5

LET'S GET FUNCTIONAL

I've mentioned several times that F# is a functional language, but as you've learned from previous chapters you can build rich applications in F# without using any functional techniques. Does that mean that F# isn't really a functional language? No. F# is a general-purpose, multiparadigm language that allows you to program in the style most suited to your task. It is considered a functional-first language, meaning that its constructs encourage a functional style. In other words, when developing in F# you should favor functional approaches whenever possible and switch to other styles as appropriate.

In this chapter, we'll see what functional programming really is and how functions in F# differ from those in other languages. Once we've established that foundation, we'll explore several data types commonly used with functional programming and take a brief side trip into lazy evaluation.

What Is Functional Programming?

Functional programming takes a fundamentally different approach toward developing software than object-oriented programming. While object-oriented programming is primarily concerned with managing an ever-changing system state, functional programming emphasizes immutability and the application of deterministic functions. This difference drastically changes the way you build software, because in object-oriented programming you're mostly concerned with defining classes (or structs), whereas in functional programming your focus is on defining functions with particular emphasis on their input and output.

F# is an impure functional language where data is immutable by default, though you can still define mutable data or cause other side effects in your functions. Immutability is part of the functional concept called *referential transparency*, which means that an expression can be replaced with its result without affecting the program's behavior. For example, if you can replace `let sum = add 5 10` with `let sum = 15` without otherwise affecting the program's behavior, then `add` is said to be referentially transparent. But immutability and referential transparency are only two aspects of functional programming, and they certainly don't make a language functional on their own.

Programming with Functions

If you've never done any "real" functional programming, F# will forever change the way you think about functions because its functions closely resemble mathematical functions in both structure and behavior. For example, Chapter 3 introduced the `unit` type, but I avoided discussing its importance in functional programming. Unlike C# and Visual Basic, F# makes no distinction between functions that return values and those that don't. In fact, every function in F# accepts exactly one input value and returns exactly one output value. The `unit` type enables this behavior. When a function doesn't have any specific input (no parameters), it actually accepts `unit`. Similarly, when a function doesn't have any specific output, it returns `unit`.

The fact that every F# function returns a value allows the compiler to make certain assumptions about your code. One important assumption is that the result of the last evaluated expression in a function is the function's return value. This means that although `return` is a keyword in F#, you don't need to explicitly identify return values.

Functions as Data

A defining (and arguably the most important) characteristic of any functional language is that it treats functions like any other data type. The .NET Framework has always supported this concept to some degree with delegation, but until relatively recently delegation was too cumbersome to be viable in all but a few limited scenarios. Only when LINQ was introduced

with the goodness of lambda expressions and the built-in generic delegate types (`Action` and `Func`) did delegation reach its full potential. F# uses delegation behind the scenes, but unlike C# and Visual Basic, its syntax abstracts away the delegation with the `->` token. The `->` token, generally read as "goes to" or "returns," identifies a value as a *function value* where the data type specified on the left is the function's input type and the data type on the right is its return type. For example, the signature for a function that both accepts and returns a string is `string -> string`. Similarly, a parameterless function that returns a string is represented as `unit -> string`.

Signatures become increasingly complex when you begin working with *higher-order functions*—functions that accept or return other functions. Higher-order functions are used extensively in F# (and functional programming in general) because they allow you to isolate common parts of functions and substitute the parts that change.

In some ways, higher-order functions are to functional programming what interfaces are to object-oriented programming. For example, consider a function that applies a transformation to a string and prints the result. Its signature might look something like `(string -> string) -> string -> unit`. This simple notation goes a long way toward making your code more comprehensible than when you're dealing with the delegates directly.

NOTE *You can use the function signatures in type annotations whenever you're expecting a function. As with other data types, though, the compiler can often infer the function type.*

Interoperability Considerations

Despite the fact that F# functions are ultimately based on delegation, be careful when working with libraries written in other .NET languages, because the delegate types aren't interchangeable. F# functions rely on the overloaded `FSharpFunc` delegate types, whereas traditional .NET delegates are often based on the `Func` and `Action` types. If you need to pass `Func` and `Action` delegates into an F# assembly, you can use the following class to simplify the conversion.

```
open System.Runtime.CompilerServices

[<Extension>]
type public FSharpFuncUtil =
  [<Extension>]
  static member ToFSharpFunc<'a, 'b> (func : System.Func<'a, 'b>) =
    fun x -> func.Invoke(x)

  [<Extension>]
  static member ToFSharpFunc<'a> (act : System.Action<'a>) =
    fun x -> act.Invoke(x)
```

The `FSharpFuncUtil` class defines the overloaded `ToFSharpFunc` method as traditional .NET extension methods (via the `ExtensionAttribute` on both the class and methods) so you can easily call them from another language. The

first overload handles converting single-parameter `Func` instances, while the second handles single-parameter `Action` instances. These extension methods don't cover every use case, but they're certainly a good starting point.

Currying

Functions in F# work a bit differently than you're probably accustomed to. For example, consider the simple `add` function, introduced in Chapter 2.

```
let add a b = a + b
```

You might think that `add` accepts two parameters, but that's not how F# functions work. Remember, in F# every function accepts exactly one input and returns exactly one output. If you create the preceding binding in FSI or hover over the name in Visual Studio, you'll see that its signature is:

```
val add : a:int -> b:int -> int
```

Here, the name `add` is bound to a function that accepts an integer (`a`) and returns a function. The returned function accepts an integer (`b`) and returns an integer. Understanding this automatic function chaining—called *currying*—is critical to using F# effectively because it enables several other features that affect how you design functions.

To better illustrate how currying actually works, let's rewrite `add` to more closely resemble the compiled code.

```
> let add a = fun b -> (+) a b;;

val add : a:int -> b:int -> int
```

The most significant thing here is that both this and the previous version have exactly the same signature. Here, though, `add` accepts only a single parameter (`a`) and returns a separate function as defined by a lambda expression. The returned function accepts the second parameter (`b`) and invokes the multiplication operator as another function call.

Partial Application

One of the capabilities unlocked by curried functions is partial application. *Partial application* allows you to create new functions from existing ones simply by supplying some of the arguments. For example, in the case of `add`, you could use partial application to create a new `addTen` function that always adds 10 to a number.

```
> let addTen = add 10;;

val addTen : ❶(int -> int)
```

```
> addTen 10;;
val it : int = 20
```

Notice at ❶ how `addTen`'s definition and signature are listed. Although we didn't explicitly include any parameters in the definition, the signature is still a function that both accepts and returns an integer. The compiler evaluated the curried `add` function as far as it could with the provided arguments (just `10`, in this case) and bound the resulting function to the name, `addTen`.

