CHAPTER 3

Functional Programming

You saw in Chapter 1 that pure functional programming treats everything as a value, including functions. Although F# is not a pure functional language, it does encourage you to program in the functional style; that is, it encourages you to use expressions and computations that return a result, rather than statements that result in some side effect. In this chapter, you'll survey the major language constructs of F# that support the functional programming paradigm and learn how they make it easier to program in the functional style.

Literals

Literals represent constant values and are useful building blocks for computations. F# has a rich set of literals, summarized in Table 3-1.

Table 3-1. *F# Literals*

Example	F# Type	.NET Type	Description
`"Hello\t ", "World\n"`	`string`	`System.String`	A string in which a backslash (\) is an escape character
`@"c:\dir\fs", @""""`	`string`	`System.String`	A verbatim string where a backslash (\) is a regular character
`"bytesbytesbytes"B`	`byte array`	`System.Byte[]`	A string that will be stored as a byte array
`'c'`	`char`	`System.Char`	A character
`true, false`	`bool`	`System.Boolean`	A Boolean
`0x22`	`int/int32`	`System.Int32`	An integer as a hexadecimal
`0o42`	`int/int32`	`System.Int32`	An integer as an octal
`0b10010`	`int/ int32`	`System.Int32`	An integer as a binary
`34y`	`sbyte`	`System.SByte`	A signed byte

Table 3-1. *Continued*

Example	F# Type	.NET Type	Description
34uy	byte	System.Byte	An unsigned byte
34s	int16	System.Int16	A 16-bit integer
34us	uint16	System.UInt16	An unsigned 16-bit integer
34l	int/int32	System.Int32	A 32-bit integer
34ul	uint32	System.UInt32	An unsigned 32-bit integer
34n	nativeint	System.IntPtr	A native-sized integer
34un	unativeint	System.UIntPtr	An unsigned native-sized integer
34L	int64	System.Int64	A 32-bit integer
34UL	uint64	System.Int64	An unsigned 32-bit integer
3.0F, 3.0f	float32	System.Single	A 32-bit IEEE floating-point number
3.0	float	System.Double	A 64-bit IEEE floating-point number
3474262622571I	bigint	Microsoft.FSharp.Math.BigInt	An arbitrary large integer
474262612536171N	bignum	Microsoft.FSharp.Math.BigNum	An arbitrary large number

In F#, string literals can contain newline characters, and regular string literals can contain standard escape codes. Verbatim string literals use a backslash (\) as a regular character, and two double quotes ("") are the escape for a quote. You can define all integer types using hexadecimal and octal by using the appropriate prefix and postfix indicator. The following example shows some of these literals in action, along with how to use the F# `printf` function with a `%A` pattern to output them to the console. The `printf` function interprets the `%A` format pattern using a combination of F#'s reflection (covered in Chapter 7) and the .NET `ToString` method, which is available for every type, to output values in a human-readable way.

```
// some strings
let message = "Hello
World\r\n\t!"
let dir = @"c:\projects"
```

```
// a byte array
let bytes = "bytesbytesbytes"B

// some numeric types
let xA = 0xFFy
let xB = 0o7777un
let xC = 0b10010UL

// print the results
let main() =
    printfn "%A" message
    printfn "%A" dir
    printfn "%A" bytes
    printfn "%A" xA
    printfn "%A" xB
    printfn "%A" xC

// call the main function
main()
```

The results of this example, when compiled and executed, are as follows:

```
"Hello
World
        !"
"c:\projects"
[|98uy; 121uy; 116uy; 101uy; 115uy; 98uy; 121uy; 116uy; 101uy; 115uy; 98uy;
  121uy; 116uy; 101uy; 115uy|]
-1y
4095un
18UL
```

Functions

In F#, functions are defined using the keyword `fun`. The function's arguments are separated by spaces, and the arguments are separated from the function body by a left ASCII arrow (`->`).

Here is an example of a function that takes two values and adds them together:

```
fun x y -> x + y
```

Notice that this function does not have a name; this is a sort of function literal. Functions defined in this way are referred to as *anonymous functions*, *lambda functions*, or just *lambdas*.

The idea that a function does not need a name may seem a little strange. However, if a function is to be passed as an argument to another function, it may not need a name, especially if the task it's performing is relatively simple.

If you need to give the function a name, you can bind it to an identifier, as described in the next section.

Identifiers and let Bindings

Identifiers are the way you give names to values in F# so you can refer to them later in a program. You define an identifier using the keyword `let` followed by the name of the identifier, an equal sign, and an expression that specifies the value to which the identifier refers. An expression is any piece of code that represents a computation that will return a value. The following expression shows a value being assigned to an identifier:

```
let x = 42
```

To most people coming from an imperative programming background, this will look like a variable assignment. There are a lot of similarities, but a key difference is that in pure functional programming, once a value is assigned to an identifier, it does not change. This is why I will refer to them throughout this book as *identifiers*, rather than as *variables*.

■ **Note** Under some circumstances, you can redefine identifiers, which may look a little like an identifier changing value, but is subtly different. Also, in imperative programming in F#, in some circumstances, the value of an identifier can change. In this chapter, we focus on functional programming, where identifiers do not change their values.

An identifier can refer to either a value or a function, and since F# functions are really values in their own right, this is hardly surprising. This means F# has no real concept of a function name or parameter name; these are just identifiers. You can bind an anonymous function to an identifier the same way you can bind a string or integer literal to an identifier:

```
let myAdd = fun x y -> x + y
```

However, as it is very common to need to define a function with a name, F# provides a short syntax for this. You write a function definition the same way as a value identifier, except that a function has two or more identifiers between the `let` keyword and the equal sign, as follows:

```
let raisePowerTwo x = x ** 2.0
```

The first identifier is the name of the function, `raisePowerTwo`, and the identifier that follows it is the name of the function's parameter, `x`. If a function has a name, it is strongly recommended that you use this shorter syntax for defining it.

The syntax for declaring *values* and *functions* in F# is indistinguishable because functions *are* values, and F# syntax treats them both similarly. For example, consider the following code:

```
let n = 10

let add a b = a + b
```

```
let result = add n 4

printfn "result = %i" result
```

On the first line, the value `10` is assigned to the identifier `n`; then on the second line, a function, `add`, which takes two arguments and adds them together, is defined. Notice how similar the syntax is, with the only difference being that a function has parameters that are listed after the function name. Since everything is a value in F#, the literal `10` on the first line is a value, and the result of the expression `a + b` on the next line is also a value that automatically becomes the result of the `add` function. Note that there is no need to explicitly return a value from a function as you would in an imperative language.

The results of this code, when compiled and executed, are as follows:

```
result = 14
```

Identifier Names

There are some rules governing identifier names. Identifiers must start with an underscore (_) or a letter, and can then contain any alphanumeric character, underscore, or a single quotation mark ('). Keywords cannot be used as identifiers. As F# supports the use of a single quotation mark as part of an identifier name, you can use this to represent "prime" to create identifier names for different but similar values, as in this example:

```
let x = 42
let x' = 43
```

F# supports Unicode, so you can use accented characters and letters from non-Latin alphabets as identifier names:

```
let 标识符 = 42
```

If the rules governing identifier names are too restrictive, you can use double tick marks (``) to quote the identifier name. This allows you to use any sequence of characters—as long as it doesn't include tabs, newlines, or double ticks—as an identifier name. This means you could create an identifier that ends with a question mark, for example (some programmers believe it is useful to have names that represent Boolean values end with a question mark):

```
let ``more? `` = true
```

This can also be useful if you need to use a keyword as an identifier or type name:

```
let ``class`` = "style"
```

For example, you might need to use a member from a library that was not written in F# and has one of F#'s keywords as its name (you'll learn more about using non-F# libraries in Chapter 4). Generally, it's best to avoid overuse of this feature, as it could lead to libraries that are difficult to use from other .NET languages.

Scope

The *scope* of an identifier defines where you can use an identifier (or a type, as discussed in the "Defining Types" section later in this chapter) within a program. It is important to have a good understanding of scope, because if you try to use an identifier that's not in scope, you will get a compile error.

All identifiers—whether they relate to functions or values—are scoped from the end of their definitions until the end of the sections in which they appear. So, for identifiers that are at the top level (that is, identifiers that are not local to another function or other value), the scope of the identifier is from the place where it's defined to the end of the source file. Once an identifier at the top level has been assigned a value (or function), this value cannot be changed or redefined. An identifier is available only after its definition has ended, meaning that it is not usually possible to define an identifier in terms of itself.

You will have noticed that in F#, you never need to explicitly return a value; the result of the computation is automatically bound to its associated identifier. So, how do you compute intermediate values within a function? In F#, this is controlled by whitespace. An indentation creates a new scope, and the end of this scope is signaled by the end of the indentation. Indention means that the `let` binding is an intermediate value in the computation that is not visible outside this scope. When a scope closes (by the indentation ending), and an identifier is no longer available, it is said to *drop out of scope* or to be *out of scope*.

To demonstrate scope, the next example shows a function that computes the point halfway between two integers. The third and fourth lines show intermediate values being calculated.

```
// function to calculate a midpoint
let halfWay a b =
    let dif = b - a
    let mid = dif / 2
    mid + a

// call the function and print the results
printfn "(halfWay 5 11) = %i" (halfWay 5 11)
printfn "(halfWay 11 5) = %i" (halfWay 11 5)
```

First, the difference between the two numbers is calculated, and this is assigned to the identifier `dif` using the `let` keyword. To show that this is an intermediate value within the function, it is indented by four spaces. The choice of the number of spaces is left to the programmer, but the convention is four. After that, the example calculates the midpoint, assigning it to the identifier `mid` using the same indentation. Finally, the desired result of the function is the midpoint plus a, so the code can simply say `mid + a`, and this becomes the function's result.

■ **Note** You cannot use tabs instead of spaces for indenting, because these can look different in different text editors, which causes problems when whitespace is significant.

The results of this example are as follows:

```
(halfWay 5 11) = 8
(halfWay 11 5) = 8
```

THE F# LIGHTWEIGHT SYNTAX

By default, F# is whitespace-sensitive, with indentation controlling the scope of identifiers. The language F# was based on, Objective Caml (OCaml), is not whitespace-sensitive. In OCaml, scope is controlled though the use of the in keyword. For example the `halfWay` function from the previous example would look like the following (note the additional `in` keyword in the middle two lines):

```
let halfWay a b =
    let dif = b - a in
    let mid = dif / 2 in
    mid + a
```

The F# whitespace-sensitive syntax is said to be a *lightweight* syntax, because certain keywords and symbols–such as `in`, `;`, `begin`, and `end`–are optional. This means the preceding function definition will be accepted by the F# compiler even with the additional `in` keywords. If you want to force the use of these keywords, add the declaration `#light "off"` to the top of each source file.

