of the simulation. Both of these are logically related to the simulation state, so in C# we'll add them to the Simulation class.

In F#, we'll add them to the `Simulation` type using type augmentations, which we discussed in chapter 9. The following snippet shows the types of these operations using C# syntax:

```
class Simulation {
   public static Simulation InitialState { get; }   #A
   public Simulation Step();                        #B
}
```

#A Generates the initial state
#B Performs simulation step and returns a new state

If you're writing the sample code as you read the book, you can implement these on your own in some very simple way. For now, the `Step` method can just return the original state or it can move all the animals by one pixel in some direction, just so that we can tell that the simulation is actually running. The `InitialState` property should generate a couple of randomly located animals and predators. We'll get back to these methods after we finish implementing the machinery that runs the simulation.

Next we need the ability to draw the simulation state. In C#, we'll make this part of the `MainForm` class. In F#, the form is a global value, so we can implement the drawing code as a simple function. The operation will simply iterate over all the animals and predators in the current simulation state and draw them on the form using `System.Drawing` classes. The C# method has this signature:

```
void DrawState(Simulation state)
```

I won't present the full code here, but now you'll recognize it when we call it.

At this point, we have everything we need to run the simulation, even though we haven't yet implemented any interesting algorithms for animal and predator movement. Let's put everything we've got together, so we can test it before we start making the animals behave more intelligently.

RUNNING THE SIMULATION

We'll run the simulation as fast as the computer is able to, so we'll implement it as a loop that runs the `Step` method, redraws the form and then starts again until the form is closed. We don't want to block the main application thread, because that would make the application unresponsive, so we'll run the simulation as a background process. The F# and C# versions are implemented in rather different ways, so we'll look at both of them. Listing 14.23 shows the C# code, which explicitly creates a thread.

Listing 14.23 Running simulation on a thread (C#)

```
private void MainForm_Load(object sender, EventArgs e) {
   Thread th = new Thread(() => {
      Simulation state = Simulation.InitialState;              #1
      while(this.Visible) {
         this.Invoke(new Action(() => DrawState(state)));      #2
         state = state.Step();                                 #A
      }
   });
   th.Start();                                                 #B
}
```

#1 Mutable variable that holds the current state
#2 Redraw the form
#A Calculate the new state
#B Start the simulation loop thread

This method is the only part of the C# version of the simulation where we need to use mutable state. In particular, we create a variable that holds the current state of the simulation (#1). We run the simulation on a thread in a `while` loop and we redraw the form, calculate the new state and store this state in the same local variable for each iteration. In C#, we can't write the code without mutation, but we won't need this in F#.

Another notable point is the way we update the form (#2). In Windows Forms, it is only allowed to access controls from the main GUI thread. We use the `Invoke` method that takes a delegate and runs in on the GUI thread.

We don't use threads explicitly in the F# version, because we can start the simulation using asynchronous workflows. Also, we can replace the mutable variable and imperative while loop using recursion. You can see the source code in listing 14.24.

Listing 14.24 Running simulation using recursion and 'async' (F#)

```
let rec runSimulation(state:Simulation) =
   form.Invoke(new Action(fun () -> drawState(state))) |> ignore      #A
   if (form.Visible) then
      runSimulation(state.Step())                                     #1

Async.Spawn(async { runSimulation(Simulation.InitialState) })         #2
```

#A Redraw the form
#1 Tail-recursive call with a new state
#2 Start the computation asynchronously

The loop that runs the simulation is implemented as a recursive function. We don't need to worry about running out of stack space, because the recursive call is tail-recursive (#1). The lack of tail recursion in C# is the only thing that prevents us from using the same technique there.

The function is an ordinary function that loops in a blocking way while the form is visible, but we can still use asynchronous workflows to launch it in the background. This isn't really related to the typical asynchronous programming as we discussed it in the previous chapter–`Async.Spawn` is just a simple way to start executing the function on a separate thread. The workflow just calls the recursive function that blocks the thread and runs the simulation loop.

If you've implemented the `Step` method and added the code to draw the simulation, you should have a working application by now. Now that we've got the skeleton in place, we can work on making the animals and predators behave intelligently, and parallelizing the `Step` function.

14.3.4 Implementing smart animals and predators

Before we can start implementing the algorithm for animals and predators, we'll need a couple of helper functions. We'll look at their type signatures, which should give you enough information to understand how they work and also to implement them on your own if you want to. The following snippet shows their commented F# type signatures:

```
/// Returns the distance between two specified locations
val distance : Location -> Location -> float

/// Returns 10 check-points on the path between the specified locations
val getPathPoints : Location * Location -> seq<Location>

/// Returns the specified number of randomly generated locations
val randomLocations : int -> seq<Location>
```

The first function can be implemented using `Math.Pow` and `Math.Sqrt`. Note that we've given the function several parameters, which allows us to use partial application. This is convenient in F# when we want to calculate the distance of a collection of locations from one specific point. The second and third functions can be implemented using sequence expressions in F# and iterators in C#.

We'll need to call the `randomLocations` function from multiple threads running in parallel, so we need to use the `SafeRandom` module we created earlier. Each call to `randomLocations` first creates a new random number generator using `SafeRandom.New()` and then uses this generator repeatedly to build the result. The result is a lazy sequence, so it will be actually generated on demand. As it happens, we'll need all the items in the sequence to calculate the location of animal, so this doesn't make a big difference.

The algorithms that compute the new locations of animals and predators look quite similar in C# and F#, because we can implement them using the same collection processing functions. In F# and other standard functional languages, these are the part of standard libraries and in C# 3.0 they are available using LINQ. We'll look at these algorithms in the next two sections, showing each in a single language.

MOVING ANIMALS IN F#

Let's start with a function that takes the location of a single animal and the current state as arguments and returns the animal's new location. We'll have around 100 of animals in the simulation and we'll need to calculate the new location for all of them. This means it's probably not worth making the function run its logic in parallel within a single call. Instead, we'll just parallelize the many calls to the function later. Working out exactly where to split the computation is an important part of parallelizing an application.

Listing 14.25 implements an animal's behavior by generating 10 random locations in the world and working out which is the safest. It does this by looking at the direct path to the location and calculating how close the nearest predator is.

Listing 14.25 Implementing the animal behavior (F#)

```
let moveAnimal (state:Simulation) (animPos:Location) =
   let nearestPredatorFrom(pos) =                                     #1
      state.Predators |> Seq.map (distance pos) |> Seq.min

   let nearestPredatorOnPath(target) =                                #2
      getPathPoints(animPos, target)
      |> Seq.map nearestPredatorFrom |> Seq.min

   let target =
      randomLocations(10)                                             #3
      |> Seq.max_by nearestPredatorOnPath                             #3
   animPos + (target - animPos) * (20.0 / (distance target animPos))    #4
```

#1 Get the distance between 'pos' and the nearest predator
#2 Check safety of the path to the 'target'
#3 Choose the best of the generated locations
#4 Move the animal by 20 points in that direction

The code starts off by implementing two local utility functions. The first one (#1) uses `Seq.map` to calculate the distance between each predator and the specified location and then uses `Seq.min` to find get the shortest of those distances. The second one (#2) looks for the nearest predator on the path between the animal's current location and a specified destination by checking several points on the path between them.

Next we choose a target location for the animal. We generate 10 random locations and choose the one the animal can reach while staying as far away from predators as possible (#3). We do this with `Seq.max_by`, which returns the element for which the given function returns the largest value. In our case, the function returns the shortest distance between the predators and the path from the animal's current location to the randomly generated target. Finally, we use the overloaded operators of the `Location` type to calculate and return a new location of the animal. Each time the function is called, the animal moves 20 points in the best generated direction.

We'll use a similar algorithm to move the predators - but obviously with a different aim. The predator will also generate some random locations and then move in the best possible direction. The following section shows the C# version of the code.

MOVING PREDATORS IN C#

The algorithm for determining the best target for a predator is a bit more difficult. We're going to make the predator follow a path towards the largest number of animals and the smallest number of other predators. The C# method that implements the algorithm is shown in listing 14.26. It is a part of the `Simulation` class, which lets us access other predators and animals easily. (This is why we don't need to take the current state as a parameter like we did in the F# animal behavior function; the current state is available as `this`.)

Listing 14.26 Implementing the predator behavior (C#)

```
int LocationsClose(IEnumerable<Location> an, Location pos) {          #1
   return an.Where(a => Distance(a, pos) < 50).Count();
}
int LocationsOnPath(IEnumerable<Location> an,
      Location pfrom, Location ptarget) {                             #2
   return GetPathPoints(pfrom, ptarget)
      .Sum(pos => LocationsClose(an, pos));
}
Location MovePredator(Location predPos) {
   var target = RandomLocations(20).MaxBy(pos =>                        #3
      LocationsOnPath(Animals, predPos, pos) -                          #3
      LocationsOnPath(Predators, predPos, pos) * 3);                    #3

   return predPos + (target - predPos) *                                #A
      (10.0 / Distance(target, predPos));                               #A
}
```

#1 Count locations close to the given point
#2 Count locations close to the specified path
#3 Select path with many animals and a few predators
#A Move predator by 10 points

In the F# code for animal movement, we started by implementing two local helper functions. In C#, we implement similar helpers as ordinary methods. In principle, we could also use local lambda functions, but I decided to use a more typical C# approach to make the code simpler.

The first helper method (#1) takes a collection of locations (which can be our collection of animals or predators) and counts how many are close to the specified point. The second helper (#2) counts locations that are close to a whole path. This is done by generating a collection of points on that path, calling the first method on each of these points and summing the results.

To implement the predator behavior, we generate 20 random locations and choose the one with the largest number of animals and smallest number of predators (#3) close to the path between the predator's current location and the target. For each random location, we compute this "score" with two calls to `LocationsOnPath`. We multiply the count of nearby predators by a constant to make it more significant, because the number of predators in the whole simulation is smaller. The `MaxBy` extension method returns the location with the largest score. This method isn't a standard LINQ operator, but you can find its implementation in the complete simulation source code.

Now that we have functions for calculating new locations for both animals and predators, we can finally implement the larger step function of the simulation. It will need to calculate new locations of all the animals and predators, so this will be the best place to introduce parallelism into the simulation.

14.3.5 Running the simulation in parallel

To run the simulation in parallel, we'll use a combination of task-based parallelism using `Future` and declarative parallelism using PLINQ (and the `Parallel` module in F#). To calculate the new state of the simulation, we need to perform two basic tasks - move all the animals and predators. With the algorithms above, these two tasks take roughly the same time, so this would be enough on a dual core machine.

However, splitting the work into just two tasks isn't the best option if we have a machine with more than two processors, or if one of the tasks takes longer than the other. At this point, we can use declarative data parallelism, because we can calculate the new location of each animal and predator independently: we can view it as a list processing operation. Listing 14.27 uses both of these techniques to implement an F# function that runs the simulation.

Listing 14.27 Generating random state and running a simulation step (F#)

```
let simulationStep(state) =
   let futureAnimals = Future.Create(fun () ->                         #1
      state.Animals
         |> Parallel.of_seq
         |> Parallel.map (moveAnimal state)
         |> List.of_seq)
   let predators =                                                     #2
      state.Predators
         |> Parallel.of_seq                                            #3
         |> Parallel.map (movePredator state)                          #3
         |> List.of_seq                                                #3
   { Animals = futureAnimals.Value; Predators = predators }

type Simulation with                                                   #4
   member x.Step() = simulationStep x                                  #A
   static member InitialState =
      { Animals = randomLocations(150) |> List.of_seq                  #B
        Predators = randomLocations(15) |> List.of_seq }               #B
```

#1 Process animals as a task
#2 Process predators immediately
#3 Get new predator locations using PLINQ
#4 Add members to the 'Simulation' type
#A Runs a single simulation step
#B Generates random initial state

The listing implements the simulation step as an F# function and then makes this function part of the `Simulation` type using type augmentation (#4). It also adds the `InitialState` property that simply generates 100 random locations for animals and 10 randomly located predators.

In the simulation step, we use `Future` to process the animals as a task on a different thread (#1). The second task which processes predators is executed on the primary thread (#2), so we only need a single `Future` value. This is similar to the tree processing code we discussed earlier. Each task creates a new list with locations for the next simulation step (#3). The list processing is further parallelized using the `Parallel` module.

Tweaking the performance

Tweaking the code to get the maximal performance is always difficult and it requires a lot of experimentation. If you run the simulation with any of the two parallelization techniques, you should get a reasonable speedup. On my dual core machine it is about

155% times the speed of a sequential implementation when we just use Future, and 175% when using both PLINQ and Future. You can try various configurations to find the best performance on your system.

The C# implementation of the Step method is very similar to the F# version, as you can see from listing 14.28.

Listing 14.28 Running the simulation step in parallel (C#)

```
public Simulation Step() {
   var futureAnimals = Future.Create(() =>                          #1
      Animals.AsParallel()
         .Select(a => MoveAnimal(a))
         .ToList();                                                 #2
   var predators =
      Predators.AsParallel()                                        #3
         .Select(p => MovePredator(p))
         .ToList();
   return new Simulation(futureAnimals.Value, predators.Value);     #4
}
```

Just like in the F# version, we create a single Future value to process animals (#1) and run the code that processes predators on the primary thread (#2). Each list processing task uses the AsParallel method so the query operators are executed in parallel. We only need the Select operator to get a new location for every animal or predator, so we use the extension method directly rather than writing a query expression. Finally we create a new Simulation object (#4) that holds the new state.

As you can see, running the simulation in parallel wasn't difficult. Again, this was because we used functional techniques in our application design. The data structure representing the state is immutable, and in every step of the simulation we create a new state. This means that we can't run into race conditions while updating the state from multiple threads running concurrently.

14.4 Summary

In this chapter, we reviewed three different approaches for writing parallel applications in a functional style. Two of these techniques are based on essential aspects of functional programming.

Declarative programming lends itself to data parallelization, and PLINQ makes this particularly easy. We can use this from both C# and F#, and a wrapper module makes the F# code more idiomatic than working with PLINQ directly. Both C# and F# use higher order functions to represent the work to be done, either directly or through C# query expressions.

The second technique is task-based parallelism. This is made simpler by using the immutable data structures we're used to in functional programming. We can spawn multiple tasks to calculate different parts of the result and then just assemble these sub-results; immutability guarantees that tasks can work independently and will not corrupt each other.

We've also seen how to parallelize applications that use mutable state, but keep the mutation local. This is a valid and useful approach which allows us to use arrays in a functional way. When we create a new array the result of an operation, we can initialize the array in parallel. We saw how helper functions can make this even simpler, and we implemented a parallel version of array mapping in the `Array2.Parallel` module.

Code is only really useful when it's part of an application, so we looked at two complete applications in this chapter. When an application is designed in a functional manner from the start, the changes needed to introduce parallelism are relatively straightforward. In fact, we could use the techniques from this chapter to parallelize all the applications we created when talking about functional architecture in chapters 7 and 8.

In the next chapter, we're going to leave the realm of asynchronous and parallel computing and look at how we can express logic and behavior as clearly as possible. Some of the aspects of F# which make it so expressive can also be applied in C# 3.0 thanks to features such as lambda functions and extension methods. We're going to look at a famous functional approach for creating animations as our main example, but the same ideas can also be used in other domains.

15 Creating domain specific language for animations

In this chapter we're going to talk about language oriented programming. I briefly mentioned this approach in the first two chapters of the book. Its goal is to develop libraries that allow us to write code using as natural syntax as possible. Language oriented programming is in some sense a third paradigm that can be used in F# together with functional and object-oriented style, however its less formally defined and relies very much on an intuitive sense. This style is applicable to C# too, so we'll mix F# and C# examples again in this chapter. We'll use this approach to create a library for creating declarative animations. Note that we've already seen a brief example of this library in chapter 1, because it is a great example of writing readable and declarative functional code.

Just like any programming paradigm, the language oriented style gives us mainly a new point of view and it doesn't precisely specify a technology that we have to use. For that reason, we'll start this chapter by briefly talking about this programming style and about techniques that we can use in both C# and F# when using this style. After the brief introduction to the language oriented programming, we'll use language oriented techniques to create the animation library.

15.1 What is language oriented programming?

The language oriented programming is useful when we have a family of similar problems that we need to solve. Animations are a great example of this problem, because you may want to create hundreds of different animations. In the language oriented style, we start by creating a "*language*" for solving problems from this family and then use the language we created for solving all the particular problems. Implementing the language is a difficult problem, but solving problems using the language is a relatively easy task, supporting division of work in larger teams. I originally put the term *language* in double quotes, because it can mean a different thing depending on the particular technique we're using.

The key aspect of the language that we create is that it is tailored for the particular problem and it is as easy to use as possible. These kinds of languages are called *domain specific languages*, because they are used only for solving problems from one domain. This is what makes the style effective. When the language is limited to some domain, it can be easier to use than a *general purpose language* (which is on the other hand a language that allows us to solve any programming problem like for example C#). In the next section, we'll quickly talk about different types of domain specific languages.

15.1.1 Types of domain specific languages

Domain specific languages can be divided into external and internal. The difference between these two types is whether the domain specific language (DSL) exists as a language alone or whether it is embedded in some other, host, language. In this chapter, we'll focus only on internal DSLs, so I'll first briefly introduce external DSLs and then we'll turn our attention to the kind that we're interested in.

The term *external* means that the language is a completely separate language that has its own syntax. A good example is the SQL language, which has syntax and a specification. It can be used only for solving limited types of problems (data querying) and it cannot be used for solving any problem in general. However, it is a really good in doing the job that it was designed for. Implementing and designing external DSLs is quite difficult, because we have to create our own parser to read the source code and interpreter (or even a compiler) for the language. Also, designing a language from scratch requires a lot of experience.

Another example of an external DSL, which is easier to create, is an XML file with a well defined schema. In that case, our language specifies only the schema - the names of the elements that we expect and what are valid values. Working with code written in this language is easier, because we can use existing libraries for XML processing. For example, you can think of an XML configuration file as a language that describes the configuration.

On the other hand, internal DSLs are languages that are embedded inside general purpose languages such as C# or F# and use the syntax of that language. You can think for example of LINQ queries in C# 3.0. The syntax is part of the C# language, but it really looks like a separate language embedded in C#. However, LINQ isn't the best example, because it is baked in the language and you can't implement similar extension yourself. The point of internal DSLs is that you can create your own thanks to the flexibility of the hosting language. In the next three sections, I'll briefly show possible ways for creating an embedded language inside F# and C#.

15.1.2 Using abstract value representations in F#

In this section we'll look at the simplest way for embedding a language in F#. This only works for very simple languages. The language shouldn't contain any complicated program code specifying a behavior and most of the problems should be described just by combining primitive objects or collections of them and specifying their properties. In the second chapter, I mentioned that language oriented style is used very frequently in LISP. The following sidebar shows a brief motivating example using that language.

Embedded languages in LISP

As I already mentioned, the LISP family of languages is very suitable for creating domain specific languages. Thanks to its simplistic syntax, it is possible to encode almost any problem in a language that is consistent with the LISP programming style. Moreover, LISP allows us to treat the code as a data or as a program code, whichever is more suitable at the moment. This means that you can largely customize the language. Rather than running the code, we can read it as data and run it in any way we want.

The following brief example shows how we could describe a simple animation using the LISP syntax. The code describes two moving circles. The green one is rotating around the point (0, 0) in the 100 pixel distance in speed 2 and the second one is moving between the two specified points:

```
(compose
   (circle(green (rotating 0 0 100 2)))
   (circle(red (linear -100 0 100 0)))
)
```

We created our own primitives for describing the animation such as `compose`, `circle` or `rotating` and used these to specify how the animation looks. Once we have the description, we could write a LISP program that reads the embedded animation code and runs it using graphical user interface.

The code we've seen in the sidebar doesn't look like a program code that would run. It looks more like a code that constructs some value with a tree like structure. The value is just a data without any executable code. It describes the animation and we have to implement a function that takes the value and gives it some interpretation, most likely by showing it as a running animation.

When the code in the language looks like a value, we can implement it in F# by creating types that allow us to create values specifying all the properties we want. The most common F# type for this kind of problems is discriminated union, but you can also use lists (to represent collections of things) or records for primitives with large number of properties. The listing 15.1 shows a simple type declaration that defines the types for describing animations just like the one in the previous sidebar.