Currying applies arguments one at a time, from left to right, so partially applied arguments must correspond to the function's first parameters.

WARNING *Once you're comfortable with currying and partial application, you may start thinking that you could simulate them in C# or Visual Basic by returning `Func` or `Action` instances. Don't. Neither language is designed to support this type of functional programming, so simulating these concepts is inelegant at best and immensely error prone at worst.*

Pipelining

Another feature often associated with currying (and used extensively in F#) is pipelining. *Pipelining* allows you to create your own function chains by evaluating one expression and sending the result to another function as the final argument.

Forward Pipelining

Usually you'll send values forward to the next function using the *forward pipelining operator* (`|>`). If you don't want to do anything with a function's result when it returns something other than `unit`, you can pipe the result forward to the `ignore` function like this:

```
add 2 3 |> ignore
```

Pipelining isn't restricted to simple scenarios like ignoring a result. As long as the last argument of the receiving function is compatible with the source function's return type, you can create complex function chains. For example, suppose you have a list of daily temperatures in degrees Fahrenheit and want to find the average temperature, convert it to Celsius, and print the result. You could do it the old-fashioned, procedural way by defining a binding for each step, or you could use pipelining to chain the steps like this:

```
let fahrenheitToCelsius degreesF = (degreesF - 32.0) * (5.0 / 9.0)

let marchHighTemps = [ 33.0; 30.0; 33.0; 38.0; 36.0; 31.0; 35.0;
                       42.0; 53.0; 65.0; 59.0; 42.0; 31.0; 41.0;
                       49.0; 45.0; 37.0; 42.0; 40.0; 32.0; 33.0;
                       42.0; 48.0; 36.0; 34.0; 38.0; 41.0; 46.0;
                       54.0; 57.0; 59.0 ]
```

```
marchHighTemps
|> List.average
|> fahrenheitToCelsius
|> printfn "March Average (C): %f"
```

Here the `marchHighTemps` list is piped to the `List` module's `average` function. The `average` function is then evaluated and its result passed on to the `fahrenheitToCelsius` function. Finally, the average temperature in Celsius is passed along to `printfn`.

Backward Pipelining

Like its forward counterpart, the *backward pipelining operator* (`<|`) sends the result of an expression to another function as the final argument, but does it from right to left instead. Because it changes precedence within an expression, the backward pipelining operator is sometimes used as a replacement for parentheses.

The backward pipelining operator can change the semantics of your code. For instance, in the `fahrenheitToCelsius` example in the previous section, the emphasis is on the list of temperatures because that's what's listed first. To change the semantics to emphasize the output, you could place the `printfn` function call ahead of the backward pipelining operator.

```
printfn "March Average (F): %f" <| List.average marchHighTemps
```

Noncurried Functions

Although pipelining is typically associated with curried functions, it also works with noncurried functions (like methods) that accept only a single argument. For instance, to force a delay in execution you could pipe a value into the `TimeSpan` class's static `FromSeconds` method and then send the resulting `TimeSpan` object to `Thread.Sleep`, as shown here.

```
5.0
|> System.TimeSpan.FromSeconds
|> System.Threading.Thread.Sleep
```

Because neither the `TimeSpan` class nor the `Thread` class is defined in F#, the functions aren't curried, but you can see how we can chain these functions together with the forward pipelining operator.

Function Composition

Like pipelining, *function composition* allows you to create function chains. It comes in two forms: forward (`>>`) and backward (`<<`).

Function composition is subject to the same rules as pipelining regarding inputs and outputs. Where function composition differs is that instead of

defining a one-time operation, the composition operators actually generate new functions. Continuing with our average temperature example, you could easily create a new function from the `List.average` and `fahrenheitToCelsius` functions with the forward composition operator.

```
> let averageInCelsius = List.average >> fahrenheitToCelsius;;

val averageInCelsius : (float list -> float)
```

The composition operator results in a new function that accepts a list of floats and returns a float. Now, instead of calling the two functions independently, you can simply call `averageInCelsius` instead.

```
printfn "March average (C): %f" <| averageInCelsius marchHighTemps
```

As with pipelining, you can compose functions from noncurried functions. For instance, you could compose the forced delay example from "Noncurried Functions" on page 108 as well.

```
> let delay = System.TimeSpan.FromSeconds >> System.Threading.Thread.Sleep;;

val delay : (float -> unit)
```

As you might expect, you can now call the `delay` function to temporarily pause execution.

```
> delay 5.0;;
val it : unit = ()
```

Recursive Functions

There are typically three looping constructs associated with imperative code: `while` loops, simple `for` loops, and enumerable `for` loops. Because each relies on a state change to determine when the exit criteria have been met, you'll need to take a different approach to looping when writing purely functional code. In functional programming, the preferred looping mechanism is *recursion*. A *recursive function* is one that calls itself either directly or indirectly through another function.

Although methods within a type are implicitly recursive, `let`-bound functions, such as those defined within a module, are not. To make a `let`-bound function recursive, you must include the `rec` keyword in its definition, as this factorial function illustrates.

```
let rec factorial v =
  match v with | 1L -> 1L
               | _ -> v * factorial (v - 1L)
```

The `rec` keyword instructs the compiler to make the function name available within the function but does not otherwise change the function's signature (`int64 -> int64`).

Tail-Call Recursion

The preceding factorial example is simple, but it suffers from a major flaw. For example, consider what happens when you call `factorial 5`. On each recursive iteration (other than when the value is 1), the function calculates the product of `v` and the factorial of `v - 1`. In other words, calculating the factorial for a given value inherently requires each subsequent factorial call to complete. At run time, it looks a bit like this:

```
5L * (factorial 4L)
5L * (4L * (factorial 3L))
5L * (4L * (3L * (factorial 2L)))
-- snip --
```

The preceding snippet shows that each call is added to the stack. It's unlikely that this would be a problem with a factorial function, since the calculation can quickly overflow the data type, but more complex recursion scenarios could result in running out of stack space. To address this problem, you can revise the function to use a *tail call* by removing the dependency on subsequent iterations, as shown here:

```
❶ let factorial v =
    let ❷rec fact c p =
      match c with | 0L -> p
                   | _ -> ❸fact <| c - 1L <| c * p
    ❹fact v 1L
```

The revised factorial function ❶ creates and then calls a nested recursive function, `fact` ❷, to isolate the implementation details. The `fact` function accepts both the current iteration value (`c`) and the product calculated by the previous iteration (`p`). At ❸ (the nonzero case), the `fact` function makes the recursive call. (Notice how only the arguments to the recursive call are calculated here.) Finally, to initiate recursion, the `factorial` function ❹ invokes the first `fact` iteration, passing the supplied value and `1L`.