I believe that significant whitespace is a much more intuitive way of programming, because it helps the programmer decide how the code should be laid out. Therefore, in this book, I cover the F# lightweight syntax.

Identifiers within functions are scoped to the end of the expression in which they appear. Ordinarily, this means they are scoped until the end of the function definition in which they appear. So, if an identifier is defined inside a function, it cannot be used outside it. Consider the next example:

```
let printMessage() =
    let message = "Help me"
    printfn "%s" message

printfn "%s" message
```

This attempts to use the identifier `message` outside the function `printMessage`, which is out of scope. When trying to compile this code, you'll get the following error message:

```
Prog.fs(34,17): error: FS0039: The value or constructor 'message' is not defined.
```

Identifiers within functions behave a little differently from identifiers at the top level, because they can be redefined using the `let` keyword. This is useful because it means that you do not need to keep inventing names to hold intermediate values. To demonstrate, the next example shows a mathematical puzzle implemented as an F# function. Here, you need to calculate a lot of intermediate values that you don't particularly care about; inventing names for each one these would be an unnecessary burden on the programmer.

```
open System

let readInt() = int (Console.ReadLine())

let mathsPuzzle() =
    printfn "Enter day of the month on which you were born: "
    let input = readInt()
    let x = input * 4 // multiply it by 4
    let x = x + 13 // add 13
    let x = x * 25 // multiply the result by 25
    let x = x - 200 // subtract 200
    printfn "Enter number of the month you were born: "
    let input = readInt()
    let x = x + input
    let x = x * 2 // multiply by 2
    let x = x - 40 // subtract 40
    let x = x * 50 // multiply the result by 50
    printfn "Enter last two digits of the year of your birth: "
    let input = readInt()
    let x = x + input
    let x = x - 10500 // finally, subtract 10,500
    printf "Date of birth (ddmmyy): %i" x

mathsPuzzle()
```

The results of this example, when compiled and executed, are as follows:

```
Enter day of the month on which you were born: 23
Enter number of the month you were born: 5
Enter last two digits of the year of your birth: 78
Date of birth (ddmmyy): 230578
```

Note that this is different from changing the value of an identifier. Because you're redefining the identifier, you're able to change the identifier's type, as shown in the next example, but you still retain type safety.

■ **Note** *Type safety*, sometimes referred to as *strong typing*, basically means that F# will prevent you from performing an inappropriate operation on a value; for example, you can't treat an integer as if it were a floating-point number. I discuss types and how they lead to type safety in the "Types and Type Inference" section later in this chapter.

```
let changeType () =
    let x = 1             // bind x to an integer
    let x = "change me"   // rebind x to a string
    let x = x + 1         // attempt to rebind to itself plus an integer
    printfn "%s" x
```

This example will not compile, because on the third line, the value of x changes from an integer to the string "change me", and then on the fourth line, it tries to add a string and an integer, which is illegal in F#, so you get a compile error:

```
prog.fs(55,13): error: FSOOO1: This expression has type
    int
but is here used with type
    string
stopped due to error
```

If an identifier is redefined, its old value is available while the definition of the identifier is in progress. But after it is defined—that is, at the end of the expression—the old value is hidden. If the identifier is redefined inside a new scope, the identifier will revert to its old value when the new scope is finished.

The next example defines a message and prints it to the console. It then redefines this message inside an *inner function* called innerFun, which also prints the message. Then it calls the function innerFun, and finally prints the message a third time.

```
let printMessages() =
    // define message and print it
    let message = "Important"
    printfn "%s" message;
    // define an inner function that redefines value of message
    let innerFun () =
        let message = "Very Important"
        printfn "%s" message
    // call the inner function
    innerFun ()
    // finally print the first message again
    printfn "%s" message

printMessages()
```

The results of this example, when compiled and executed, are as follows:

```
Important
Very Important
Important
```

A programmer from the imperative world might have expected that `message`, when printed out for the final time, would be bound to the value `Very Important`, rather than `Important`. It holds the value `Important` because the identifier `message` is rebound, rather than assigned, to the value `Very Important` inside the function `innerFun`, and this binding is valid only inside the scope of the function `innerFun`. Therefore, once this function has finished, the identifier `message` reverts to holding its original value.

■ **Note** Using inner functions is a common and excellent way of breaking up a lot of functionality into manageable portions, and you will see their usage throughout the book. They are sometimes referred to as *closures* or *lambdas*, although these two terms have more specific meanings. A *closure* means that the function uses a value that is not defined at the top level. A *lambda* is an anonymous function.

Capturing Identifiers

You have already seen that in F#, you can define functions within other functions. These functions can use any identifier in scope, including definitions that are also local to the function where they are defined. Because these inner functions are values, they could be returned as the result of the function or passed to another function as an argument. This means that although an identifier is defined within a function, so it is not visible to other functions, its actual lifetime may be much longer than the function in which it is defined. Let's look at an example to illustrate this point. Consider the following function, defined as `calculatePrefixFunction`:

```
// function that returns a function to
let calculatePrefixFunction prefix =
    // calculate prefix
    let prefix' = Printf.sprintf "[%s]: " prefix
    // define function to perform prefixing
    let prefixFunction appendee =
        Printf.sprintf "%s%s" prefix' appendee
    // return function
    prefixFunction

// create the prefix function
let prefixer = calculatePrefixFunction "DEBUG"

// use the prefix function
printfn "%s" (prefixer "My message")
```

This function returns the inner function it defines, `prefixFunction`. The identifier `prefix'` is defined as local to the scope of the function `calculatePrefixFunction`; it cannot be seen by other functions outside `calculatePrefixFunction`. The inner function `prefixFunction` uses `prefix'`, so when `prefixFunction` is returned, the value `prefix'` must still be available. `calculatePrefixFunction` creates the function `prefixer`. When `prefixer` is called, you see that its result uses a value that was calculated and associated with `prefix'`:

```
[DEBUG]: My message
```

Although you should have an understanding of this process, most of the time you don't need to think about it, because it doesn't involve any additional work by the programmer. The compiler will automatically generate a *closure* to handle extending the lifetime of the local value beyond the function in which it is defined. The .NET garbage collection will automatically handle clearing the value from memory. Understanding this process of identifiers being captured in closures is probably more important when programming in imperative style, where an identifier can represent a value that changes over time. When programming in the functional style, identifiers will always represent values that are constant, making it slightly easier to figure out what has been captured in a closure.

The use Binding

It can be useful to have some action performed on an identifier when it drops out of scope. For example, it's important to close file handles when you've finished reading or writing to the file, so you may want to close the file as soon as the identifier that represents it drops out of scope. More generally, anything that is an operation system resource—such as network socket—or is precious because it's expensive to create or a limited number is available—such as a database connection—should be closed or freed as quickly as possible.

In .NET, objects that fall into this category should implement the `IDisposable` interface (for more information about objects and interfaces, see Chapter 5). This interface contains one method, `Dispose`, which will clean up the resource; for example, in the case of a file, it will close the open file handle. So, in many cases, it's useful to call this method when the identifier drops out of scope. F# provides the `use` binding to do just that.

A `use` binding behaves the same as a `let` binding, except that when the variable drops out of scope, the compiler automatically generates code to ensure that the `Dispose` method will be called at the end of the scope. The code generated by the compiler will always be called, even if an exception occurs (see the "Exceptions and Exception Handling" section later in this chapter for more information about exceptions). To illustrate this, consider the following example:

```
open System.IO

// function to read first line from a file
let readFirstLine filename =
    // open file using a "use" binding
    use file = File.OpenText filename
    file.ReadLine()

// call function and print the result
printfn "First line was: %s" (readFirstLine "mytext.txt")
```

Here, the function `readFirstLine` uses the .NET Framework method `File.OpenText` to open a text file for reading. The `StreamReader` that is returned is bound to the identifier `file` using a `use` binding. The example then reads the first line from the file and returns this as a result. At this point, the identifier file drops out of scope, so its `Dispose` method will be called and close the underlying file handle.

Note the following important constraints on the use of `use` bindings:

- You can use `use` bindings only with objects that implement the `IDisposable` interface.
- `use` bindings cannot be used at the top level. They can be used only within functions, because identifiers at the top level never go out of scope.

Recursion

Recursion means defining a function in terms of itself; in other words, the function calls itself within its definition. Recursion is often used in functional programming where you would use a loop in imperative programming. Many believe that algorithms are much easier to understand when expressed in terms of recursion rather than loops.

To use recursion in F#, use the `rec` keyword after the `let` keyword to make the identifier available within the function definition. The following example shows recursion in action. Notice how on the fifth line, the function makes two calls to itself as part of its own definition.

```
// a function to generate the Fibonacci numbers
let rec fib x =
    match x with
    | 1 -> 1
    | 2 -> 1
    | x -> fib (x - 1) + fib (x - 2)

// call the function and print the results
printfn "(fib 2) = %i" (fib 2)
printfn "(fib 6) = %i" (fib 6)
printfn "(fib 11) = %i" (fib 11)
```

The results of this example, when compiled and executed, are as follows:

```
(fib 2) = 1
(fib 6) = 8
(fib 11) = 89
```

This function calculates the *n*th term in the Fibonacci sequence. The Fibonacci sequence is generated by adding the previous two numbers in the sequence, and it progresses as follows: 1, 1, 2, 3, 5, 8, 13, …. Recursion is most appropriate for calculating the Fibonacci sequence, because the definition of any number in the sequence, other than the first two, depends on being able to calculate the previous two numbers, so the Fibonacci sequence is defined in terms of itself.

Although recursion is a powerful tool, you should be careful when using it. It is easy to inadvertently write a recursive function that never terminates. Although intentionally writing a program that does not terminate is sometimes useful, it is rarely the goal when trying to perform calculations. To ensure that recursive functions terminate, it is often useful to think of recursion in terms of a base case and a recursive case:

- The *recursive case* is the value for which the function is defined in terms of itself. For the function `fib`, this is any value other than 1 and 2.
- The *base case* is the nonrecursive case; that is, there must be some value where the function is not defined in terms of itself. In the `fib` function, 1 and 2 are the base cases.

Having a base case is not enough in itself to ensure termination. The recursive case must tend toward the base case. In the `fib` example, if `x` is greater than or equal to 3, then the recursive case will tend toward the base case, because `x` will always become smaller and at some point reach 2. However, if `x` is less than 1, then `x` will grow continually more negative, and the function will recurse until the limits of the machine are reached, resulting in a stack overflow error (`System.StackOverflowException`).

The previous code also uses F# pattern matching, which is discussed in the "Pattern Matching" section later in this chapter.

