CHAPTER 17

# Language-Oriented Programming: Advanced Techniques

Chapters 3 to 6 covered three well-known programming paradigms in F#: *functional*, *imperative*, and *object* programming. Throughout this book, however, you have in many ways been exploring what is essentially a fourth programming paradigm: *language-oriented programming*. In this chapter, you will focus on advanced aspects of language-oriented programming through language-integrated, domain-specific languages and meta-programming.

The word *language* can have a number of meanings in this context. For example, take the simple language of arithmetic expressions and algebra that you learned in high school mathematics, made up of named variables, such as $x$ and $y$, and composite expressions, such as $x+y$, $xy$, $-x$, and $x^2$. For the purposes of this chapter, this language can have a number of manifestations:

- One or more *concrete representations*: for example, using an ASCII text format or an XML representation of arithmetic expressions.
- One or more *abstract representations*: for example, as F# types and values representing the normalized form of an arithmetic expression tree.
- One or more *computational representations*, either by functions that compute the values of arithmetic expressions or via other engines that perform analysis, interpretation, compilation, execution, or transformation on language fragments. These can be implemented in F#, in another .NET language, or in external engines.

The language-oriented programming techniques covered in this book (including some in this chapter) are:

- Manipulating unstructured text and binary representations of languages, including writing parsers and lexers (Chapter 8)
- Manipulating semi-structured language representations, such as XML (Chapter 8)
- Using F# functions, types, and active patterns for abstract and symbolic representations of languages (Chapters 3, 9 and 12)
- Using F# sequence expressions, asynchronous expressions, and, more generally, F# computation expressions for tightly language-integrated representations of some languages (Chapters 9, 11, and 13, and this chapter)

D. Syme et al., *Expert F# 3.0*

- Using the F# "dynamic" operators to give a slightly improved syntax for accessing information stored using dynamic (untyped) representation techniques (this chapter)
- Using F# reflection and quotations to represent languages via meta-programming (this chapter)

As you can see, language-oriented programming isn't a single technique; sometimes you work with fully concrete representations (for example, reading bits from disk) and sometimes with fully computational representations (for example, defining and using functions that compute the value of arithmetic expressions). Most often, you work somewhere in between (for example, manipulating abstract syntax trees). These tasks require different techniques, and there are trade-offs among choosing to work with different kinds of representations. For example, if you're generating human-readable formulae, then you may need to store more concrete information; but if you're interested just in evaluating arithmetic expressions, then a purely computational encoding may be more effective. You see some of those trade-offs in the different techniques described above.

---

■ **Note**: The term *language-oriented programming* was originally applied to F# by Robert Pickering in the Apress book *Beginning F#*, and it really captures a key facet of F# programming. Thanks, Robert!

---

# Computation Expressions

Chapter 3 introduced a useful notation for generating sequences of data, called *sequence expressions*. For example:

```
> seq { for i in 0 .. 3 -> (i, i * i) };;

val it : seq<int * int> = seq [(0, 0); (1, 1); (2, 4); (3, 9)]
```

Sequence expressions are used extensively throughout this book. For example, Chapter 9 uses sequence expressions for a range of sequence-programming tasks.

Likewise, Chapter 11 introduced a useful notation for generating individual results *asynchronously*, meaning that the result of a computation is eventually delivered to a continuation, a form of *future*, *task*, or *promise*. Asynchronous computations can also represent actions that do not "block" .NET threads when waiting for I/O. For example, consider the body of an agent from Chapter 11, which asynchronously waits for a message and then recursively loops—this is a form of asynchyronous state machine.

```
let rec loop n =
    async { printfn "n = %d, waiting..." n
            let! msg = inbox.Receive()
            return! loop (n + msg) }
```

It turns out that both sequence expressions and asynchronous expressions are just two instances of a more general construct called *computation expressions*. These are also sometimes called *workflows*, although they bear only a passing similarity to the workflows used to model business processes. The general form of a computation expression is `builder { comp-expr }`. Table 17-1 shows the primary constructs that can be used within the braces of a computation expression and how these constructs are de-sugared by the F# compiler given a computation expression builder `builder`.

The three most important applications of computation expressions in F# programming are:

- General-purpose programming with sequences, lists, and arrays
- Parallel, asynchronous, and concurrent programming using asynchronous workflows, discussed in detail in Chapter 11
- Database queries, by quoting a workflow and translating it to SQL via the .NET LINQ libraries, a technique demonstrated in Chapter 13

This section covers briefly how computation expressions work through some simple examples.

***Table 17-1.*** *Main constructs in computation expressions and their ee-sugaring*

| Construct | De-sugared Form |
|---|---|
| `let pat = expr in cexpr` | `let pat = expr in "cexpr"` |
| `let! pat = expr in cexpr` | `b.Bind (expr, (fun pat -> "cexpr"))` |
| `use val = expr in cexpr` | `b.Using(expr,  (fun val -> "cexpr"))` |
| `use! val = expr in cexpr` | `b.Bind (expr, (fun x -> "use val = expr in cexpr"))` |
| `do expr in cexpr` | `expr; b.Delay (fun () -> "cexpr"))` |
| `do! expr in cexpr` | `b.Bind (expr, (fun () -> "cexpr"))` |
| `for pat in expr do cexpr` | `b.For (expr, (fun pat -> "cexpr"))` |
| `while expr do cexpr` | `b.While ((fun () -> expr), b.Delay (fun () -> "cexpr"))` |
| `try cexpr1 with val -> expr2` | `b.TryWith(b.Delay(fun () -> "cexpr1"),`<br>`(fun val -> "cexpr2"))` |
| `try cexpr finally expr` | `b.TryFinally(b.Delay(fun () -> "cexpr"),`<br>`(fun () -> expr))` |
| `if expr then cexpr1 else cexpr2` | `if expr then "cexpr1" else "cexpr2"` |
| `if expr then cexpr` | `if expr then "cexpr" else b.Zero()` |
| `cexpr1; cexpr2` | `v.Combine ("cexpr1", b.Delay(fun () -> "cexpr2"))` |
| `yield expr` | `b.Yield expr` |
| `yield! expr` | `b.YieldFrom expr` |
| `return expr` | `b.Return expr` |
| `return! expr` | `b.ReturnFrom expr` |

■ **Note**: If you've never seen F# workflows or Haskell monads before, you might find that workflows take a bit of getting used to. They give you a way to write computations that may behave and execute quite differently than normal programs do.