Listing 15.1 Specifying animations using discriminated unions (F#)

```
type AnimatedLocation =                                  #A
   | Static of PointF
   | Rotating of PointF * float32 * float32
```

```
      | Linear of PointF * PointF

   type Animation =                                  #B
      | Circle of AnimatedLocation
      | Compose of Animation * Animation
```

#A Specifies movement of an object
#B Type of values that represent animations

Using this type, we can now create values that describe very simple animations. In some sense, the type above specifies the syntax we can use when using the embedded language for creating animations, because it specifies what a valid animation value is[***]. The following example shows a simple animation doing the same thing as the one in the sidebar above:

```
let animation =
   Compose(
      Circle(Brushes.Green, Rotating(PointF(0.f, 0.f), 100.f, 2.f)),
      Circle(Brushes.Red, Linear(PointF(-100.f, 0.f), PointF(100.f, 0.f)))
   )
```

This example creates a value with a tree like structure specifying that we want to create an animation composed from two circles. The animation is created by specifying how the location of object changes and the language above allows us to create objects that are static, rotating or moving along some line.

In C#, we could get similar results using carefully designed classes and the new C# 3.0 features such as object initializers and collection initializers. However, the principle would be the same. The embedded language is used to create values that describe the problem. The use of discriminated unions or collections and objects is only influenced by what is the easiest way to create these kinds of values in the host language[†††]. In the next section, we'll look at another technique that we can use to create domain specific languages in C#.

15.1.3 Using fluent interface in C#

Fluent interface is probably the most widely known way for creating embedded languages in object oriented languages such as Java or C#. Unlike simple discriminated unions or classes created using the object initializer syntax, fluent interface allows us to hide the internal representation of the values that the language constructs.

The key idea behind fluent interface is that we create some object and then use a chain of method to specify properties and also behavior of that object. The listing 15.2 shows how we could use this style to describe the animation we've been using as an example in the previous section.

[***] Expert F# [Syme et.al, 2007] calls this type of languages *abstract syntax representations*

[†††] Fowler calls similar types of DSLs *literal collection expressions* [Fowler, 2008]

Listing 15.2 Creating animations using fluent interface (C#)

```
var animation = new Circle()                                    #1
   .WithColor(Brushes.Green)
   .RotateAround(0, 0).RotateDistance(100).RotateSpeed(2)       #2
   .ComposeWith(new Circle()                                    #3
      .WithColor(Brushes.Red)
      .LinearFrom(-100, 0).LinearTo(100, 0)                     #B
    );
```

#1 Create first circle
#2 Specify its behavior
#3 Compose with second circle
#B Specify the movement

The listing starts by constructing a circle (#1) and then setting its properties in a method chain. As a part of this method chain, we also specify the behavior (how the circle is animating) using three method calls (#2). In this part of the method call chain we specify individual properties of the rotation. After configuring the circle, we use `ComposeWith` method to compose the first circle with a second circle (#3). The second circle is provided as an argument and we use similar method chain to configure its properties.

In imperative programming languages, implementing fluent interfaces require creating a wrapper type that configures the created underlying mutable object[‡‡‡]. However, in functional programming this style looks very natural. In fact, it is quite similar to what we've seen already when talking about LINQ, where we also use method chains to specify the query. Let me demonstrate this using the following query:

```
var q = data.Where(c => c.Country == "London")
            .OrderBy(c => c.Name)
            .Select(c => c.Name);
```

Functional paradigm largely supports this style of programming. When using immutable data structures, every operation returns a new value instead of altering an existing one. This means that it has to returns some object as a result and this supports chaining of method calls just like in LINQ or the example above. This also makes the program more declarative, because the code in the embedded language is just a single expression that describes what we want to achieve. In fact, when writing functional code in C#, it often uses the fluent interface style to some extent. The C# version of the library for creating animations that we'll create later in this chapter will be no difference. In the next section, we'll look at a typical way for creating internal DSLs in functional programming languages.

[‡‡‡] Fowler describes how to do this [Fowler, 2008] and calls the construct *Expression Builder*

15.1.4 Using combinator libraries in F#

We've already seen a simple way to create embedded languages in F# using discriminated unions. Using this technique, we could easily specify the syntax of the embedded language, but the language wasn't very extensible, because the type fully specifies what we can do using it. Also, we couldn't use sophisticated internal representation, because the embedded programs created just simple values with a tree structure.

When using combinator libraries, the library gives us a couple of primitive values (such as primitive animating objects) and functions or operators to compose them, which are usually called *combinators*. This approach is very extensible, because we can create our own more sophisticated primitives (just by composing provided primitives) and also create our functions for composing objects in sophisticated ways (again, just by using various primitive combinators together).

We'll use this programming style when creating the F# version of the animation library, so let me demonstrate the idea using an example that we'll be able to write at the end of the chapter. The following snippet shows an animation with a sun and two rotating planets:

```
let planets =
   sun -- (rotate 150.0f 1.0f earth)
       -- (rotate 200.0f 0.7f mars)
```

The animation is composed from three primitives using a primitive combinator "`--`". We're using three different solar objects, which are primitives that we defined for a more specific type of animations (animations showing solar system). It also uses a rotate primitive, which gives us a way to specify how an object rotates. This looks like a basic primitive, but as we'll see it is derived from the only elementary primitive for specifying movement of an object. The figure 15.1 shows what we'll get when we run the animation above.

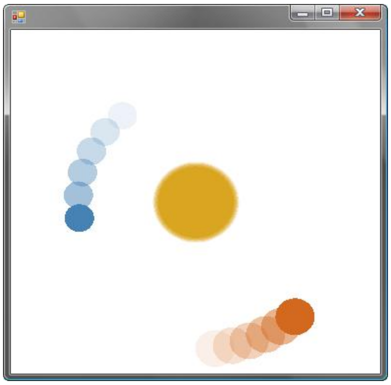


Figure 15.1 Running planet simulation with mars and the earth rotating around the sun.

Combinator libraries are quite popular in the functional programming community and have been used for a wide range of libraries. Perhaps the best known example is the Parsec library for creating parsers [Leijen, Meijer, 2001]. In this chapter, we'll see that they can be elegantly used for describing animations and we'll also see that using this style in F# is closely related to using the fluent interface style in the C# language.

15.2 Introducing functional animations

The idea of expressing animations in a functional language as a domain specific language (or a combinator library) comes from a Haskell project called Fran, which was created by Conal Elliott and Paul Hudak in 1997 [Elliot, Hudak, 1997]. Fran stands for functional reactive animations and the library allows you create animations and also specify how the animation reacts to events such as mouse clicks.

The library that we'll implement in this chapter is largely motivated by Fran. However, we'll focus on the animations alone and we won't talk in detail about reacting to events. However, we'll briefly look how the library could be extended to support reacting to events in the next chapter, where we'll talk about reactive GUI programming. Animations can be elegantly modeled using time-varying values. In Fran, these values are called behaviors and we'll follow this naming. The following note explains what a behavior is.

WHAT IS A BEHAVIOR?

Behavior is a time-varying value. It can be represented as a composite value, whose actual value may be different depending on the time. We talked about composite values earlier in the book, so for example `Option<int>` is a composite value that can have actual value of type integer or a special value `None`. Similarly, we'll have a type `Behavior<int>`, whose actual integer value can be different depending on the time.

Behaviors are an essential part of our animation framework, because we can use them for specifying locations of objects. When the location changes depending on the time, it means that the whole object will be moving. We'll start this chapter by talking about behaviors and we'll get to animations relatively late. However, when we'll start talking about animations, you'll see that we're already done, because we'll already have everything we'll need.

15.3 Working with behaviors

As I already mentioned, behaviors are largely independent from the animation library. They just represent a value, which is varying in the time but doesn't necessary have to be related to any graphical drawing. Using behaviors, we can for example create a time-varying integer, which changes with the time. We'll see that we can for example view the value of this integer at the specified time.

This means that we can start by implementing behaviors independently from the rest of the animation library. In this section, we'll look how behaviors can be represented and how we can create basic behaviors. However, animated integer is far less interesting than a running animation, so once we'll implement basic behaviors, we'll look how to use them to create animations. At the end of the chapter, we'll get back to behaviors and we'll implement various useful operators for creating and working with them.

15.3.1 Representing time-varying values

I we've seen in section 15.3, we can represent behaviors or time-varying values using some composite type, so the type of behaviors will be `Behavior<'a>`. It is important to realize that from the user perspective, it is not interesting what the internal representation is. Our library will provide basic functions for creating behaviors, so the user will build behaviors only using these functions. Let's now look what would be a good representation of behaviors.

REPRESENTING BEHAVIORS IN F#

One possible representation is to store the initial value and some difference that specifies how the value changes over time. For example, if we had initial value 10 and 1 as the difference per second, then the value after 15 seconds would be 25. However, this isn't very flexible and we could represent only very limited kinds of animated values. A better representation for our purpose will be a function that returns the actual value if we give it the time as an argument. This allows us to represent animated value of any kind. You can see the F# type declarations in listing 15.3.

Listing 15.3 Representing behaviors using functions (F#)

```
type BehaviorContext =                               #1
   { Time : float32 }
type Behavior<'a> =                                  #2
   | BH of (BehaviorContext -> 'a)                   #A
```

#1 Arguments for evaluating time-varying values
#2 Single case discriminated union
#A Function evaluates the actual value

The simplest possible representation of the behavior would be just a function of type `float32 -> 'a`, which would return the value when we give it the current time. In the listing above, we already did the next step of the iterative development process and we're using a bit more complicated version, which will make behaviors easier to use and also extend in the future. First of all, we're using a simple record type (#1) to wrap the current time. This allows us to add new information that can be used by the behavior other than just the time.

We're also using a single-case discriminated union to wrap the function (#2). This is quite useful because it gives the name to the type and allows us to hide the internal representation of the type. We'll see how exactly this can be done in chapter 18. The point is that the user of our library will just see `Behavior<int>` without knowing that it is internally a function. Also, using single-case discriminated union is syntactically quite convenient, because we can use pattern matching to access the function value. We'll see this shortly, but let's first look at the equivalent declaration in C#.

REPRESENTING BEHAVIORS IN C#

When thinking about the representation in C#, we have a fewer options to choose from. Somewhat surprisingly, we still have other options than using a class. We could represent the behavior directly using a delegate such as `Func<float, T>`. However, similarly to F#, it is better to hide the internal representation and give the type a name. You can see the C# representation of behaviors in listing 15.4.

Listing 15.4 Representing behaviors (C#)

```
internal class BehaviorContext {                                        #1
   public BehaviorContext(float time) {
      this.time = time;
   }
   private readonly float time;
   public float Time { get { return time; } }                           #2
}
public class Behavior<T> {
   internal Func<BehaviorContext, T> BehaviorFunc { get; set; }         #3
}
```

#1 Immutable animation state
#2 Gets current time of the animation
#3 Function that calculates the value

The representation in C# is essentially the same as in F#. We're using a simple immutable class to store the current time (#1). We'll pass instances of this class as an argument to various functions, so we want to make sure that it cannot be modified. For this reason we implement the `Time` property (#2) using read-only field. The representation of behavior is a generic class with a single property of type `Func<DrawingContext, T>` (#3), which corresponds to the function value wrapped inside an F# discriminated union.

I mentioned that we want to hide the representation of the `Behavior` type from the user, so we marked the property as `internal`. Similarly, the `BehaviorContext` type is also internal. Instead of constructing the behaviors directly, the user will create behavior using one of primitive functions that we provide. In the next section, we're going to look at these primitives. We'll start by looking at the C# version of these functions and then we'll implement the same primitives in F#. After that we'll spend some time using the F# interactive to explore how behaviors work.

15.3.2 Creating behaviors in C#

As I mentioned earlier we'll start with only a few basic functions for creating behaviors. After we'll have a nicer way for visualizing behaviors (by animating graphical objects), we'll get back to this topic and add more interesting constructs. Clearly, the simplest method that creates a behavior will just take the same function as the function used by our underlying representation.

However, this will be just an internal method that will make it easier for us to write other primitives. Since we'll wrap the creation of the object in a method, we'll be able to use the type inference of method type arguments in C#. This is similar to helper methods that we implemented earlier, such as `Option.Some`. You can see the code for this helper method in listing 15.5.

Listing 15.5 Creating behavior from a function (C#)

```
public class Behavior {
   internal static Behavior<T> Create<T>(Func<BehaviorContext, T> f) {
      return new Behavior<T> { BehaviorFunc = f };
   }
}
```

This source code uses C# 3.0 object initializer syntax to specify the value of the `BehaviorFunc` property when creating the object. We'll see later that we can create all kinds of behaviors without using this method, so we can mark it as internal. This means that the internal representation of behaviors can stay fully hidden.

As a next step, we can use the `Behavior.Create` method to create a couple of primitive constructs that we'll make available to the user. In C#, we'll expose them as static methods and properties in a static class. You can see this utility class in the listing 15.6.

Listing 15.6 Primitive behaviors (C#)

```
static class Time {
   public static Behavior<float> Current {                           #1
      get { return Behavior.Create(ctx => ctx.Time); }
   }
   public static Behavior<float> Wiggle {                            #2
      get { return Behavior.Create(ctx =>
               (float)Math.Sin(ctx.Time * Math.PI)); }
   }
   public static Behavior<T> Always<T>(T v) {                        #3
      return Behavior.Create(ctx => v);
   }
   public static Behavior<float> Anim(this float v) {                #4
      return Behavior.Create(ctx => v);
   }
}
```

#1 Behavior that represents the current time
#2 A value oscillating between 1 and -1
#3 Create constant behavior from a value
#4 Extension method for floats only

The first three constructs are quite straightforward. The `Current` property returns a behavior that represents the current time (#1). The property named `Wiggle` calculates sine function of the time (#2). This will be quite useful in the animation, because sine can be used for creating circular movements, so we'll use it later to create rotating drawings. Finally, the `Always` method creates a behavior that has always the same value, which is specified as an argument (#3). This method is generic, so we can use it to create constant behaviors of any type.

We'll use behaviors for creating animations, but there is another way to visualize them. We can draw a graph of the value depending on the time. The figure 15.2 shows the three primitive behaviors that we just implemented. The function that draws the screenshot is available as part of the online source code, so you can use it when experimenting with behaviors.

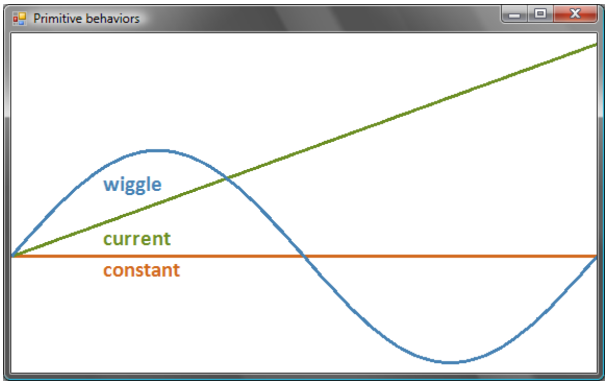


Figure 15.2 Primitive behaviors during the first two seconds. The value of 'current' ranges from 0 to 2 and 'wiggle' oscillates between +1 and -1.

The last construct in the previous listing is an extension method for the C# `float` type (#4). The C# syntax allows us to call methods directly on numeric literals, so we can use this method to write for example `0.5f.Always()`. This is syntactically very simple, which is one of the goals of domain specific languages. In the examples we'll see later, this construct will be quite frequent and you'll see that using the `Time.Always` method would definitely complicate these samples.

You may be wondering why we didn't create just a single generic method that would be also an extension method. That would of course work, but we'd add `Always` method to all types that we'll work with. This looks like overkill, because we won't need to create behaviors from most of the types. On the other hand, for floats, this extension method makes a good sense, because we'll need to create constant behaviors from floating point numbers relatively often.

In the next section, we'll briefly look at the implementation of the same primitives in F#. We already know how they work, so this will be quite easy. As a next step, we'll experiment with them in the F# interactive to learn how we can use them later in the chapter.

15.3.3 Creating simple behaviors in F#

Let's start by duplicating the functionality, which we just implemented in C#. The listing 15.7 shows how to implement two behavior values (called `wiggle` and `time`) and a function for creating constant behaviors (called `always`).

Listing 15.7 Primitive behavior functions and values (F#)

```
> open System;;
> let sample(a) = BH(a);;                                          #1
val sample : (BehaviorContext -> 'a) -> Behavior<'a>

> let always(n) = sample(fun _ -> n)                               #A
  let time      = sample(fun t -> t.Time)                          #B
  let wiggle    = sample(fun t -> sin(t.Time * float32 Math.PI))   #C
  ;;
val always : 'a -> Behavior<'a>                                    #2
val time : Behavior<float32>                                       #2
val wiggle : Behavior<float32>                                     #2
```

#1 Creates behavior from a function
#A Returns always the same value
#B Return the current time
#C Sine of the current time
#2 Function 'always' and behavior values 'time' and 'wiggle'

The listing starts by creating a utility function called `sample` (#1), which is similar to the previous `Behavior.Create` method. We could of course use the discriminated union constructor `BH` directly, but we want to make sure that the internal representation isn't unnecessarily exposed, so we'll create behavior values using this function. The name `sample`, reflects the fact that the function can be used to get the individual observation at selected times, which is called sampling in statistics.

Once we have the utility function, we create three primitives just as we did in the previous C# listing. You can also see the type signatures inferred by F# interactive (#2). The `always` is a generic function and the two other values are simply numeric behaviors. A one thing that we haven't implemented yet is a syntactically friendlier way to construct constant numeric behaviors. Using the function above, we could write `(always 0.5f)`, which isn't as elegant as it could be. We can use the same approach as in C# and define an extension member for the type `float32`:

```
type System.Single with
   member x.always = always(x)
```

When implementing extension members we have to use the full .NET name of `float32` which is `System.Single`. Extension members in F# aren't limited to methods, so we could implement the extension above as a property. The following example shows how to use it:

```
> let v = 123.0f.always
val v : Behavior<System.Single>
```

As you can see, this is syntactically very convenient and it will make the code of our animations quite elegant. Now that we know how to create primitive behaviors, we'll take a look how we can manipulate with them. The best way for doing explorative style of programming like this is to use the F# interactive tool.

15.3.4 Calculating with behaviors in F#

In this section, we'll write a few utility functions for working with behaviors. Even though we'll just experiment with them, we'll find later that most of the code we write in this section is extremely useful for implementing our animation sample. We'll first implement all of the functions in F# and test them in F# interactive and we'll see how to reimplement the most important functions in C# in a later section.

The first thing that we'll need to implement in order to test any code that uses behaviors is to write a function that reads the value of a behavior at the specified time. We'll implement such function in the next section.

READING VALUES

Calculating a value of the behavior at the given time is very easy. Because the internal representation is a function that gives us the value when it gets the time as an argument, we just need to execute this function. The listing 15.8 shows a function `readValue`, which takes a behavior and a time and returns the value. Once we have this function, we use it to read values of the primitive behaviors we created above.

Listing 15.8 Reading values of behaviors at the specified time (F# interactive)

```
> let readValue(BH(animFunc), time) =                        #1
     animFunc { Time = time };;                               #2
val readValue : Behavior<'a> * float32 -> 'a

> readValue(42.0f.always, 1.5f);;                             #A
val it : System.Single = 42.0f

> readValue(time, 1.5f);;                                     #B
val it : float32 = 1.5f

> readValue(wiggle, 1.5f);;                                   #C
val it : float32 = -1.0f
```

#1 Extract function value using pattern matching
#2 Run the function
#A Get value of a constant behavior
#B Time gives 1.5f after 1.5 seconds
#C Value of wiggle in the lowest peak

In F#, we can use pattern matching anywhere where we can bind a value. This includes the declaration of function arguments. In the listing above, we used a pattern in the function declaration to extract the function carried by behavior (#1). The discriminated union representing the behavior contains only a single case, so this pattern cannot fail and so it is perfectly fine to use it this way. As a next step, we construct a `BehaviorContext` value and pass it as an argument to the function (#2), which calculates the value of the behavior at the specified time.

The next couple of lines show how we can get values of the primitive behaviors that we implemented earlier. As you can see, all of them yield the expected results, so the interactive development style once again helped us to make sure that we're writing correct code. Reading values of primitive behaviors is a good start. The next question is, how can we create some more sophisticated behavior? Let's say that we for example wanted to create behavior that represents the square of the current time. We could write this using `sample`, but that's quite complicated. Ideally, we just want to apply the `square` function to a behavior! In the next section, we'll see how to do that.

APPLYING FUNCTION TO A BEHAVIOR

When explaining what a behavior is earlier, I said that behavior is a composite value and I draw the similarity between `Behavior<int>` and `Option<int>`. Both of these types are composite values that contain some value, but in some unusual way. The option type is unusual, because it may be empty and behavior is unusual, because the value depends on the time.

This analogy will be surprisingly helpful. If you remember chapter 6, where we talked about higher order functions for working with option values, you may also remember that we used a function `Option.map`, which applied a specified function to the value carried by the option. It turns out that we can implement exactly same `map` function for behaviors as well. Let's look at the listing 15.9, which shows how to do that and we'll talk about the details after that.