Operators

In F#, you can think of *operators* as a more aesthetically pleasing way to call functions.

F# has two different kinds of operators:

- A *prefix* operator is an operator where the operands come after the operator.
- An *infix* operator comes in between the first and second operands.

F# provides a rich and diverse set of operators that you can use with numeric, Boolean, string, and collection types. The operators defined in F# and its libraries are too numerous to be covered in this section, so rather than looking at individual operators, we'll look at how to use and define operators in F#.

As in C#, F# operators are overloaded, meaning you can use more than one type with an operator; however, unlike in C#, both operands must be the same type, or the compiler will generate an error. F# also allows users to define and redefine operators.

Operators follow a set of rules similar to C#'s for operator overloading resolution; therefore, any class in the BCL, or any .NET library, that was written to support operator overloading in C# will support it in F#. For example, you can use the + operator to concatenate strings, as well as to add a `System.TimeSpan` to a `System.DataTime`, because these types support an overload of the + operator. The following example illustrates this:

```
let rhyme = "Jack " + "and " + "Jill"

open System
let oneYearLater =
    DateTime.Now + new TimeSpan(365, 0, 0, 0, 0)
```

Unlike functions, operators are not values, so they cannot be passed to other functions as parameters. However, if you need to use an operator as a value, you can do this by surrounding it with parentheses. The operator will then behave exactly like a function. Practically, this has two consequences:

- The operator is now a function, and its parameters will appear after the operator:

  ```
  let result = (+) 1 1
  ```

- As it is a value, it could be returned as the result of a function, passed to another function, or bound to an identifier. This provides a very concise way to define the `add` function:

```
let add = (+)
```

You'll see how using an operator as a value can be useful later in this chapter when we look at working with lists.

Users can define their own operators or redefine any of the existing ones if they want (although this is not always advisable, because the operators then no longer support overloading). Consider the following perverse example that redefines `+` to perform subtraction:

```
let (+) a b = a - b
printfn "%i" (1 + 1)
```

User-defined (*custom*) operators must be nonalphanumeric and can be a single character or a group of characters. You can use the following characters in custom operators:

```
!$%&*+-./<=>?@^|~
```

The syntax for defining an operator is the same as using the `let` keyword to define a function, except the operator replaces the function name and is surrounded by parentheses so the compiler knows that the symbols are used as a name of an operator, rather than as the operator itself. The following example shows defining a custom operator, `+*`, which adds its operands and then multiplies them:

```
let ( +* ) a b = (a + b) * a * b
printfn "(1 +* 2) = %i" (1 +* 2)
```

The results of this example, when compiled and executed, are as follows:

```
(1 +* 2) = 6
```

Unary operators always come before the operand. User-defined binary operators are prefix if they start with an exclamation mark (`!`), a question mark (`?`), or a tilde (`~`); they are infix, with operators coming between the operands, for everything else.

Function Application

Function application, also sometimes referred to as *function composition* or *composing functions*, simply means calling a function with some arguments. The following example shows the function `add` being defined and then applied to two arguments. Notice that the arguments are not separated with parentheses or commas; only whitespace is needed to separate them.

```
let add x y = x + y

let result = add 4 5

printfn "(add 4 5) = %i" result
```

The results of this example, when compiled and executed, are as follows:

```
(add 4 5) = 9
```

In F#, a function has a fixed number of arguments and is applied to the value that appears next in the source file. You do not necessarily need to use parentheses when calling functions, but F# programmers often use them to define which function should be applied to which arguments. Consider the simple case where you want to add four numbers using the `add` function. You could bind the result of each function call to new identifier, but for such a simple calculation, this would be very cumbersome:

```
let add x y = x + y

let result1 = add 4 5
let result2 = add 6 7

let finalResult = add result1 result2
```

Instead, it often better to pass the result of one function directly to the next function. To do this, you use parentheses to show which parameters are associated with which functions:

```
let add x y = x + y

let result =
    add (add 4 5) (add 6 7)
```

Here, the second and third occurrences of the `add` function are grouped with the parameters `4`, `5` and `6`, `7`, respectively, and the first occurrence of the `add` function will act on the results of the other two functions.

F# also offers another way to compose functions, using the *pipe-forward* operator (`|>`). This operator has the following definition:

```
let (|>) x f = f x
```

This simply means it takes a parameter, `x`, and applies that to the given function, `f`, so that the parameter is now given before the function. The following example shows a parameter, `0.5`, being applied to the function `System.Math.Cos` using the pipe-forward operator:

```
let result = 0.5 |> System.Math.Cos
```

This reversal can be useful in some circumstances, especially when you want to chain many functions together. Here is the previous `add` function example rewritten using the pipe-forward operator:

```
let add x y = x + y

let result = add 6 7 |> add 4 |> add 5
```

Some programmers think this style is more readable, as it has the effect of making the code read in a more right-to-left manner. The code should now be read as "add 6 to 7, then forward this result to the next function, which will add 4, and then forward this result to a function that will add 5." A more detailed explanation of where it's appropriate to use this style of function application can be found in Chapter 4.

This example also takes advantage of the capability to partially apply functions in F#, as discussed in the next section.

Partial Application of Functions

F# supports the partial application of functions (these are sometimes called *partial* or *curried* functions). This means you don't need to pass all the arguments to a function at once. Notice that the final example in the previous section passes a single argument to the `add` function, which takes two arguments. This is very much related to the idea that functions are values.

Because a function is just a value, if it doesn't receive all its arguments at once, it returns a value that is a new function waiting for the rest of the arguments. So, in the example, passing just the value `4` to the `add` function results in a new function, which I named `addFour`, because it takes one parameter and adds the value `4` to it. At first glance, this idea can look uninteresting and unhelpful, but it is a powerful part of functional programming that you'll see used throughout the book.

This behavior may not always be appropriate. For example, if the function takes two floating-point parameters that represent a point, it may not be desirable to have these numbers passed to the function separately, because they both make up the point they represent. You may alternatively surround a function's parameters with parentheses and separate them with commas, turning them into a *tuple*. You can see this in the following code:

```
let sub (a, b) = a - b

let subFour = sub 4
```

When attempting to compile this example, you will receive the following error message:

```
prog.fs(15,19): error: FS0001: This expression has type
    int
but is here used with type
    'a * 'b
```

This example will not compile because the `sub` function requires both parameters to be given at once. `sub` now has only one parameter, the tuple `(a, b)`, instead of two, and although the call to `sub` in the second line provides only one argument, it's not a tuple. So, the program does not type check, as the code is trying to pass an integer to a function that takes a tuple. Tuples are discussed in more detail in the "Defining Types" section later in this chapter.

In general, functions that can be partially applied are preferred over functions that use tuples. This is because functions that can be partially applied are more flexible than tuples, giving users of the function more choices about how to use them. This is especially true when creating a library to be used by other programmers. You may not be able to anticipate all the ways your users will want to use your functions, so it is best to give them the flexibility of functions that can be partially applied.

Pattern Matching

Pattern matching allows you to look at the value of an identifier and then make different computations depending on its value. It might be compared to the `switch` statement in C++ and C#, but it is much more powerful and flexible. Programs that are written in the functional style tend to be written as series of transformations applied to the input data. Pattern matching allows you to analyze the input data and decided which transformation should be applied to it, so pattern matching fits in well with programming in the functional style.

The pattern-matching construct in F# allows you to pattern match over a variety of types and values. It also has several different forms and crops up in several places in the language, including its exception-handling syntax, which is discussed in the "Exceptions and Exception Handling" section later in this chapter.

The simplest form of pattern matching is matching over a value. You have already seen this earlier in this chapter, in the "Recursion" section, where it was used to implement a function that generated numbers in the Fibonacci sequence. To illustrate the syntax, the next example shows an implementation of a function that will produce the Lucas numbers, a sequence of numbers as follows: 1, 3, 4, 7, 11, 18, 29, 47, 76, …. The Lucas sequence has the same definition as the Fibonacci sequence; only the starting points are different.

```
// definition of Lucas numbers using pattern matching
let rec luc x =
    match x with
    | x when x <= 0 -> failwith "value must be greater than 0"
    | 1 -> 1
    | 2 -> 3
    | x -> luc (x - 1) + luc (x - 2)

// call the function and print the results
printfn "(luc 2) = %i" (luc 2)
printfn "(luc 6) = %i" (luc 6)
printfn "(luc 11) = %i" (luc 11)
printfn "(luc 12) = %i" (luc 12)
```

The results of this example, when compiled and executed, are as follows:

```
(luc 2) = 3
(luc 6) = 18
(luc 11) = 199
(luc 12) = 322
```

The syntax for pattern matching uses the keyword `match`, followed by the identifier that will be matched, then the keyword `with`, then all the possible matching rules separated by vertical bars (`|`). In the simplest case, a rule consists of either a constant or an identifier, followed by an arrow (`->`), and then by the expression to be used when the value matches the rule. In this definition of the function `luc`, the second two cases are literals—the values `1` and `2`—and these will be replaced with the values `1` and `3`, respectively. The fourth case will match any value of `x` greater than 2, and this will cause two further calls to the `luc` function.

The rules are matched in the order in which they are defined, and the compiler will issue an error if pattern matching is incomplete; that is, if there is some possible input value that will not match any rule. This would be the case in the `luc` function if you had omitted the final rule, because any values of `x` greater than 2 would not match any rule. The compiler will also issue a warning if there are any rules that will never be matched, typically because there is another rule in front of them that is more general. This would be the case in the `luc` function if the fourth rule were moved ahead of the first rule. In this case, none of the other rules would ever be matched, because the first rule would match any value of `x`.

You can add a `when` guard (as in the first rule in the example) to give precise control about when a rule fires. A `when` guard is composed of the keyword `when` followed by a Boolean expression. Once the rule is matched, the `when` clause is evaluated, and the rule will fire only if the expression evaluates to `true`. If the expression evaluates to `false`, the remaining rules will be searched for another match. The first rule is designed to be the function's error handler. The first part of the rule is an identifier that will match any integer, but the `when` guard means the rule will match only those integers that are less than or equal to zero.

If you want, you can omit the first `|`. This can be useful when the pattern match is small and you want to fit it on one line. You can see this in the next example, which also demonstrates the use of the underscore (`_`) as a *wildcard*.

```
let booleanToString x =
    match x with false -> "False" | _ -> "True"
```

The `_` will match any value and is a way of telling the compiler that you're not interested in using this value. For example, in this `booleanToString` function, you do not need to use the constant `true` in the second rule, because if the first rule is matched, you know that the value of `x` will be `true`. Moreover, you do not need to use `x` to derive the string `"True"`, so you can ignore the value and just use `_` as a wildcard.