Although the recursive call is still present in the code, when the F# compiler detects that no iteration is dependent on subsequent iterations, it optimizes the compiled form by replacing the recursion with an imperative loop. This allows the system to iterate as long as necessary. You can observe this optimization by examining the stack traces for each version by inserting a breakpoint and looking at the call stack window (if you're running this as a console application) or by printing out the stack information returned from `System.Diagnostics.StackTrace`, as shown here. (Note that your namespaces will likely vary.)

```
Standard recursion
   at FSI_0024.printTrace()
   at FSI_0028.factorial(Int64 v)
   at FSI_0028.factorial(Int64 v)
   at FSI_0028.factorial(Int64 v)
   at FSI_0028.factorial(Int64 v)
   at FSI_0028.factorial(Int64 v)
   at <StartupCode$FSI_0029>.$FSI_0029.main@()
   -- snip --

Tail recursion
   at FSI_0024.printTrace()
   at FSI_0030.fact@75-8(Int64 c, Int64 p)
   at <StartupCode$FSI_0031>.$FSI_0031.main@()
   -- snip --
```

Mutually Recursive Functions

When two or more functions call each other recursively, they are said to be *mutually recursive*. Like mutually recursive types (described in Chapter 4), mutually recursive functions must be defined together with the `and` keyword. For example, a Fibonacci number calculation is easily expressed through mutual recursion.

```
let fibonacci n =
  let rec f = function
              | 1 -> 1
              | n -> g (n - 1)
  and g = function
          | 1 -> 0
          | n -> g (n - 1) + f (n - 1)
  f n + g n
```

The preceding `fibonacci` function defines two mutually recursive functions, `f` and `g`. (The function keyword inside each is a shortcut for pattern matching.) For all values other than 1, `f` calls `g`. Similarly, `g` recursively calls itself and `f`.

Because the mutual recursion is hidden inside `fibonacci`, consumers of this code can simply call `fibonacci` directly. For example, to compute the sixth number in the Fibonacci sequence you'd write:

```
> fibonacci 6;;
val it : int = 8
```

Mutual recursion can be useful, but this example is really only good for illustrating the concept. For performance reasons, a more realistic Fibonacci example would likely forego mutual recursion in favor of a technique called *memoization*, where expensive computations are performed once and the results are cached to avoid calculating the same values multiple times.

Lambda Expressions

If you've ever used LINQ or done any other functional programming, you're probably already familiar with *lambda expressions* (or *function expressions*, as they're sometimes called). Lambda expressions are used extensively in functional programming. In brief, they provide a convenient way to define simple, single-use, anonymous (unnamed) functions. Lambda expressions are typically favored over let-bound functions when the function is significant only within its context (such as when filtering a collection).

Lambda expression syntax is similar to that of a function value except that it begins with the `fun` keyword, omits the function identifier, and uses the arrow token (`->`) in place of an equal sign. For example, you could express the Fahrenheit-to-Celsius conversion function inline as a lambda expression and immediately evaluate it like this:

```
(fun degreesF -> (degreesF - 32.0) * (5.0 / 9.0)) 212.0
```

Although defining ad hoc functions like this is certainly one use for lambda expressions, they're more commonly created inline with calls to higher-order functions, or included in pipeline chains.

Closures

Closures enable functions to access values visible in the scope where a function is defined regardless of whether that value is part of the function. Although closures are typically associated with lambda expressions, nested functions created with `let` bindings can be closures as well, since ultimately they both compile to either an `FSharpFunc` or a formal method. Closures are typically used to isolate some state. For instance, consider the quintessential closure example—a function that returns a function that manipulates an internal counter value, as shown here:

```
let createCounter() =
  let count = ref 0
  (fun () -> count := !count + 1
             !count)
```

The `createCounter` function defines a reference cell that's captured by the returned function. Because the reference cell is in scope when the returned function is created, the function has access to it no matter when it's called. This allows you to simulate a stateful object without a formal type definition.

To observe the function modifying the reference cell's value, we just need to invoke the generated function and call it like this:

```
let increment = createCounter()
for i in [1..10] do printfn "%i" (increment())
```

Functional Types

F# includes native support for several additional data types. These types—tuples, records, and discriminated unions—are typically associated with functional programming, but they're often useful in mixed-paradigm development as well. While each of these types has a specific purpose, they're all intended to help you remain focused on the problem your software is trying to solve.

Tuples

The most basic functional type is the *tuple*. Tuples are a convenient way to group a number of values within a single immutable construct without creating a custom type. Tuples are expressed as comma-delimited lists and are sometimes enclosed in parentheses. For example, the following two definitions representing geometric points as tuples are equally valid.

```
> let point1 = 10.0, 10.0;;

val point1 : float * float = (10.0, 10.0)

> let point2 = (20.0, 20.0);;

val point2 : float * float = (20.0, 20.0)
```

The signature for a tuple type includes the type of each value separated by an asterisk (*). The asterisk is used as the tuple element delimiter for mathematical reasons: Tuples represent the Cartesian product of all values their elements contain. Therefore, to express a tuple in a type annotation, you write it as an asterisk-delimited list of types like this:

```
let point : float * float = 0.0, 0.0
```

Despite some syntactic similarities, particularly when the values are enclosed in parentheses, it's important to recognize that other than the fact that they contain multiple values, tuples aren't collections; they simply group a fixed number of values within a single construct. The tuple types don't implement `IEnumerable<'T>`, so they can't be enumerated or iterated over in an enumerable `for` loop, and individual tuple values are exposed only through properties with nonspecific names like `Item1` and `Item2`.

TUPLES IN .NET

Tuples have always been part of F# but were only introduced to the larger .NET Framework with .NET 4. Prior to .NET 4, the tuple classes were located in the `FSharp.Core` library, but they have since been moved to `mscorlib`. This difference is only really important if you intend to write cross-language code against earlier versions of the .NET Framework, because it affects which assembly you reference.