Listing 15.9 Implementing 'map' for behaviors (F# interactive)

```
> module Behavior =                                                  #A
     let map f (BH(fvalue)) =
        sample(fun t -> f(fvalue(t)));;                              #1
module Behavior =
   val map : ('a -> 'b) -> Behavior<'a> -> Behavior<'b>

> let squared = time |> Behavior.map (fun n -> n * n);;              #2
val squared : Behavior<float32>

> readValue(squared, 9.0f);;                                         #3
val it : float32 = 81.0f
```

#A Declare the function in a module
#1 Create behavior that applies 'f' to all values
#2 Behavior representing a square of the current time
#3 Get the value after 9 seconds

The first argument of `Behavior.map` is a function (`f`) that we want to apply to values of the behavior. The second argument is the behavior itself and we immediately extract the underlying function that represents it (`fvalue`). To build the result, we need to create a new behavior, so we construct its underlying representation, which is again a function. To construct it, we use a lambda function that takes the time as the argument. It first runs `fvalue` to get the value of the original behavior at the specified time. Once it has the value, it runs the `f` function to get the final result. In fact, the body is just composing the functions, so we could also write `sample(fvalue >> f)` as the implementation.

As you can see, we can now use `Behavior.map` to perform any calculation with the values carried by the behavior. The second command (#2) shows that we can calculate square of other primitive behaviors. The behavior that we get as a result doesn't execute the square function that we provided until we ask it for an actual value at the specified time. When we do this (#3), it executes the function that we returned as a result from `map`. This function then gets the current time (by evaluating value of the `time` primitive) and then runs the square function that we provided.

This may look a bit tricky, so a good way to understand what is going on is to add `printfn` construct to the function that calculates the square. This will clarify when is the function executed and with what arguments. You can also take a look at the figure 15.3 a couple of pages later, which shows the graph of the `squared` behavior together with a behavior that we'll create in the next section.

In this section, we did an amazing progress. We can now take a primitive behavior and construct a derived behavior using almost any calculation. In fact, we could now implement the `wiggle` primitive just by applying sine function to the `time` primitive using `map`. However, there are still things that we cannot do. For example, what if we wanted to add two behaviors? Is there a more elegant way to do this than using the `sample` primitive?

TURNING FUNCTIONS INTO "BEHAVIOR FUNCTIONS"

The `Behavior.map` function takes two arguments. In the example above, we specified both of them. However, partial function application allows us to call the function only with a single argument. Using the function in this way will give us an interesting insight. In the following example, we specify only the first argument (a function) and we'll use simple function abs, which returns absolute value of an integer:

```
> abs;;
val it : (int -> int)                             #A

> let absB = Behavior.map abs;;
val absB : (Behavior<int> -> Behavior<int>)       #B
```
#A Works with integers
#B Works with behaviors of integer

On the first line, you can see the type of the `abs` function. The second line shows what happens if we call `Behavior.map` with `abs` as the first and only argument. The type of the result is a function that takes `Behavior<int>` and returns `Behavior<int>`. This means that we used `Behavior.map` to create a function that calculates an absolute value of a behavior! We could use this trick to turn any function that takes a single parameter into a function that does the same thing for behaviors. Before we'll continue, let me briefly discuss this type of functions in general.

Lifting of functions and operators

The construct that we've just seen is a well known concept in functional programming and it is usually called *lifting*. In some sense, we could even call it a functional design pattern. Lifting allows us to transform a function that works with values into a function that does the same thing in a different setting [HaskellWiki]. Interestingly, lifting is hidden in one C# 2.0 language feature, so we can demonstrate it using familiar language. If we want to create a primitive value such as `int`, which can have a `null` value, we can use C# 2.0 nullable types:

```
int? num1 = 14;
int? num2 = null;
```

So far nothing we haven't seen anything interesting. We declared two nullable int values. One of them contains a real integer value and the other doesn't have a value. However, did you know that you can write following:

```
int? sum1 = num1 + num2;
int? sum2 = num1 + num1;
```

The result of the first calculation will be `null`, because at least one of the arguments is `null`. The result of the second expression will be 28, because both arguments of the "+" operator have a value. In this example, the C# compiler takes the ordinary "+" operator, which works with integers and creates a lifted "+" operator that works with nullable types. This operation is very similar to what we want to do with behaviors.

In the sidebar, we've seen a lifting of operators that take two arguments. The `Behavior.map` implements lifting for functions of single argument, so the remaining thing is to implement lifting for functions of multiple arguments. The listing 15.10 shows the remaining lifting functions.

Listing 15.10 Lifting functions of multiple arguments (F#)

```
module Behavior =                                                    #A
   (...)                                                             #B

   let lift1 f v =                                                   #1
      map f v
   let lift2 f (BH(fv1)) (BH(fv2)) =                                 #2
      sample(fun t -> f (fv1(t)) (fv2(t)))
   let lift3 f (BH(fv1)) (BH(fv2)) (BH(fv3)) =
      sample(fun t -> f (fv1(t)) (fv2(t)) (fv3(t)))

val lift1 : ('a -> 'b)             -> B<'a> -> B<'b>                 #3
val lift2 : ('a -> 'b -> 'c)       -> B<'a> -> B<'b> -> B<'c>        #3
val lift3 : ('a -> 'b -> 'c -> 'd) -> B<'a> -> B<'b> -> B<'c> -> B<'d>   #3
```

#A Place functions to the Behavior module
#B 'Behavior.map' omitted in this listing
#1 Does the same thing as 'map'
#2 Lifting for functions of two-arguments
#3 'B' is an abbreviation of 'Behavior'

The listing above first shows how to implement the lifting functions and then separately shows their signatures. The implementation of lift1 function is trivial, because it does the same thing as `Behavior.map` (#1). However, this is largely thanks to the partial function application, so when we'll implement the same functionality in C#, these functions will be different. The implementation of `lift2` and `lift3` (#2) is similar to the `map` function which we've seen earlier and I'm sure you'd be now able to implement lifting for functions of more than three arguments as well.

At this point, we'll able to implement any computations that work with behavior without using the low-level `sample` primitive. Any computation that you can think of can be implemented just using the `time` primitive and one of the lifting functions. The following snippet answers the question that motivated this section - how can we add two behaviors?

```
> let added = lift2 (+) wiggle time;;
val added : Behavior<float32>
```

The example uses `lift2` function with (+) operator as the first argument and two primitive behaviors as next arguments. If we read the value of the returned behavior, it will get values of the two behaviors used as arguments and add them together. We can visualize this behavior just as we did earlier for three primitive behaviors. The figure 15.3 shows the behavior above and a slightly modified behavior calculating square of the time (I modified the computation, so that the graph better fits in the screenshot).

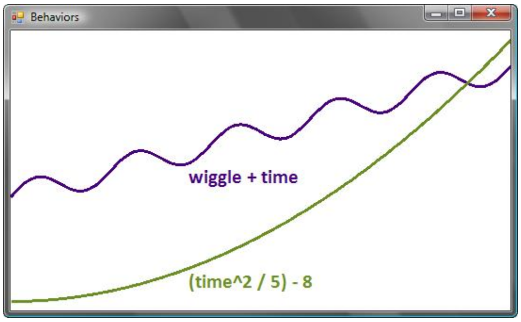


Figure 15.3 Graph showing values of two complex behaviors during the first 10 seconds.

The example above could be written even more elegantly. We'll see later that we can implement standard F# operators for behaviors, so we'll be eventually able to write just `wiggle + time`, but we'll do that later. In this section, we implemented `Behavior.map` and a family for lifting functions for behaviors in F#. The next section shows how to do the same thing in C#.

15.3.5 Implementing lifting and map in C#

Lifting functions and map are essential for constructing behaviors, so we'll need them in the C# version of the project as well. After the previous discussion about the F# version, you already have some idea what should these functions do, so we won't discuss everything in detail. However, the C# version will have some interesting aspects as well.

Whenever we had a `map` function in F#, we used the name `Select` in C#. This is the standard terminology used in LINQ, so we'll stay consistent and we'll implement the analogous C# construct as a `Select` extension method for behaviors. I earlier mentioned that in C#, there is a difference between the `Select` method and methods that implement the lifting. The best way to understand the difference is to look at the function signatures:

```
// Apply the function 'f' to values of the behavior 'behavior'
Behavior<R> Selelct<T, R>(Behavior<T> behavior, Func<T, R> f);

// Returns a function that applies 'f' to the given behavior
Func<Behavior<T>, Behavior<R>> Lift1<T, R>(Func<T, R> f);
```

The method `Lift1` takes just a single argument (a function) and returns a lifted function. The `Select` method takes the function to apply and also the behavior, so it can immediately construct a new behavior using this function. The implementation of these functions will be similar, so we can still see that they are related, but we cannot easily implement them using exactly the same code. The listing 15.11 shows the implementation of `Select` and `Lift1`. It also adds a method for lifting of functions with two arguments.

Listing 15.11 Lifting methods and 'Select' (C#)

```
public static class Behavior {
   // (...)                                                            #A

   public static Behavior<R> Selelct<T, R>
         (this Behavior<T> behavior, Func<T, R> f) {
      return Create(ctx => f(behavior.BehaviorFunc(ctx)));            #1
   }

   public static Func<Behavior<T>, Behavior<R>>
         Lift1<T, R> (Func<T, R> f) {
      return behavior => Create(ctx => f(behavior.BehaviorFunc(ctx)));  #2
   }

   public static Func<Behavior<T1>, Behavior<T2>, Behavior<R>>
         Lift2<T1, T2, R>(Func<T1, T2, R> f) {
      return (b1, b2) => Create(ctx =>                                 #3
         f(b1.BehaviorFunc(ctx), b2.BehaviorFunc(ctx)));               #3
   }
}
```

#A Earlier 'Create' method omitted
#1 Return a behavior that applies the function
#2 Returns function returning the behavior
#3 …similarly for two arguments

We'll add all the methods to the static class Behavior, which already contains the internal `Create` method. The implementation of Select (#1) is a literal translation of the F# version. It constructs a behavior and gives it a function that calculates the value at the specified time using the original behavior (`behavior`) and the provided function. The second method (#2) is more interesting, because it returns a function. This means that we'll return a lambda function that takes the behavior as an argument and then does the same thing as the previous method. Finally, we also implement `Lift2`, which is very similar, but works with functions of two arguments. The listing doesn't show implementation of `Lift3`, but you could implement it yourself using the similar pattern.

Using the methods above, we can construct the same behaviors as in the previous F# example. The best way to create behavior representing the squared time is to use the `Select` extension method. To add two primitive behaviors, we'll create a lifted addition and then use it:

```
var squared = time.Select(t => t * t);                        #A

var plusB = Behavior.Lift2((float a, float b) => a + b);      #B
var added = plusB(Time.Current, Time.Wiggle);                 #C
```

#A Square of the time
#B Lifted addition
#C Adding two behaviors

The first example should be fairly straightforward. It uses the `Select` method to specify a function that will be used for calculating values of the `squared` behavior. The second example first declares a value `plusB`, which is a function that can add two behaviors of type float. The overall type of this function is quite long:

```
Func<Behavior<float>, Behavior<float>, Behavior<float>>
```

Luckily, we can use the C# type inference in this case and we don't have to write this type explicitly. Once we have this lifted plus operator, we can use it to add two behaviors. We add together behaviors that represent the current time and the wiggle primitive and the result will be again a behavior (more specifically `Behavior<float>`). When we called one of the processing functions `Select`, you may have been wondering whether there is any relation between behaviors and LINQ, which also uses method called Select. The following sidebar answers this question.

Behaviors and LINQ

The signature of the `Select` method in the listing 15.11 has the same structure as the signature of the `Select` method that you can use in LINQ when writing queries. This isn't accidental and the fact is that you can use the C# 3.0 query syntax for creating behaviors. You can for example write the following perfectly valid code:

```
var squared = from t in Time.Wiggle select t * t;
```

This means exactly the same thing as the squared behavior, which we declared in the sample above, because this is the translation that LINQ uses. This type of query is quite interesting, because the source of the values (`Time.Wiggle`) contains potentially infinite number of values. However, the query is evaluated only when we need a value of the behavior `squared` at some specified time. We won't discuss LINQ queries for behaviors in larger detail in this chapter however we could implement the `SelectMany` query operator, which would allow us to write for example the following:

```
var added = from a in Time.Current
            from b in Time.Wiggle
            select a + b;
```

This is definitely an interesting alternative to using lifting explicitly. We could also implement an F# computation expression builder for creating behaviors. You can find implementations of these interesting extensions on the book web site www.functional-programming.net.

Behaviors are essential and the most difficult part of our animation framework. At this point we already implemented enough of what we'll need to create all the animations later in the chapter, so it is the time to look at the second component that we'll use for creating animations - the code that represents and works with graphical drawings.

15.4 Working with drawings

Similarly as when designing behaviors, we have to start by answering the question: "What is a graphical drawing?" Technically speaking, what is the right way for representing graphics in our animations? We'll take a look at this problem in the first section. However, we already know what we will need to do with drawings. First of all, we need to be able to compose them. The animation will be described in terms of drawings that are moving and are composed to form a single drawing. In the future, we could also support other geometrical transformations such as scaling and skewing.

15.4.1 Representing drawings

In the second chapter, we used graphical drawings as a sample problem that could be implemented using discriminated unions. This would be a good choice for diagramming application, where the application needs to understand the structure of shapes. However, in this chapter we'll use a more extensible representation. In C#, drawing will be simply an interface with a method that knows how to draw it. This could be represented more simply as a function, but as with behaviors, we want to hide the internal representation.

The F# version will follow the C# style and it will use interfaces as well. The reason for this is that the code will be using more .NET functionality, so it will be more consistent. Since the representations are quite similar this time, we can discuss both them side-by-side in the listing 15.12.

Listing 15.12 Representing drawings in C# and F#

```
using System.Drawing;                        open System.Drawing

interface IDrawing {                 #1      type Drawing =                   #2
   void Draw(Graphics gr);                      abstract Draw :
}                                                  Graphics -> unit

class Drawing : IDrawing {           #3      let drawing(f) =                 #4
   public Action<Graphics>                     { new Drawing with
      DrawFunc { get; set; }                         member x.Draw(gr) =
   public void Draw(Graphics gr) {                      f(gr) }
```

```
            DrawFunc(g);
         }
      }
```

Even though the architectural idea is the same in both languages, the implementation uses different techniques. In both C# and F#, we first define an interface (#1) (#2). As I mentioned earlier, we can omit the starting "`I`" from the F# interface name as long as it is used only from F#, because F# unifies all type declarations.

Later in the code, we'll need an easy way for creating drawings. The drawing is specified just by the drawing function, so we want to be able to create it just by specifying the drawing code using lambda function syntax. In C#, we create a simple class (#3) that implements the interface and has a property of type `Action<Graphics>`. The property represents the function that is called when the drawing should draw itself. In F#, we could use object expressions every time we'll need to create `Drawing` value later in the code, but we implement a utility function to simplify this task. The function `drawing` (#4) takes a function that does the drawing as an argument and returns a `Drawing` value that will use this function. This allows us to use lambda function, which is syntactically simpler than object expression.

The declaration above shows that object oriented and functional concepts can be used efficiently together. The interface declaration uses classical object oriented idea, because the sample we're discussing in this chapter is already a more evolved application. However, for the implementation, we can still use the simplicity of functional style. We'll see this simplicity in the next section where we'll implement a first concrete drawing.

15.4.2 Creating and composing drawings

In this section, we'll implement our first primitive drawing, which will be an circle. We could similarly implement many other types of drawings, but we'll look only at one example and you can add additional drawings yourself. Instead, we'll discuss other ways to create drawings, which will be later important for our animation code. In particular we'll see that we can create a composed drawing from two or more other drawings. However, let's start by implementing the circle.

CREATING AND MOVING CIRCLES

The function that implements the drawing gets the `Graphics` object as an argument. The object has `FillCircle` method, so the implementation will be really simple. Perhaps a more interesting aspect is that the listing contains a very low amount of additional noise and adding another drawing would take only about 3 lines of code. The listing 15.13 shows both C# and F# version.

Listing 15.13 Creating circle in F# and C#

```
// C# version
public static class Drawings {                                            #A
   public static IDrawing Circle(Brush brush, float size) {
      return new Drawing { DrawFunc = gr =>                              #1
         gr.FillEllipse(brush, -size/2.0f, -size/2.0f, size, size)
      };
   }
}

// F# version
module Drawings =                                                        #B
   let circle brush size =
      drawing(fun g ->                                                   #2
         g.FillEllipse(brush, -size/2.0f, -size/2.0f, size, size))
```

#A Static class for creating drawings
#1 Specify the drawing function using lambda
#B In F# we use module with functions
#2 Create drawing using higher order function

To make the code more organized, we placed all functions and methods for creating drawings in an organization unit named `Drawings`. In C# we'll use static class with drawings implemented as static methods and in F# we'll use a module containing functions. The C# code that implements the `Circle` method creates a new `Drawing` object (#1) and specifies its `DrawFunc` property using object initializer syntax. The lambda function is rather simple and just calls the `FillEllipse` method with the given brush and the specified size.

In F#, we implement circle as a simple function that takes the brush and the size as two arguments. The function doesn't take parameters as a tuple, which is usually the preferred way when creating functional libraries using the language oriented approach. The function uses the higher order function `drawing` that we implemented above and gives it a lambda function that does the draws the circle (#2). We'll use circle as the only primitive drawing for now and we'll now look what could be done just using one primitive.

If we created two circles using the function above, the center of both of them would be the point `(0,0)`. This means that if we composed two circles, we wouldn't get very interesting results. The code above allows us to specify their size, but it appears we forgot that we'll also need to specify location! Good news is that we haven't forgotten about that, because we'll use a different approach for specifying the location. We'll create an circle with the center in the point `(0,0)` and then move it to any point we'll need.

We'll implement moving of a drawing as a function (or method) that takes a drawing and X and Y coordinates as arguments. It then returns a result that draws the specified drawing translated by the given offset. How can we implement this functionality? We could draw the original drawing to a bitmap and then draw the bitmap to the specified coordinates, but there is even a simpler solution, because the `Graphics` object that we're using to do the drawing supports translation transformation directly. The listing 15.14 shows how we can use it. This time, we'll look only at the function (or a method) and we'll start with the F# version.

Listing 15.14 Translating drawings in F# and C#

```
// F# version
let moveXY x y (img:Drawing) =
   drawing(fun g ->                                                    #1
      g.TranslateTransform(x, y)                                       #A
      img.Draw(g)                                                      #B
      g.TranslateTransform(-x, -y) )                                   #C

// C# version
public static IDrawing MoveXY(this IDrawing img, float x, float y) {    #2
   return new Drawing { DrawFunc = g => {                              #3
      g.TranslateTransform(x, y);
      img.Draw(g);
      g.TranslateTransform(-x, -y); }
   };
}
```

#1 Return a new translated drawing
#A All drawing will be translated
#B Run the original drawing
#C Reset the transform
#2 Extension method
#3 Return translated 'Drawing' object

The C# version implements `MoveXY` as an extension method, which means that we'll be able to call it using the dot-notation. Since we want to make it available for any object that implements the `IDrawing` interface, using extension methods is the only option. The implementation code uses exactly the same pattern as we've seen when implementing the circle. The F# function uses the `drawing` primitive to specify how to draw the translated image (#1). In C# we return a `Drawing` object and specify the drawing function (#3).

The implementation is slightly more interesting this time. It changes the origin of the coordinate system used when drawing on the graphics using the `TranslateTransform` method. This means that if we run the original drawing (for example circle), it will still draw on the point `(0,0)`, but the point will actually be somewhere else on the graphics. Once we can move drawings around, we can finally create something else than just circles with the same center. However, we don't want to work with collection of drawings, so the next question is how could we create a single drawing from two other drawings?

COMPOSING DRAWINGS

If we composed drawings by storing all the drawings in a collection, we'd have to duplicate many functions. For example, we might want to move all drawings in a collection, but we couldn't simply use `moveXY`, because it works only with single drawing. Instead, we want to create a single drawing value that will draw all the composed drawings. In this section we'll create a function that allows us to compose drawings. To understand how exactly the function works, we can look at its type signature:

```
val compose : Drawing -> Drawing -> Drawing
```

The function takes two drawing values as arguments and returns a single drawing. Note that we don't need to specify any offsets to define the positions of drawings, because we can just move the arguments before calling `compose` using the `moveXY` function from the previous section. Implementation of this function is actually quite simple and you can see it the listing 15.15.