Another useful feature of pattern matching is that you can combine two patterns into one rule through the use of the vertical bar (`|`). The next example, `stringToBoolean`, demonstrates this.

```
// function for converting a boolean to a string
let booleanToString x =
    match x with false -> "False" | _ -> "True"

// function for converting a string to a boolean
let stringToBoolean x =
    match x with
    | "True" | "true" -> true
    | "False" | "false" -> false
    | _ -> failwith "unexpected input"

// call the functions and print the results
printfn "(booleanToString true) = %s"
    (booleanToString true)
printfn "(booleanToString false) = %s"
    (booleanToString false)
printfn "(stringToBoolean \"True\") = %b"
    (stringToBoolean "True")
```

```
printfn "(stringToBoolean \"false\") = %b"
    (stringToBoolean "false")
printfn "(stringToBoolean \"Hello\") = %b"
    (stringToBoolean "Hello")
```

The first two rules have two strings that should evaluate to the same value, so rather than having two separate rules, you can just use | between the two patterns. The results of this examples, when compiled and executed, are as follows:

```
(booleanToString true) = True
(booleanToString false) = False
(stringToBoolean "True") = true
(stringToBoolean "false") = false
Microsoft.FSharp.Core.FailureException: unexpected input
   at FSI_0005.stringToBoolean(String x)
   at <StartupCode$FSI_0005>.$FSI_0005.main@()
```

It is also possible to pattern match over most of the types defined by F#. The next two examples demonstrate pattern matching over tuples, with two functions that implement a Boolean And and Or using pattern matching. Each takes a slightly different approach.

```
let myOr b1 b2 =
    match b1, b2 with
    | true, _ -> true
    | _, true -> true
    | _ -> false

let myAnd p =
    match p with
    | true, true -> true
    | _ -> false

printfn "(myOr true false) = %b" (myOr true false)
printfn "(myOr false false) = %b" (myOr false false)
printfn "(myAnd (true, false)) = %b" (myAnd (true, false))
printfn "(myAnd (true, true)) = %b" (myAnd (true, true))
```

The results of these examples, when compiled and executed, are as follows:

```
(myOr true false) = true
(myOr false false) = false
(myAnd (true, false)) = false
(myAnd (true, true)) = true
```

The myOr function has two Boolean parameters, which are placed between the match and with keywords and separated by commas to form a tuple. The myAnd function has one parameter, which is itself a tuple. Either way, the syntax for creating pattern matches for tuples is the same and similar to the syntax for creating tuples.

If it's necessary to match values within the tuple, the constants or identifiers are separated by commas, and the position of the identifier or constant defines what it matches within the tuple. This is shown in the first and second rules of the myOr function and in the first rule of the myAnd function. These rules match parts of the tuples with constants, but you could use identifiers if you want to work with the separate parts of the tuple later in the rule definition. Just because you're working with tuples doesn't mean you always need to look at the various parts that make up the tuple.

The third rule of myOr and the second rule of myAnd show the whole tuple matched with a single _ wildcard character. This, too, could be replaced with an identifier if you want to work with the value in the second half of the rule definition.

Because pattern matching is such a common task in F#, the language provides alternative shorthand syntax. If the sole purpose of a function is to pattern match over something, then it may be worth using this syntax. In this version of the pattern-matching syntax, you use the keyword function, place the pattern where the function's parameters would usually go, and then separate all the alternative rules with |. The next example shows this syntax in action in a simple function that recursively processes a list of strings and concatenates them into a single string.

```
// concatenate a list of strings into single string
let rec conactStringList =
    function head :: tail -> head + conactStringList tail
           | [] -> ""

// test data
let jabber = ["'Twas "; "brillig, "; "and "; "the "; "slithy "; "toves "; "..."]
// call the function
let completJabber = conactStringList jabber
// print the result
printfn "%s" completJabber
```

The results of this example, when compiled and executed, are as follows:

```
'Twas brillig, and the slithy toves ...
```

Pattern matching is one of the fundamental building blocks of F#, and we'll return to it several times in this chapter. We'll look at pattern matching over lists, with record types and union types, and with exception handling. The most advanced use of pattern matching is discussed in the "Active Patterns" section toward the end of the chapter. You can find details of how to pattern match over types from non-F# libraries in Chapters 4.

Control Flow

F# has a strong notion of *control flow*. In this way. it differs from many pure functional languages, where the notion of control flow is very loose, because expressions can be evaluated in essentially any order. The strong notion of control flow is apparent in the `if … then … else …` expression.

In F#, the `if … then … else …` construct is an expression, meaning it returns a value. One of two different values will be returned, depending on the value of the Boolean expression between the `if` and `then` keywords. The next example illustrates this. The `if … then … else …` expression is evaluated to return either `"heads"` or `"tails"`, depending on whether the program is run on an even second or an odd second.

```
let result =
    if System.DateTime.Now.Second % 2 = 0 then
        "heads"
    else
        "tails"

printfn "%A" result
```

It's interesting to note that the `if … then … else …` expression is just a convenient shorthand for pattern matching over a Boolean value. So the previous example could be rewritten as follows:

```
let result =
    match System.DateTime.Now.Second % 2 = 0 with
    | true -> "heads"
    | false ->  "tails"
```

The `if … then … else …` expression has some implications that you might not expect if you are more familiar with imperative-style programming. F#'s type system requires that the values being returned by the `if … then … else …` expression must be the same type, or the compiler will generate an error. So, if in the previous example, you replaced the string `"tails"` with an integer or Boolean value, you would get a compile error. If you really require the values to be of different types, you can create an `if … then … else …` expression of type `obj` (F#'s version of `System.Object`), as shown in the next example, which prints either `"heads"` or `false` to the console.

```
let result =
    if System.DateTime.Now.Second % 2 = 0 then
        box "heads"
    else
        box false

printfn "%A" result
```

Imperative programmers may be surprised that an `if … then … else …` expression must have an `else` if the expression returns a value. This is pretty logical when you think about it and considering the examples you've just seen. If the `else` were removed from the code, the identifier `result` could not be

assigned a value when the `if` evaluated to false, and having uninitialized identifiers is something that F# (and functional programming in general) aims to avoid. There is a way for a program to contain an `if … then` expression without the `else`, but this is very much in the style of imperative programming, so I discuss it in Chapter 4.

Lists

F# *lists* are simple collection types that are built into F#. An F# list can be an *empty list*, represented by square brackets (`[ ]`), or it can be another list with a value concatenated to it. You concatenate values to the front of an F# list using a built-in operator that consists of two colons (`::`), pronounced "cons." The next example shows some lists being defined, starting with an empty list on the first line, followed by two lists where strings are placed at the front by concatenation:

```
let emptyList = []
let oneItem = "one " :: []
let twoItem = "one " :: "two " :: []
```

The syntax to add items to a list by concatenation is a little verbose, so if you just want to define a list, you can use shorthand. In this shorthand notation, you place the list items between square brackets and separate them with a semicolon (`;`), as follows:

```
let shortHand = ["apples "; "pears"]
```

Another F# operator that works on lists is the at symbol (`@`), which you can use to concatenate two lists together, as follows:

```
let twoLists = ["one, "; "two, "] @ ["buckle "; "my "; "shoe "]
```

All items in an F# list must be of the same type. If you try to place items of different types in a list—for example, you try to concatenate a string to a list of integers—you will get a compile error. If you need a list of mixed types, you can create a list of type `obj` (the F# equivalent of `System.Object`), as in the following code:

```
// the empty list
let emptyList = []

// list of one item
let oneItem = "one " :: []

// list of two items
let twoItem = "one " :: "two " :: []

// list of two items
let shortHand = ["apples "; "pairs "]

// concatenation of two lists
let twoLists = ["one, "; "two, "] @ ["buckle "; "my "; "shoe "]
```

```
// list of objects
let objList = [box 1; box 2.0; box "three"]

// print the lists
let main() =
    printfn "%A" emptyList
    printfn "%A" oneItem
    printfn "%A" twoItem
    printfn "%A" shortHand
    printfn "%A" twoLists
    printfn "%A" objList

// call the main function
main()
```

The results of this example, when compiled and executed, are as follows:

```
[]
["one "]
["one "; "two "]
["apples "; "pairs "]
["one, "; "two, "; "buckle "; "my "; "shoe "]
[1; 2.0; "three"]
```

I discuss types in F# in more detail in the "Types and Type Inference" section later in this chapter.

F# lists are *immutable*; in other words, once a list is created, it cannot be altered. The functions and operators that act on lists do not alter them, but they create a new, modified version of the list, leaving the old list available for later use if needed. The next example shows this.

```
// create a list of one item
let one = ["one "]
// create a list of two items
let two = "two " :: one
// create a list of three items
let three = "three " :: two

// reverse the list of three items
let rightWayRound = List.rev three

// function to print the results
let main() =
    printfn "%A" one
    printfn "%A" two
    printfn "%A" three
    printfn "%A" rightWayRound
```

```
// call the main function
main()
```

An F# list containing a single string is created, and then two more lists are created, each using the previous one as a base. Finally, the `List.rev` function is applied to the last list to create a new reversed list. When you print these lists, it is easy to see that all the original lists remain unaltered:

```
["one "]
["two "; "one "]
["three "; "two "; "one "]
["one "; "two "; "three "]
```

Pattern Matching Against Lists

The regular way to work with F# lists is to use pattern matching and recursion. The pattern-matching syntax for pulling the head item off a list is the same as the syntax for concatenating an item to a list. The pattern is formed by the identifier representing the head, followed by `::` and then the identifier for the rest of the list. You can see this in the first rule of `concatList` in the next example. You can also pattern match against list constants; you can see this in the second rule of `concatList`, where there is an empty list.

```
// list to be concatenated
let listOfList = [[2; 3; 5]; [7; 11; 13]; [17; 19; 23; 29]]

// definition of a concatenation function
let rec concatList l =
    match l with
    | head :: tail -> head @ (concatList tail)
    | [] -> []

// call the function
let primes = concatList listOfList

// print the results
printfn "%A" primes
```

The results of this example, when compiled and executed, are as follows:

```
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
```

Taking the head from a list, processing it, and then recursively processing the tail of the list is the most common way of dealing with lists via pattern matching, but it certainly isn't the only thing you can do with pattern matching and lists. The following example shows a few other uses of this combination of features.

```
// function that attempts to find various sequences
let rec findSequence l =
    match l with
    // match a list containing exactly 3 numbers
    | [x; y; z] ->
        printfn "Last 3 numbers in the list were %i %i %i"
            x y z
    // match a list of 1, 2, 3 in a row
    | 1 :: 2 :: 3 :: tail ->
        printfn "Found sequence 1, 2, 3 within the list"
        findSequence tail
    // if neither case matches and items remain
    // recursively call the function
    | head :: tail -> findSequence tail
    // if no items remain terminate
    | [] -> ()

// some test data
let testSequence = [1; 2; 3; 4; 5; 6; 7; 8; 9; 8; 7; 6; 5; 4; 3; 2; 1]

// call the function
findSequence testSequence
```

The first rule demonstrates how to match a list of a fixed length—in this case, a list of three items. Here, identifiers are used to grab the values of these items so they can be printed to the console. The second rule looks at the first three items in the list to see whether they are the sequence of integers 1, 2, 3; and if they are, it prints a message to the console. The final two rules are the standard head/tail treatment of a list, designed to work their way through the list, doing nothing if there is no match with the first two rules.