Extracting Values

Tuples are often useful for returning multiple values from a function or for sending multiple values to a function without currying them. For instance, to calculate the slope of a line you could pass two points as tuples to a `slope` function. To make the function work, though, you'll need some way to access the individual values. (Fortunately, tupled values are always accessible in the order in which they're defined, so some of the guesswork is eliminated.)

When working with *pairs* (tuples containing two values like the geometric points we discussed previously), you can use the `fst` and `snd` functions to retrieve the first and second values, respectively, as shown here.

```
let slope p1 p2 =
  let x1 = fst p1
  let y1 = snd p1
  let x2 = fst p2
  let y2 = snd p2
  (y1 - y2) / (x1 - x2)

slope (13.0, 8.0) (1.0, 2.0)
```

Notice how we define bindings for the various coordinates with the `fst` and `snd` functions. As you can see, however, extracting each value this way can get pretty tedious and these functions work only with pairs; if you were to try either against a *triple* (a tuple with three values), you'd get a type mismatch. (The reason is that at their core, tuples compile to one of the nine generic overloads of the `Tuple` class.) Aside from sharing a common name, the tuple classes are independent of each other and are otherwise incompatible.

A more practical approach to extract tuple values involves introducing a *Tuple pattern*. Tuple patterns allow you to specify an identifier for each value in the tuple by separating the identifiers with commas. For example, here's the `slope` function revised to use Tuple patterns instead of the pair functions.

```
let slope p1 p2 =
  let x1, y1 = p1
  let x2, y2 = p2
  (y1 - y2) / (x1 - x2)
```

You can see how Tuple patterns may help, but you need to be careful with them. If your pattern doesn't match the number of values in the tuple, you'll get a type mismatch.

Fortunately, unlike the pair functions, resolving the problem is simply a matter of adding or removing identifiers. If you don't care about a particular value in your Tuple pattern, you can ignore it with the Wildcard pattern (_). For instance, if you have three-dimensional coordinates but care only about the z-coordinate, you could ignore the x- and y-values as follows:

```
> let _, _, z = (10.0, 10.0, 10.0);;

val z : int = 10
```

Tuple patterns aren't limited to `let` bindings. In fact, we can make a further revision to the `slope` function and include the patterns right in the function signature!

```
let slope (x1, y1) (x2, y2) = (y1 - y2) / (x1 - x2)
```

Equality Semantics

Despite the fact that they're formally reference types, each of the built-in tuple types implements the `IStructuralEquatable` interface. This ensures that all equality comparisons involve comparing the individual component values rather than checking that two tuple instances reference the same `Tuple` object in memory. In other words, two tuple instances are considered equal when the corresponding component values in each instance are the same, as shown here:

```
> (1, 2) = (1, 2);;
val it : bool = true
> (2, 1) = (1, 2);;
val it : bool = false
```

For the same reasons that the `fst` and `snd` functions work only with pairs, comparing tuples of different lengths will cause an error.

Syntactic Tuples

So far, all of the tuples we've looked have been concrete ones, but F# also includes *syntactic tuples*. For the most part, syntactic tuples are how F# works around noncurried functions in other languages. Because F# functions always accept a single parameter, but functions in C# and Visual Basic can

accept more than one, in order to call functions from libraries written in other languages you can use a syntactic tuple and let the compiler work out the details.

For example, the `String` class's `Format` method accepts both a format string and a `params` array of values. If `String.Format` were a curried function, you'd expect its signature to be something like `Format : format:string -> params args : obj [] -> string`, but it's not. Instead, if you hover your cursor over the function name in Visual Studio, you'll see that its signature is actually `Format(format:string, params args : obj []) : string`. This distinction is significant because it means that the arguments must be applied as a group rather than individually as they would with curried functions. If you were to try invoking the method as a curried F# function, you'd get an error like this:

```
> System.String.Format "hello {0}" "Dave";;

  System.String.Format "hello {0}" "Dave";;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

stdin(3,1): error FS0003: This value is not a function and cannot be applied
```

The correct way to call `String.Format` in F# is with a syntactic tuple, like this:

```
> System.String.Format ("hello {0}", "Dave");;
val it : string = "hello Dave"
```

You've probably noticed that F# generally doesn't require parentheses around arguments when calling a function; it uses parentheses primarily to establish precedence. Because functions are applied from left to right, you'll mainly use parentheses in a function call to pass the result of another function as an argument. In this case, the parentheses around the arguments are necessary. Without them, the left-to-right evaluation would cause the compiler to essentially treat the expression as `((System.String.Format "hello {0}"), "Dave")`. In general, it's good practice to include parentheses around syntactic tuples in order to remove any ambiguity.

Out Parameters

F# doesn't directly support `out` parameters—parameters passed by reference with values assigned in the method body so they can be returned to the caller. To fully support the .NET Framework, however, F# needs a way to access `out` parameter values. For example, the `TryParse` methods on the various numeric data type classes attempt to convert a string to the corresponding numeric type and return a Boolean value indicating success or failure. If the conversion succeeds, the `TryParse` methods set the `out` parameter to the appropriate converted value. For instance, calling `System.Int32.TryParse` with `"10"` would return `true` and set the `out` parameter to `10`. Similarly, calling the same function with `"abc"` would return `false` and leave the `out` parameter unchanged.

In C#, calling `System.Int32.TryParse` would look like this:

```
// C#
❶ int v;
var r = System.Int32.TryParse("10", out v);
```

The problem with `out` parameters in a functional language is that they require a side effect, as shown by the uninitialized variable at ❶. To work around this problem, the F# compiler converts the return value and `out` parameter to a pair. Therefore, when you invoke a method with an `out` parameter in F#, you treat it exactly like any other tuple-returning function.

Calling the same `Int32.TryParse` method in F# looks like this:

```
// F#
let r, v = System.Int32.TryParse "10"
```

For a behind-the-scenes look at the generated class, we can once again turn to ILSpy to see how it's represented in C#.

```
// C#
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
namespace <StartupCode$Samples>
{
  internal static class $Samples
  {
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    internal static readonly Tuple<bool, int> patternInput@3;
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    internal static readonly int v@3;
    [DebuggerBrowsable(DebuggerBrowsableState.Never)]
    internal static readonly bool r@3;
    [DebuggerBrowsable(DebuggerBrowsableState.Never), DebuggerNonUserCode, CompilerGenerated]
    internal static int init@;
  ❶static $Samples()
    {
      int item = 0;
      $Samples.patternInput@3 = ❷new Tuple<bool, int>(❸int.TryParse("10", out item), item);
      ❹$Samples.v@3 = Samples.patternInput@3.Item2;
      ❺$Samples.r@3 = Samples.patternInput@3.Item1;
    }
  }
}
```

Here, the F# compiler wrapped the `Int32.TryParse` call inside a static class. The generated class's static constructor ❶ invokes `TryParse` at ❸ and wraps the results in a tuple at ❷. Then, the internal `v@3` and `r@3` fields are assigned to the `out` parameter value and the return value at ❹ and ❺, respectively. In turn, the `v` and `r` values defined by the `let` binding are compiled to read-only properties that return the `v@3` and `r@3` values.