Listing 15.15 Creating composed drawing in F# and C#

```
let compose (img1:Drawing) (img2:Drawing) =
   drawing(fun g ->                                             #1
      img1.Draw(g)                                              #2
      img2.Draw(g) )                                            #2

public static IDrawing Compose(this IDrawing img1, IDrawing img2) {
   return new Drawing { DrawFunc = g => {                      #3
      img1.Draw(g);
      img2.Draw(g); }
   };
}
```

#1 Return a new drawing
#2 Draw both of the drawings
#3 Create composed drawing

The source code once again repeats the pattern that we've been using to create drawings. Just like in previous cases, we write a function (or a method) that contains a lambda function that does the drawing. In F#, the lambda function is wrapped inside a drawing using the `drawing` primitive (#1) and in C#, we're explicitly creating `Drawing` object (#3). When the returned composed image is drawn, the lambda function gets called and it simply invokes the `Draw` method of both of the drawings that we're composing together (#2).

Using the three functions we've just implemented, we can create any drawings that contain colorful circles at different locations. Implementing other primitive drawings should be easy task, so we'll focus on a more interesting thing now. Before moving to animations, let's briefly look how the code to create a simple drawing would look. I haven't yet mentioned how we can show a drawing in a form, because it'll be much easier to demonstrate after we start talking about animations, so for now, we'll just look at the code. The figure 15.4 shows a very simple drawing that we want to create.

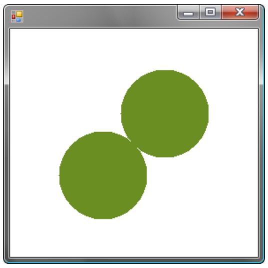


Figure 15.4 Two green circles moved using 'moveXY' and composed using 'compose'

Let's look only at the F# version of the code. We'll see more interesting C# samples once we turn everything into animations and with no doubt, you could write the C# version of the following code yourself:

```
open Drawings                                       #A
```

```
let greenCircle = circle Brushes.OliveDrab 100.0f    #B
let drawing =
   compose (moveXY -35.0f 35.0f greenCircle)         #C
            (moveXY 35.0f -35.0f greenCircle)         #C
```

#A Open module with drawing functions
#B Create a green circle
#C Compose two translated circles

The code starts by opening the Drawings module, which contains all the functions for working with drawings. Next, we create a single green circle of size 100 pixels. We duplicate the circle by moving it twice in different directions by 50 pixels and then compose the two moved drawings to get a single drawing value called `drawing`.

TRY IT YOURSELF

In this section, we implemented only a couple of basic drawing features, but there are many other things that you may want to try. First of all, we created only a single primitive drawing (a circle), so you may want to extend the code to work also for example with images. Also, we implemented only a single transformation in `moveXY`. The `Graphics` object supports other transformation such as `RotateTransform` or `ScaleTransform`, which could be used to create some very interesting effects. However, you may first want to read the last section of the chapter, so you can see how everything runs as an animation.

In this section we were talking about drawings and we've seen how to create them and compose them in interesting ways. In the previous section, we were talking about time-varying values. It seems that these two concepts are in principle all we need to create animations, so the only remaining thing is to show how we can use them together.

15.5 Creating animations

In the title of this chapter I promised that we'll implement a functional library for creating animations, but you already read about 20 pages of the chapter and you haven't seen a single animation. But let me tell you good news. If we use the two components that we just created in the right way, we're already finished with implementing animations.

Let's briefly recap what we've done so far. We created a type `Behavior<'a>`, that represents a value that changes over time and we also wrote functions such as `Behavior.lift2` that allow us to use standard functions for working with behavior values. Later, we discussed drawings and we created the type `Drawing` (in C# `IDrawing`) and a couple of functions for creating and working with drawings. How can we use these two components to create an animation?

WHAT IS AN ANIMATION?

If you think of an animation, then you can very easily describe it as a drawing that is changing in time. We've seen that we can represent a value changing in time as a behavior, so animation is in fact just a behavior of a drawing. This means that we can represent animations using the `Behavior<Drawing>` type.

As you can see, we just implemented a library for creating animations! Let's have a look how we can create an animation using the functionality we implemented earlier. We ended the last section by creating a simple drawing in F# (a value called `drawing`). We can simply turn it into an animation by using a function that creates a constant behavior:

```
> let animDrawing = always drawing;;
val animDrawing : Behavior<Drawing>
```

The type of the result is `Behavior<Drawing>`, and as we've just seen, this is a type that we'll use for representing animations. However, this isn't really an animation, because the drawing is always the same. We'll shortly see that we can use lifting functions and functions for working with drawings to create far more interesting animations, but let's first see how we could display the animation in a form.

15.5.1 Implementing animation form in F#

In this section, we'll implement a form for displaying animations. This is particularly interesting in F#, because we'll use it from the F# interactive to create and experiment with animations. At this point, the typical style of development in F# is very different from C#. In C#, we'll implement the form, create the animation, compile our application and run it. On the other hand, in F#, we'll implement the form, load it into F# interactive and then we'll try to write some animations and use the loaded form to display then and see how they work. As I mentioned repeatedly in the book, this style of interactive development is essential for F# and it helps us to make sure that our code works correctly as early as possible, because we can immediately try it.

The listing 15.16 shows the F# implementation of the form. The C# version is essentially the same and you can find it in the source code, which is available online. We won't discuss the C# code needed to display the animation, because we'd have to compile the whole application, but I'll continue to show all interesting pieces of code (such as creating some nice animations) side-by-side in both C# and F#.

Listing 15.16 Implementing form for showing animations (F#)

```
open System.Windows.Forms

type AnimationForm() as this =
   inherit Form()                                                  #1
   let emptyAnim = always(drawing(fun _ -> ()))
   let mutable startTime = DateTime.Now                            #A
   let mutable anim = emptyAnim                                    #A

   let setAnimation(newAnim) =                                     #2
      anim <- newAnim
      startTime <- DateTime.Now
```

```
    do this.SetStyle(ControlStyles.AllPaintingInWmPaint |||
                     ControlStyles.OptimizedDoubleBuffer, true)
       let tmr = new Timers.Timer(Interval = 25.0)                    #B
       tmr.Elapsed.Add(fun _ -> this.Invalidate() )                   #B
       tmr.Start()                                                    #B

    member x.Animation                                                #3
       with get() = anim
       and set(newAnim) = setAnimation(newAnim)

    override x.OnPaint(e) =                                           #4
       let wid, hgt = x.ClientSize.Width, x.ClientSize.Height
       e.Graphics.FillRectangle(Brushes.White, 0, 0, wid, hgt))       #D
       e.Graphics.TranslateTransform(float32 wid/2.0f, float32 hgt/2.0f) #D

       let elapsed = (DateTime.Now - startTime).TotalSeconds
       let drawing = readValue(anim, float32 elapsed)                 #5
       drawing.Draw(e.Graphics)                                       #5
```

#1 Inherit from the .NET 'Form' class
#A Mutable state of the object
#2 Setup a new animation
#B Redraw animation every 25ms
#3 Get or set the current animation
#4 Override method that does the drawing
#D Prepare for the drawing
#5 Get the current drawing and run it

The class declaration contains a few advanced aspects of object oriented programming in F# that we'll need to explain. The form inherits from the .NET class `Form`. This is written using the `inherit Form()` construct (#1) directly following the type declaration. The body of the class starts with a few ordinary let bindings. The first one declares an empty animation. This is a constant behavior containing a drawing (created using the `drawing` primitive) that doesn't draw anything. We use it later as an initial animation displayed on the form. When using the form, we'll create it in the F# interactive only once and we'll use mutation to imperatively set the currently displayed animation. This is a very common and perfectly valid use of mutable state when using interactive development in F#.

The mutation is done using a property named `Animation` (#3). When creating read/write property, we use the `with` keyword and specify getter and setter as two blocks of code using a syntax similar to the usual function declaration. In the setter, we invoke a utility function `setAnimation` (#2) which updates the state of the form.

The declaration of the form also contains the `as this` construct directly following the implicit constructor. This allows us to use the reference to the form in the constructor code. It is needed in the initialization when we call `SetStyle` method to avoid flickering (the `|||` operator is a binary or operator, which can also be used for working with enumerations). The `this` reference is also used when creating the timer that forces redrawing of the form.

The most interesting part of the form is in the OnPaint member (#4). It overrides the default OnPaint method of a .NET form and draws the animation. It is called repeatedly, because we created a timer that invalidates the form using Invalidate method. Drawing of the animation in the end is quite easy (#5). We use the helper function readValue, which we declared earlier when experimenting with behaviors. The function gives us drawing for the current time in the animation. Once we have the drawing, we just invoke its Draw method, which paints it on the graphics object provided by the system.

Creating the form was the only difficult thing that we had to write in order to start creating animations. Now, we can instantiate the form in F# interactive and set its property Animation to the simple drawing we created earlier (a value called animDrawing) and you should see a basic (not yet animated) drawing. To show a real animation, we'll need to use more interesting behaviors. We'll see how this can be done in the next section.

15.5.2 Creating animations using behaviors

Now that we have all the underlying machinery to create animations and a form to display them, we can start creating various animations. In this section, we'll use the drawing we already created in F# (two green circles) and we'll create an animation that moves it. In this section we'll just continue our exploration using the interactive tools that F# gives us, so we'll implement the sample only in F#. It will show us that we can use primitives that we already have to create animations and we'll see what kind of operations we need to do with animated graphics. After that we'll again implement everything in both F# and C#.

In the listing 15.17, we first create a version of the moveXY primitive that works with animations using lifting. The function originally worked with drawings and numbers, but to create an animation, we want to use it with behaviors. Using this function, we can then create an animation that moves the drawing depending on the time.

Listing 15.17 Creating simple animation (F# interactive)

```
> let af = new AnimationForm(ClientSize = Size(750, 750), Visible=true) #1
val af : AnimationForm

> let moveXY x y img = Behavior.lift3 Drawings.moveXY x y img;;         #2
val moveXY : Behavior<float32> -> Behavior<float32> ->                  #A
             Behavior<#Drawing> -> Behavior<Drawing>                    #A

> let wiggle100 = Behavior.lift2 (*) wiggle 100.0f.always;;             #3
val wiggle100 : Behavior<float32>

> af.Animation <- moveXY wiggle100 wiggle100 animDrawing                #4
```

#1 Create the animation form
#2 Lift the 'moveXY' function to work with behaviors
#A All arguments are behaviors now
#3 Multiply 'wiggle' to get a value in range -100 .. 100
#4 Create and display the animation

We start by creating moveXY function that allows us to specify behaviors as offsets when moving the drawing (#1). This is done simply by using Behavior.lift3 primitive, which turns a function with three arguments into a function that works with behaviors. As you can see from the inferred type signature, the function now takes two behaviors specifying offset and an animation (the use of `#<type>` isn't important here, so we can read it just as a type `Behavior<Drawing>`) and returns a new animation. On the next two lines, we use this primitive to create an animation, which you can see in figure 15.5.

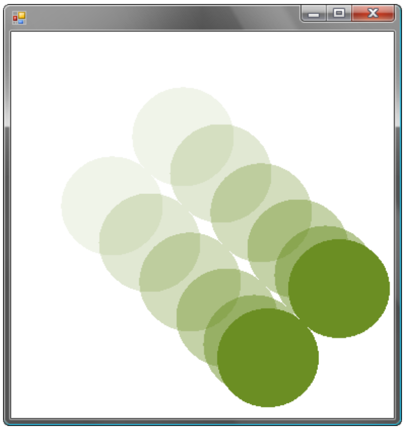


Figure 15.5 Two green circles moving from the upper left corner to the bottom right corner (shadows are added, so you can see how objects move)

To define the animation, we first need to create a behavior that will give us some reasonably large X and Y offsets. We do this by multiplying the `wiggle` primitive by a constant behavior returning always 100. This means that the value of `wiggle100` (#2) will oscillate between -100 and +100. Once we have this value, we use the new `moveXY` function that takes behaviors as its arguments. We give it two behaviors and the result is an animation that sets the coordinates of the drawing to values ranging between `(-100,-100)` and `(100,100)`.

ANIMATIONS IN A BOOK

When writing this chapter, I realized that presenting animations in a book will be quite difficult, because you won't really see any animation. To give you a better idea how the animation looks, I added shadows that show earlier locations of the objects. Implementing this attractive effect was rather easy. The only thing we need to do is to draw the animation multiple times using an earlier time when reading the drawing from a behavior and then use .NET drawing capabilities to display older animations as transparent. We won't look at the implementation in the book, but you can find it in the online source code (as an additional type of form that presents the animation).

The listing we've just seen shows that we'll need to use two kinds of operations quite often when creating animations. First of all, we need lifted versions of primitive functions such as `moveXY` or `compose` and secondly, we often need to calculate with behaviors and we had to do this by explicitly lifting "*" operator which isn't very elegant. In the next section we'll see how this could be improved.

15.5.3 Adding animation primitives

Our goal is to make the code that constructs animations as declarative and as simple as possible. For this reason, we want to use primitives that are directly designed for creating animations. We've seen that we already can do anything we need just using functions for working with behaviors and drawings, but the code would look more elegant if we created primitives for creating animations rather used lifting explicitly. Let's start by looking at functions for working with drawings.

CREATING DRAWING PRIMITIVES FOR ANIMATIONS

In the last listing, we created `moveXY` primitive that works with animations by lifting the `Drawings.moveXY` function, which works with drawings. Now we need to do the same thing for other drawings primitives - namely `circle` and `compose`. The listing 15.18 shows F# declarations for composition operator and primitive for creating animated circle. It also shows the C# version of composition (this time as an extension method) to demonstrate how to use lifting in C#. The C# implementation of other lifted operations is essentially the same, so it isn't included in the listing, but you can find it in the online source code.

Listing 15.18 Creating animation primitives using lifting in F# and C#

```
// F# version
let circle brush size =                                                #A
   Behavior.lift2 Drawings.circle brush size
let ( -- ) anim1 anim2 =                                               #1
   Behavior.lift2 Drawings.compose anim1 anim2

// C# version
public static class Anims {
   public static Behavior<IDrawing> Compose
         (this Behavior<IDrawing> anim1, Behavior<IDrawing> anim2) {    #2
      return Behavior.Lift2<IDrawing, IDrawing, IDrawing>              #B
         (Drawings.Compose)(anim1, anim2);                             #3
   }
   // (...)                                                            #C
}
```

#A All arguments are behaviors
#1 Custom operator for composing animations
#2 Extension method for animations
#B Type arguments are required
#C 'MoveXY' and 'Circle' are similar

Using lifting in F# is quite easy thanks to the type inference. We created the moveXY primitive in the previous section, so in this listing we just add the `circle` function and a custom operator "`--`" (#1) for composing animations. Both of them have two arguments, so we can use the Behavior.lift2 function to create a variant of the function that works with behaviors. The resulting function (or an operator) takes two behaviors as arguments and returns a behavior (more specifically, the type `Behavior<Drawing>`, which represents an animation). This means that the `circle` function now takes both brush and the size as a behavior and we can create circles with changing size and color.

In the C# version, we place all operations inside a static `Anims` class. If you remember chapter 6, I mentioned an analogy between custom operators in F# and extension methods in C#, so this suggests us to implement `Compose` as an extension method (#2). In the body of the method, we use lifting, to get a lifted version of the `Drawings.Compose` method. The `Lift2` method returns a function that we can call, so we immediately give it two animations as arguments and return the result, which is the composed animation. Before we look how to use the functionality we just implemented, let's do two more improvements that will allow us to do interesting things with behaviors.

CALCULATING WITH BEHAVIORS

Another operation that we'll need quite frequently is to multiply or add numeric behaviors. In the sample animation, we wanted to multiply the `wiggle` value by constant behavior `100.0f.always`. Instead of using lifting explicitly, it is more convenient to provide overloaded operators for working with numeric behaviors. The listing 15.19 shows how to implement two operators for addition and multiplication in F#.

Listing 15.19 Extension operators for calculating with behaviors (F#)

```
type Behavior<'a> with                                                #1
   static member (+) (a:Behavior<float32>, b) =
      Behavior.lift2 (+) a b
   static member (*) (a:Behavior<float32>, b) =                       #2
      Behavior.lift2 (*) a b                                          #A
```

#1 Type augmentation adding operators
#2 Multiplication for behaviors of 32bit floats
#A Lift the standard operator

In F#, we can add operators to a type using type augmentations (#1) that we've seen in chapter 9. The augmentation simply adds two static members to the type that take arguments of type `Behavior<float32>` (#2). Adding generic operators that work with any numeric type would be more difficult, so we create operators only for the numeric type we're using in this chapter. The implementation of the operator is easy, because we can again express it just using appropriate lifting function. The C# implementation is almost the same, so we won't talk about it, however you can find it in the online source code.

The second addition is more interesting. When running an animation (represented as a behavior) we'll sometimes want to run it faster or start it after some time. For example, if we had two rotating circles, we may want to rotate one of them two times faster. This could be done by creating a new primitive behavior, but there is a more elegant way. We can simply create a function that takes a behavior as an argument and returns a new behavior that runs faster or is delayed by specified number of seconds. You can see the F# version of the source code in listing 15.20.

Listing 15.20 Speeding up and delaying behaviors (F#)

```
let wait shift (BH(bfunc)) =
   sample(fun t -> bfunc { t with Time = t.Time + shift })            #1
let faster q (BH(bfunc)) =
   sample(fun t -> bfunc { t with Time = t.Time * q })                #2
```

#1 Shift the original time
#2 Scale the original time

The listing again shows only the F# version of the code, because I only wanted to demonstrate the idea. Implementing the C# version should be easy, because it uses the same pattern we've seen repeatedly earlier in section 15.3.5 and as always, the full source code is available on the book web site.

Functions in the listing work with any behaviors in general. Both of them take floating point number as the first argument and an original behavior as the second one. To create a new behavior, we have to use the low level `sample` primitive. We create a new behavior that calls the function extracted from the original behavior and gives it a different time as an argument. In the first case, the time is shifted by the specified number of seconds (#1) and in the second case it is multiplied by the provided coefficient (#2). To explain what this means, let's look at the second function and let's say we're running a behavior 2 times faster. When the actual time is 2 seconds, the returned behavior will invoke the original one with time set to 4 seconds, which means that the animation will do the movements it would usually perform in 4 seconds just in 2 seconds.

As a last thing in this chapter, we'll create a core part of a solar system simulation. It will use everything we implemented so far to create a more complicated and interesting animation and I'll use it to demonstrate how composable and reusable our solution is.

15.5.4 Creating solar system animation

The key part of the solar system animation will be a rotation of objects around each other. The library we just created allows us to compose primitives it provides into higher level primitives for our particular problem, so we can encapsulate rotation inside a reusable function (or C# method) that we'll later use to describe the simulation. This is an important property of a well designed functional library and for example, functions for working with sequences are composable exactly in the same way.

Our new primitive will rotate a provided animation around the point `(0,0)` in a specified distance and using a specified speed. You can see both F# and C# version of the primitive in the listing 15.21.

Listing 15.21 Implementing rotation in F# and C#

```
// F# function
let rotate (dist:float32) speed img =
   let pos = wiggle * dist.always                                   #A
   img |> moveXY pos (wait 0.5f pos)                                 #1
       |> faster speed                                               #2

// C# extension method
public static Behavior<IDrawing> Rotate
      (this Behavior<IDrawing> img, float dist, float speed) {
   var pos = Time.Wiggle * dist.Always();
   return img.MoveXY(pos, pos.Wait(0.5f))                            #3
             .Faster(speed);                                         #3
}
```

#A Oscillate between -dist and +dist
#1 Delay the Y-coordinate animation by one half
#2 Use the provided speed
#3 Chaining extension methods

Interestingly, we can implement the rotation just using `moveXY` function. The movement created using the `wiggle` primitive is a sinusoid, which means that it gives us values for one coordinate of the rotating object. To get the second coordinate, we need to delay the phase by one half of a second. This gives us the same value we'd get if we created a primitive using cosine function. To delay the behavior, we can simply use the `wait` function that we just implemented (#1).

Note that we can use pipelining to specify sequence of operations that should be done with an animation. After specifying the rotation, we also apply the `faster` function (#2) to specify the required speed of the rotation. In C#, we can use the same programming style thanks to the use of extension methods (#3) that take the animation as a first argument and return a new one as the result.

Using the primitive to describe rotation, we can now create our solar system animation rather easily. We'll start by creating three circles that represent solar objects (we'll have only sun, the earth and month) and then describe how they rotate around each other. The figure 15.6 shows the running animation, so you can see what we're creating.