The results of this example, when compiled and executed, are as follows:

```
Found sequence 1, 2, 3 within the list
Last 3 numbers in the list were 3 2 1
```

Although pattern matching is a powerful tool for the analysis of data in lists, it's often not necessary to use it directly. The F# libraries provide a number of higher-order functions for working with lists that implement the pattern matching for you, so you don't need repeat the code. To illustrate this, imagine you need to write a function that adds one to every item in a list. You can easily write this using pattern matching:

```
let rec addOneAll list =
    match list with
    | head :: rest ->
        head + 1 :: addOneAll rest
    | [] -> []

printfn "(addOneAll [1; 2; 3]) = %A" (addOneAll [1; 2; 3])
```

The results of this example, when compiled and executed, are as follows:

```
(addOneAll [1; 2; 3]) = [2; 3; 4]
```

However, the code is perhaps a little more verbose than you would like for such a simple problem. The clue to solving this comes from noticing that adding one to every item in the list is just an example of a more general problem: the need to apply some transformation to every item in a list. The F# core library contains a function `map`, which is defined in the `List` module. It has the following definition:

```
let rec map func list =
    match list with
    | head :: rest ->
        func head :: map func rest
    | [] -> []
```

You can see that the `map` function has a very similar structure to the `addOneAll` function from the previous example. If the list is not empty, you take the head item of the list and apply the function, `func`, that you are given as a parameter. This is then appended to the results of recursively calling `map` on the rest of the list. If the list is empty, you simply return the empty list. The `map` function can then be used to implement adding one to all items in a list in a much more concise manner:

```
let result = List.map ((+) 1) [1; 2; 3]

printfn "List.map ((+) 1) [1; 2; 3] = %A" result
```

The results of this example, when compiled and executed, are as follows:

```
(List.map ((+) 1) [1; 2; 3]) = [2; 3; 4]
```

Also note that this example uses the add operator as a function by surrounding it with parentheses, as described earlier in this chapter in the "Operators" section. This function is then partially applied by passing its first parameter but not its second. This creates a function that takes an integer and returns an integer, which is passed to the `map` function.

The `List` module contains many other interesting functions for working with lists, such as `List.filter` and `List.fold`. These are explained in more detail in Chapter 7, which describes the libraries available with F#.

List Comprehensions

List comprehensions make creating and converting collections easy. You can create F# lists, sequences, and arrays directly using comprehension syntax. I cover arrays in more detail in the next chapter. *Sequences* are collections of type `seq`, which is F#'s name for the .NET BCL's `IEnumerable` type. I describe them in the "Lazy Evaluation" section later in this chapter.

The simplest comprehensions specify ranges, where you write the first item you want, either a number or a letter, followed by two periods (`..`), and then the last item you want, all within square

brackets (to create a list) or braces (to create a sequence). The compiler then does the work of calculating all the items in the collection, taking the first number and incrementing it by 1, or similarly with characters, until it reaches the last item specified. The following example demonstrates how to create a list of numbers from 0 through 9 and a sequence of the characters from A through Z:

```
// create some list comprehensions
let numericList = [ 0 .. 9 ]
let alpherSeq = seq { 'A' .. 'Z' }

// print them
printfn "%A" numericList
printfn "%A" alpherSeq
```

The results of this example are as follows:

```
[0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
seq ['A'; 'B'; 'C'; 'D'; ...]
```

To create more interesting collections, you can also specify a step size for incrementing numbers (note that characters do not support this type of list comprehension). You place the step size between the first and last items, separated by an extra pair of periods (`..`). The next example shows a list containing multiples of 3, followed by a list that counts backward from 9 to 0:

```
// create some list comprehensions
let multiplesOfThree = [ 0 .. 3 .. 30 ]
let revNumericSeq = [ 9 .. -1 .. 0 ]

// print them
printfn "%A" multiplesOfThree
printfn "%A" revNumericSeq
```

The results of this example are as follows:

```
[0; 3; 6; 9; 12; 15; 18; 21; 24; 27; 30]
[9; 8; 7; 6; 5; 4; 3; 2; 1; 0]
```

List comprehensions also allow loops to create a collection from another collection. The idea is that you enumerate the old collection, transform each of its items, and place any generated items in the new collection. To specify such a loop, use the keyword `for`, followed by an identifier, followed by the keyword `in`, at the beginning of the list comprehension. The next example creates a sequence of the squares of the first ten positive integers.

```
// a sequence of squares
let squares =
    seq { for x in 1 .. 10 -> x * x }
```

```
// print the sequence
printfn "%A" squares
```

The example uses `for` to enumerate the collection `1 .. 10`, assigning each item in turn to the identifier `x`. It then uses the identifier `x` to calculate the new item, in this case multiplying `x` by itself to square it. The results of this example are as follows:

```
seq [1; 4; 9; 16; ...]
```

The use of the F# keyword `yield` gives you a lot of flexibility when defining list comprehensions. The `yield` keyword allows you to decide whether or not a particular item should be added to the collection. For example, consider the following example:

```
// a sequence of even numbers
let evens n =
    seq { for x in 1 .. n do
            if x % 2 = 0 then yield x }

// print the sequence
printfn "%A" (evens 10)
```

The goal is to create a collection of even numbers. So you test each number in the collection you are enumerating to see if it is a multiple of two. If it is, you return it using the `yield` keyword; otherwise, you perform no action. The results of this example are as follows:

```
seq [2; 4; 6; 8; ...]
```

It's also possible to use list comprehensions to iterate in two or more dimensions by using a separate loop for each dimension. The next example defines a function called `squarePoints` that creates a sequence of points forming a square grid, each point represented by a tuple of two integers.

```
// sequence of tuples representing points
let squarePoints n =
    seq { for x in 1 .. n do
            for y in 1 .. n do
                yield x, y }

// print the sequence
printfn "%A" (squarePoints 3)
```

The results of this example are as follows:

```
seq [(1, 1); (1, 2); (1, 3); (2, 1); ...]
```

You'll look at using comprehensions with arrays and collections from the .NET Framework BCL in Chapter 4.

Types and Type Inference

F# is a *strongly typed* language, which means you cannot use a function with a value that is inappropriate. You cannot call a function that has a string as a parameter with an integer argument; you must explicitly convert between the two. The way the language treats the type of its values is referred to as its *type system*. F# has a type system that does not get in the way of routine programming. In F#, all values have a type, and this includes values that are functions.

Ordinarily, you don't need to explicitly declare types; the compiler will work out the type of a value from the types of the literals in the function and the resulting types of other functions it calls. If everything is OK, the compiler will keep the types to itself; only if there is a type mismatch will the compiler inform you by reporting a compile error. This process is generally referred to as *type inference*. If you want to know more about the types in a program, you can make the compiler display all inferred types with the `-i` switch. Visual Studio users get tooltips that show types when they hover the mouse pointer over an identifier.

The way type inference works in F# is fairly easy to understand. The compiler works through the program, assigning types to identifiers as they are defined, starting with the top leftmost identifier and working its way down to the bottom rightmost. It assigns types based on the types it already knows—that is, the types of literals and (more commonly) the types of functions defined in other source files or assemblies.

The next example defines two F# identifiers and then shows their inferred types displayed on the console with the F# compiler's `-i` switch.

```
let aString = "Spring time in Paris"
let anInt = 42
```

```
val aString : string
val anInt : int
```

The types of these two identifiers are unsurprising—`string` and `int`, respectively. The syntax used by the compiler to describe them is fairly straightforward: the keyword `val` (meaning "value") and then the identifier, a colon, and finally the type.

The definition of the function `makeMessage` in the next example is a little more interesting.

```
let makeMessage x = (Printf.sprintf "%i" x) + " days to spring time"
let half x = x / 2
```

```
val makeMessage : int -> string
val half : int -> int
```

Note that the `makeMessage` function's definition is prefixed with the keyword `val`, just like the two values you saw before; even though it is a function, the F# compiler still considers it to be a value. Also, the type itself uses the notation `int -> string`, meaning a function that takes an integer and returns a string. The `->` between the type names (an *ASCII arrow*, or just *arrow*) represents the transformation of the function being applied. The arrow represents a transformation of the value, but not necessarily the

type, because it can represent a function that transforms a value into a value of the same type, as shown in the `half` function on the second line.

The types of functions that can be partially applied and functions that take tuples differ. The following functions, `div1` and `div2`, illustrate this.

```
let div1 x y = x / y
let div2 (x, y) = x / y

let divRemainder x y = x / y, x % y
```

```
val div1 : int -> int -> int
val div2 : int * int -> int
val divRemainder : int -> int -> int * int
```

The function `div1` can be partially applied, and its type is `int -> int -> int`, representing that the arguments can be passed in separately. Compare this with the function `div2`, which has the type `int * int -> int`, meaning a function that takes a pair of integers—a tuple of integers—and turns them into a single integer. You can see this in the function `div_remainder`, which performs integer division and also returns the remainder at the same time. Its type is `int -> int -> int * int`, meaning a curried function that returns an integer tuple.

The next function, `doNothing`, looks inconspicuous enough, but it is quite interesting from a typing point of view.

```
let doNothing x = x
```

```
val doNothing : 'a -> 'a
```

This function has the type `'a -> 'a`, meaning it takes a value of one type and returns a value of the same type. Any type that begins with a single quotation mark (`'`) means a *variable* type. F# has a type, `obj`, that maps to `System.Object` and represents a value of any type, a concept that you will probably be familiar with from other languages common language runtime (CLR)–based programming languages (and indeed, many languages that do not target the CLR). However, a variable type is not the same. Notice how the type has an `'a` on both sides of the arrow. This means that, even though the compiler does not yet know the type, it knows that the type of the return value will be the same as the type of the argument. This feature of the type system, sometimes referred to as *type parameterization*, allows the compiler to find more type errors at compile time and can help avoid casting.

■ **Note** The concept of a variable type, or type parameterization, is closely related to the concept of *generics* that were introduced in CLR version 2.0 and have now become part of the ECMA specification for CLI version 2.0. When F# targets a CLI that has generics enabled, it takes full advantage of them by using them anywhere it finds an undetermined type. Don Syme, the creator of F#, designed and implemented generics in the .NET CLR before he started working on F#. One might be tempted to infer that he did this so he could create F#!