Record Types

Like tuples, *record types* allow you to group values in a single immutable construct. You might think of them as bridging the functional gap between tuples and your own classes. Record types provide many of the same conveniences as tuples, like simple syntax and value equality semantics, while offering you some control over their internal structure and allowing you to add custom functionality.

Defining Record Types

Record type definitions consist of the `type` keyword, an identifier, and a list of labels with type annotations all enclosed in braces. For example, this listing shows a simple record type representing an RGB color.

```
> type rgbColor = { R : byte; G : byte; B : byte };;

type rgbColor =
  {R: byte;
   G: byte;
   B: byte;}
```

If you take a peek at what the compiler generates from this definition, you'll see a sealed class with read-only properties, equality semantics, and a single constructor to initialize all values.

NOTE *When defining record types on a single line, you must separate each label and type annotation pair by semicolons. If you place each pair on a separate line, you can safely omit the semicolons.*

Creating Records

New records are created via *record expressions*. Record expressions allow you to specify a value for each label in the record type. For example, you could create a new `rgbColor` instance using a record expression, as shown next. (Note that, as when defining a record type, you must separate each label or assignment pair by semicolons or place it on a line of its own.)

```
> let red = { R = 255uy; G = 0uy; B = 0uy };;

val red : rgbColor = {R = 255uy;
                      G = 0uy;
                      B = 0uy;}
```

Notice that nowhere in the record expression do we include an explicit reference to the `rgbColor` type. This is another example of F#'s type inference engine at work. Based on the labels alone, the compiler was able to infer that we were creating an instance of `rgbColor`. Because the compiler relies on the labels rather than position to determine the correct type, order doesn't matter. This means that you can place the label and value pairs in any order. Here, we create an `rgbColor` instance with the labels in `G`, `B`, `R` order.

```
> let red = { G = 0uy; B = 0uy; R = 255uy };;

val red : rgbColor = {R = 255uy;
                      G = 0uy;
                      B = 0uy;}
```

Unlike with tuples, we don't need to use special value extraction functions like `fst` or `snd` with record types, because each value can be accessed by its label. For instance, a function that converts an `rgbColor` value to its hexadecimal string equivalent might look like this:

```
let rgbColorToHex (c : rgbColor) =
  sprintf "#%02X%02X%02X" c.R c.G c.B
```

Avoiding Naming Conflicts

The compiler can usually infer the correct type, but it's possible to define two record types with the same structure. Consider what happens when you add a `color` type with the same structure as `rgbColor`.

```
> type rgbColor = { R : byte; G : byte; B : byte }
type color = { R : byte; G : byte; B : byte };;

type rgbColor =
  {R: byte;
   G: byte;
   B: byte;}
type color =
  {R: byte;
   G: byte;
   B: byte;}

> let red = { R = 255uy; G = 0uy; B = 0uy };;

val red : ❶color = {R = 255uy;
                    G = 0uy;
                    B = 0uy;}
```

Despite having two record types with the same structure, type inference still succeeds, but notice at ❶ that the resulting type is `color`. Due to F#'s top-down evaluation, the compiler uses the most recently defined type that matches the labels. If your goal was to define `red` as `color` you'd be fine, but if you wanted `rgbColor` instead you'd have to be a bit more explicit in your record expression and include the type name, as shown here:

```
> let red = { ❶rgbColor.R = 255uy; G = 0uy; B = 0uy };;

val red : ❷rgbColor = {R = 255uy;
                       G = 0uy;
                       B = 0uy;}
```

By qualifying one of the names with the type name at ❶, you bypass type inference and the correct type is resolved ❷. (Although you can technically qualify the type on any name, the convention is to do it on either the first one or all of them.)

Copying Records

Not only can you use record expressions to create new record instances from scratch, but you can also use them to create new record instances from existing ones by copying values forward and setting new values for one or more properties. The alternate syntax, called a *copy and update record expression*, makes it easy to create yellow from red, as shown here:

```
> let red = { R = 255uy; G = 0uy; B = 0uy }
let yellow = { red with G = 255uy };;

val red : color = {R = 255uy;
                   G = 0uy;
                   B = 0uy;}
val yellow : color = {R = 255uy;
                      G = 255uy;
                      B = 0uy;}
```

To specify new values for multiple properties, separate them with semicolons.

Mutability

Like virtually everything else in F#, record types are immutable by default. However, because their syntax is so convenient, they're commonly used in place of classes. In many cases, though, these scenarios require mutability. To make record type properties mutable within F#, use the `mutable` keyword just as with a `let` binding. For instance, you could make all of `rgbColor`'s members mutable like this:

```
> type rgbColor = { mutable R : byte
                   mutable G : byte
                   mutable B : byte };;

type rgbColor =
  {mutable R: byte;
   mutable G: byte;
   mutable B: byte;}
```

When a record type property is mutable, you can change its value with the standard assignment operator (`<-`) like this:

```
let myColor = { R = 255uy; G = 255uy; B = 255uy }
myColor.G <- 100uy
```

CLIMUTABLE

Although record types support binary serialization by default, other forms of serialization require a default constructor and writable properties. To allow for more situations where record types can be used in favor of classes, the F# team introduced the `CLIMutable` attribute in F# 3.0.

Decorating a record type with this attribute instructs the compiler to include a default constructor and to make the generated properties read/write, but the compiler doesn't expose those capabilities within F#. Even though the generated properties are writable, unless they're explicitly marked as mutable with the `mutable` keyword in the record type definition, their values can't be changed in F# code. For this reason, be careful when using `CLIMutable` record types across language boundaries to ensure that you don't inadvertently change something.