## F# WORKFLOWS AND HASKELL MONADS

Computation expressions are the F# equivalent of monadic syntax in the programming language Haskell. Monads are a powerful and expressive design pattern; they are characterized by a generic type `M<'T>` combined with at least two operations:

```
bind : M<'T> -> ('T ->  M<'U>) -> M<'U>
return : 'T -> M<'T>
```

These correspond to the primitives `let!` and `return` in the F# computation-expression syntax. Several other elements of the computation-expression syntax can be implemented in terms of these primitives, but the F# de-sugaring process leaves this up to the implementer of the workflow, because sometimes, derived operations can have more efficient implementations. Well-behaved monads should satisfy three important rules, called the *monad laws*.

F# uses the terms *computation expression* and *workflow* for four reasons:

- When the designers of F# talked with the designers of Haskell about this, they agreed that the word *monad* is obscure and sounds a little daunting and that using other names might be wise.
- There are some technical differences: for example, some F# workflows can be combined with imperative programming, utilizing the fact that workflows can have side effects that aren't tracked by the F# type system. In Haskell, all side-effecting operations must be lifted into the corresponding monad. The Haskell approach has some important advantages: you can know for sure what side effects a function can have by looking at its type. It also, however, makes it more difficult to use external libraries from within a computation expression.
- F# workflows can be reified using F# quotations, providing a way to execute the workflow by alternative means—for example, by translation to SQL. This gives them a different role in practice, because they can be used to model both concrete languages and computational languages.
- F# workflows can also be used to embed computations that generate multiple results (called *monoids*), such as sequence expressions (also known as *comprehension syntax*). These generally use `yield` and `yield!` instead of `return` and `return!` and often have no `let!`.

## An Example: Success/Failure Workflows

Perhaps the simplest kind of workflow is one in which failure of a computation is made explicit: for example, in which each step of the workflow may either *succeed*, by returning a result `Some(v)`, or *fail*, by returning the value `None`. You can model such a workflow using functions of type `unit -> 'T option`—that is, functions that may or may not compute a result. In this section, assume that these functions are pure and terminating: they have no side effects, raise no exceptions, and always terminate.

Whenever you define a new kind of workflow, it's useful to give a name to the type of values/objects generated by that workflow. In this case, let's call them `Attempt` objects:

```
type Attempt<'T> = (unit -> 'T option)
```

Of course, you can use regular functional programming to start to build `Attempt<'T>` objects:

```
let succeed x = (fun () -> Some(x)) : Attempt<'T>
let fail = (fun () -> None) : Attempt<'T>
let runAttempt (a : Attempt<'T>) = a()
```

These conform to the types:

```
val succeed : x:'T -> Attempt<'T>
val fail : Attempt<'T>
val runAttempt : a:Attempt<'T> -> 'T option
```

Using only normal F# expressions to build `Attempt` values can be a little tedious and lead to a proliferation of many different functions that stitch together `Attempt` values in straightforward ways. Luckily, as you've seen with sequence expressions, F# comes with predefined syntax for building objects such as `Attempt` values. You can use this syntax with a new type by defining a *builder* object that helps stitch together the fragments that make up the computation expression. Here's an example of the signature of an object you have to define in order to use workflow syntax with a new type (note that this is a type signature for an object, not actual code—we show how to define the `AttemptBuilder` type and its members later in this section):

```
type AttemptBuilder =
    member Bind : p:Attempt<'T> * ('T -> Attempt<'U>) -> Attempt<'U>
    member Delay : f:(unit -> Attempt<'T>) -> Attempt<'T>
    member Return : x:'T -> Attempt<'T>
    member ReturnFrom : x:Attempt<'T> -> Attempt<'T>
```

Typically, there is one global instance of each such builder object. For example:

```
let attempt = new AttemptBuilder()
```

```
val attempt : AttemptBuilder
```

First, let's see how you can use the F# syntax for workflows to build `Attempt` objects. You can build `Attempt` values that always succeed:

```
> let alwaysOne = attempt { return 1 };;

val alwaysOne : Attempt<int>

> let alwaysPair = attempt { return (1,"two") };;

val alwaysPair : Attempt<int * string>

> runAttempt alwaysOne;;

val it : int option = Some 1

> runAttempt alwaysPair;;

val it : (int * string) option = Some (1, "two")
```

Note that `Attempt` values such as `alwaysOne` are just functions; to run an `Attempt` value, just apply it. These correspond to uses of the `succeed` function, as you will see shortly.

You can also build more interesting `Attempt` values that check a condition and return different `Attempt` values on each branch, as shown in this example:

```
> let failIfBig n = attempt {if n > 1000 then return! fail else return n};;

val failIfBig : n:int -> Attempt<int>

> runAttempt (failIfBig 999);;

val it : int option = Some 999

> runAttempt (failIfBig 1001);;

val it : int option = None
```

Here, one branch uses `return!` to return the result of running another `Attempt` value, and the other uses `return` to give a single result. These correspond to `yield!` and `yield` in sequence expressions.

Next, you can build `Attempt` values that sequence together two `Attempt` values by running one, getting its result, binding it to a variable, and running the second. You do this by using the syntax form `let! pat = expr`, which is unique to computation expressions:

```
> let failIfEitherBig (inp1, inp2) = attempt {
        let! n1 = failIfBig inp1
        let! n2 = failIfBig inp2
        return (n1, n2)};;

val failIfEitherBig : inp1:int * inp2:int -> Attempt<int * int>

> runAttempt (failIfEitherBig (999, 998));;

val it : (int * int) option = Some (999, 998)

runAttempt (failIfEitherBig (1003, 998));;

val it : (int * int) option = None

> runAttempt (failIfEitherBig (999, 1001));;

val it : (int * int) option = None
```

Let's look at this more closely. First, what does the first `let!` do? It runs the `Attempt` value `failIfBig inp1`, and if this returns `None`, the whole computation returns `None`. If the computation on the right delivers a value (that is, returns `Some`), then it binds the result to the variable `n1` and continues. Note the following for the expression `let! n1 = failIfBig inp1`:

- The expression on the right (`failIfBig inp1`) has type `Attempt<int>`.
- The variable on the left (`n1`) is of type `int`.

This is somewhat similar to a sequence of normal `let` binding, but `let!` also controls whether the rest of the computation is executed. In the case of the `Attempt` type, it executes the rest of the computation only when it receives a `Some` value. Otherwise, it returns `None`, and the rest of the code is never executed.