The function doNothingToAnInt, shown next, is an example of a value being constrained—a *type constraint*. In this case, the function parameter x is constrained to be an int. It is possible to constrain any identifier, not just function parameters, to be of a certain type, though it is more typical to need to constrain parameters. The list stringList here shows how to constrain an identifier that is not a function parameter.

```
let doNothingToAnInt (x: int) = x
let intList = [1; 2; 3]

let (stringList: list<string>) = ["one"; "two"; "three"]
```

```
val doNothingToAnInt _int : int -> int
val intList : int list
val stringList : string list
```

The syntax for constraining a value to be of a certain type is straightforward. Within parentheses, the identifier name is followed by a colon (:), followed by the type name. This is also sometimes called a *type annotation*.

The intList value is a list of integers, and the identifier's type is int list. This indicates that the compiler has recognized that the list contains only integers, and in this case, the type of its items is not undetermined but is int. Any attempt to add anything other than values of type int to the list will result in a compile error.

The identifier stringList has a type annotation. Although this is unnecessary, since the compiler can resolve the type from the value, it is used to show an alternative syntax for working with undetermined types. You can place the type between angle brackets after the type that it is associated with instead of just writing it before the type name. Note that even though the type of stringList is constrained to be list<string> (a list of strings), the compiler still reports its type as string list when displaying the type, and they mean exactly the same thing. This syntax is supported to make F# types with a type parameter look like generic types from other .NET libraries.

Constraining values is not usually necessary when writing pure F#, though it can occasionally be useful. It's most useful when using .NET libraries written in languages other than F# and for interoperation with unmanaged libraries. In both these cases, the compiler has less type information, so it is often necessary to give it enough information to disambiguate things.

Defining Types

The type system in F# provides a number of features for defining custom types. All of F#'s type definitions fall into two categories:

- *Tuples* or *records*, which are a set of types composed to form a composite type (similar to structs in C or classes in C#)
- *Sum* types, sometimes referred to as *union* types

Tuple and Record Types

Tuples are a way of quickly and conveniently composing values into a group of values. Values are separated by commas and can then be referred to by one identifier, as shown in the first line of the next example. You can then retrieve the values by doing the reverse, as shown in the second and third lines, where identifiers separated by commas appear on the left side of the equal sign, with each identifier receiving a single value from the tuple. If you want to ignore a value in the tuple, you can use _ to tell the compiler you are not interested in the value, as in the second and third lines.

```
let pair = true, false
let b1, _ = pair
let _, b2 = pair
```

Tuples are different from most user-defined types in F# because you do not need to explicitly declare them using the type keyword. To define a type, you use the type keyword, followed by the type name, an equal sign, and then the type you are defining. In its simplest form, you can use this to give an alias to any existing type, including tuples. Giving aliases to single types is not often useful, but giving aliases to tuples can be very useful, especially when you want to use a tuple as a type constraint. The next example shows how to give an alias to a single type and a tuple, and also how to use an alias as a type constraint.

```
type Name = string
type Fullname = string * string

let fullNameToSting (x: Fullname) =
    let first, second = x in
    first + " " + second
```

Record types are similar to tuples in that they compose multiple types into a single type. The difference is that in record types, each *field* is named. The next example illustrates the syntax for defining record types.

```
// define an organization with unique fields
type Organization1 = { boss: string; lackeys: string list }
// create an instance of this organization
let rainbow =
    { boss = "Jeffrey";
      lackeys = ["Zippy"; "George"; "Bungle"] }

// define two organizations with overlapping fields
type Organization2 = { chief: string; underlings: string list }
type Organization3 = { chief: string; indians: string list }

// create an instance of Organization2
let (thePlayers: Organization2) =
    { chief = "Peter Quince";
      underlings = ["Francis Flute"; "Robin Starveling";
                    "Tom Snout"; "Snug"; "Nick Bottom"] }
```

```
// create an instance of Organization3
let (wayneManor: Organization3) =
    { chief = "Batman";
      indians = ["Robin"; "Alfred"] }
```

You place field definitions between braces and separate them with semicolons. A field definition is composed of the field name followed by a colon and the field's type. The type definition `Organization1` is a record type where the field names are unique. This means you can use a simple syntax to create an instance of this type where there is no need to mention the type name when it is created. To create a record, you place the field names followed by equal signs and the field values between braces ({}), as shown in the `Rainbow` identifier.

F# does not force field names to be unique, so sometimes the compiler cannot infer the type of a field from the field names alone. In such a case, the compiler cannot infer the type of the record. To create records with nonunique fields, the compiler needs to statically know the type of the record being created. If the compiler cannot infer the type of the record, you need to use a type annotation, as described in the previous section. Using a type annotation is illustrated by the types `Organization2` and `Organization3`, and their instances `thePlayers` and `wayneManor`. You can see the type of the identifier given explicitly just after its name.

Accessing the fields in a record is fairly straightforward. You simply use the syntax `record` identifier name, followed by a dot, followed by field name. The following example illustrates this, showing how to access the `chief` field of the `Organization` record.

```
// define an organization type
type Organization = { chief: string; indians: string list }

// create an instance of this type
let wayneManor =
    { chief = "Batman";
      indians = ["Robin"; "Alfred"] }

// access a field from this type
printfn "wayneManor.chief = %s" wayneManor.chief
```

Records are immutable by default. To an imperative programmer, this may sound like records are not very useful, since there will inevitably be situations where you need to change a value in a field. For this purpose, F# provides a simple syntax for creating a copy of a record with updated fields. To create a copy of a record, place the name of that record between braces, followed by the keyword `with`, followed by a list of fields to be changed, with their updated values. The advantage of this is that you don't need to retype the list of fields that have not changed. The following example demonstrates this approach. It creates an initial version of `wayneManor` and then creates `wayneManor'`, in which `"Robin"` has been removed.

```
// define an organization type
type Organization = { chief: string; indians: string list }

// create an instance of this type
let wayneManor =
    { chief = "Batman";
      indians = ["Robin"; "Alfred"] }
```

```
// create a modified instance of this type
let wayneManor' =
    { wayneManor with indians = [ "Alfred" ] }

// print out the two organizations
printfn "wayneManor = %A" wayneManor
printfn "wayneManor' = %A" wayneManor'
```

The results of this example, when compiled and executed, are as follows:

```
wayneManor = {chief = "Batman";
 indians = ["Robin"; "Alfred"];}
wayneManor' = {chief = "Batman";
 indians = ["Alfred"];}
```

Another way to access the fields in a record is using pattern matching; that is, you can use pattern matching to match fields within the record type. As you would expect, the syntax for examining a record using pattern matching is similar to the syntax used to construct it. You can compare a field to a constant with *field* = *constant*. You can assign the values of fields with identifiers with *field* = *identifier*. You can ignore a field with *field* = _. The `findDavid` function in the next example illustrates using pattern matching to access the fields in a record.

```
// type representing a couple
type Couple = { him : string ; her : string }

// list of couples
let couples =
    [ { him = "Brad" ; her = "Angelina" };
      { him = "Becks" ; her = "Posh" };
      { him = "Chris" ; her = "Gwyneth" };
      { him = "Michael" ; her = "Catherine" } ]

// function to find "David" from a list of couples
let rec findDavid l =
    match l with
    | { him = x ; her = "Posh" } :: tail -> x
    | _ :: tail -> findDavid tail
    | [] -> failwith "Couldn't find David"

// print the results
printfn "%A" (findDavid couples)
```

The first rule in the `findDavid` function is the one that does the real work, checking the `her` field of the record to see whether it is `"Posh"`, David's wife. The `him` field is associated with the identifier `x` so it can be used in the second half of the rule.

The results of this example, when compiled and executed, are as follows:

```
Becks
```

It's important to note that you can use only literal values when you pattern match over records like this. So, if you wanted to generalize the function to allow you to change the person you are searching for, you would need to use a `when` guard in your pattern matching:

```
let rec findPartner soughtHer l =
    match l with
    | { him = x ; her = her } :: tail when her = soughtHer -> x
    | _ :: tail -> findPartner soughtHer tail
    | [] -> failwith "Couldn't find him"
```

Field values can also be functions. Since this technique is mainly used in conjunction with a mutable state to form values similar to objects, I will cover that usage in Chapter 5.

Union or Sum Types

Union types, sometimes called *sum types* or *discriminated unions*, are a way of bringing together data that may have a different meaning or structure.

You define a union type using the `type` keyword, followed by the type name, followed by an equal sign, just as with all type definitions. Then comes the definition of the different *constructors*, separated by vertical bars. The first vertical bar is optional.

A constructor is composed of a name that must start with a capital letter, which is intended to avoid the common bug of getting constructor names mixed up with identifier names. The name can optionally be followed by the keyword `of` and then the types that make up that constructor. Multiple types that make up a constructor are separated by asterisks. The names of constructors within a type must be unique. If several union types are defined, then the names of their constructors can overlap; however, you should be careful when doing this, because it can be that further type annotations are required when constructing and consuming union types.

The next example defines a type `Volume` whose values can have three different meanings: liter, US pint, or imperial pint. Although the structure of the data is the same and is represented by a float, the meanings are quite different. Mixing up the meaning of data in an algorithm is a common cause of bugs in programs, and the `Volume` type is, in part, an attempt to avoid this.

```
type Volume =
    | Liter of float
    | UsPint of float
    | ImperialPint of float

let vol1 = Liter 2.5
let vol2 = UsPint 2.5
let vol3 = ImperialPint (2.5)
```

The syntax for constructing a new instance of a union type is the constructor name followed by the values for the types, with multiple values separated by commas. Optionally, you can place the values in parentheses. You use the three different `Volume` constructors to construct three different identifiers: `vol1`, `vol2`, and `vol3`.