Additional Members

Because record types are really just syntactic sugar for classes, you can define additional members just as you would on a class. For example, you could augment `rgbColor` with a method that returns its hexadecimal string equivalent like this:

```
type rgbColor = { R : byte; G : byte; B : byte }
                member x.ToHexString() =
                  sprintf "#%02X%02X%02X" x.R x.G x.B
```

Now you can call the `ToHexString` method on any `rgbColor` instance.

```
> red.ToHexString();;
val it : string = "#FF0000"
```

Additional members on record types can also be static. For example, suppose you wanted to expose a few common colors as static properties on a record type. You could do this:

```
type rgbColor = { R : byte; G : byte; B : byte }
                -- snip --
                static member Red = { R = 255uy; G = 0uy; B = 0uy }
                static member Green = { R = 0uy; G = 255uy; B = 0uy }
                static member Blue = { R = 0uy; G = 0uy; B = 255uy }
```

The static `Red`, `Green`, and `Blue` properties behave like any other static member and can be used anywhere you need an `rgbColor` instance.

```
> rgbColor.Red.ToHexString();;
val it : string = "#FF0000"
```

You can also create custom operators for your record types as static members. Let's implement the addition operator to add two `rgbColor` instances.

```
open System
type rgbColor = { R : byte; G : byte; B : byte }
                -- snip --
                static member (+) (l : rgbColor, r : rgbColor) =
                  { R = Math.Min(255uy, l.R + r.R)
                    G = Math.Min(255uy, l.G + r.G)
                    B = Math.Min(255uy, l.B + r.B) }
```

The operator overload on `rgbColor` is defined and invoked like any other operator:

```
> let yellow = { R = 255uy; G = 0uy; B = 0uy } +
               { R = 0uy; G = 255uy; B = 0uy };;

val yellow : rgbColor = {R = 255uy;
                         G = 255uy;
                         B = 0uy;}
```

Discriminated Unions

Discriminated unions are user-defined data types whose values are restricted to a known set of values called *union cases*. There are no equivalent structures in the other popular .NET languages.

At first glance, you might mistake some simple discriminated unions for enumerations because their syntax is so similar, but they're entirely different constructs. For one, enumerations simply define labels for known integral values, but they aren't restricted to those values. By contrast, the only valid values for discriminated unions are their union cases. Furthermore, each union case can either stand on its own or contain associated immutable data.

The built-in `Option<'T>` type highlights each of these points. We're really only interested in its definition here, so let's take a look at that.

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

`Option<'T>` defines two cases, `None` and `Some`. `None` is an empty union case, meaning that it doesn't contain any associated data. On the other hand, `Some` has an associated instance of `'T` as indicated by the `of` keyword.

To demonstrate how discriminated unions enforce a specific set of values, let's define a simple function that accepts a generic option and writes out the associated value when the option is `Some`, or `"None"` when the option is `None`:

```
let showValue (v : _ option) =
  printfn "%s" (match v with
                | Some x -> x.ToString()
                | None -> "None")
```

When we invoke this function, we simply need to provide one of the option cases:

```
> Some 123 |> showValue;;
123
val it : unit = ()
> Some "abc" |> showValue;;
abc
val it : unit = ()
> None |> showValue;;
None
val it : unit = ()
```

Notice how in each of the three calls to `showValue`, we specified only the union case names. The compiler resolved both `Some` and `None` as `Option<'T>`. (In the event of a naming conflict, you can qualify the case names with the discriminated union name just as you would with a record type.) However, if you were to call `showValue` with a value other than `Some` or `None`, the compiler will raise an error like this:

```
> showValue "xyz";;

  showValue "xyz";;
  ----------^^^^^

stdin(9,11): error FS0001: This expression was expected to have type
    Option<'a>
but here has type
    string
```

Defining Discriminated Unions

Like other types, discriminated union definitions begin with the `type` keyword. Union cases are delimited with bars. The bar before the first union case is optional, but omitting it when there's only one case can be confusing because it will make the definition look like a type abbreviation. In fact, if you omit the bar in a single-case discriminated union and there is no data associated with the case, the compiler will treat the definition as a type abbreviation when there is a naming conflict with another type.

The normal rules for identifiers apply when you are defining union cases, with one exception: Union case names must begin with an uppercase letter to help the compiler differentiate union cases from other identifiers in pattern matching. If a case name does not begin with an uppercase letter, the compiler will raise an error.

In practice, discriminated unions typically serve one of three purposes:

- Representing simple object hierarchies
- Representing tree structures
- Replacing type abbreviations

Simple Object Hierarchies

Discriminated unions are commonly used to represent simple object hierarchies. In fact, they excel at this task so much that they're often used as a substitute for formal classes and inheritance.

Imagine working on a system that needs some basic geometry functionality. In an object-oriented environment, such functionality would probably consist of an `IShape` interface and a number of concrete shape classes like `Circle`, `Rectangle`, and `Triangle`, with each implementing `IShape`. A possible implementation might look like this:

```
type IShape = interface end

type Circle(r : float) =
  interface IShape
  member x.Radius = r

type Rectangle(w : float, h : float) =
  interface IShape
  member x.Width = w
  member x.Height = h

type Triangle(l1 : float, l2 : float, l3 : float) =
  interface IShape
  member x.Leg1 = l1
  member x.Leg2 = l2
  member x.Leg3 = l3
```

Discriminated unions offer a cleaner alternative that is less prone to side effects. Here's what that same object hierarchy might look like as a discriminated union:

```
type Shape =
/// Describes a circle by its radius
| Circle of float
/// Describes a rectangle by its width and height
| Rectangle of ❶float * float
/// Describes a triangle by its three sides
| Triangle of ❷float * float * float
```

IT'S BIGGER ON THE INSIDE

Discriminated unions are much more complex than their syntax might lead you to believe. Each discriminated union compiles to an abstract class responsible for handling equality and comparison semantics as well as type checking and union case creation. Similarly, each union case compiles to a class that is both nested within and inherits from the union class. The union case classes define the properties and backing stores for each of their associated values along with an internal constructor.

Although it's possible to replicate some of the discriminated union functionality within other languages, doing so is nontrivial. Proving just how complex discriminated unions really are, inspecting the compiled `Shape` type we just defined in ILSpy reveals nearly 700 lines of C# code!

The `Shape` type defines three cases: `Circle`, `Rectangle`, and `Triangle`. Each case has at least one attached value specific to the shape it represents. Notice at ❶ and ❷ how the tuple syntax is used to associate multiple data values with a case. But despite using the tuple syntax, cases don't actually compile to tuples. Instead, each associated data item compiles to an individual property that follows the tuple naming pattern (that is, `Item1`, `Item2`, and so on). This distinction is important because there's no direct conversion from a union case to a tuple, meaning that you can't use them interchangeably. The only real exception to this rule is that when the types are wrapped in parentheses the compiler will interpret the grouping as a tuple. In other words, the compiler treats `of string * int` and `of (string * int)` differently; the former is tuple-like, while the latter actually is a tuple. Unless you really need a true tuple, though, use the default format.