You can use normal `let` bindings in computation expressions. For example:

```
let sumIfBothSmall (inp1, inp2) =
    attempt { let! n1 = failIfBig inp1
              let! n2 = failIfBig inp2
              let sum = n1 + n2
              return sum}
```

In this case, the `let` binding executes exactly as you would expect; it takes the expression `n1+n2` and binds its result to the value `sum`. To summarize, you've seen that computation expressions let you:

- Use an expression-like syntax to build `Attempt` computations
- Sequence these computations together using the `let!` construct
- Return results from these computations using `return` and `return!`
- Compute intermediate results using `let`

Workflows let you do a good deal more than this, as you will see in the sections that follow.

## Defining a Workflow Builder

Listing 17-1 shows the implementation of the workflow builder for `Attempt` workflows; this is the simplest definition for `AttemptBuilder`.

***Listing 17-1.*** *Defining a workflow builder*

```
let succeed x = (fun () -> Some(x))
let fail = (fun () -> None)
let runAttempt (a : Attempt<'T>) = a()
let bind p rest = match runAttempt p with None -> fail | Some r -> (rest r)
let delay f = (fun () -> runAttempt (f()))
let combine p1 p2 = (fun () -> match p1() with None -> p2() | res -> res)

type AttemptBuilder() =

    /// Used to de-sugar uses of 'let!' inside computation expressions.
    member b.Bind(p, rest) = bind p rest

    /// Delays the construction of an attempt until just before it is executed
    member b.Delay(f) = delay f

    /// Used to de-sugar uses of 'return' inside computation expressions.
    member b.Return(x) = succeed x

    /// Used to de-sugar uses of 'return!' inside computation expressions.
    member b.ReturnFrom(x : Attempt<'T>) = x

    /// Used to de-sugar uses of 'c1; c2' inside computation expressions.
    member b.Combine(p1 : Attempt<'T>, p2 : Attempt<'T>) = combine p1 p2

    /// Used to de-sugar uses of 'if .. then ..' inside computation expressions when
    /// the 'else' branch is empty
    member b.Zero() = fail

let attempt = new AttemptBuilder()
```

The inferred types here are:

```
type AttemptBuilder =
  class
    new : unit -> AttemptBuilder
    member Bind : p:Attempt<'T> * rest:('T -> Attempt<'U>) -> Attempt<'U>
    member Combine : p1:Attempt<'T> * p2:Attempt<'T> -> Attempt<'T>
    member Delay : f:(unit -> Attempt<'T>) -> Attempt<'T>
    member Return : x:'T -> Attempt<'T>
    member ReturnFrom : x:Attempt<'T> -> Attempt<'T>
    member Zero : unit -> Attempt<'T>
  end

val attempt : AttemptBuilder
```

F# implements workflows by de-sugaring computation expressions using a builder. For example, given the previous `AttemptBuilder`, the workflow

```
attempt { let! n1 = failIfBig inp1
          let! n2 = failIfBig inp2
          let sum = n1 + n2
          return sum}
```

de-sugars to

```
attempt.Bind(failIfBig inp1, (fun n1 ->
    attempt.Bind(failIfBig inp2, (fun n2 ->
        let sum = n1 + n2
        attempt.Return sum))))
```

One purpose of the F# workflow syntax is to make sure you don't have to write this sort of thing by hand. The de-sugaring of the workflow syntax is implemented by the F# compiler. Table 17-2 shows some of the typical signatures that a workflow builder needs to implement.

***Table 17-2.*** *Some typical Workflow-builder members as required by the F# compiler*

| Member | Description |
|---|---|
| `member Bind : M<'T> * ('T -> M<'U>) -> M<'U>` | Used to de-sugar `let!` and `do!` within computation expressions. |
| `member Return : 'T -> M<'T>` | Used to de-sugar `return` within computation expressions. |
| `member ReturnFrom : M<'T> -> M<'T>` | Used to de-sugar `return!` within computation expressions. |
| `member Delay : (unit -> M<'T>) -> M<'T>` | Used to ensure that side effects within a computation expression are performed when expected. |
| `member For : seq<'T> * ('T -> M<'U>) -> M<'U>` | Used to de-sugar `for ... do ...` within computation expressions. `M<'U>` can optionally be `M<unit>`. |
| `member While : (unit -> bool) * M<'T> -> M<'T>` | Used to de-sugar `while ... do ...` within computation expressions. `M<'T>` may optionally be `M<unit>`. |

| Member | Description |
|---|---|
| `member Using : 'T * ('T -> M<'T>) -> M<'T> when 'T :> IDisposable` | Used to de-sugar `use` bindings within computation expressions. |
| `member Combine : M<'T> * M<'T> -> M<'T>` | Used to de-sugar sequencing within computation expressions. The first `M<'T>` may optionally be `M<unit>`. |
| `member Zero : unit -> M<'T>` | Used to de-sugar empty `else` branches of `if/then` constructs within computation expressions. |

Most of the elements of a workflow builder are usually implemented in terms of simpler primitives. For example, assume you're defining a workflow builder for some type `M<'T>` and you already have implementations of functions `bindM` and `returnM` with the types:

```
val bindM : M<'T> -> ('T -> M<'U>) -> M<'U>
val returnM : 'T -> M<'T>
```

You can implement `Delay` using the functions:

```
let delayM f = bindM (returnM ()) f
```

You can now define an overall builder in terms of all four functions:

```
type MBuilder() =
    member b.Return(x) = returnM x
    member b.Bind(v, f) = bindM v f
    member b.Delay(f) = delayM f
```

`Let` and `Delay` may also have more efficient direct implementations, however, which is why F# doesn't insert the previous implementations automatically.

## Workflows and Untamed Side Effects

It's possible, and in some cases even common, to define workflows that cause side effects. For example, you can use `printfn` in the middle of an `Attempt` workflow:

```
let sumIfBothSmall (inp1, inp2) =
    attempt { let! n1 = failIfBig inp1
              printfn "Hey, n1 was small!"
              let! n2 = failIfBig inp2
              printfn "n2 was also small!"
              let sum = n1 + n2
              return sum }
```

Here's what happens when you call this function:

```
> runAttempt(sumIfBothSmall (999, 999));;

Hey, n1 was small!
n2 was also small!
val it : int option = Some 1998
```

```
> runAttempt(sumIfBothSmall (999, 999));;

Hey, n1 was small!

val it : int option = None
```

Side effects in workflows must be used with care, particularly because workflows are typically used to construct delayed or on-demand computations. In the previous example, printing is a fairly benign side effect. More significant side effects, such as mutable state, can also be sensibly combined with some kinds of workflows, but be sure you understand how the side effect will interact with the particular kind of workflow you're using. This example allocates a piece of mutable state that is local to the `Attempt` workflow, and this is used to accumulate the sum:

```
let sumIfBothSmall (inp1, inp2) =
    attempt { let sum = ref 0
              let! n1 = failIfBig inp1
              sum := sum.Value + n1
              let! n2 = failIfBig inp2
              sum := sum.Value + n2
              return sum.Value}
```

We leave it as an exercise for you to examine the de-sugaring of this workflow to see that the mutable reference is indeed local, in the sense that it doesn't escape the overall computation and that different executions of the same workflow use different reference cells.