To deconstruct the values of union types into their basic parts, you always use pattern matching. When pattern matching over a union type, the constructors make up the first half of the pattern-matching rules. You don't need a complete list of rules, but if the list is incomplete, there must be a default rule, using either an identifier or a wildcard to match all remaining rules. The first part of a rule for a constructor consists of the constructor name followed by identifiers or wildcards to match the various values within it. The following `convertVolumeToLiter`, `convertVolumeUsPint`, and `convertVolumeImperialPint` functions demonstrate this syntax:

```
// type representing volumes
type Volume =
    | Liter of float
    | UsPint of float
    | ImperialPint of float

// various kinds of volumes
let vol1 = Liter 2.5
let vol2 = UsPint 2.5
let vol3 = ImperialPint 2.5

// some functions to convert between volumes
let convertVolumeToLiter x =
    match x with
    | Liter x -> x
    | UsPint x -> x * 0.473
    | ImperialPint x -> x * 0.568
let convertVolumeUsPint x =
    match x with
    | Liter x -> x * 2.113
    | UsPint x -> x
    | ImperialPint x -> x * 1.201
let convertVolumeImperialPint x =
    match x with
    | Liter x -> x * 1.760
    | UsPint x -> x * 0.833
    | ImperialPint x -> x

// a function to print a volume
let printVolumes x =
    printfn "Volume in liters = %f,
in us pints = %f,
in imperial pints = %f"
        (convertVolumeToLiter x)
        (convertVolumeUsPint x)
        (convertVolumeImperialPint x)
```

```
// print the results
printVolumes vol1
printVolumes vol2
printVolumes vol3
```

The results of these examples, when compiled and executed, are as follows:

```
Volume in liters = 2.500000,
in us pints = 5.282500,
in imperial pints = 4.400000
Volume in liters = 1.182500,
in us pints = 2.500000,
in imperial pints = 2.082500
Volume in liters = 1.420000,
in us pints = 3.002500,
in imperial pints = 2.500000
```

An alternative solution to this problem is to use F#'s units of measure. This is discussed in the "Units of Measure" section later in the chapter.

Type Definitions with Type Parameters

Both union and record types can be parameterized. Parameterizing a type means leaving one or more of the types within the type being defined to be determined later by the consumer of the types. This is a similar concept to the variable types discussed earlier in this chapter. When defining types, you must be a little more explicit about which types are variable.

F# supports two syntaxes for type parameterization. In the first, you place the type being parameterized between the keyword type and the name of the type, as follows:

```
type 'a BinaryTree =
| BinaryNode of 'a BinaryTree * 'a BinaryTree
| BinaryValue of 'a

let tree1 =
    BinaryNode(
        BinaryNode ( BinaryValue 1, BinaryValue 2),
        BinaryNode ( BinaryValue 3, BinaryValue 4) )
```

In the second syntax, you place the types being parameterized in angle brackets after the type name, as follows:

```
type Tree<'a> =
| Node of Tree<'a> list
| Value of 'a
```

```
let tree2 =
    Node( [ Node( [Value "one"; Value "two"] ) ;
         Node( [Value "three"; Value "four"] ) ]  )
```

Like variable types, the names of type parameters always start with a single quote (') followed by an alphanumeric name for the type. Typically, just a single letter is used. If multiple parameterized types are required, you separate them with commas. You can then use the type parameters throughout the type definition. The previous examples defined two parameterized types using the two different syntaxes that F# offers. The `BinaryTree` type used OCaml-style syntax, where the type parameters are placed before the name of the type. The `tree` type used .NET-style syntax, with the type parameters in angle brackets after the type name.

The syntax for creating and consuming an instance of a parameterized type does not change from that of creating and consuming a nonparameterized type. This is because the compiler will automatically infer the type parameters of the parameterized type. You can see this in the following construction of `tree1` and `tree2`, and their consumption by the functions `printBinaryTreeValues` and `printTreeValues`:

```
// definition of a binary tree
type 'a BinaryTree =
    | BinaryNode of 'a BinaryTree * 'a BinaryTree
    | BinaryValue of 'a

// create an instance of a binary tree
let tree1 =
    BinaryNode(
        BinaryNode ( BinaryValue 1, BinaryValue 2),
        BinaryNode ( BinaryValue 3, BinaryValue 4) )

// definition of a tree
type Tree<'a> =
    | Node of Tree<'a> list
    | Value of 'a

// create an instance of a tree
let tree2 =
    Node( [ Node( [Value "one"; Value "two"] ) ;
        Node( [Value "three"; Value "four"] ) ] )

// function to print the binary tree
let rec printBinaryTreeValues x =
    match x with
    | BinaryNode (node1, node2) ->
        printBinaryTreeValues node1
        printBinaryTreeValues node2
    | BinaryValue x ->
        printf "%A, " x
```

```
// function to print the tree
let rec printTreeValues x =
    match x with
    | Node l -> List.iter printTreeValues l
    | Value x ->
        printf "%A, " x

// print the results
printBinaryTreeValues tree1
printfn ""
printTreeValues tree2
```

The results of this example, when compiled and executed, are as follows:

```
1, 2, 3, 4,
"one", "two", "three", "four",
```

You may have noticed that although I've discussed defining types, creating instances of them, and examining these instances, I haven't discussed updating them. It is not possible to update these kinds of types, because the idea of a value that changes over time goes against the idea of functional programming. However, F# does have some types that are updatable, and I discuss them in Chapter 4.

Recursive Type Definitions

Ordinarily, the scope of a type definition is from where it is declared forward to the end of the source file in which it is declared. If a type needs to reference a type declared later, it cannot typically do so. The only reason you'll need to do this is if types are *mutually recursive*.

F# provides a special syntax for defining types that are mutually recursive. The types must be declared together, in the same block. Types declared in the same block must be declared next to each other; that is, without any value definitions in between, and the keyword `type` is replaced by the keyword `and` for every type definition after the first one.

Types declared in this way are not any different from types declared the regular way. They can reference any other type in the block, and they can even be mutually referential.

The next example shows how you might represent an XML tree in F#, using union types and record types. Two types in this example are mutually recursive, `XmlElement` and `XmlTree`, declared in the same block. If they were declared separately, `XmlElement` would not be able to reference `XmlTree` because `XmlElement` is declared before `XmlTree`; because their declarations are joined with the keyword `and`, `XmlElement` can have a field of type `XmlTree`.

```
// represents an XML attribute
type XmlAttribute =
    { AttribName: string;
      AttribValue: string; }
```

```
// represents an XML element
type XmlElement =
    { ElementName: string;
      Attributes: list<XmlAttribute>;
      InnerXml: XmlTree }

// represents an XML tree
and XmlTree =
  | Element of XmlElement
  | ElementList of list<XmlTree>
  | Text of string
  | Comment of string
  | Empty
```

Active Patterns

Active patterns provide a flexible new way to use F#'s pattern-matching constructs. They allow you to execute a function to see whether a match has occurred or not, which is why they are called *active*. Their design goal is to permit you to make better reuse of pattern-matching logic in your application.

All active patterns take an input and then perform some computation with that input to determine whether a match has occurred. There are two sorts of active patterns:

- *Complete active patterns* allow you to break a match down into a finite number of cases.
- *Partial active patterns* can either match or fail.

First, we'll look at complete active patterns.

Complete Active Patterns

The syntax for defining an active pattern is similar to the syntax for defining a function. The key difference is that the identifier that represents an active pattern is surrounded by *banana brackets*, which are formed of parentheses and vertical bars (`(| |)`). The names of the different cases of the active pattern go between the banana brackets, separated by vertical bars. The body of the active pattern is just an F# function that must return each case of the active pattern given in the banana brackets. Each case may also return additional data, just like a union type. This can be seen in the first part of the following example, which shows an active pattern for parsing input string data.

```
open System

// definition of the active pattern
let (|Bool|Int|Float|String|) input =
    // attempt to parse a bool
    let success, res = Boolean.TryParse input
    if success then Bool(res)
    else
```

```
        // attempt to parse an int
        let success, res = Int32.TryParse input
        if success then Int(res)
        else
            // attempt to parse a float (Double)
            let success, res = Double.TryParse input
            if success then Float(res)
            else String(input)

// function to print the results by pattern
// matching over the active pattern
let printInputWithType input =
    match input with
    | Bool b -> printfn "Boolean: %b" b
    | Int i -> printfn "Integer: %i" i
    | Float f -> printfn "Floating point: %f" f
    | String s -> printfn "String: %s" s

// print the results
printInputWithType "true"
printInputWithType "12"
printInputWithType "-12.1"
```

The pattern is designed to decide if the input string is a Boolean, integer, floating-point, or string value. The case names are `Bool`, `Int`, `Float`, and `String`. The example uses the `TryParse` method provided by the base class library to decide, in turn, if the input value is a Boolean, integer, or floating-point value; if it is not one of these, then it is classified as a string. If parsing is successful, this is indicated by returning the case name along with the value parsed.

In the second half of the example, you see how the active pattern is used. The active pattern allows you to treat a string value as if it were a union type. You can match against each of the four cases and recover the data returned by the active pattern in a strongly typed manner.

The results of this example, when compiled and executed, are as follows:

```
Boolean: true
String: 12
Floating point: -12.100000
```

Incomplete Active Patterns

To define an incomplete active pattern, you use a syntax similar to that for a complete active pattern. An incomplete active pattern has only one case name, which is placed between banana brackets, as with the complete active pattern. The difference is that an incomplete active pattern must be followed by a vertical bar and an underscore to show it is incomplete (as opposed to a complete active pattern with just one case).

Remember that the key difference between complete and incomplete active patterns is that complete active patterns are guaranteed to return one of their cases; whereas active patterns either

match or fail to match. So, an incomplete active pattern is the `option` type. The `option` type is simple union type that is already built into the F# base libraries. It has just two cases: `Some` and `None`. It has the following definition:

```
type option<'a> =
    | Some of 'a
    | None
```

This type, as it name suggests, is used to represent either the presence or absence of a value. So, an incomplete active pattern returns either `Some`, along with any data to be returned, to represent a match, or `None` to represent failure.

All active patterns can have additional parameters, as well as the input they act on. Additional parameters are listed before the active pattern's input.

The next example reimplements the problem from the previous example using an incomplete active pattern that represents the success or failure of a .NET regular expression. The regular expression pattern will be given as a parameter to the active pattern.

```
open System.Text.RegularExpressions

// the definition of the active pattern
let (|Regex|_|) regexPattern input =
    // create and attempt to match a regular expression
    let regex = new Regex(regexPattern)
    let regexMatch = regex.Match(input)
    // return either Some or None
    if regexMatch.Success then
        Some regexMatch.Value
    else
        None

// function to print the results by pattern
// matching over different instances of the
// active pattern
let printInputWithType input =
    match input with
    | Regex "$true|false^" s -> printfn "Boolean: %s" s
    | Regex @"$-?\d+^" s -> printfn "Integer: %s" s
    | Regex "$-?\d+\.\d*^" s -> printfn "Floating point: %s" s
    | _ -> printfn "String: %s" input

// print the results
printInputWithType "true"
printInputWithType "12"
printInputWithType "-12.1"
```

While complete active patterns behave in exactly the same way as a union type—meaning the compiler will raise a warning only if there are missing cases—an incomplete active pattern will always

require a final catch-all case to avoid the compiler raising a warning. However incomplete active patterns do have the advantage that you can chain multiple active patterns together, and the first case that matches will be the one that is used. This can be seen in the preceding example, which chains three of the regular expression active patterns together. Each active pattern is parameterized with a different regular expression pattern: one to match Boolean input, another to match integer input, and the third to match floating-point input.