As you'd expect, creating `Shape` instances is the same as creating `Option<'T>` instances. For example, here's how to create an instance of each case:

```
let c = Circle(3.0)
let r = Rectangle(10.0, 12.0)
let t = Triangle(25.0, 20.0, 7.0)
```

One of the major annoyances with the tuple syntax for multiple associated values is that it's easy to forget what each position represents. To work around the issue, include XML documentation comments—like those preceding each case in this section's `Shape` definition—as a reminder.

Fortunately, relief is available. One of the language enhancements in F# 3.1 is support for named union type fields. The refined syntax resembles a hybrid of the current tupled syntax and type-annotated field definitions. For example, under the new syntax, `Shape` could be redefined as follows.

```
type Shape =
| Circle of Radius : float
| Rectangle of Width : float * Height : float
| Triangle of Leg1 : float * Leg2 : float * Leg3 : float
```

For discriminated unions defined with the F# 3.1 syntax, creating new case instances is significantly more developer friendly—not only because the labels appear in IntelliSense, but also because you can use named arguments like this:

```
let c = Circle(Radius = 3.0)
let r = Rectangle(Width = 10.0, Height = 12.0)
let t = Triangle(Leg1 = 25.0, Leg2 = 20.0, Leg3 = 7.0)
```

Tree Structures

Discriminated unions can also be *self-referencing*, meaning that the data associated with a union case can be another case from the same union. This is handy for creating simple trees like this one, which represents a rudimentary markup structure:

```
type Markup =
| ContentElement of string * ❶Markup list
| EmptyElement of string
| Content of string
```

Most of this definition should be familiar by now, but notice that the `ContentElement` case has an associated string and list of `Markup` values.

The nested `Markup` list ❶ makes it trivial to construct a simple HTML document like the following. Here, `ContentElement` nodes represent elements (such as `html`, `head`, and `body`) that contain additional content, while `Content` nodes represent raw text contained within a `ContentElement`.

```
let movieList =
  ContentElement("html",
    [ ContentElement("head", [ ContentElement("title", [ Content "Guilty Pleasures" ])])
      ContentElement("body",
        [ ContentElement("article",
            [ ContentElement("h1", [ Content "Some Guilty Pleasures" ])
              ContentElement("p",
                [ Content "These are "
                  ContentElement("strong", [ Content "a few" ])
                  Content " of my guilty pleasures" ])
              ContentElement("ul",
                [ ContentElement("li", [ Content "Crank (2006)" ])
                  ContentElement("li", [ Content "Starship Troopers (1997)" ])
                  ContentElement("li", [ Content "RoboCop (1987)" ])])])])])
```

To convert the preceding tree structure to an actual HTML document, you could write a simple recursive function with a match expression to handle each union case, like this:

```
let rec toHtml markup =
  match markup with
  | ❶ContentElement (tag, children) ->
        use w = new System.IO.StringWriter()
        children
          |> Seq.map toHtml
          |> Seq.iter (fun (s : string) -> w.Write(s))
        sprintf "<%s>%s</%s>" tag (w.ToString()) tag
  | ❷EmptyElement (tag) -> sprintf "<%s />" tag
  | ❸Content (c) -> sprintf "%s" c
```

The `match` expression is used here roughly like a `switch` statement in C# or a `SELECT CASE` statement in Visual Basic. Each match case, denoted by a vertical pipe (`|`), matches against an Identifier pattern that includes the union case name and identifiers for each of its associated values. For instance, the match case at ❶ matches `ContentElement` items and represents the associated values with the `tag` and `children` identifiers within the case body (the part after the arrow). Likewise, the match cases at ❷ and ❸ match the `EmptyElement` and `Content` cases, respectively. (Note that because match expressions return a value, each match case's return type must be the same.)

Invoking the `toHtml` function with `movieList` results in the following HTML (formatted for readability). As you look over the resulting HTML, try tracing each element back to its node in `movieList`.

```
<html>
  <head>
    <title>Guilty Pleasures</title>
  </head>
  <body>
    <article>
        <h1>Some Guilty Pleasures</h1>
        <p>These are <strong>a few</strong> of my guilty pleasures</p>
        <ul>
            <li>Crank (2006)</li>
            <li>Starship Troopers (1997)</li>
            <li>RoboCop (1987)</li>
        </ul>
    </article>
  </body>
</html>
```

Replacing Type Abbreviations

Single-case discriminated unions can be a useful alternative to type abbreviations, which, while nice for aliasing existing types, don't provide any additional type safety. For instance, suppose you've defined `UserId` as an

alias for `System.Guid` and you have a function `UserId -> User`. Although the function accepts `UserId`, nothing prevents you from sending in an arbitrary `Guid`, no matter what that `Guid` actually represents.

Let's extend the markup examples from the previous section to show how single-case discriminated unions can solve this problem. If you wanted to display the generated HTML in a browser, you could define a function like this:

```
open System.IO

❶ type HtmlString = string

let displayHtml (html ❷: HtmlString) =
  let fn = Path.Combine(Path.GetTempPath(), "HtmlDemo.htm")
  let bytes = System.Text.UTF8Encoding.UTF8.GetBytes html
  using (new FileStream(fn, FileMode.Create, FileAccess.Write))
        (fun fs -> fs.Write(bytes, 0, bytes.Length))
  System.Diagnostics.Process.Start(fn).WaitForExit()
  File.Delete fn
```

The actual mechanics of the `displayHtml` function aren't important for this discussion. Instead, focus your attention on ❶ the `HtmlString` type abbreviation and ❷ the type annotation explicitly stating that the `html` parameter is an `HtmlString`.

It's clear from the signature that the `displayHtml` function expects the supplied string to contain HTML, but because `HtmlString` is merely a type abbreviation there's nothing ensuring that it actually is HTML. As written, both `movieList |> toHtml |> displayHtml` and `"abc123" |> displayHtml` are valid.