As mentioned, workflows are nearly always delayed computations. As you saw in Chapter 4, delayed computations and side effects can interact. For this reason, the de-sugaring of workflow syntax inserts a `Delay` operation around the entire workflow. This

```
let printThenSeven = attempt { printf "starting..."; return 3 + 4 }
```

de-sugars to

```
let printThenSeven = attempt.Delay(fun () -> printf "starting..."; attempt.Return(3 + 4))
```

This means that "starting . . ." is printed each time the `printThenSeven` attempt object is executed.

## Computation Expressions with Custom Query Operators

F# 3.0 includes a set of extensions to computation expressions that allow builders to define additional "custom operators" associated with a computation-expression builder. This is particularly used to define query-like languages that progressively add constraints, sorting, and other declarations to a query. For example, we can change the "Attempt" builder above to use a custom query operator "condition" (to replace if/then):

```
type Attempt<'T> = (unit -> 'T option)
let succeed x = (fun () -> Some(x))
let fail = (fun () -> None)
let runAttempt (a : Attempt<'T>) = a()
let bind p rest = match runAttempt p with None -> fail | Some r -> (rest r)
let delay f = (fun () -> runAttempt (f()))
let condition p guard = (fun () ->
    match p() with
    | Some x when guard x -> Some x
```

```
    | _ -> None)

type AttemptBuilder() =
    member b.Return(x) = succeed x
    member b.Bind(p, rest) = bind p rest
    member b.Delay(f) = delay f

    [<CustomOperation("condition",MaintainsVariableSpaceUsingBind = true)>]
    member x.Condition(p, [<ProjectionParameter>] b) = condition p b

let attempt = new AttemptBuilder()
```

Note that customer operations are declared using the CustomOperation attribute. Loosely speaking, a custom operation gets to operate on the "whole" computation—any values already declared in the computation expression are packaged up into a tuple, the operation is applied, and the values are then unpackaged. The technique used to package/unpackage is either "`return/let!`" (`MaintainsVariableSpaceUsingBind` is `true`), or "`yield/for`" (`MaintainsVariableSpaceUsingBind` is `false`). Parameters to custom operations can access the variables defined in the computation expression through the `ProjectionParameter` attribute.

For example, a workflow to generate a pair of random numbers in the unit circle is:

```
let randomNumberInCircle =
    attempt { let x, y = rand(), rand()
              condition (x * x + y * y < 1.0)
              return (x, y) }
```

Note that this is simply an alternative to an if/then expression. For F# 3.0, you must use either "control flow" operators, such as if/then/else, or "custom" operators, such as condition, in your own computation expressions. Attempts to combine these are unlikely to be satisfying.

Custom operators are very rarely defined in F# 3.0 programming and are primarily for use with F# 3.0 query expressions, used in Chapter 13.

## Example: Probabilistic Workflows

Workflows provide a fascinating way to embed a range of nontrivial, nonstandard computations into F#. To give you a feel for this, this section defines a *probabilistic* workflow. That is, instead of writing expressions to compute, say, integers, you instead write expressions that compute *distributions* of integers. This case study is based on a paper by Ramsey and Pfeffer from 2002.

For the purposes of this section, you're interested in distributions over discrete domains characterized by three things:

- You want to be able to sample from a distribution (for example, sample an integer or a coin flip).
- You want to compute the support of a distribution: that is, a set of values in which all elements outside the set have zero chance of being sampled.
- You want to compute the expectation of a function over the distribution. For example, you can compute the probability of selecting element `A` by evaluating the expectation of the function (`fun x -> if x = A then 1.0 else 0.0`).

You can model this notion of a distribution with abstract objects. Listing 17-2 shows the definition of a type of distribution values and an implementation of the basic primitives `always` and `coinFlip`, which help build distributions.

***Listing 17-2.*** *Implementing probabilistic modeling using computation expressions*

```
type Distribution<'T when 'T : comparison> =
    abstract Sample : 'T
    abstract Support : Set<'T>
    abstract Expectation: ('T -> float) -> float

let always x =
    { new Distribution<'T> with
          member d.Sample = x
          member d.Support = Set.singleton x
          member d.Expectation(H) = H(x) }

let rnd = System.Random()

let coinFlip (p : float) (d1 : Distribution<'T>) (d2 : Distribution<'T>) =
    if p < 0.0 || p > 1.0 then failwith "invalid probability in coinFlip"
    { new Distribution<'T> with
          member d.Sample =
              if rnd.NextDouble() < p then d1.Sample else d2.Sample
          member d.Support = Set.union d1.Support d2.Support
          member d.Expectation(H) =
             p * d1.Expectation(H) + (1.0 - p) * d2.Expectation(H) }
```

The types of these primitives are:

```
type Distribution<'T when 'T : comparison> =
  interface
    abstract member Expectation : ('T -> float) -> float
    abstract member Sample : 'T
    abstract member Support : Set<'T>
  end

val always : x:'T -> Distribution<'T> when 'T : comparison
val coinFlip :
  p:float -> d1:Distribution<'T> -> d2:Distribution<'T> -> Distribution<'T>
    when 'T : comparison
```

The simplest distribution is `always x`; this is a distribution that always samples to the same value. Its expectation and support are easy to calculate. The expectation of a function `H` is just `H` applied to the value, and the support is a set containing the single value `x`. The next distribution defined is `coinFlip`, which is a distribution that models the ability to choose between two outcomes.

Listing 17-3 shows how you can define a workflow builder for distribution objects.