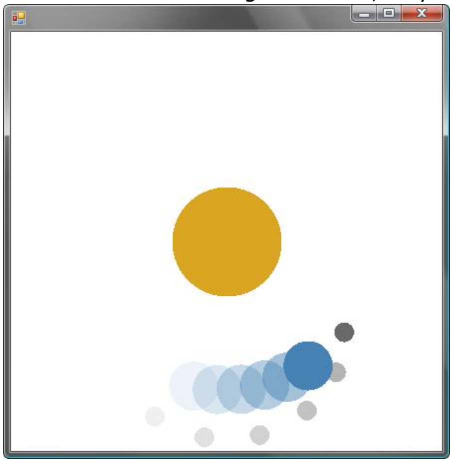


Figure 15.6 Running solar system simulation; Moon is rotating around the earth and both of them are rotating around sun.

Let's now look at the source code. The listing 15.22 shows both of the versions side by side, so that we can see how some constructs in F# and C# correspond to each other.

Listing 15.22 Creating solar system animation

in F# and C#

```
// F# version
let sun   = circle (always Brushes.Goldenrod) 100.0f.always             #A
let earth = circle (always Brushes.SteelBlue) 50.0f.always              #A
let moon  = circle (always Brushes.DimGray)   20.0f.always              #A

let planets =
   sun -- rotate 150.0f 1.0f                                            #1
      (earth -- rotate 50.0f 12.0f moon)                                #2

// C# version
var sun   = Anims.Cirle(Time.Always(Brushes.Goldenrod), 100.0f.Always());
var earth = Anims.Cirle(Time.Always(Brushes.SteelBlue), 50.0f.Always());
var moon  = Anims.Cirle(Time.Always(Brushes.DimGray),   20.0f.Always());

var planets = sun.Compose(
   earth.Compose(moon.Rotate(50.0f, 12.0f))                             #2
        .Rotate(150.0f, 1.0f));                                         #1
```

#A Create planets with constant size and color
#1 Rotate the earth with moon around the sun
#2 Rotate moon around the earth

Is there a way for using (#2) two times in the listing above?

The code that constructs planets is quite simple. The only notable thing is that we're using a circle primitive for creating animations, so we have to provide both brush and the size as a behavior. This is quite interesting, because we could for example create a shining sun that is growing bigger and whose color changes.

Composing the animation from rotating objects is far more interesting. I'll start explaining it from the middle. We use the `rotate` function to create a moon that rotates around the center in the distance 50 pixels. We compose this animation with the earth, which isn't rotating (#2), so the result is a moon rotating around the earth. The type of this result is just an animation, so we can again start rotating it in the distance 150 pixels (#3). If we compose the resulting animation with a sun (that isn't moving), we'll get the animation where the earth is rotating around the sun. This way of composing rotations shows that the framework is quite composable. Before we conclude this chapter, let me mention a few more extensions that we could make to the animation library.

Taking the animation library further

There are many interesting additions that we could make to the library. I already mentioned that we could add more primitive drawings and transformations. We should of course add lifted versions of those, so they can be easily used when creating animations. However, there are other even more interesting options.

We could add an additional primitive just like `wiggle` that would give us a location of the mouse cursor. We could implement this for example by adding the mouse location to the `BehaviorContext` type. This is quite interesting because it would allow us to create animations that depend on the mouse location. A more sophisticated extension could allow us to create non-linear dynamic systems. We could add a primitive that tells us how quickly is a certain behavior value changing and we could then use it to create system that depends on how quickly its state is changing.

The animation library we created is indeed far from being a robust physical simulation, but it shows an interesting direction. Some of the ideas that I outlined in this sidebar will be available on the book web site after the book is published.

The animation library implemented as a domain specific language is an interesting example of a very useful functional programming style. This style can be of course used for developing a wide variety of applications, so in the next section we'll briefly sketch a domain specific language for a completely different area.

15.6 Developing financial modeling language

So far in this chapter, we've seen most of the ideas that you need to know if you plan to design your own domain specific language. To give you some idea how this could be done for a more business oriented problem, we'll briefly sketch a language that can be used for modeling financial contracts. This example is motivated by an article by Simon Peyton Jones et al. *Composing contracts: an adventure in financial engineering* [Jones, Eber, Seward, 2000]. In this section, we'll implement only the most basic parts of the language, so you can look at the article for more information.

15.6.1 Defining the primitives

Similarly as when creating the animation language, we'll need to start by defining the type of the values we're working with and by implementing a couple of primitives that can be later composed. Our primitive data type will be called `Contract` and it will represent trades that can occur at some particular date and time.

Declaring the contract type

As you can see in listing 15.23, we're using similar technique as when declaring behaviors and we're creating a discriminated union with a single discriminator that contains a function that calculates the list of possible trades.

Listing 15.23 Type representing financial contracts (F# interactive)

```
> type Contract =
     | CF of (DateTime -> seq<int * string>);;                    #1
(...)
> let eval (CF f) dt = f(dt) |> List.of_seq;;                     #2
val eval : Contract -> DateTime -> (int * string) list
```
#1 Contract can calculate it's trades

#2 Gets a list of trades at particular date

The function that represents the actual contract takes a single argument and returns a sequence of tuples (#1). When we call it with a particular date as an argument, it will generate all trades that can occur at the given date. The trade is represented simply as a tuple containing a number of stocks that we want to buy or sell and the name of the stock. We'll use positive numbers to represent buying and negative values to represent selling of stocks.

The second part of the listings implements an `eval` function (#2) that evaluates the contract at some time and returns the list of trades. We're using a sequence to represent the trades in the contract, because that makes the code more general and we could in principle also represent infinite number of possible operations. However, the `eval` function returns a list, because we expect that the overall result will be finite.

Implementing combinators

Once we have the data type representing values of our language, we need to implement a couple of primitive functions for creating and composing these values. In case of behaviors, we created primitive values such as `wiggle` and we declared lifted operators for composing them. In case of contracts, we'll start with a function `trade` that creates a contract representing a single purchase that can occur at any time. To compose contracts, we'll provide a function `combine`, which unions trades of the two provided contracts.

The listing 15.24 shows the implementation of these two functions as well as functions for restricting the dates when the contracts can occur and a function for creating trades where we're selling some stocks.

Listing 15.24 Combinators for creating and composing contracts (F# interactive)

```
> let trade amount what = CF(fun _ ->                                #1
     seq { yield amount, what })
  let combine (CF a) (CF b) = CF(fun now ->                          #2
     Seq.concat [ a(now); b(now) ])
  ;;
val trade : int -> string -> Contract
val combine : Contract -> Contract -> Contract

> let after dt (CF f) = CF(fun now ->                                #3
     seq { if now >= dt then yield! f(now) })
  let until dt (CF f) = CF(fun now ->                                #3
     seq { if now <= dt then yield! f(now) })
  let give (CF f) = CF(fun now ->                                    #4
     seq { for am, itm in f(now) -> -am, itm })
  ;;
val after : DateTime -> Contract -> Contract
val until : DateTime -> Contract -> Contract
val give : Contract -> Contract
```

#1 Single trade of specified number of stocks
#2 Concatenate trades of two contracts
#3 Limit the date when contract is active
#4 Change sale to purchase and conversely

A single trade that can occur at any time is represented as a function that ignores its parameter (a date when we're evaluating the contract) and returns a sequence with a single element (#1). Composition is also easy (#2), because we simply concatenate all trades of the two underlying contracts that can occur at the given date.

The next two primitives let us limit the date when a contract is active (#3). We implemented them by creating a function that tests whether the date when we're evaluating the contract matches the condition of the primitive. When the test succeeds, it returns all underlying trades using the `yield!` primitive, otherwise it returns an empty sequence. Finally, the last primitive can be used to change whether a contract is sale or a purchase of the specified stocks. We implement it by iterating over all the underlying trades of a contract and changing positive amounts to negative and vice versa.

As I wrote earlier, the goal of this section is only to sketch how a language for describing financial contracts might look like. However, even with the very limited example that we've just implemented, we can describe many interesting things.

15.6.2 Using the modeling language

Perhaps the most valuable thing about domains specific languages is that we can use the basic primitives provided by the library designer to create more complicated functions for composing contracts. This makes the library quite flexible, because the users of the library (in our case financial experts) can create the primitives that they need. As the designers of the core library, we only need to provide primitives that are rich enough to allow that.

In the listing 15.25, we'll briefly look at two such functions that are defined in terms of the primitives we've seen in the previous section. It defines function for specifying time interval within which a trade can occur and a function that creates a trade valid at one specific date.

Listing 15.25 Implementing derived financial contract functions (F# interactive)

```
> let between dateFrom dateTo contract =
      after dateFrom (until dateTo contract);;
val between : DateTime -> DateTime -> Contract -> Contract

> let tradeAt date ammount what =
      between date date (trade ammount what);;
val tradeAt : DateTime -> int -> string -> Contract
```

The first function is composed from the two primitives that we defined for restricting the date of the contract. It takes the starting date and the ending date and a contract and returns a contract that can happen at any time within the specified interval. The second function creates a primitive trade that can occur only at the precisely specified date. It uses `trade` function to construct elementary trade and then limits it validity using the `between` function. Note that `after` and `until` function use operators that allow equality (>= and <=), so the use of between is reasonable.

Equipped with these functions for creating and composing contracts, let's now try to write some contract and evaluate what trades can occur as part of the contract at two distinct dates. The listing 15.26 shows a contract where we're willing to sell 500 stocks of Google at one particular date and to buy 1000 stocks of Microsoft at any time within the specified 10 days.

Listing 15.26 Creating and evaluating sample contract (F# interactive)

```
> let dfrom, dto = DateTime(2009, 4, 10), DateTime(2009, 4, 20)
  let itstocks =
     combine (give (tradeAt (DateTime(2009, 4, 15)) 500 "GOOG"))     #1
             (between dfrom dto (trade 1000 "MSFT"));;               #1
val itstocks : Contract = CF <fun:trade@6>

> eval itstocks (DateTime(2009, 4, 14));;                            #2
val it : (int * string) list = [(1000, "MSFT")]

> eval itstocks (DateTime(2009, 4, 15));;                            #2
val it : (int * string) list = [(1000, "MSFT"); (-500, "GOOG")]
```

#1 Describe contract using the DSL
#2 Get actual trades at two distinct dates

The listing starts by creating values that represent two dates between which we're willing to purchase Microsoft stocks. Then we define a value `itstocks` that represents our contract. We're using the `combine` primitive to merge two possible trades (#1). The first one is selling of the Google stocks. One way to construct sale is to create a contract that represents a purchase of the stocks (we construct that using the `tradeAt` function that we implemented in the previous listing) and then use the `give` primitive to change purchase into a sale. This way we can create reusable trades and then use them when writing both sales and purchases.

Once we've defined the contract, we can evaluate it. The contract in our language represents a specification of trades that can occur at some specified dates, so we can evaluate it to get the possible trades at some time. As you can see in the listing, for the first date, the result is only a purchase of Microsoft stocks, but for the second date, we'll get both of the trades.

Representing contracts as abstract values

In this example, we represented contracts in a way that is quite similar to how we earlier represented behaviors. We've essentially used a function that calculates the trades and then wrote combinators that compose these functions. This is one of the two basic techniques that I mentioned in the beginning of the chapter.

When working with contracts, we could use the second technique as well and we could design a discriminated union abstractly representing the contract. It would have options that roughly correspond to the basic primitives of the domain specific language:

```
type Contract =
    | Exchange of int * string
    | After of DateTime * Contract
    | Until of DateTime * Contract
    | Combine of Contract * Contract
```

As you can see the type is recursive, so we can compose the elementary value `Exchange` that represents a single trade with other trades using `Combine`, limit their validity using `After` and `Until` and so on.

The difference between these two techniques is that when using abstract value representations, we can write all sorts of processing functions for the language. We could for example easily add a function that takes `Contract` value and evaluates its overall risk and so on. On the other hand, when we use a function type under the hood, we cannot observe many properties of the value once it is created and we can only execute it. In reality, it would be probably better to represent contracts using abstract values, but I wanted to demonstrate how you can use the same technique we've seen earlier in the chapter for creating a language for two different domains.

Clearly, the domain specific language that we've sketched in this section was very limited and simplistic, but it demonstrated that the approach is very powerful and that it can be used for a wide variety of problem domains. It definitely isn't limited to describing animations and financial contracts and I'm sure you already have some ideas how you could use it for solving the problems that you're concerned with.

15.7 Summary

We started the chapter by talking about the language oriented programming style and in particular about various techniques for creating internal domain specific languages. I briefly mentioned techniques like literal expressions that can be used in both F# and C#, fluent interfaces that are particularly useful in C# and combinator libraries which are used in functional programming languages.

Later, we created a language for describing animations. We divided that into two unrelated concepts - behaviors and drawings. We provided a few primitives such as `wiggle`, `time` and `circle` and operations for composing them such as overloaded operators for behaviors or `moveXY` function for drawings. Using these primitives we could compose anything we wanted, so we don't need to know anything about the underlying representation and the user of our library can just think about problems using those simple primitives. Next, we've seen that well designed libraries can be nicely composed, because we could create an animation library just by composing two unrelated concepts - behaviors and drawings. Finally, we also briefly sketched a domain specific library for a completely different problem, which is modeling of financial contracts.

In the next chapter, we'll turn our attention back to asynchronous workflows that we've seen in chapter 13, but we're going to use them differently. We'll look at developing applications that react to external events including events from the user interface. In general, we'll talk about writing *reactive applications* and the F# techniques that we can use.

16 Developing reactive functional programs

In this chapter, we're look at a few techniques for creating user interfaces and dealing with the input from the user or other external events. We'll also discuss one interesting mechanism available in F# that can be used for creating concurrent programs. This sounds like somewhat unrelated topics, but we'll see many similarities. All of the libraries and examples we'll see in this chapter share a similar architecture, so let's first briefly look at the reactive architecture in general.

When implementing imperative or functional application with the usual architecture, the code we write drives the execution of the application and controls what happens in the next step. However, for some problems such as GUI applications, this architecture doesn't work very well. For example a windows application needs to handle a large number of various user interface events; it may need to respond to a completion of asynchronous web service requests or for example to a stat update from some background computation. The execution of this type of applications is controlled by the events and the application is concerned with *reacting* to them. For this reason, this principle is sometimes called *inversion of control* and is sometimes anecdotally referred to as *The Hollywood Principle*[§§§].

The standard .NET way for writing this kind of applications is to use event handlers. However, when using event handlers, we always need some local mutable state, which

[§§§] "Don't call us, we will call you".

means that it is in some way against the functional principles. On the other hand, this is the most straightforward way. We've already seen how to use it in chapters showing some graphical user interface, so we won't spend a long time discussing this programming style. Instead, we'll focus at some of the appealing alternatives that F# gives us.

We will start by looking at the declarative way to handle events, which is somewhat similar to the elegant declarative list processing that we've seen in some of the early chapters. Then we'll look at using asynchronous workflows for event handling, which gives us a way to revert back the inversion of control and again write the code in a way where we control (or at least appear to control) what the application is doing. Finally, we'll look at working with state in an application like that and we'll also briefly look at message passing concurrency, which is a powerful technique for writing multi-threaded applications.

16.1 Reactive programming using events

With no doubt, you already know how to write application that reacts to events in C# and we've seen that the same technique can be used in F# as well. The usual way is register a callback function (or a method) with the event. When the event occurs, the callback function is called and it can react to the event, for example by updating the state of the application or by doing changes in its user interface.

We'll shortly see that there are other ways for handling events, but let's first review the usual style using one example. The code in listing 16.1 monitors changes in the file system using the `FileSystemWatcher` class. Once initialized, the watcher triggers an event every time some file is created, renamed or deleted.

Listing 16.1 Monitoring file system events (F#)

```
open System.IO
let w = new FileSystemWatcher("C:\\Temp", EnableRaisingEvents = true) #A

let isNotHidden(fse:RenamedEventArgs) =                              #1
   let hidden = FileAttributes.Hidden                                #1
   (File.GetAttributes(fse.FullPath) &&& hidden) <> hidden           #1

w.Renamed.Add(fun fse ->                                             #2
   if isNotHidden(fse) then                                          #B
      printfn "%s renamed to %s" fse.OldFullPath fse.FullPath)       #B
```

#A Initialize the watcher
#1 Test attributes of the file
#2 Register the event handler
#B Report only visible files

The listing starts by initializing the `FileSystemWathcher` object and we also set the `EnableRaisingEvents` property during the construction, to activate the monitoring. The next few lines (#1) show a simple function that checks whether a file is not marked as hidden. The argument to this function is a class derived from `EventArgs` that carries information about the event triggered by the watcher.

The last part of the code (#2) registers an event handler that will be called when a file is renamed. In F#, events are represented in a different way than in other .NET languages. In C#, event is a special member of the class and you can work with it only by using one of the operators for adding (+=) or removing (-=) event handlers. On the other hand, in F# events appear as standard members of type `IEvent<'T>` where the `T` parameter specifies the value carried by the event (derived from `EventArgs`). This type has an `Add` method that we can use for registering a callback function. The type representing events also has `AddHandler` and `RemoveHandler` methods, so you can still use delegates if you want to be able to remove the registered callback later.

The example above uses the `Add` method and gives it a lambda function as an argument (#2). The function reacts by printing information about the renamed file, but we don't want to react to every event. Instead we want to display the message only when the affected file is not marked as hidden. To do this, we simply write an `if` condition inside the callback function.

This of course works fine, but as we'll see in the next section, F# allows us to write the filtering of events in a more declarative way, which makes the program easier to read and also gives us better ways for factoring our code. Later we'll see that the same principles can be also to some extent applied in C#.

16.1.1 Introducing event functions

Working with events by directly providing callback function isn't very declarative. We're imperatively adding the event handler and the whole behavior is wrapped inside the callback function, so let's now think how we could write the same thing in a more declarative style. We've seen that one way for making code declarative is to use higher order functions. The best examples are functions for working with lists such as `List.filter`. If we had a list of events from the file system watcher (called `fswList`), we could factor the code into two parts. The first one would filter the events to select only those that we're interested in and the second part would print the information. The first part might look something like this:

```
let renamedVisible =
   fswList |> List.filter isNotHidden
```

The snippet uses the `isNotHidden` function as an argument to the function that filters the list. The second part could use the `List.iter` function to perform printing of every item in the list.

As we'll see in the listing 16.2, we can use exactly the same pattern when working with events. We can think of events in a similar way as we think of lists. Events also carry a sequence of values, with the difference that the values are not available immediately. A new value appears every time the event is triggered. This sequence of event arguments can be filtered in a similar way as collections. We can use `Event.filter` function to create an event that is triggered when the source event produces a value that matches provided predicate (a function returning `bool`).

Listing 16.2 Filtering events using Event.filter function (F# interactive)

```
> let renamedVisible =                                              #1
     w.Renamed |> Event.filter isNotHidden                          #1
val renamedVisible : IEvent<RenamedEventArgs>                       #2

> renamedVisible |> Event.listen (fun fse ->                        #3
     printfn "%s renamed to %s" fse.OldFullPath fse.FullPath)
val it : unit
```

#1 Filter renames of hidden files
#2 Result is a filtered event
#3 Print file name when event occurs

The first command (#1) filters the event in a similar way in which we filtered a list of values. As you can see by looking at the type of the result (#2), the function creates a new event object. The returned event listens to the event of the file system watcher and when a file is renamed, it uses the provided filtering function to test whether the value carried by the event should be ignored or not. If the filtering function returns `false`, the resulting event is triggered, otherwise the current occurrence of the event is ignored.

The next line registers a function that prints information about the renamed file with the filtered event. We're using another function for working with events called `Event.listen`. This function does the same thing as the Add method that we were using earlier, but it allows us to write the whole event processing code in a more uniform way just using higher order functions.

Before we discuss benefits of this programming style, let's look at the table 16.1, which shows several of the most important functions for working with events, including those that we've used in the previous listing. As you can see, many of them very closely correspond to a function for working with sequences.

Event function	Type of the function and description
Event.filter	`('T -> bool) -> IEvent<'TDel,'T> -> IEvent<'T>` Returns event that is triggered only when the source event occurs and when the value carried by the event matches the predicate specified as the first argument. This function corresponds to `List.filter` for lists.
Event.map	`('T -> 'R) -> IEvent<'TDel,'T> -> IEvent<'R>` Returns an event that is triggered every time the source event is triggered. The value carried by the returned event is calculated from the source value using the function given as the first argument. This corresponds to the `List.map` function.
Event.listen	`('T -> unit) -> IEvent<'TDel,'T> -> unit` Registers a callback function for the specified event. The function provided as the first argument is called whenever the event given as the second argument occurs. This function is similar to `List.iter` function for lists.
Event.scan	`('S -> 'T -> 'S) -> 'S -> IEvent<'TDel,'T> ->`

	`IEvent<'S>` This function creates event with internal state. The initial state is given as the second argument and it is updated every time the source event occurs using the function given as the first argument. The returned event reports the accumulated state every time the source event is triggered and state is recomputed.
Event.merge	`IEvent<'TDel1,'T> -> IEvent<'TDel2,'T> -> IEvent<'T>` Creates an event that is triggered when either of the events passed as arguments occurs. Note that the type of the values carried by the events (T) has to be same for both of the events given as arguments.