To introduce a bit more type safety, we can replace the `HtmlString` definition with a single-case discriminated union, like this:

```
type HtmlString = | HtmlString of string
```

Now that `HtmlString` is a discriminated union, we need to change the `displayHtml` function to extract the associated string. We can do this in one of two ways. The first option requires us to change the function's signature to include an Identifier pattern. Alternatively, we can leave the signature alone and introduce an intermediate binding (also using an Identifier pattern) for the associated value. The first option is cleaner, so that's the approach we'll use.

```
let displayHtml (HtmlString(html)) =
  let fn = Path.Combine(Path.GetTempPath(), "HtmlDemo.htm")
  let bytes = System.Text.UTF8Encoding.UTF8.GetBytes html
  using (new FileStream(fn, FileMode.Create, FileAccess.Write))
        (fun fs -> fs.Write(bytes, 0, bytes.Length))
  System.Diagnostics.Process.Start(fn).WaitForExit()
  File.Delete fn
```

To call the `displayHtml` function, we only need to wrap the string from the `toHtml` function in an `HtmlString` instance and pass it to `displayHtml` as follows:

```
HtmlString(movieList |> toHtml) |> displayHtml
```

Finally, we can further simplify this code by revising the `toHtml` function to return an `HtmlString` instead of a string. One approach would look like this:

```
let rec toHtml markup =
  match markup with
  | ContentElement (tag, children) ->
        use w = new System.IO.StringWriter()
        children
          |> Seq.map toHtml
          |> Seq.iter (fun ❶(HtmlString(html)) -> w.Write(html))
        HtmlString (sprintf "<%s>%s</%s>" tag (w.ToString()) tag)
  | EmptyElement (tag) -> HtmlString (sprintf "<%s />" tag)
  | Content (c) -> HtmlString (sprintf "%s" c)
```

In this revised version, we've wrapped each case's return value in an `HtmlString` instance. Less trivial, though, is ❶, which now uses an Identifier pattern to extract the HTML from the recursive result in order to write the raw text to the `StringWriter`.

With the `toHtml` function now returning an `HtmlString`, passing its result to `displayHtml` is simplified to this:

```
movieList |> toHtml |> displayHtml
```

Single-case discriminated unions can't guarantee that any associated values are actually correct, but they do offer a little extra safety in that they force developers to make conscious decisions about what they're passing to a function. Developers could create an `HtmlString` instance with an arbitrary string, but if they do they'll be forced to think about whether the data is correct.

Additional Members

Like record types, discriminated unions also allow additional members. For example, we could redefine the `toHtml` function as a method on the `Markup` discriminated union as follows:

```
type Markup =
| ContentElement of string * Markup list
| EmptyElement of string
| Content of string
```

```
member x.toHtml() =
  match x with
  | ContentElement (tag, children) ->
        use w = new System.IO.StringWriter()
        children
          |> Seq.map (fun m -> m.toHtml())
          |> Seq.iter (fun (HtmlString(html)) -> w.Write(html))
        HtmlString (sprintf "<%s>%s</%s>" tag (w.ToString()) tag)
  | EmptyElement (tag) -> HtmlString (sprintf "<%s />" tag)
  | Content (c) -> HtmlString (sprintf "%s" c)
```

Calling this method is like calling a method on any other type:

```
movieList.toHtml() |> displayHtml
```

Lazy Evaluation

By default, F# uses *eager evaluation*, which means that expressions are evaluated immediately. Most of the time, eager evaluation will be fine in F#, but sometimes you can improve perceived performance by deferring execution until the result is actually needed, through *lazy evaluation*.

F# supports a few mechanisms for enabling lazy evaluation, but one of the easiest and most common ways is through the use of the `lazy` keyword. Here, the lazy keyword is used in conjunction with a series of expressions that includes a delay to simulate a long-running operation.

```
> let lazyOperation = lazy (printfn "evaluating lazy expression"
                            System.Threading.Thread.Sleep(1000)
                            42);;

val lazyOperation : Lazy<int> = Value is not created.
```

You can see the `lazy` keyword's impact. If this expression had been eagerly evaluated, `evaluating lazy expression` would have been printed and there would have been an immediate one-second delay before it returned `42`. Instead, the expression's result is an instance of the built-in `Lazy<'T>` type. In this case, the compiler inferred the return type and created an instance of `Lazy<int>`.

NOTE *Be careful using the `lazy` type across language boundaries. Prior to F# 3.0, the `Lazy<'T>` class was located in the `FSharp.Core` assembly. In .NET 4.0, `Lazy<'T>` was moved to `mscorlib`.*

The Lazy<'T> instance created by the lazy keyword can be passed around like any other type, but the underlying expression won't be evaluated until you force that evaluation by either calling the Force method or accessing its Value property, as shown next. Convention generally favors the Force method, but it doesn't really matter whether you use it or the Value property to force evaluation. Internally, Force is just an extension method that wraps the Value property.

```
> lazyOperation.Force() |> printfn "Result: %i";;
evaluating lazy expression
Result: 42
val it : unit = ()
```

Now that we've forced evaluation, we see that the underlying expression has printed its message, slept, and returned 42. The Lazy<'T> type can also improve application performance through memoization. Once the associated expression is evaluated, its result is cached within the Lazy<'T> instance and used for subsequent requests. If the expression involves an expensive or time-consuming operation, the result can be dramatic.

To more effectively observe memoization's impact, we can enable timing in FSI and repeatedly force evaluation as follows:

```
> let lazyOperation = lazy (System.Threading.Thread.Sleep(1000); 42)
#time "on";;

val lazyOperation : Lazy<int> = Value is not created.


--> Timing now on

> lazyOperation.Force() |> printfn "Result: %i";;
Result: 42
Real: ❶00:00:01.004, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()
> lazyOperation.Force() |> printfn "Result: %i";;
Result: 42
Real: ❷00:00:00.001, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()
> lazyOperation.Force() |> printfn "Result: %i";;
Result: 42
Real: ❸00:00:00.001, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()
```

As you can see at ❶, the first time Force is called we incur the expense of putting the thread to sleep. The subsequent calls at ❷ and ❸ complete instantaneously because the memoization mechanism has cached the result.

Summary

As you've seen in this chapter, functional programming requires a different mindset than object-oriented programming. While object-oriented programming emphasizes managing system state, functional programming is more concerned with program correctness and predictability through the application of side-effect-free functions to data. Functional languages like F# treat functions as data. In doing so, they allow for greater composability within systems through concepts like higher-order functions, currying, partial application, pipelining, and function composition. Functional data types like tuples, record types, and discriminated unions help you write correct code by letting you focus on the problem you're trying to solve instead of attempting to satisfy the compiler.