***Listing 17-3.*** *Defining a builder for probabilistic modeling using computation expressions*

```
let bind (dist : Distribution<'T>) (k : 'T -> Distribution<'U>) =
    { new Distribution<'U> with
```

```
        member d.Sample =
            (k dist.Sample).Sample
        member d.Support =
            Set.unionMany (dist.Support |> Set.map (fun d -> (k d).Support))
        member d.Expectation H =
            dist.Expectation(fun x -> (k x).Expectation H) }

type DistributionBuilder() =
    member x.Delay f = bind (always ()) f
    member x.Bind(d, f) = bind d f
    member x.Return v = always v
    member x.ReturnFrom vs = vs

let dist = new DistributionBuilder()
```

The types of these primitives are:

```
val bind :
  dist:Distribution<'T> -> k:('T -> Distribution<'U>) -> Distribution<'U>
    when 'T : comparison and 'U : comparison val dist: DistributionBuilder
```

Listing 17-4 shows the all-important `bind` primitive; it combines two distributions, using the sample from the first to guide the sample from the second. The support and expectation are calculated by taking the support from the first and splaying it over the support of the second. The expectation is computed by using the first distribution to compute the expectation of a function derived from the second. These are standard results in probability theory, and they are the basic machinery you need to get going with some interesting modeling.

Before you begin using workflow syntax, you define two derived functions to compute distributions. Listing 17-4 shows the additional derived operations for distribution objects that you use later in this example.

***Listing 17-4.*** *Defining the derived operations for probabilistic modeling using computation expressions*

```
let weightedCases (inp : ('T * float) list) =
    let rec coinFlips w l =
        match l with
        | [] -> failwith "no coinFlips"
        | [(d, _)] -> always d
        | (d, p) :: rest -> coinFlip (p / (1.0 - w)) (always d) (coinFlips (w + p) rest)
    coinFlips 0.0 inp

let countedCases inp =
    let total = Seq.sumBy (fun (_, v) -> v) inp
    weightedCases (inp |> List.map (fun (x, v) -> (x, float v / float total)))
```

The two functions `weightedCases` and `countedCases` build distributions from the weighted selection of a finite number of cases. The types are:

```
val weightedCases :
  inp:('T * float) list -> Distribution<'T> when 'T : comparison
val countedCases :
  inp:('a * int) list -> Distribution<'a> when 'a : comparison
```

For example, here is the distribution of outcomes on a fair European roulette wheel:

```
type Outcome = Even | Odd | Zero
let roulette = countedCases [ Even,18; Odd,18; Zero,1]
```

You can now use sampling to draw from this distribution:

```
> roulette.Sample;;
val it : Outcome = Even

> roulette.Sample;;
val it : Outcome = Odd
```

You can compute the expected payout of a $5 bet on Even, where you would get a $10 return:

```
> roulette.Expectation (function Even -> 10.0 | Odd -> 0.0 | Zero -> 0.0);;

val it : float = 4.864864865
```

Now, let's model another scenario. Let's say you have a traffic light with the following probability distribution for showing red/yellow/green:

```
type Light =
    | Red
    | Green
    | Yellow

let trafficLightD = weightedCases [Red, 0.50; Yellow, 0.10; Green, 0.40]
```

Drivers are defined by their behavior with respect to a traffic light. For example, a cautious driver is highly likely to brake on a yellow light and always stops on a red:

```
type Action = Stop | Drive

let cautiousDriver light =
    dist { match light with
           | Red -> return Stop
           | Yellow -> return! weightedCases [Stop, 0.9; Drive, 0.1]
           | Green -> return Drive}
```

An aggressive driver is unlikely to brake on yellow and may even go through a red light:

```
let aggressiveDriver light =
    dist { match light with
           | Red -> return! weightedCases [Stop, 0.9; Drive, 0.1]
           | Yellow -> return! weightedCases [Stop, 0.1; Drive, 0.9]
           | Green -> return Drive}
```

This gives the value of the light showing in the other direction:

```
let otherLight light =
    match light with
    | Red -> Green
    | Yellow -> Red
    | Green -> Red
```

You can now model the probability of a crash between two drivers given a traffic light. Assume there is a 10 percent chance that two drivers going through the intersection will avoid a crash:

```
type CrashResult = Crash | NoCrash

// Where the suffix D means distribution
let crash (driverOneD, driverTwoD, lightD) =
    dist { // Sample from the traffic light
           let! light = lightD

           // Sample the first driver's behavior given the traffic light
           let! driverOne = driverOneD light

           // Sample the second driver's behavior given the traffic light
           let! driverTwo = driverTwoD (otherLight light)

           // Work out the probability of a crash
           match driverOne, driverTwo with
           | Drive, Drive -> return! weightedCases [Crash, 0.9; NoCrash, 0.1]
           | _ -> return NoCrash}
```

You can now instantiate the model to a cautious/aggressive driver pair, sample the overall model, and compute the overall expectation of a crash as approximately 3.7 percent:

```
> let model = crash (cautiousDriver, aggressiveDriver, trafficLightD);;

val model : Distribution<CrashResult>

> model.Sample;;

val it : CrashResult = NoCrash
...
> model.Sample;;

val it : CrashResult = Crash

> model.Expectation (function Crash -> 1.0 | NoCrash -> 0.0);;

val it : float = 0.0369
```

■ **Note**: This section showed how to define a simplistic embedded *computational probabilistic modeling language*. There are many more efficient and sophisticated techniques to apply to the description, evaluation, and analysis of probabilistic models than those shown here, and you can make the implementation of the primitives shown here more efficient by being more careful about the underlying computational representations.

## Combining Workflows and Resources

In some situations, workflows can sensibly make use of transient resources, such as files. The tricky thing is that you still want to be careful about closing and disposing of resources when the workflow is complete or when it's no longer being used. For this reason, the workflow type must be carefully designed to correctly dispose of resources halfway through a computation, if necessary. This is useful, for example, in sequence expressions, such as this one that opens a file and reads lines on demand:

```
let linesOfFile(fileName) =
    seq { use textReader = System.IO.File.OpenText(fileName)
          while not textReader.EndOfStream do
              yield textReader.ReadLine()}
```

Chapter 4 discussed the construct `use pat = expr`. As shown in Table 17-2, you can also use this construct within workflows. In this case, the `use pat = expr` construct de-sugars into a call to `seq.Using`. In the case of sequence expressions, this function is carefully implemented to ensure that `textReader` is kept open for the duration of the process of reading from the file. Furthermore, the `Dispose` function on each generated `IEnumerator` object for a sequence calls the `textReader.Dispose()` method. This ensures that the file is closed even if you enumerate only half of the lines in the file. Workflows thus allow you to scope the lifetime of a resource over a delayed computation.