The results of this example, when compiled and executed, are as follows:

```
Boolean: true
String: 12
Floating point: -12.1
```

Units of Measure

Units of measure are an interesting addition to the F# type system. They allow you to classify numeric values into different units. The idea of this is to prevent you from accidentally using a numeric value incorrectly—for example, adding together a value that represents inches with a value that represents centimeters without first performing the proper conversion.

To define a unit of measure, you declare a type name and prefix it with the attribute `Measure`. Here is an example of creating a unit of type meters (abbreviated to m):

```
[<Measure>]type m
```

By default, units of measure work with floating-point values—that is, `System.Double`. To create a value with a unit, you simply postfix the value with the name of the unit in angled brackets. So, to create a value of the meter type, use the following syntax:

```
let meters = 1.0<m>
```

Now we are going to revisit the example from the "Defining Types" section, which used union types to prevent various units of volume from being mixed up. This example implements something similar using units of measure. It starts by defining a unit of measure for liters and another for pints. Then it defines two identifiers that represent different volumes: one with a pint unit and one with a liter. Finally, it tries to add these two values, an operation that should result in an error, since we can't add pints and liters without first converting them.

```
[<Measure>]type liter
[<Measure>]type pint

let vol1 = 2.5<liter>
let vol2 = 2.5<pint>

let newVol = vol1 + vol2
```

This program will not compile, resulting in the following error:

```
Program.fs(7,21): error FS0001: The unit of measure 'pint' does not match the unit of measure
'liter'
```

The addition or subtraction of different units of measure is not allowed, but the multiplication or division of different units of measure is allowed and will create a new unit of measure. For example, you know that to convert a pint to a liter, you need to multiply it by the ratio of liters to pints. One liter is made up of approximately 1.76 pints, so you can now calculate the correct conversion ratio in the program:

```
let ratio =  1.0<liter> / 1.76056338<pint>
```

The identifier `ratio` will have the type `float<liter/pint>`, which makes it clear that it is the ratio of liters to pints. Furthermore, when a value of type `float<pint>` is multiplied by a value of type `float<liter/pint>`, the resulting type will automatically be of type `float<liter>`, as you would expect. This means you can now write the following program, which ensures that pints are safely converted to liters before adding them:

```
// define some units of measure
[<Measure>]type liter
[<Measure>]type pint

// define some volumes
let vol1 = 2.5<liter>
let vol2 = 2.5<pint>

// define the ratio of pints to liters
let ratio =  1.0<liter> / 1.76056338<pint>

// a function to convert pints to liters
let convertPintToLiter pints =
    pints * ratio

// perform the conversion and add the values
let newVol = vol1 + (convertPintToLiter vol2)
```

Exceptions and Exception Handling

Defining exceptions in F# is similar to defining a constructor of a union type, and the syntax for handling exceptions is similar to pattern matching.

You define exceptions using the `exception` keyword, followed by the name of the exception, and then optionally the keyword `of` and the types of any values the exception should contain, with multiple types separated by asterisks. The next example shows the definition of an exception, `WrongSecond`, which contains one integer value.

```
exception WrongSecond of int
```

You can raise exceptions with the `raise` keyword, as shown in the `else` clause in the following `testSecond` function. F# also has an alternative to the `raise` keyword, the `failwith` function, as shown in the following `if` clause. If, as is commonly the case, you just want to raise an exception with a text description of what went wrong, you can use `failwith` to raise a generic exception that contains the text passed to the function.

```
// define an exception type
exception WrongSecond of int

// list of prime numbers
let primes =
    [ 2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59 ]

// function to test if current second is prime
let testSecond() =
    try
        let currentSecond = System.DateTime.Now.Second in
        // test if current second is in the list of primes
        if List.exists (fun x -> x = currentSecond) primes then
            // use the failwith function to raise an exception
            failwith "A prime second"
        else
            // raise the WrongSecond exception
            raise (WrongSecond currentSecond)
    with
    // catch the wrong second exception
    WrongSecond x ->
        printf "The current was %i, which is not prime" x

// call the function
testSecond()
```

As shown in `testSecond`, the `try` and `with` keywords handle exceptions. The expressions that are subject to error handling go between the `try` and `with` keywords, and one or more pattern-matching rules must follow the `with` keyword. When trying to match an F# exception, the syntax follows that of trying to match an F# constructor from a union type. The first half of the rule consists of the exception name, followed by identifiers or wildcards to match values that the exception contains. The second half of the rule is an expression that states how the exception should be handled. One major difference between this and the regular pattern-matching constructs is that no warning or error is issued if pattern matching is incomplete. This is because any exceptions that are unhandled will propagate until they reach the top level and stop execution. The example handles exception `wrongSecond`, while leaving the exception raised by `failwith` to propagate.

F# also supports a `finally` keyword, which is used with the `try` keyword. You can't use the `finally` keyword in conjunction with the `with` keyword. The `finally` expression will be executed whether or not an exception is thrown. The next example shows a `finally` block being used to ensure a file is closed and disposed of after it is written to:

```
// function to write to a file
let writeToFile() =
    // open a file
    let file = System.IO.File.CreateText("test.txt")
    try
        // write to it
        file.WriteLine("Hello F# users")
    finally
        // close the file, this will happen even if
        // an exception occurs writing to the file
        file.Dispose()

// call the function
writeToFile()
```

■ **Caution** Programmers coming from an OCaml background should be careful when using exceptions in F#. Because of the architecture of the CLR, throwing an exception is pretty expensive—quite a bit more expensive than in OCaml. If you throw a lot of exceptions, profile your code carefully to decide whether the performance costs are worth it. If the costs are too high, revise the code appropriately.

Lazy Evaluation

Lazy evaluation goes hand in hand with functional programming. The theory is that if there are no side effects in the language, the compiler or runtime is free to choose the evaluation order of expressions.

As you know, F# allows functions to have side effects, so it's not possible for the compiler or runtime to have a free hand in function evaluation; therefore, F# is said to have a strict evaluation order, or to be a *strict language*. You can still take advantage of lazy evaluation, but you must be explicit about which computations can be delayed—that is, evaluated in a lazy manner.

You use the keyword `lazy` to delay a computation (invoke lazy evaluation). The computation within the lazy expression remains unevaluated until evaluation is explicitly forced with the `force` function from the `Lazy` module. When the `force` function is applied to a particular lazy expression, the value is computed, and the result is cached. Subsequent calls to the `force` function return the cached value–whatever it is—even if this means raising an exception.

The following code shows a simple use of lazy evaluation:

```
let lazyValue = lazy ( 2 + 2 )
let actualValue = Lazy.force lazyValue

printfn "%i" actualValue
```

The first line delays a simple expression for evaluation later. The next line forces evaluation. Finally, the value is printed.

The value has been cached, so any side effects that take place when the value is computed will occur only the first time the lazy value is forced. This is fairly easy to demonstrate, as shown by the next example.

```
let lazySideEffect =
    lazy
        ( let temp = 2 + 2
          printfn "%i" temp
          temp )

printfn "Force value the first time: "
let actualValue1 = Lazy.force lazySideEffect
printfn "Force value the second time: "
let actualValue2 = Lazy.force lazySideEffect
```

In this example, a lazy value has a side effect when it is calculated: it writes to the console. To show that this side effect takes place only once, it forces the value twice. As you can see from the result, writing to the console takes place only once:

```
Force value the first time:
4
Force value the second time:
```

Laziness can also be useful when working with collections. The idea of a lazy collection is that elements in the collection are calculated on demand. Some collection types also cache the results of these calculations, so there is no need to recalculate elements. The collection most commonly used for lazy programming in F# is the `seq` type, a shorthand for the BCL's `IEnumerable` type. `seq` values are created and manipulated using functions in the `Seq` module. Many other values are also compatible with the type `seq`; for example, all F# lists and arrays are compatible with this type, as are most other collection types in the F# libraries and the .NET BCL.

Possibly the most important function for creating lazy collections, and probably the most difficult to understand, is `unfold`. This function allows you to create a lazy list. What makes it complicated is that you must provide a function that will be repeatedly evaluated to provide the elements of the list. The function passed to `Seq.unfold` can take any type of parameter and must return an `option` type. An `option` type is a union type that can be either `None` or `Some(x)`, where `x` is a value of any type. `None` is used to represent the end of a list. The `Some` constructor must contain a tuple. The first item in the tuple represents the value that will become the first value in the list. The second value in the tuple is the value that will be passed into the function the next time it is called. You can think of this value as an accumulator.

The next example shows how this works. The identifier `lazyList` will contain three values. If the value passed into the function is less than 13, it appends the list using this value to form the list element, and then adds 1 to the value passed to the list. This will be the value passed to the function the next time it is called. If the value is greater than or equal to 13, the example terminates the list by returning `None`.

```
// the lazy list definition
let lazyList =
    Seq.unfold
        (fun x ->
```

```
            if x < 13 then
                // if smaller than the limit return
                // the current and next value
                Some(x, x + 1)
            else
                // if great than the limit
                // terminate the sequence
                None)
        10

// print the results
printfn "%A" lazyList
```

The results of this example, when compiled and executed, are as follows:

```
10
11
12
```

Sequences are useful to represent lists that don't terminate. A nonterminating list can't be represented by a classic list, which is constrained by the amount of memory available. The next example demonstrates this by creating `fibs`, an infinite list of all the Fibonacci numbers. To display the results conveniently, the example uses the function `Seq.take` to turn the first 20 items into an F# list, but carries on calculating many more Fibonacci numbers, as it uses F# `bigint` integers, so it is not limited by the size of a 32-bit integer.

```
// create an infinite list of Fibonacci numbers
let fibs =
    Seq.unfold
        (fun (n0, n1) ->
            Some(n0, (n1, n0 + n1)))
        (1I,1I)

// take the first twenty items from the list
let first20 = Seq.take 20 fibs

// print the finite list
printfn "%A" first20
```

The results of this example are as follows:

```
[1I; 1I; 2I; 3I; 5I; 8I; 13I; 21I; 34I; 55I; 89I; 144I; 233I; 377I; 610I; 987I;
 1597I; 2584I; 4181I; 6765I]
```

Note that both of these sequences could also be created using the list comprehension discussed earlier in this chapter. If list comprehensions are based on sequences, they are automatically lazy.

Summary

In this chapter, you looked at the major functional programming constructs in F#. This is the core of the language, and I hope you've developed a good feel for how to approach writing algorithms and handling data in F#. The next chapter covers imperative programming, and you'll see how to mix functional and imperative programming techniques to handle tasks such as input and output.