Table 16.1 Overview of some interesting higher order functions for working with events

The table shows a couple of things that are worth explaining. First of all, the type representing the event used as an input for all the functions is different than the result. The input type has two type parameters. The second one is the value carried by the event and the first one (named `'TDel`) is a .NET delegate used when registering handlers for the event. The result type is a special type of events that is used in F# and uses a generic delegate named `Handler<'T>` that is available in the F# library. This means that the `IEvent<'T>` type is actually just a shortcut for a type `IEvent<Handler<'T>, 'T>`. In F#, we'll use the simplest type most of the time and the version with two type parameters is used only when accessing delegates declared in an existing .NET type.

The `Event.scan` function also deserves an explanation, because it looks a bit more complicated than the others. The signature looks a bit similar to the `List.fold_left` function. Both of the functions take an initial state and a function that knows how to calculate a new state from the original state and an element from the list or value carried by the event. The difference is that the `fold_left` function returns the result of accumulating all the elements of the list. This is of course impossible for events, because we don't know when the event will happen for the last time. So, instead of waiting for the last element, the `Event.scan` function returns an event that is triggered every time the internal state is recalculated. We'll see an example showing how useful the function is in the next section, but let me first return to our previous example for a second.

Probably the larger benefit of using higher order functions for working with events is that we can write the handling in a more declarative way. In the previous listing, we replaced an imperative `if` in the body of the event handler with a declarative filtering, but we can take the example even further. If we create a function that formats the information carried by the `RenamedEventArgs` (called for example `formatFileEvent`) then we can write the whole event handling as a single very succinct expression.

Listing 16.3 Declarative event handling (F#)

```
w.Renamed
   |> Event.filter isNotHidden              #1
   |> Event.map formatFileEvent             #2
   |> Event.listen (printfn "%s")           #3
```

#1 Filter renames of hidden files
#2 Create event carrying formatted strings
#3 Output the carried message

The listing 16.3 implements the same functionality as our first listing, using two helper functions and the functions from the `Event` module. Once we know we can think of event as series of values, the code should be easy to read. Instead of imperatively specifying "what to do" when event occurs, we declaratively specify aspects of the required result. The first line specifies what kind of events we are interested in (#1), the second one specifies what information is important for us (#2) and the last line gives a way for displaying the formatted information (#3).

The declarative style is one of the benefits, but this way of working with events gives us a richer way to factor the code. For example, we could omit the last line to create an event that can be used in several other places of the application. Then we could for example use `Event.listen` with `MessageBox.Show` as an argument to display the notifications in a graphical form. To become more familiar with this concept, we'll look at another slightly more complicated example in the next section.

16.1.2 Creating simple reactive application

Let's now look how we can use the processing function when writing a simple Windows Forms application. The main form of the application is displayed in the figure 16.1 and you can probably already guess what it is supposed to do.

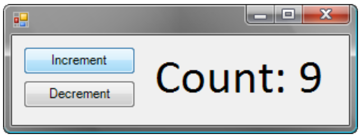


Figure 16.1 The number displayed in label is changed by clicking on the buttons.

If we implemented this application in the usual way, we'd create a mutable field (or mutable ref cell in F#). Then we'd write an event handler that would be called when either of the buttons is clicked. The event handler would test which of the buttons was clicked and it would increment or decrement the mutable state and display it on the label.

Now, how can we implement the same thing using the functions for working with events that we introduced in the previous section? One of the nice things of many declarative libraries is that the code written using them can be very nicely visualized. This is true for events as well, so you can see a diagram demonstrating our solution in figure 16.2.

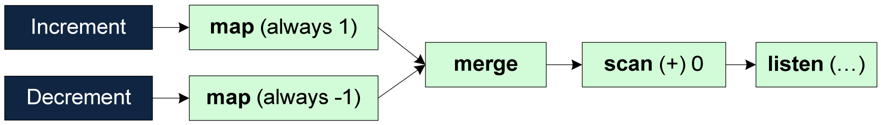


Figure 16.2 Event processing pipeline used in the sample application; boxes on the left represent source events and light boxes represent events created using processing functions.

The idea is that we'll take the click events and turn them into an event that carries an integer value. We'll do this using a helper function named `always`. It returns a function that ignores its argument and always returns the same value. We'll use it to create events that will carry either +1 or -1 depending on which of the buttons was clicked. Then we can merge these two events and use the `Event.scan` function to sum the values carried by the events.

The code needed to build the user interface isn't very interesting, so we'll look only at the part needed to setup the event processing. You can see the code that encodes the pipeline from the previous figure in the listing 16.4.

Listing 16.4 Pipeline for handling events (F#)

```
let always x = (fun _ -> x)                                 #A
let incEvt = (btnUp.Click |> Event.map (always 1))          #1
let decEvt = (btnDown.Click |> Event.map (always -1))       #1

Event.merge incEvt decEvt                                   #2
   |> Event.scan (+) 0                                      #3
   |> Event.listen (fun sum ->
      lbl.Text <- sprintf "Count: %d" sum)                  #B
```

#A Create function that always returns 'x'
#1 Create events that carry +1 or -1 values
#2 Merge the events
#3 Calculate summary of carried values
#B Display the result

To make the code more readable, we don't encode the whole pipeline as a single expression (even though it would be possible). Instead, we first declare two helper values that represent events (#1). The type of both `incEvt` and `decEvt` values is `IEvent<int>`, which means that they represent events. The value carried by the event raised by the "Increment" button is always +1 and the value of the other event is always -1. To generate the value, we're using a function `always` from the previous chapter that returns a function ignoring the argument and returning always the same value. The value that is ignored in the example above is `EventArgs` argument of the `Click` event.

As a next step, we merge these events together to create an event that will be triggered every time either of the buttons is clicked. The event carries integer values, so we can use `Event.scan` to sum the values starting with 0 as an initial value. We're using the plus operator for aggregation, so in every click, the aggregation will add +1 or -1. Finally, we use the `Event.listen` function to specify a handler that displays the current sum of clicks.

The ability to work with events as if they were values of type `IEvent` is a special feature of the F# language, because F# automatically wraps .NET events into this type. Using the same principle in C# is a bit difficult, but it is possible and it nicely demonstrates the power of declarative programming style and LINQ, so we'll look at it at least briefly.

16.1.3 Declarative event processing in C# 3.0

To use events as first-class values in C#, we first need to create our own implementation of the `IEvent<T>` type for C#. This will be an interface containing two methods for doing the usual operations with events - one for attaching and one for removing an event handler. We will not discuss the full implementation in the book and we'll instead use existing project called Reactive LINQ. You can find more information about it in the series of articles starting with article *Introducing Reactive LINQ* [Petricek, 2008]. All source code needed to run the examples from this section is of course available on the book web site.

Let's now look how we could implement the demo with `FileSystemWatcher` in C#. The Reactive LINQ library gives us an `IEvent<T>` type and a couple of extension methods for doing the same things as `Event.filter`, `Event.map` and others. The library follows the standard C# naming, so the corresponding extension methods for working with event values are called `Where` and `Select`.

However, the problem that we have to workaround in C# is that events (such as `watcher.Renamed`) are not first-class values and so they cannot be passed as an argument to a method. This means that we have to first convert them into the `IEvent<T>` representation. The Reactive LINQ library provides a method `Reactive.Attach` that takes the name of the event as a string and creates an event value of type `IEvent`.

I mentioned that the methods for working with events are called `Where` and `Select`. This is very important, because it also means that we can use the syntactic sugar available in C# and use the LINQ query syntax to write the event processing code instead of calling these methods explicitly. The listing 16.5 uses Reactive LINQ to display notification when a visible file gets renamed.

Listing 16.5 Working with events using LINQ (C#)

```
var watcher = new FileSystemWatcher("C:\\Temp") {                        #A
   EnableRaisingEvents = true };
var watcherEvt = Reactive.Attach<RenamedEventArgs>                       #1
   (watcher, "Renamed");

var renamedEvt =
   from fse in watcherEvt                                                #2
   where IsNotHidden(fse)                                                #2
   select String.Format("{0} renamed to {1}",                            #2
      fse.OldFullPath, fse.FullPath);                                    #2

renamedEvt.Listen(Console.WriteLine);                                    #3
```

#A Initialize the file system watcher
#1 Convert event to a value
#2 Filter events and yield string with file names
#3 Print the information when event occurs

After initializing the `FileSystemWatcher` object, we use the `Attach` method to turn the `watcher.Renamed` event into a first-class value (#1) represented using the `IEvent<RenamedEventArgs>` interface. The `Attach` method takes a single type argument that specifies the type of values carried by the event and a single argument which is the name of the event. It uses reflection under the hood, so we have to be careful to specify the name correctly.

Most of the processing code is implemented as a single LINQ query (#2) that uses a single helper method `IsNotHidden` to filter renames of hidden files. The C# compiler translates the query to ordinary calls to `Where` and `Select` extension methods, so there is nothing magical going on. Most of the code directly corresponds to what we've just seen in F#. Finally, the last line uses an extension method `Listen` to register a handler for the filtered event. We're using simply the `Console.WriteLine` method, so the string carried by the event will be printed to the screen.

In the last few sections, we've seen how to create events that are constructed from other events using higher order functions or using LINQ queries. However, we still haven't seen how to declare a new event. In C# this is done using the well known `event` keyword, but the technique used in F# differs, so we'll discuss it in the next section.

16.1.4 Declaring events in F#

When declaring a new event, we need two things. First of all, we need to create `IEvent<'T>` value that we could publish and that others could use for listening to our newly created event. As a second thing, we also need a way to trigger the event. In C#, the event can be triggered using the method invocation syntax, but only from the class where it was declared. When we create a new event in F#, we'll get a function value for triggering it.

Let's look at an example showing how this looks in practice. Probably the most common scenario for working with events is when we need to expose event as a member of some object in a similar way as for example Windows Forms controls. The listing 16.6 shows a simple concrete object type (a class) that exposes one event and one method that sometimes triggers it.

Listing 16.6. Declaring event as a class member (F# interactive)

```
> type Counter() =
     let mutable num = 0
     let ev = new Event<_>()                                          #1

     member x.SignChanged = ev.Publish                                #2
     member x.Add(n) =
        num <- num + n
        if (sign(num - n) <> sign(num)) then
```

```
            ev.Trigger(num);;                                          #3

> let c = Counter()
  c.SignChanged |> Event.listen (printfn "Number: %d");;

> c.Add(10);;
Number: 10                                                             #A
> c.Add(10);;
> c.Add(-30);;
Number: -10                                                            #B
```

#1 Create a new event
#2 Publish the 'IEvent' value
#3 Trigger the event using provided member
#A Sign changed from 0 to 1
#B Sign changed to -1

The Counter class contains a single mutable field that stores the current number. The Add method can be used for changing the state of the object and we want to trigger the event SignChanged when the sign of the stored number changes. When declaring a new event, we use the Event class from the F# library. This object contains a Publish member that returns the corresponding IEvent<'T> value that can be listened to and a Trigger member for running the event.

The previous listing shows the typical way of working with events in a type declaration. We store the instance of the Event class as a local value (#1) and we expose the event value returned by the Publish member as a public member of the class, so that the users can listen to the event, but cannot trigger it. Finally, when the conditions of the event arise, we run it using the Trigger member (#3).

DECLARING C# COMPATIBLE EVENTS

The technique we've used in this section creates events that can be naturally used from F# however they won't appear as standard C# events. First of all, F# uses its own delegate type (Handler<T>). If you want to use some other delegate, you can create the event using a class Event<'TDel, 'T>, which allows you to specify the type of the delegate as the first argument. (…)

In the last few sections, we learned many things about events and we've seen some of the benefits of using events as first-class values. Most notably, the fact that we can use higher order functions for working with events. In the next section, we'll extend our example from the previous chapter with a useful function that will take event value as an argument, to demonstrate how a function like that could be designed and implemented.

16.2 Creating reactive animations

When implementing the library for creating animations in the previous chapter, I wrote that the library is largely influenced by functional reactive programming. However, we focused only on the part that implements animations, so the examples from the previous chapter couldn't react to events such as mouse clicks. Implementing a complete library for functional reactive programming is outside of the scope of this book, but we can look at least at one example that shows the relation between behaviors (from the previous chapter) and events that we discussed in the previous sections. This will also show some of the possibilities that F# gives us by treating events as first-class values.

As you may remember from the previous chapter, behavior is a value that can vary in time. For example an ellipse whose location is changing depending on the time. In this section, we'll create a function named `switch`, which allows us to create behaviors that change when some external event occurs. We'll use it to create an animation that starts as a static image and becomes animating faster every time you click on the form.

16.2.1 Using the switch function

We'll start by looking at the example first and describe the implementation of the `switch` function later. As we've seen repeatedly in the previous chapters, a good way to understand what a function does is to look at its type, so let's examine the type first:

```
val switch : Behavior<'T> -> IEvent<'Del, Behavior<'T>> -> Behavior<'T>
```

The result of the function is a behavior that represents a value of `'T` varying in time. This means that the function somehow constructs a behavior using the first two arguments. The first argument represents an initial behavior. Before the event occurs, the returned behavior will be the same as the one provided as the first argument.

The most interesting thing is the second argument. It is an event that carries values of type `Behavior<'T>`. This means that every time the event is triggered, it will give us a new behavior that we can use instead of the initial behavior (or instead of the previous behavior). Every time the event occurs, the `switch` function will under the hood replace the returned behavior with the one obtained from the event. You may be thinking that event containing a behavior as a value looks a bit complicated. That's probably true, but we'll see shortly that events like this can be constructed quite easily.

If you look at the type of the `switch` function in Visual Studio or F# interactive, it will also print a `when` clause that specifies restriction for the `'Del` type. In particular, it specifies that the delegate type should take the value carried by the event (`Behavior<'T>`) as an argument, so that it's compatible with the event. However, this is just a technical detail that we don't have to worry about, because we'll use only created declared in F#.

Now that we know enough about the `switch` function, let's look how we can use it. The listing 16.7 first creates a simple rotating circle similar to those from the previous chapter. Then it constructs an event that is triggered when the user clicks on the form and carries a new behavior - the same animation running a bit faster. Finally, it uses the `switch` function to construct a behavior that's changing with every click.

Listing 16.7 Animation with changing speed (F#)

```
let af = new AnimationForm(ClientSize = Size(400, 400), Visible=true)
let greenCircle = circle (cns Brushes.OliveDrab) 100.0f.anim          #A
let rotatingCircle = rotate 100.0f 1.0f greenCircle                  #A

let circleEvt =                                                      #1
  af.Click
  |> Event.map (always 0.1f)                                         #2
  |> Event.scan (+) 0.0f                                             #2
  |> Event.map (fun x -> faster x rotatingCircle)                    #3

let init = faster 0.0f rotatingCircle                                #B
af.Animation <- switch init circleEvt                                #4
```

#A Create rotating circle with a constant size
#1 Event carrying behaviors as a value
#2 Adds 0.1 to the initial speed 0.0
#3 Create a new faster animation
#B Initial animation is suspended
#4 Animation that speeds-up with clicks

The listing first creates a standard behavior `rotatingCircle` that represents an animated circle that's rotating using a constant speed. As a next step, it constructs the event that yields new behaviors (#1). We're using the same trick that we used when counting the number of clicks on a button to create an event that will yield a number specifying the speed, which increments with every click. The last call to `Event.map` in the pipeline (#3) turns the event carrying the speed into an event that carries a behavior. It changes the speed of the original rotating circle every time by calling the faster function with the new speed as an argument.

Once we have the event, we can finally use the `switch` function. First we create an initial behavior, which is the circle with the rotation speed set to zero. Then we use this behavior and the event declared earlier to create the final animation (#4). You can see how the final animation looks in the figure 16.3. The figure shows the animation after about 3 and 13 clicks.

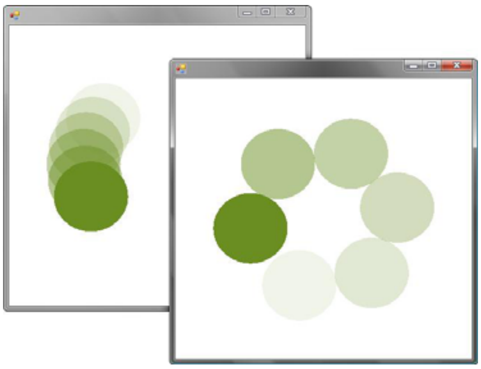


Figure 16.3 Two forms showing the animation running using different speeds after several mouse clicks

Thanks to the combination of first-class events and behaviors, we can write quite interesting animations in a fully declarative way. In the next section, we'll look under the hood and we'll discuss the implementation of the `switch` function.

16.2.2 Implementing switch function

I already sketched how the `switch` function might work in the previous section, so let's now look at the full source in listing 16.8. The key idea is that the function will return a behavior that uses an actual behavior stored in a mutable variable. Every time the event occurs, we'll update the mutable variable, so the returned behavior will start behaving differently. Note that this use of mutable state is completely hidden from the user, so the code that we wrote as an end-user was declarative and free of any visible side-effects.

Listing 16.8 Implementing the switch function (F#)

```
let switch init evt =
   let current = ref init                                         #1
   evt |> Event.listen (fun arg -> current := arg)                #2
   sample(fun ctx ->
      let (BH(f)) = !current                                      #3
      f(ctx))                                                     #3
```

#1 Store the actual behavior in a ref cell
#2 Update the behavior
#3 Get the current behavior and run it

The function first declares a mutable variable (using the F# ref cells that we discussed in chapter 8). The initial value of the ref cell is set to the initial behavior (#1). Next, we setup a handler for the event that can yield a new behavior (#2). When the event occurs, we set the value of the ref cell to the new behavior that we obtained from the event. We don't worry about the thread safety in this example, because when we use the switch function only with Windows Forms events, the state will be always accessed only from the (single) GUI thread. Finally, the behavior that's returned from the function is constructed using the `sample` primitive from the previous chapter. When the lambda function gets called to get the value of the behavior at the specified time, we simply dereference the current behavior and use it to process the request.

The functions from `Event` module are useful if the logic of the event handling isn't very complicated. If the reaction to an event is always the same and if you need to filter the event or combine it with other events, then the declarative style is very useful. However, describing more complex logic declaratively using events may not be easily possible. In the next section we'll look at another technique that uses asynchronous workflows from chapter 13 for handling of GUI events.

16.2 Programming user interface using workflows

When designing applications that don't react to external events, we have rich ways for describing the control flow of the application, such as `if-then-else` expressions, `for` loops and `while` loops in imperative languages or recursion and higher order functions in functional languages. Constructs like this make it very easy to describe what the application does. The control flow is clearly visible in the source code, so drawing a flowchart that describes it is a straightforward task.

Unfortunately, understanding reactive applications is much more difficult. A usual C# application or GUI control that needs to react to multiple events has some mutable state and when an event occurs, it updates the state and perhaps runs some action in response to the event, depending on the current state. In this encoding, it is quite difficult to understand what the states of the application and transitions between them are. Using asynchronous workflows, we can write the code in a way that makes the control flow of the application visible even for reactive applications.

16.2.1 Waiting for events asynchronously

The reason why we cannot use standard control flow constructs to drive reactive applications is that we don't have any way for waiting for an event to occur. Writing a function that runs in a loop and checks whether an event has occurred is difficult to implement, but more importantly, it is also a bad practice, because it would block the executing thread. As we've seen in chapter 13, asynchronous workflows allow us to write code that looks like sequential, can contain waiting for external events (such as completion of an asynchronous I/O operation), but is executed asynchronously without blocking the thread.

So far, we've seen only asynchronous methods that perform I/O operations, but there is also a primitive that stops the asynchronous workflow and resumes it when the specified event (of type `IEvent<'Del, 'T>`) occurs. The primitive is called `AwaitEvent` and is available as a member of the `Async` type. Currently, the primitive isn't a part of the F# libraries, so you can find it the full source code on the book web site. Let's start by looking at the type signature of the primitive:

```
val AwaitEvent : IEvent<'Del, 'T> -> Async<'T>
```

The type shows us that the function is quite simple. It takes event as an argument and returns a value that we can use inside an asynchronous workflow using the `let!` keyword. One important difference between events and `Async<'T>` values is that asynchronous workflow can be executed at most once, while events can be triggered multiple times. This means that the `AwaitEvent` function waits for the *first occurrence* of the event and then resumes the asynchronous workflow. Let's now look how to use this function in practice.