## Recursive Workflow Expressions

Like functions, workflow expressions can be defined recursively. Many of the best examples are generative sequences. For example:

```
let rnd = System.Random()

let rec randomWalk k =
    seq { yield k
          yield! randomWalk (k + rnd.NextDouble() - 0.5) }
```

```
> randomWalk 10.0;;

val it : seq<float> = seq [10.0; 10.44456912; 10.52486359; 10.07400056; ...]

> randomWalk 10.0;;

val it : seq<float> = seq [10.0; 10.03566833; 10.12441613; 9.922847582; ...]
```

# Using F# Reflection

The final topics in this chapter are *F# quotations*, which provide a way to get at a representation of F# expressions as abstract syntax trees, and *reflection*, which lets you get at representations of assemblies, type definitions, and member signatures. Let's look at reflection first.

## Reflecting on Types

One of the simplest uses of reflection is to access the representation of types and generic type variables using the `typeof` operator. For example, `typeof<int>` and `typeof<'T>` are both expressions that generate values of type `System.Type`. Given a `System.Type` value, you can use the .NET APIs to access the `System.Reflection.Assembly` value that represents the .NET assembly that contains the definition of the type (.NET assemblies are described in Chapter 19). You can also access other types in the `System.Reflection` namespace, such as `MethodInfo`, `PropertyInfo`, `MemberInfo`, and `ConstructorInfo`. The following example examines the names associated with some common types:

```
> let intType = typeof<int>;;

val intType : System.Type = System.Int32

> intType.FullName;;

val it : string = "System.Int32"

> intType.AssemblyQualifiedName;;

val it : string =
  "System.Int32, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

> let intListType = typeof<int list>;;

val intListType : System.Type =
  Microsoft.FSharp.Collections.FSharpList'1[System.Int32]

> intListType.FullName;;

val it : string =
  "Microsoft.FSharp.Collections.FSharpList'1[[System.Int32, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]"
```

## Schema Compilation by Reflecting on Types

The F# library includes the namespace `Microsoft.FSharp.Reflection`, which contains types and functions that extend the functionality of the `System.Reflection` namespace of .NET.

You can use the combination of .NET and F# reflection to provide generic implementations of language-related transformations. This section gives one example of this powerful technique. Listing 17-5 shows the definition of a *generic schema reader compiler*, in which a data schema is described using F# types and the schema compiler helps convert untyped data from comma-separated value text files into this data schema.

*Listing 17-5. Using Types and Attributes to Guide Dynamic Schema Compilation*

```
open System
open System.IO
open System.Globalization
open Microsoft.FSharp.Reflection
```

```
    /// An attribute to be added to fields of a schema record type to indicate the
    /// column used in the data format for the schema.
    type ColumnAttribute(col : int) =
        inherit Attribute()
        member x.Column = col

    /// SchemaReader builds an object that automatically transforms lines of text
    /// files in comma-separated form into instances of the given type 'Schema.
    /// 'Schema must be an F# record type where each field is attributed with a
    /// ColumnAttribute attribute, indicating which column of the data the record
    /// field is drawn from. This simple version of the reader understands
    /// integer, string and DateTime values in the CSV format.
    type SchemaReader<'Schema>() =

        // Grab the object for the type that describes the schema
        let schemaType = typeof<'Schema>

        // Grab the fields from that type
        let fields = FSharpType.GetRecordFields(schemaType)

        // For each field find the ColumnAttribute and compute a function
        // to build a value for the field
        let schema =
            fields |> Array.mapi (fun fldIdx fld ->
                let fieldInfo = schemaType.GetProperty(fld.Name)
                let fieldConverter =
                    match fld.PropertyType with
                    | ty when ty = typeof<string> -> (fun (s : string) -> box s)
                    | ty when ty = typeof<int> -> (System.Int32.Parse >> box)
                    | ty when ty = typeof<DateTime> ->
                        (fun s -> box (DateTime.Parse(s, CultureInfo.InvariantCulture)))
                    | ty -> failwithf "Unknown primitive type %A" ty

                let attrib =
                    match fieldInfo.GetCustomAttributes(typeof<ColumnAttribute>, false) with
                    | [|(:? ColumnAttribute as attrib)|] -> attrib
                    | _ -> failwithf "No column attribute found on field %s" fld.Name
                (fldIdx, fld.Name, attrib.Column, fieldConverter))

        // Compute the permutation defined by the ColumnAttribute indexes
        let columnToFldIdxPermutation c =
            schema |> Array.pick (fun (fldIdx, _, colIdx,_) ->
                if colIdx = c then Some fldIdx else None)

        // Drop the parts of the schema we don't need
        let schema =
            schema |> Array.map (fun (_, fldName, _, fldConv) -> (fldName, fldConv))

        // Compute a function to build instances of the schema type. This uses an
        // F# library function.
        let objectBuilder = FSharpValue.PreComputeRecordConstructor(schemaType)
```

```
    // OK, now we're ready to implement a line reader
    member reader.ReadLine(textReader : TextReader) =
        let line = textReader.ReadLine()
        let words = line.Split([|',','|]) |> Array.map(fun s -> s.Trim())
        if words.Length <> schema.Length then
            failwith "unexpected number of columns in line %s" line
        let words = words |> Array.permute columnToFldIdxPermutation

        let convertColumn colText (fieldName, fieldConverter) =
           try fieldConverter colText
           with e ->
               failwithf "error converting '%s' to field '%s'" colText fieldName

        let obj = objectBuilder (Array.map2 convertColumn words schema)

        // OK, now we know we've dynamically built an object of the right type
        unbox<'Schema>(obj)

    /// This reads an entire file
    member reader.ReadFile(file) =
        seq { use textReader = File.OpenText(file)
              while not textReader.EndOfStream do
                  yield reader.ReadLine(textReader)}
```

The type of the SchemaReader is simple:

```
type SchemaReader<'Schema> =
  class
    new : unit -> SchemaReader<'Schema>
    member ReadFile : file:string -> seq<'Schema>
    member ReadLine : textReader:System.IO.TextReader -> 'Schema
  end
```

First, see how the SchemaReader is used in practice. Let's say you have a text file containing lines such as:

```
Steve, 12 March 2010, Cheddar
Sally, 18 Feb 2010, Brie
...
```

It's reasonable to want to convert this data to a typed data representation. You can do this by defining an appropriate record type along with enough information to indicate how the data in the file map into this type. This information is expressed using *custom attributes*, which are a way to add extra meta-information to assembly, type, member, property, and parameter definitions. Each custom attribute is specified as an instance of a typed object, here ColumnAttribute, defined in Listing 17-5. The suffix Attribute is required when defining custom attributed but can be dropped when using them:

```
type CheeseClub =
    { [<Column(0)>] Name : string
      [<Column(2)>] FavouriteCheese : string
      [<Column(1)>] LastAttendance : System.DateTime }
```

You can now instantiate the SchemaReader type and use it to read the data from the file into this typed format:

```
> let reader = new SchemaReader<CheeseClub>();;

val reader : SchemaReader<CheeseClub>

> fsi.AddPrinter(fun (c : System.DateTime) -> c.ToString());;
> System.IO.File.WriteAllLines("data.txt",
    [|"Steve, 12 March 2010, Cheddar"; "Sally, 18 Feb 2010, Brie"|]);;

> reader.ReadFile("data.txt");;

val it : seq<CheeseClub>
 = seq
    [{Name = "Steve";
      FavouriteCheese = "Cheddar";
      LastAttendance = 12/03/2010 00:00:00;};
     {Name = "Sally";
      FavouriteCheese = "Brie";
      LastAttendance = 18/02/2010 00:00:00;}]
```

There is something somewhat magical about this; you've built a layer that automatically does the impedance matching between the untyped world of a text-file format into the typed world of F# programming. Amazingly, the SchemaReader type itself is only about 50 lines of code. The comments in Listing 17-5 show the basic steps being performed. The essential features of this technique are:

1. The schema information is passed to the SchemaReader as a type variable. The SchemaReader then uses the typeof operator to extract a System.Type representation of the schema type.
2. The information needed to drive the transformation process comes from custom attributes. Extra information can also be supplied to the constructor of the SchemaReader type if necessary.
3. The let bindings of the SchemaReader type are effectively a form of precomputation (they can also be seen as a form of compilation). They precompute as much information as possible given the schema. For example, the section analyzes the fields of the schema type and computes functions for creating objects of the field types. It also computes the permutation from the text file columns to the record fields.
4. The data objects are ultimately constructed using reflection functions, in this case a function computed by Microsoft.FSharp.Reflection.Value.GetRecordConstructor or primitive values parsed using System.Int32.Parse and similar functions. This and other functions for creating F# objects dynamically are in the Microsoft.FSharp.Reflection library. Other functions for creating other .NET objects dynamically are in the System.Reflection library.
5. The member bindings of SchemaReader interpret the residue of the precomputation stage, in this case using the information and computed functions to process the results of splitting the text of a line.

This technique has many potential applications and has been used for CSV file reading, building F#-centric serializers/deserializers, and building generic strongly typed database schema access.

## Using the F# Dynamic Reflection Operators

F# lets you define two special operators, `(?)` and `(?<-)`, to perform dynamic lookups of objects. These are conceptually very simple operators, but they add interesting new opportunities for interoperability between dynamic data and static data in F# programming.

These operators implicitly translate their second argument to a string, if it's a simple identifier. That is, a use of these operators is translated as:

```
expr ? nm            ®   (?) expr "nm"
expr1 ? nm <- expr2  ®   (?<-) expr1 "nm" expr2
```

This means that the operators can be used to simulate a dynamic lookup of a property or a method on an object. This dynamic lookup can use any dynamic/reflective technique available to you. One such technique is to use .NET reflection to look up and/or set the properties of an object:

```
open System.Reflection

let (?) (obj : obj) (nm : string) : 'T =
    obj.GetType().InvokeMember(nm, BindingFlags.GetProperty, null, obj, [||])
    |> unbox<'T>

let (?<-) (obj : obj) (nm : string) (v : obj) : unit =
    obj.GetType().InvokeMember(nm, BindingFlags.SetProperty, null, obj, [|v|])
    |> ignore
```

Now, you can use the operators to dynamically query data:

```
type Record1 = {Length : int; mutable Values : int list}

let obj1 = box [1; 2; 3]
let obj2 = box {Length = 4; Values = [3; 4; 5; 7]}

let n1 : int = obj1?Length
let n2 : int = obj2?Length
let valuesOld : int list = obj2?Values
```

Here, both `obj1` and `obj2` have type `obj`, but you can do dynamic lookups of the properties `Length` and `Values` using the `?` operator. Of course, these uses aren't strongly statically typed—this is why you need the type annotations `: int` and `: int list` to indicate the return type of the operation. Given the earlier definition of the `(?<-)` operator, you can also set a property dynamically::

```
obj2?Values <- [7; 8; 9]
let valuesNew : int list = obj2?Values
```

Using the `(?)` and `(?<-)` operators obviously comes with strong drawbacks: you lose considerable type safety, and performance may be affected by the use of dynamic techniques. Their use is recommended only when you're consistently interoperating with weakly typed objects, or when you continually find yourself doing string-based lookup of elements of an object.

# Using F# Quotations

The other side to reflective meta-programming in F# is *quotations*. These allow you to reflect over expressions in much the same way that you've reflected over types in the previous section. It's simple to get going with F# quotations; you open the appropriate modules and surround an expression with <@ . . . @> symbols:

```
> open Microsoft.FSharp.Quotations;;

> let oneExpr = <@ 1 @>;;

val oneExpr : Expr<int> = Value (1)

> let plusExpr = <@ 1 + 1 @>;;

val plusExpr : Expr<int> = Call (None, op_Addition, [Value (1), Value (1)])
```

You can see here that the act of quoting an expression gives you back the expression as data. Those familiar with Lisp or Scheme know a sophisticated version of this in the form of Lisp quotations, and those familiar with C# 3.0 will find it familiar, because C# uses similar mechanisms for its lambda expressions. F# quotations are distinctive partly because they're *typed* (like C# lambda expressions) and because the functional, expression-based nature of F# means that so much of the language can be quoted and manipulated relatively easily.

Chapter 13 uses F# queries that implicitly convert F# quotations to SQL via the .NET LINQ library. Perhaps the most important application is in Chapter 14, where quotations are converted to JavaScript when using WebSharper. This may be implemented by a function with a type such as

```
val CompileToJavaScript : Expr<'T> -> string
```

### WHAT ARE F# QUOTATIONS FOR?

The primary rationale for F# quotations is to allow fragments of F# syntax to be executed by alternative means: for example, as an SQL query via LINQ or by running on another device, such as a GPU or as JavaScript in a client-side Web browser. F# aims to leverage heavy-hitting external components that map subsets of functional programs to other execution machinery. Another example use involves executing a subset of F# array code by dynamic generation of Fortran code and invoking a high-performance vectorizing Fortran compiler. The generated DLL is loaded and invoked dynamically.