COUNTING MOUSE CLICKS

We'll start by looking at simple example that's similar to what we've seen in the beginning of the chapter and we'll implement a demo that counts the number of clicks and displays it on a label. This could be implemented using `Event.scan` and the source code would be shorter, but as we'll see later `AwaitEvent` is a far more powerful construct. You can see the source code in the listing 16.9.

Listing 16.9 Counting clicks using asynchronous workflows (F#)

```
let frm, lbl = new Form(...), new Label(...)          #A

let rec loop(count) = async {                         #1
  let! args = Async.AwaitEvent(lbl.MouseDown)         #2
  lbl.Text <- sprintf "Clicks: %d" count
  return! loop(count + 1) }                           #3

do
   Async.Spawn(loop(1))                               #4
   Application.Run(frm)
```

#A Create the user interface (omitted)
#1 'Infinite' asynchronous loop
#2 Wait for the next click
#3 Loop with incremented count
#4 Start the loop without blocking

The essential part of the application that implements the counting is a single recursive function that's implemented as an asynchronous workflow (#1). The function appears to create an infinite loop, which may look suspicious for the first time. However, the construct is completely valid, because it starts by waiting for a `MouseDown` event (#2). This is done asynchronously, which means that the workflow will just install the event handler and the rest of it will be executed when you click on the label.

In the introduction, I wrote that the `AwaitEvent` primitive waits only for the first event, because asynchronous workflows can yield only a single value. As you can see in this example, if we want to handle every occurrence of the event, we can simply use recursive loop to setup the waiting again for the next occurrence. In addition, the `loop` function allows us to store the current state as parameters of the function. In fact, this way of expressing computations is very similar to primitive recursive functions that we've seen in the first sections of the book.

As I wrote earlier, the example we've just seen could be easily implemented using the `Event.scan` function, so let's look at a slightly more complicated problem now.

LIMITING THE SPEED OF CLICKS

Let's say that we'd like to limit the rate of clicks. For example, we want the count to stay the same at least for one second after it gets incremented by clicking on the label. One way for implementing this is to add another parameter to the `loop` function of type `DateTime` that will store the last time of a successful click. When the event occurs inside the loop, we could then check the difference from the current time and the last time and increase the count only when the difference is larger than the limit.

However, there is a much simpler way. In chapter 13 we implemented a primitive `Async.Sleep` that allows us to stop the workflow for a specified time. If we use it somewhere in the loop function, it will sleep for one second before reacting to the next event, which is exactly what we wanted. The method for sleeping the workflow for is also available in the F# library as an extension method for the `Thread` class (located in the

`System.Threading` namespace), so everything we have to do is to add the following line before the line that last line that runs the recursion:

```
    do! Thread.AsyncSleep(1000)
```

This is already something that would be quite difficult to do using the functions from the `Event` module. Just for curiosity, you can find the solution using `Event` functions in the online source code and it's about 8 lines long and a bit tricky to understand. However, the control flow of this example was still pretty simple. In the next section, we'll look at a more sophisticated example that better demonstrates the capabilities of using asynchronous workflows for GUI programming.

16.2.2 Drawing rectangles

A problem that's surprisingly difficult to solve in a functional way in F# is drawing of graphical objects on a Windows Forms form. When drawing a rectangle, the user starts by pressing the mouse button in one of the corners, then moves the cursor to the opposite corner and then releases the button. While moving the cursor with the button pressed, the application should draw the current shape of the rectangle and when the button is released it should be finally applied to a bitmap or stored in the list of vector shapes.

The usual implementation would use a mutable flag specifying whether we're currently drawing and a mutable variable for storing the last location specifying where the user pressed the mouse button. Then we'd handle `MouseDown`, `MouseUp` and `MouseMove` events and appropriately modify the state when one of them fires. However, if we think of the control flow of the application, we can see that it's actually quite simple. You can see a flowchart that shows it in the figure 16.4.

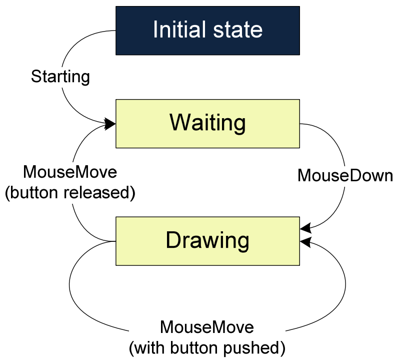


Figure 16.4 When the application is 'Waiting' we can press button to start 'Drawing'. In this state, we can either continue 'Drawing' by moving the mouse or complete the task and change the state of the application back to 'Waiting' by releasing the button.

Before we'll look at encoding of the state machine from the figure above into actual F# program using asynchronous workflows, we'll need to write a single utility function to make the application complete.

IMPLEMENTING PROGRAM FUNDAMENTALS

We'll improve the application a little bit later, but let's start with just an empty form on which we can draw rectangles. The code in listing 16.10 shows the code necessary to create the form and a function `drawRectangle` that draws a rectangle on the form using the specified color and two of any corner points of the rectangle.

Listing 16.10 Creating user interface and drawing utility (F#)

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

let form = new Form(ClientSize=Size(800, 600))

let drawRectangle(clr, (x1, y1), (x2, y2)) =                      #1
   use gr = form.CreateGraphics()
   use br = new SolidBrush(clr)
   let left, top = min x1 x2, min y1 y2                          #A
   let width, height = abs(x1 - x2), abs(y1 - y2)               #A
   gr.FillRectangle(Brushes.White, form.ClientRectangle)         #B
   gr.FillRectangle(br, Rectangle(left, top, width, height))
```

#1 Points are represented as tuples
#A Calculate upper left and lower right point
#B Clear the window using white color

The code in the listing is very straightforward. It is worth commenting that the function `drawRectangle` takes all the parameters as a tuple, so it can be used in a way that's consistent with calling .NET methods. In addition, its second and third parameters are nested tuples that represent X and Y coordinates of the point. As we'll see shortly, this makes the rest of the code a bit easier.

IMPLEMENTING THE DRAWING STATE MACHINE

Now that we have all the basics of the application, we can implement the drawing of rectangles. As we've seen in figure 16.4, the process can be represented as a state machine with two states ('Waiting' and 'Drawing') that have various transitions between them. When programming using asynchronous workflows, we can use a direct translation and create a single function for each of the states. The transitions between them can be encoded as function calls or as returning of a value from a function.

For our example this means that we'll have two functions called `drawingLoop` and `waitingLoop`. The first of them also needs to remember some state, which is done by passing parameters to the function. You can see the full source code that implements the drawing in listing 16.11.

Listing 16.11 Workflow for drawing rectangles (F#)

```
let rec drawingLoop(clr, from) = async {
   let! move = Async.AwaitEvent(form.MouseMove)                        #1
   if (move.Button &&& MouseButtons.Left) = MouseButtons.Left then
      drawRectangle(clr, from, (move.X, move.Y))                       #2
      return! drawingLoop(clr, from)                                   #2
   else
      return (move.X, move.Y) }                                        #3

let waitingLoop() = async {
   while true do                                                       #4
      let! down = Async.AwaitEvent(form.MouseDown)
      let downPos = (down.X, down.Y)
      if (down.Button &&& MouseButtons.Left) = MouseButtons.Left then
         let! upPos = drawingLoop(Color.IndianRed, downPos)            #5
         do printfn "Drawn rectanlge (%A, %A)" downPos upPos }
```

#A Wait for the next mouse action
#2 Refresh rectangle and continue in the 'Drawing' state
#3 Return end location to the 'Waiting' state
#4 Repeat after drawing finishes
#5 Transition to the 'Drawing' state

The most direct way to encode the state machine would be to use only recursive calls between the two functions using the `return!` keyword. In the listing above, we did a minor change to this encoding, which makes the code a bit more readable. The `waitingLoop` function contains an infinite `while` loop (#4) that waits until the user pushes the left button and then transfers the control to the `drawingLoop` function. When it completes, it returns the end position of the rectangle (#3) and transfers the control back (#5). We can then print the information about the drawn rectangle and wait for another mouse-down event.

On the other hand, the function that's running while the user is drawing a rectangle is looping using recursive calls. It starts by waiting for the `MouseMove` event, which is also called when the button is released (#1). Then it tests whether the button is currently pressed and when that's the case, it refreshes the view of the form (#2). This transition is represented as the arc looping in the 'Drawing' state. When the button is released, it returns the last location as a result (#2), which is the transition back to the 'Waiting' state.

That's almost everything we need to run the application. The only remaining thing is to start the asynchronous workflow that handles drawing of rectangles and run the application:

```
[<STAThread>]
do Async.Spawn(waitingLoop())
   Application.Run(form)
```

In this simple application, we have only a single asynchronous workflow that handles all the interaction with the application. If we for example wanted to allow drawing of polygons by using the right mouse button, we could implement this without doing any changes to the code we wrote now. We would simply create another workflow for drawing of polygons and start it independently using `Async.Spawn`. This way of writing the user interface code gives us a very modular way for factoring the complex interactions into separate processes.

Waiting for events and the GUI thread

The application we just implemented consists of a single running process, but it is important to realize that a process in the sense we're using doesn't correspond to a thread. In fact, even if we had multiple processes waiting for GUI events, the application would still be single threaded.

We already discussed how asynchronous workflows work in chapter 13, but let's just briefly repeat the important point. When the workflow is waiting for an asynchronous operation it doesn't occupy any thread. Instead, it just registers a callback that will resume the workflow once the asynchronous operation completes. This means that the workflow will be executed on the thread which is used by the asynchronous operation to report that it completed. For I/O events this is a thread from the thread pool, however for GUI events, this is the GUI thread.

In .NET applications (and in any Windows GUI applications in general), there is a single GUI thread that is used for processing all the incoming user interface events. For .NET this means that all the GUI events are triggered on this single thread. What does this mean for the example we just implemented? Because the only asynchronous operation we're using, the workflow will always run on the single GUI thread. This means that the technique we're using doesn't introduce any parallelism. It just gives us an easier way to write our single threaded GUI processing.

We'll shortly see a technique that allows us to integrate this form of GUI processing with other processes that can possibly run in parallel however the usual code for user interface interaction like the one we've just seen should be simple and shouldn't perform any complicated computations, so there is no need for parallelism.

The code we wrote so far isn't really a drawing application, because it doesn't store the rectangles we draw in any way. Once the drawing is finished, it just prints some information to the console and forgets the rectangle. We could store a list of rectangles as a parameter of the `waitingLoop` function (if we changed it into a recursive function), but that wouldn't work very well, because the list would be private to the drawing loop and it couldn't be accessed from other parts of the application. To store the state that's global for the whole application, we need something better.

16.3 Storing state in reactive applications

The code we wrote in the previous section to handle drawing of rectangles can be viewed as a lightweight process that runs inside the application to handle certain task. In this chapter, the task was a GUI interaction, but it could as well perform asynchronous I/O operations as we've seen in chapter 13, such as download content of a web site. As I wrote earlier, there can be multiple processes like this running in parallel. Keep in mind that a process doesn't correspond to a thread, but some of the processes might be running in parallel.

Structuring the code as processes allows us to nicely factor the code, but we haven't discussed one essential aspect yet, which is how these processes can communicate. We could of course use some global mutable variables, but that requires careful use of locking and it's generally discouraged in the functional programming. The technique we can use instead is called *message passing*. When writing application using message passing, the processes can send messages to each other and exchange all the needed state just by sending or replying to messages.

16.3.1 Creating mailbox processor

Let's now look what this means in practice. We'll extend the application from the previous section and we'll add a process that will store the current state of the application, which is the currently selected color (we'll add an option to change the color) and a list of all the created rectangles. It will handle messages that will be sent from the process for drawing rectangles or from other event handlers that we'll add to the application. In F#, the processes that can receive messages are also called *mailbox processors*. However, before we can start implementing the mailbox processor, we'll need to know what a message is.

IMPLEMENTING THE MESSAGE TYPE

Each process can handle messages of a single known type, so we'll start by declaring the type that represents message. As you can see in listing 16.12, discriminated union is the right F# type for this purpose.

Listing 16.12 The type representing messages (F#)

```
type RectData = Color * (int * int) * (int * int)                  #1

type DrawingMessage =
   | AddRectangle of RectData                                       #2
   | SetColor of Color                                              #2
   | GetRectangles of AsyncReplyChannel<list<RectData>>             #3
   | GetColor of AsyncReplyChannel<Color>                           #3
```

#1 Type alias for rectangle
#2 Messages for updating the state
#3 Messages for reading the state

The listing starts by declaring a type alias called `RectData` (#1) which is a tuple containing all information that we want to store about rectangles. The discriminated union itself then contains two types of messages. The first two messages (#2) are used for setting the current state and they carry the arguments for this operation. In case of `AddRectangle`, the processor will add the information about the newly created rectangle to an internal list and in case of `SetColor`, it will change the current color.

The next two messages (#3) look a bit trickier, because the value they carry has a type `AsyncReplyChannel<'T>`. This type allows us to create messages that send a reply back to the caller. In our case, it means that when the process receives one of these messages, it will send a reply back containing either a list of all rectangles in case of `GetRectangles` or the currently selected color in case of `GetColor`. We'll look at sending messages and waiting for the reply shortly, but let's start by implementing the mailbox processor.

IMPLEMENTING THE PROCESSOR

In general, mailbox processors can be quite complicated. They can perform various calculations in reaction to the messages they receive; they can send messages to other processors and collect the replies, or they can even start new processors. However, the mailbox processor in our example is very simple and only stores the current state of the application and handles the messages to read or update the state.

The code, which you can see in listing 16.13, follows the same pattern as the code we wrote earlier. It is implemented as a recursive function written using asynchronous workflows that maintains the current state using function parameters.

Listing 16.13 Creating the mailbox processor (F#)

```
let state = MailboxProcessor.Start(fun mbox ->                        #1
   let rec loop(clr, rects) = async {
      let! msg = mbox.Receive()                                        #2
      match msg with
      | SetColor(newClr) ->
         return! loop(newClr, rects)                                   #3
      | AddRectangle(newRc) ->
         form.Invalidate()
         return! loop(clr, rects@[newRc])                              #3
      | GetColor(chnl) ->
         chnl.Reply(clr)                                               #4
         return! loop(clr, rects)
      | GetRectangles(chnl) ->
         chnl.Reply(rects)                                             #4
         return! loop(clr, rects) }
   loop(Color.IndianRed, []) )                                         #5
```

#3 and #5 appear intentionally two times; is it possible to do this?

#1 Starts by running the given function
#2 Asynchronously wait for the next message
#3 Update the state during recursive call
#4 Return the current state
#5 Start with initial color and an empty list

To create a mailbox processor, we use the `Start` member of the `MailboxProcessor` type. It initializes the mailbox for the messages and then runs the provided function (#1) to start the processing. The function returns an asynchronous workflow that can wait for messages using the `Receive` method (#2) of the mailbox that we get as an argument during the initialization.

We implemented the workflow using a recursive function called loop that takes two parameters. The parameter `clr` is the currently selected color and `rects` is a list of rectangles. When returning the workflow from the lambda function, we call the `loop` function with a red color and an empty list as the initial state (#5).

Now, let's have a look at the body of the `loop` function. It starts by receiving the next message from the mailbox (#2). The mailbox internally stores a queue with messages, so if a message is already in the queue, the message will be returned immediately. On the other hand, if the queue is empty, the `Receive` method will block the workflow (without blocking the actual thread) and resume it once a message is sent to the processor. Once we receive a message, we use pattern matching to decide how to handle it. The first two messages just modify the state of the processor, so we recursively call the `loop` function using the `return!` keyword (#3) with the updated state. Note that when we get a new rectangle, we want to add it to the end of the list to make sure that it will be displayed on the top, so we use the @ operator for concatenating lists.

The last two messages (#4) are used for reading the state of the processor and they carry a reply channel as an argument. When the processor receives the message, it uses the `Reply` method of the channel to send the list of rectangles or the current color back as a result to the caller and then loops without altering the state.

MAILBOX PROCESSORS AND CONCURRENCY

When writing mailbox processors, it is important to understand how they're executed with respect to threads. The thread that's executing the body can change when the workflow waits for some asynchronous operation, but the body will never run on multiple threads concurrently. When a message is received during processing of the previous message, it is queued for later processing. The code we just wrote doesn't perform any complicated computations, so it will almost always process the message immediately. Thanks to this design decision we don't have to concern about any possible race conditions.

Now that we have the mailbox processor ready, we can look how to modify the rest of the application, to use and update the state stored in the processor by sending messages.

16.3.2 Communicating using messages

In the last listing of the previous section we created a mailbox processor called `state` which has a type `MailboxProcessor<DrawingMessage>`. Note that the `Start` method that we used to create it was a member of a non-generic class `MailboxProcessor`, which has a same name, but is overloaded by the number of type parameters. Before we start looking at more code, let's quickly look at the table 16.2, which shows some of the important instance methods that we can invoke on the of the mailbox processor.

Event function	Type of the function and description
Post	Sends a message to the mailbox processor without waiting until for any reply. If the mailbox processor is busy, the message is stored in the queue.
PostAndReply	Sends a message that expects `AsyncReplyChannel<'T>` as an argument to the mailbox processor and blocks the calling thread until the mailbox processor invokes the `Reply` method of the channel. Then it returns the value

	sent to the channel.
AsyncPostAndReply	Similar to the `PostAndReply` with the exception that it runs asynchronously. When we invoke it from an asynchronous workflow using `let!`, it doesn't block the calling thread and the result is returned asynchronously.
Receive	We used this method when creating the mailbox processor to asynchronously receive the next message from the queue, so that we can process it inside workflow. This method shouldn't be used outside of the mailbox processor.
Scan	Similarly to `Receive`, this method shouldn't be used outside of the mailbox processor. It can be used when the processor is in a state when it cannot process all types of messages, because it allows us to return `None` for messages that cannot be processed. The unprocessed messages remain in the queue for later processing.

Table 16.2 The most important methods provided by the MailboxProcessor<'Msg> type.

Note that the `Scan` and `Receive` methods should be used only from the code running inside mailbox processor. We've seen how to use `Receive` in the previous section and we'll talk about the `Scan` method briefly later in this chapter. The remaining 3 methods can be used from any thread. Sometimes you may want to write a processor that sends a message to itself, but a more typical scenario, which we'll see shortly, is when we're sending messages to the processor from outside.

IMPROVING THE DRAWING PROCESS

Let's now look at the changes that we need to do to the drawing process if we want to allow the user to change the color of rectangles before drawing and if we want to keep all the drawn rectangles on the screen. The first thing we have change a bit is the code for drawing. The drawRectangle function originally erased the screen, which isn't desirable if we want to draw multiple rectangles. After changing this behavior, we can implement a function in listing 16.14 that draws all rectangles in the given list.

Listing 16.14 Utility function for drawing rectanlges (F#)

```
let redrawWindow(rectangles) =
   use gr = form.CreateGraphics()
   gr.FillRectangle(Brushes.White, form.ClientRectangle)
   for r in rectangles do
      drawRectangle(r)
```

The function clears the content of the form and then iterates over all the elements of the given list and draws the individual rectangles using the `drawRectangle` function. Note that the list stores rectangles as tuple with three elements (color and two opposite corners), which is compatible with the tuple expected by the `drawRectangle` function.

Now we're finally ready to modify the process that handles drawing of rectangles. As the whole code is implemented as an asynchronous workflow, we can use the asynchronous method `AsyncPostAndReply` when we need to get some information from the mailbox processor that stores the state. This is of course the preferred option when possible, because it doesn't block the calling thread. Most of the code in listing 16.15 stays the same, so I highlighted the lines that have changed.

Listing 16.15 Changes in the drawing process (F#)

```
let rec drawingLoop(clr, from) = async {
   let! move = Async.AwaitEvent(form.MouseMove)
   if (move.Button &&& MouseButtons.Left) = MouseButtons.Left then
      let! rects = state.AsyncPostAndReply(GetRectangles)              #1
      redrawWindow(rects)                                              #2
      drawRectangle(clr, from, (move.X, move.Y))                       #2
      return! drawingLoop(clr, from)
   else
      return (move.X, move.Y) }

let waitingLoop() = async {
   while true do
      let! down = Async.AwaitEvent(form.MouseDown)
      let downPos = (down.X, down.Y)
      if (down.Button &&& MouseButtons.Left) = MouseButtons.Left then
         let! clr = state.AsyncPostAndReply(GetColor)                  #3
         let! upPos = drawingLoop(clr, downPos)
         state.Post(AddRectangle(clr, downPos, upPos)) }               #4
```

#1 Get the list with existing rectangles
#2 Draw all rectangles including the new one
#3 Get the selected color
#4 Add the newly created rectangle

The first change that we have to do is in the `drawingLoop` function when updating the window to show the rectangle that the user is currently drawing. Originally, we only needed to erase the window and draw the new rectangle, but now we also need to draw all the rectangles that exist already. To do this, we obtain the list of rectangles from the mailbox processor by sending it the `GetRectangles` message (#1). The message takes an argument of type `AsyncReplyChannel<'T>` that will be used by the mailbox processor to reply to the caller, but we don't specify the channel explicitly in the code. This is possible, because the F# compiler treats the discriminated union constructor (`GetRectangles`) as a function that takes a single argument. We could write the same thing like this:

```
let! rects = state.AsyncPostAndReply(fun chnl -> GetRectangles(chnl))
```

If we write the code in this way, it is easier to see what is going on. The `AsyncPostAndReply` method creates a channel for the reply and uses the provided lambda function to create the message that carries the channel. The message is then sent to the mailbox processor and the workflow is suspended until a reply is sent to the channel. Once we receive the reply with a list of rectangles, we can draw them including the one that's being drawn right now (#2). Note that the reply can be sent on a background thread.

This isn't a problem, because we're drawing using the `CreateGraphics` method, which doesn't have to be called from the GUI thread.

The second change we did is in the `waitingLoop` function. Once the user starts drawing the rectangle, we first read the currently selected color (#3). The color can be changed from the application user interface (we'll shortly see how), so it is important to get the color after the call to `AwaitEvent` completes. If we placed it before `AwaitEvent`, the user could change the color, but we wouldn't know that because the `AwaitEvent` primitive can block for a very long time. Once we get the color, we can call the `drawingLoop` function to handle the input of a rectangle and finally, we use the `Post` method to send all the information about the newly created rectangle to the mailbox processor (#4).

ADDING THE USER INTERFACE

The user interface of the application will be quite simple, but we'll need to call the mailbox processor from various places to work with the current application state. First of all, we'll add a handler for the `Paint` event, so the application redraws the rectangles when some part of the window is erased by Windows. Secondly, we'll add a toolbar with a single button that allows you to change the current color, so in the end you should be able to create drawings like the one in listing 16.5, which shows the running application.

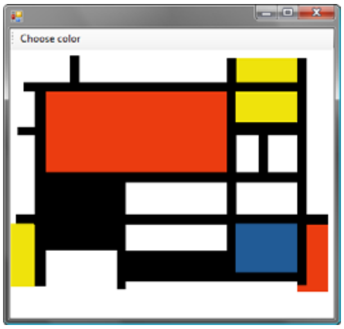


Figure 16.5 Running application with a drawing consisting only of rectangles.

You can find the code that creates the user interface in listing 16.16. The details of the code that creates the toolbar using `ToolStrip` and `ToolStripButton` controls are omitted, but you can find them in the source code available on the book web site.

Listing 16.16 Implementing the user interface (F#)

```
let tools, btnColor = new ToolStrip(...), new ToolStripButton(...)    #A

btnColor.Click.Add(fun _ ->
   use dlg = new ColorDialog()
```

```
      if (dlg.ShowDialog() = DialogResult.OK) then                     #B
         state.Post(SetColor(dlg.Color)) )                             #1

   form.Paint.Add(fun e ->
      let rects = state.PostAndReply(GetRectangles)                    #2
      redrawWindow(rects) )

   [<STAThread>]
   do Async.Spawn(waitingLoop())                                       #C
      Application.Run(form)
```

#A Create the GUI controls
#B Show dialog for color selection
#1 Send the selected color to mailbox processor
#2 Get a list with current rectangles
#C Start the process for drawing rectangles

Most of the code should be fairly straightforward. It creates the user interface and then registers a handler for two events. We don't need to do any filtering or other processing of the events, so we're using directly the `Add` method instead of using functions from the `Event` module. The first handler displays the `ColorDialog`, so the user can select a new color and if a color is selected, it posts a message with the new color to the mailbox processor (#1). We don't need to wait for any reply to this message, so the operation is done without blocking the thread.

The second event handler is for the `Paint` event and it needs to obtain the list with currently displayed rectangles first. To do this, we can use the `PostAndReply` method (#2), which constructs the message with a reply channel and then waits until the mailbox sends a reply. This method blocks the thread, so it should be used only rarely, in cases where we cannot complete the operation asynchronously. Updating the window of the application is definitely one of these situations, so this use is correct.

So far, we've been using the mailbox processor object directly. This is all right in the earlier phase of the development, but once the application becomes larger or if we want to turn a part of the application into a separate library, it is better to encapsulate the mailbox processor in an object. In the next section, we'll look how to do this.

16.3.3 Encapsulating mailbox processors

When encapsulating the mailbox processor, we'll change the global value representing the processor into a local field of an object and we'll add methods that send the messages to the private mailbox. This also has the benefit that we don't have to expose all of the messages if some of them are intended only for an internal use.

When doing a change like this, we don't need to modify the message processing code in any way. You can see the declaration of the concrete object type in the listing 16.17. Because the processing code stays the same, most of it is omitted in the listing.

Listing 16.17 Encapsulating mailbox processor into a type (F#)

```
   type DrawingState() =
      let mbox = MailboxProcessor.Start(fun mbox ->                   #1
         let rec loop(clr, rects) = async {
            let! msg = mbox.Receive()
            // Message processing code
         }
         loop(Color.Black, []) )

      member x.Setcolor(clr) =
         mbox.Post(SetColor(clr))                                 #2
      member x.AddRectangle(rc) =
         mbox.Post(AddRectangle(rc))                              #2
      member x.AsyncGetRectangles() =
         mbox.AsyncPostAndReply(GetRectangles)                    #3
      member x.AsyncGetColor() =
         mbox.AsyncPostAndReply(GetColor)                         #3
      member x.GetRectangles() =
         mbox.PostAndReply(GetRectangles)                         #4

   let state = new DrawingState()
```

#1 Private mailbox processor value
#2 Non-blocking operations without return value
#3 Asynchronous operations for reading the state
#4 Blocking call for obtaining rectangle list

To create a local mailbox processor inside the class declaration, we use a local `let` binding (#1). This becomes a part of the constructor of the class, which means that the mailbox will be started when the instance is created. Values declared using local `let` bindings are turned into local fields, so they are accessible from anywhere inside the class.

The members of the type are mostly boilerplate code. Members that update the state of the mailbox processor and don't wait for any return value (#2) send the message using the `Post` method. The second group of members (#3) that read the state is implemented using the `AsyncPostAndReply` method. Note that we're using the `Async` prefix in the name of these members. This is a standard notation used across the entire F# library to denote members that can be accessed only form asynchronous workflows. Finally, the last method (#4) is the single blocking member of the class.

Once we encapsulate the mailbox processor inside a class, we of course have to modify the rest of the code where it is accessed. Instead of sending a message explicitly, we can simply call one of the methods. The following snippet shows two sample calls from inside of an asynchronous workflow:

```
let! clr = state.AsyncGetColor()
state.Setcolor(clr)
```

We'll look at one more improvement shortly, so you'll see a few examples from the actual application in a second. Once we encapsulate the mailbox processor in a class, it can be compiled into an F# library and distributed as a reusable component. Using methods that return asynchronous workflow (the type `Async<'T>`) from C# is unfortunately difficult so if you want to create a component usable from C#, it is better to also provide methods that take a delegate as an argument and run it when the asynchronous operation completes.

In the next section we'll add one more feature to our application to show another aspect of event handling using `AwaitEvent` primitive inside asynchronous workflows.

16.3.4 Waiting for multiple events

In all the examples of using `AwaitEvent` so far, we've been waiting only for a single event. The rectangle drawing application first waits for the `MouseDown` event and then repeatedly waits for `MouseMove`. However, what if we wanted to wait either for `MouseMove` event or for some other event that could be used to cancel the drawing?

In this section, we'll look at an example showing how to wait for multiple events. We'll keep the existing code and add the ability to cancel the drawing process. When the user hits the `Esc` key, we'll stop the `drawingLoop` returning `None` as the result. To do this, we need to wait for the `MouseMove` or the `KeyDown` event and handle the one that occurs first. You can find the modified code for the `drawingLoop` function in the listing 16.18.

Listing 16.18 Drawing rectangle with cancelation using 'Esc' key (F#)

```
let rec drawingLoop(clr, from) = async {
   let! args = Async.AwaitEvent(form.MouseMove, form.KeyDown)            #1
   match args with
   | Choice1Of2(move) when                                               #2
         (move.Button &&& MouseButtons.Left) = MouseButtons.Left ->      #2
      let! rects = state.AsyncGetRectangles()                            #A
      redrawWindow(rects)
      drawRectangle(clr, from, (move.X, move.Y))
      return! drawingLoop(clr, from)
   | Choice1Of2(move) ->                                                 #3
      return Some(move.X, move.Y)
   | Choice2Of2(key) when key.KeyCode = Keys.Escape ->                   #4
      form.Invalidate();
      return None
   | _ -> return! drawingLoop(clr, from) }                               #5
```

#1 Wait for any of the specified events
#2 Continue drawing
#A Obtain list of rectangles using method
#3 Button released, return the rectangle
#4 Esc key pressed, return 'None'
#5 Otherwise wait for another event

In all the previous examples, we used the `AwaitEvent` method only with a single event as an argument. However, the method is overloaded and allows us to specify multiple events. In that case, the method will wait until the first of the provided events occur and it will ignore any other occurrences. In our case, this means that the call (#1) will block until either a mouse is moved or a key is pressed and then it will run the processing code. If the processing ends with a recursive call, then the `AwaitEvent` will be called again to wait for the next event, but in other case, the next occurrences will be ignored.

When the `AwaitEvent` returns, we want to know which of the events occurred first and what argument it carried. Also, the values carried by the events can be different for all the provided events. In this situation, the method cannot simply return the carried argument, so let's look what is the type of the returned value. The F# library contains a generic discriminated union type `Choice`, which can represent one of several choices. The type is overloaded by the number of type parameters. In the example above, we have two different choices, so the type of the `args` value is `Choice<MouseEventArgs, KeyEventArgs>`.

When the `MouseMove` event occurs first, the returned value will use the discriminated union constructor `Choice1Of2` carrying information about the mouse event, otherwise the constructor `Choice2Of2` will be used with a value of type `KeyEventArgs`. When waiting for multiple events, the names of the cases would be `Choice1Of3` and so on.

The code that chooses how to react to the event when waiting for multiple events can be nicely written using pattern matching. The first branch (#2) is called when mouse moves while still holding the button pressed. In that case we update the window and continue drawing. If the mouse moves and the button is no longer pressed, the next case (#3) will be called. This means that the user finished drawing, so we can return the end location of the rectangle.

Finally, the last two cases specify reaction to the `KeyDown` event. We're again using the when clause to determine whether the pressed key is the `Esc` key. If that's the case, we cancel the drawing process and return `None` as the result, otherwise we ignore the keyboard event and continue waiting for another event. Note that we changed the return type of the function. Previously it was `Async<int * int>`, which is an asynchronous workflow returning a location, but now that we return either `Some` or `None`, the return type is `Async<option<int * int>>`. This means that we'll also have to do a minor adjustment to the `waitingLoop` function, so that it sends the `AddRectangle` message only when a rectangle is actually drawn. This is quite a simple change, so you can find it in the full source code on the book web site.

We started this section by discussing how to use mailbox processor to store the state of the application in a scenario where we need to handle various events. In all the examples, we limited ourselves only to events coming from the user interface. However, an important feature of mailbox processors is that they can be also used in scenarios involving concurrency. We'll briefly take a look at this topic in the next section.

16.4 Message passing concurrency

When talking about the development of concurrent programs in chapter 14, we focused mostly on techniques where we avoid using mutable state. Without mutable state, we can then run several parts of a computation in parallel, because they cannot interfere with each other. This works very well for many data processing problems that can be implemented in functional way, but there are also problems where the processes need to exchange information more frequently.

The most widely known solution is using the *shared memory* and protecting the access to the shared state using locks. The problem with this technique is that using locks correctly is quite difficult. You have to make sure that all the shared memory is properly locked (to avoid *race conditions* when multiple threads write to the same location). Another difficulty is that when acquiring locks not carefully we can cause a *deadlock*, which means that two threads become blocked, waiting for the other to complete, and can never resume.

The MailboxProcessor<'Msg> type in F# can be used for implementing concurrent programs using so called *message passing* concurrency. This approach isn't as widely known, but has been successfully used in a functional language called Erlang [Armstrong, 1996]. We've seen this approach already when storing the state of our rectangle drawing application, but we haven't in detail discussed how the technique can be used in a truly concurrent scenario.

In this section, we'll look at using mailbox processor from multiple threads to demonstrate this approach. We'll use an example with a single mailbox processor and multiple asynchronous workflows (running on multiple threads) that access it. More sophisticated programs that use message passing concurrency often use multiple mailbox processors that communicate with each other.

16.4.1 Creating state machine processor

The mailbox processor we created earlier for storing the state of the rectangle drawing application was quite simple. It was able to process 4 different messages and it maintained some local state, but regardless of the state, it was always able to process any message that it received immediately. However, this may not always be the case. For example, if a single mailbox processor sends a message to two other processors, it may need to collect the replies from these processors before reacting to any other message.

As we'll see, we can write mailbox processors that represent a state machine in a very similar way to what we used when implementing the state machine for handling events when drawing rectangles using asynchronous workflows. Let's first look at the messages that the processor will handle and then we'll talk about its possible states:

```
type Message =
  | ModifyState of int
  | Block
  | Resume
```

The mailbox processor will store an integer value and the ModifyState message can be used for updating it. For simplicity, we don't have any message for reading it and the processor will just print the number to the console every time it is updated. The two other messages are quite interesting. If the process is in the initial state and it receives Block, it stops processing all the ModifyState messages and it waits for Resume. As we'll see shortly, messages that are sent to the processor when it is in the blocked state aren't lost. The processor internally has a queue where the messages are stored, so once we resume it again, it will process all the messages it received while it was blocked.

Let's now look at the listing 16.19, which shows the implementation of the mailbox processor. Similarly to the earlier example, we're encoding the state machine using two recursive functions (using asynchronous workflows) that call each other.

Listing 16.19 Mailbox processor using state machine (F#)

```
let mbox = MailboxProcessor.Start(fun mbox ->
   let rec processing(n) = async {                          #1
      printfn "Processing: %d" n
      let! msg = mbox.Receive()                             #2
      match msg with
      | ModifyState(by) -> return! processing(n + by)
      | Resume -> return! processing(n)
      | Block -> return! blocked(n) }
   and blocked(n) =                                         #3
      printfn "Blocking"
      mbox.Scan(fun msg ->                                  #4
        match msg with
        | Resume -> Some(async {                            #5
            printfn "Resuming"
            return! processing(n) })
        | _ -> None)                                        #6
   processing(0) )
```

#1 Represents the active state
#2 Process any message
#3 Represents the blocked state
#4 Only process the 'Resume' message
#5 Return workflow to continue with
#6 Other messages cannot be processed now

The implementation of the mailbox processor consists of two functions and both of them return an asynchronous workflow. The processor is started by calling the `processing` function (#1) with zero as the initial state. In this state, we can handle all the messages, so we can simply use the `Receive` primitive (#2) that asynchronously returns the next message. If the message is `ModifyState`, then we update the number and continue in the `processing` state. The `Resume` message doesn't make much sense in this state (because we haven't received the `Block` message yet), so we can ignore it. Finally, when we receive the `Block` message, we need to do something to stop processing all messages other than `Resume`, so we call the `blocked` function (#3) that represents the second state.

When the processing is blocked, we have to use the `Scan` primitive (#4), because it allows us to specify what messages we can handle and what messages should remain in the queue for later processing. The `Scan` member takes a function as an argument and the function specifies what to do when a message is received. In our example, when the message is `Resume`, we return an asynchronous workflow (#5) that the `Scan` member will run. The workflow prints a message to the console and then continues by executing the `processing` function and switching back to the active state. When the processor receives any other message in the `blocked` state, the `Scan` primitive will run the provided lambda function and will get `None` as the result. This means that it cannot process the message, so it adds the message to the queue and waits for another one.

Note that the mailbox processor as we implemented it doesn't work well in the case when we have multiple threads sending the `Block` and `Resume` messages, because if it receives a `Block` message when it's already blocked, it doesn't handle it and instead continues processing once it receives the first `Resume` message. To solve this more properly, we'd have to handle `Block` messages in the blocked state and increment some number representing the count of `Block` messages. The `Resume` message would decrement it and we'd resume the processing only after the number reached zero again. However, this won't be a problem in the example we'll look at now, because we'll create only a single thread that will repeatedly block and resume the processor.

16.4.2 Accessing mailbox concurrently

The mailbox processor handles only a single message at time, but it can be safely accessed from multiple threads. All the methods for posting message to the processor (such as `Post` and `PostAndReply`) are thread-safe. Let's now look at an example showing how we can use the mailbox processor we just implemented from three different threads.

In the listing 16.20 we create two threads that repeatedly perform some computation and once they finish computing, they send a state update to the mailbox processor (in our simplified example the threads will just sleep for some time and then generate a random number). Next, we create a single thread that repeatedly sends the `Block` and `Resume` messages to the processor.

Listing 16.20 Sending messages from multiple threads (F#)

```
let modifyThread() =                                             #A
   let rnd = new Random()
   while true do
      Thread.Sleep(500)
      mbox.Post(ModifyState(rnd.Next(11) - 5))                   #1

let blockThread() =
   while true do
      Thread.Sleep(2000)
      mbox.Post(Block)                                           #2
      Thread.Sleep(1000)                                         #2
      mbox.Post(Resume)                                          #2

for proc in [ blockThread; modifyThread; modifyThread ] do
   Async.Spawn(async { proc() })
```

#A Thread performing calculations
#1 Send an update to the mailbox
#2 Block the processing for one second

The code for the threads is quite simple. Both of them contain an infinite loop that would perform some computation in a real application and both of them occasionally send messages to the mailbox to synchronize. The first function uses only the `ModifyState` message and the second one first sends the message to block the thread, then waits for some time and then unblocks it (#2). Finally, we're using the `Async.Spawn` method to start executing the functions in a thread pool threads. We create a list of function values representing the processes to run and then start each of them in a `for` loop. Note that the list contains two times the `modifyThread` function, so we'll have two threads sending updates to the state.

Let's now briefly analyze the behavior of the application when we execute it (either as a standalone application or in F# interactive). It will start by processing the incoming `ModifyState` messages for about 2 seconds. Then the blocking thread sends the `Block` message, so nothing will happen for the next 1 second. After that, the mailbox processor will be resumed and it'll process all the queued `ModifyState` messages, so it'll almost immediately update the state several times. Then it'll continue running, processing messages as they arrive for the next 2 seconds until the next `Block` message is received.

Even though this example doesn't implement any particularly useful behavior, it should give you a pretty good idea how to use mailbox processors in a real-world application that needs to synchronize the state using message passing concurrency.

16.5 Summary

In this chapter, we covered various aspects of development of reactive applications in the functional style. We started by talking about first-class events in F#, which is the ability to use event as a standard value that can be passed as an argument or returned from a function. This allows us to use higher order functions (such as `Event.filter` or `Event.map`) when writing code that processes events, which in turn makes the code more declarative in a same way as processing of lists using higher order functions or LINQ queries. The relation with LINQ is quite interesting and we briefly mentioned that in principle, we could use LINQ queries in C# for event processing as well.

However, for more dynamic types of behavior, the declarative programming using higher order functions doesn't work that well, so we looked at another technique. We've seen that we can use workflows introduced chapter 13 for asynchronously waiting until an event occurs, which allows us to write complex event handling without the inversion of control, which means that the control flow is managed by our application. This also makes it much easier to encode control structures where the process can transition between several states, because the code directly corresponds to a state machine diagram that you may draw.

Finally, we faced the problem how to store state in an application that is encoded asynchronous workflows that handle GUI events. We've seen that this can be done by using message passing techniques and we introduced the `MailboxProcessor<'T>` type that implements this programming model in F#. This type can be also used in concurrent scenarios, so we wrapped up with an example showing how to use it from multi-threaded application.

Unfortunately, most of the examples we've seen in this chapter rely on asynchronous workflows, so they can't be directly implemented in C#. In the chapter 13 I mentioned various projects that attempt to bring similar concepts to C# such as the Concurrency and Coordination Runtime [Richter, 2006], but none of them provides the same clarity as F#. The message passing concurrency techniques from the later of the chapter exist in many different forms. The implementation that's available in F# is very close to the Erlang style message passing (see for example Concurrent Programming in Erlang [Armstrong, 1996]), but there are other alternatives. One of them is also available as a library for C# 2.0, so you can also take a look at the Joins Concurrency Library [Russo, 2007].