This effectively means that you can convert from a computational representation of a language (for example, regular F# functions and F# workflow expressions) to an abstract syntax representation of the same language. This is a powerful technique, because it lets you design using a computational model of the language (for example, sampling from a distribution or running queries against local data) and then switch to a more concrete abstract syntax representation of the same programs in order to analyze, execute, print, or compile those programs in other ways.

# Example: Using F# Quotations for Error Estimation

Listing 17-6 shows a prototypical use of quotations, in this case to perform error estimation on F# arithmetic expressions.

***Listing 17-6.*** *Error analysis on F# expressions implemented with F# quotations*

```
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

type Error = Err of float

let rec errorEstimateAux (e : Expr) (env : Map<Var, _>) =
    match e with
    | SpecificCall <@@ (+) @@> (tyargs, _, [xt; yt]) ->
        let x, Err(xerr) = errorEstimateAux xt env
        let y, Err(yerr) = errorEstimateAux yt env
        (x + y, Err(xerr + yerr))

    | SpecificCall <@@ (-) @@> (tyargs, _, [xt; yt]) ->
        let x, Err(xerr) = errorEstimateAux xt env
        let y, Err(yerr) = errorEstimateAux yt env
        (x - y, Err(xerr + yerr))

    | SpecificCall <@@ ( * ) @@> (tyargs, _, [xt; yt]) ->
        let x, Err(xerr) = errorEstimateAux xt env
        let y, Err(yerr) = errorEstimateAux yt env
        (x * y, Err(xerr * abs(y) + yerr * abs(x) + xerr * yerr))

    | SpecificCall <@@ abs @@> (tyargs, _, [xt]) ->
        let x, Err(xerr) = errorEstimateAux xt env
        (abs(x), Err(xerr))

    | Let(var, vet, bodyt) ->
        let varv, verr = errorEstimateAux vet env
        errorEstimateAux bodyt (env.Add(var, (varv, verr)))

    | Call(None, MethodWithReflectedDefinition(Lambda(v, body)), [arg]) ->
        errorEstimateAux (Expr.Let(v, arg, body)) env

    | Var(x) -> env.[x]

    | Double(n) -> (n, Err(0.0))

    | _ -> failwithf "unrecognized term: %A" e

let rec errorEstimateRaw (t : Expr) =
    match t with
    | Lambda(x, t) ->
        (fun xv -> errorEstimateAux t (Map.ofSeq [(x, xv)]))
```

```
    | PropertyGet(None, PropertyGetterWithReflectedDefinition(body), []) ->
        errorEstimateRaw body
    | _ -> failwithf "unrecognized term: %A - expected a lambda" t

let errorEstimate (t : Expr<float -> float>) = errorEstimateRaw t
```

The inferred types of the functions are:

```
type Error = | Err of float
val errorEstimateAux : e:Expr -> env:Map<Var,(float * Error)> -> float * Error
val errorEstimateRaw : t:Expr -> (float * Error -> float * Error)
val errorEstimate :
  t:Expr<(float -> float)> -> (float * Error -> float * Error)
```

That is, `errorEstimate` is a function that takes an expression for a `float -> float` function and returns a function value of type `float * Error -> float * Error`.

Let's see it in action. First, define the function `err` and a pretty-printer for `float * Error` pairs, here using the Unicode symbol for error bounds on a value:

```
> let err x = Err x;;

val err : x:float -> Error

> fsi.AddPrinter (fun (x : float, Err v) -> sprintf "%g±%g" x v);;
> errorEstimate <@ fun x -> x + 2.0 * x + 3.0 * x * x @> (1.0, err 0.1);;

val it : float * Error = 6±0.93

> errorEstimate <@ fun x -> let y = x + x in y * y + 2.0 @> (1.0, err 0.1);;

val it : float * Error = 6±0.84
```

The key aspects of the implementation of `errorEstimate` are:

- The `errorEstimate` function converts the input expression to a raw expression, which is an untyped abstract syntax representation of the expression designed for further processing. It then calls `errorEstimateRaw`. Traversals are generally much easier to perform using raw terms.
- The `errorEstimateRaw` function then checks that the expression given is a lambda expression, using the active pattern `Lambda` provided by the `Microsoft.FSharp.Quotations.Patterns` module.
- The `errorEstimateRaw` function then calls the auxiliary function `errorEstimateAux`. This function keeps track of a mapping from variables to value/error estimate pairs. It recursively analyzes the expression looking for `+`, `-`, `*` and `abs` operations. These are all overloaded operators and hence are called *generic functions* in F# terminology, so the function uses the active pattern `SpecificCall` to detect applications of these operators. At each point, it performs the appropriate error estimation.

- For variables, the environment map `env` is consulted. For constants, the error is zero.
- Two additional cases are covered in `errorEstimateAux` and `errorEstimateRaw`. The `Let` pattern allows you to include expressions of the form `let x = e1 in e2` in the subset accepted by the quotation analyzer. The `MethodWithReflectedDefinition` case allows you to perform analyses on some function calls, as you will see next.

## Resolving Reflected Definitions

One problem with meta-programming with explicit `<@ ... @>` quotation marks alone is that you can't analyze very large programs, because the entire expression to be analyzed must be delimited by these markers. This is solved in F# by allowing you to tag top-level `member` and `let` bindings as reflected. This ensures that their definitions are persisted to a table attached to their compiled DLL or EXE. These functions can also be executed as normal F# code. For example, here is a function whose definition is persisted:

```
[<ReflectedDefinition>]
let poly x = x + 2.0 * x + 3.0 * (x * x)
```

You can retrieve definitions such as this using the `MethodWithReflectedDefinition` and `PropertyGetterWithReflectedDefinition` active patterns, as shown in Listing 17-6. You can now use this function in a regular `<@ ... @>` quotation and thus analyze it for errors:

```
> errorEstimate <@ poly @> (3.0, err 0.1);;

val it : float * Error = 36±2.13
> errorEstimate <@ poly @> (30271.3, err 0.0001);;

val it : float * Error = 2.74915e+09±18.1631
```

# Summary

This chapter covered key topics in a programming paradigm, *language-oriented programming*, that is central to F#. In previous chapters, you saw some techniques for traversing abstract syntax trees. These language-representation techniques give you powerful ways to manipulate concrete and abstract syntax fragments.

In this chapter, you saw two language-representation techniques that are more tightly coupled to F#: F# computation expressions, which are useful for embedded computational languages involving sequencing, and F# quotations, which let you give an alternative meaning to existing F# program fragments. Along the way, the chapter touched on reflection and its use in mediating between typed and untyped representations.

In the next chapter, you'll look at some of the interoperability mechanisms that come with the .NET implementation of F#.