

Metaprogramming in .NET

Kevin Hazzard
Jason Bock

FOREWORD BY Rockford Lhotka





Metaprogramming in .NET

by Kevin Hazzard
Jason Bock

Chapter 6

Copyright 2013 Manning Publications

brief contents

PART 1 DEMYSTIFYING METAPROGRAMMING1

- 1 ■ Metaprogramming concepts 3
- 2 ■ Exploring code and metadata with reflection 41

PART 2 TECHNIQUES FOR GENERATING CODE63

- 3 ■ The Text Template Transformation Toolkit (T4) 65
- 4 ■ Generating code with the CodeDOM 101
- 5 ■ Generating code with Reflection.Emit 139
- 6 ■ Generating code with expressions 171
- 7 ■ Generating code with IL rewriting 199

PART 3 LANGUAGES AND TOOLS221

- 8 ■ The Dynamic Language Runtime 223
- 9 ■ Languages and tools 267
- 10 ■ Managing the .NET Compiler 287



Generating code with expressions

This chapter covers

- Using code as data
- Using the power of expression trees
- Improving code with expressions

As you saw in chapter 5, you can use the `Reflection.Emit` APIs to create dynamic code that you can execute at runtime. This requires intimate knowledge of IL. Let's be honest: learning IL isn't a skill set that most .NET developers have, nor is it one they necessarily want to acquire, even if they're interested in metaprogramming techniques. The reason is simple: writing code in IL can easily lead to incorrect implementations and requires a mental model of code execution in .NET that's not as intuitive as the one a high-level language provides.

Fortunately, there's another API in .NET that lets you create code without having to know anything about IL. This is the Expression API that exists within the LINQ world. In this chapter, you'll see how you can view your code as data in a way that will make metaprogramming much easier to do in .NET. When something is easier to accomplish, you end up using it more, which is why learning about expressions to generate dynamic code is advantageous. You'll end up seeing scenarios in your code where you can use expressions to handle certain problems elegantly. Let's start by looking at how expressions work at a higher level.

6.1 **Expression-oriented programming**

This section covers how expressions work and why they're so desirable in the world of metaprogramming. In essence, *expressions* are a representation of code via a data structure, and you'll see how you can use this representation (coding structures as data) effectively and the flexibility you gain with this view.

6.1.1 **Understanding code as data**

In chapter 1 we spent a fair amount of time defining metaprogramming. You may recall that section “Creating IL at runtime using expression trees” (a part of section 1.2.3) gave a high-level overview of expressions. After that overview, you saw metaprogramming techniques in action that were at the lower level of IL. That gave you a solid (and we think necessary) understanding of the inner workings of .NET, but more often than not, you don't need to write code in IL. You're finally going to come back to expressions, the focus of this chapter. Although that section gave a good overview of expressions, it's time to narrow the discussion to a discrete example that you'll use as a starting point into how expressions work in .NET.

Consider the following function:

```
public int Add(int x, int y)
{
    return x + y;
}
```

Writing this as an expression in .NET isn't too hard—you'll see an example of this momentarily. But consider this nomenclature:

```
(+ x y)
```

If you've spent any time even glossing over the Lisp language, you know where this line of code comes from. Even if you've never seen Lisp before, you probably could guess that this code is taking two variables and adding them together. There's a big reason why you're seeing a little bit of Lisp at this point, because at its core Lisp is all about expressions. For example:

```
(1 2 add)
```

This is a list in Lisp that contains three atoms: 1, 2, and *add*. Note that there's no difference in the formatting between a list and how the add operator works. They're both expressions. Code and data are both expressed the same way. This arrangement allows you to handle a function as if it were a data structure, which allows tremendous flexibility for a developer to modify and alter code in a fairly natural way. If you write your code in IL, there's no natural way to modify that code in the same way you'd modify a list of items. If you can treat your code like data, it becomes more natural to modify code as you would modify data. That's why a concept like expressions has been around for so long. This flexibility can also give rise to some mind-bending implementations, but we'll keep our dive into Lisp at this more moderate level for now.

NOTE Lisp is one of the oldest programming languages and is considered to be the source and inspiration for many ideas you probably take for granted in many languages, such as trees and self-hosted compilers, among other things. We don't expect you to become proficient in Lisp in any fashion—we're not even remotely close to being Lisp experts ourselves—but spending some time learning a little about Lisp is always a good thing. You can find out more about Lisp from <http://landoflisp.com>. Also, *The Joy of Clojure* by Amit Rathore (Manning, 2011) covers a fairly recent language called Clojure that's heavily influenced by Lisp. Check out <http://manning.com/rathore/>.

How would you take the add C# function you saw in the beginning of this section and turn it into an expression? Like this:

```
Expression<Func<int, int, int>> add = (x, y) => x + y;
```

That's not as concise as the Lisp syntax, but that's how it works in C#. The add variable after this point is a lambda expression. It's not a function you can execute. In fact, if you tried to write this

```
Expression<Func<int, int, int>> add = (x, y) => x + y;
var result = add(2, 3);
```

you'd get a compiler error stating that you're trying to use add, which is a variable, like a method, which it isn't. It's an expression tree at this point. To execute this code, you need to do this:

```
Expression<Func<int, int, int>> add = (x, y) => x + y;
var result = add.Compile()(2, 3);
```

The Compile() method takes the expression tree and turns it into something that the runtime can execute—namely, IL. Figure 6.1 shows what the tree looks like from a logical perspective.

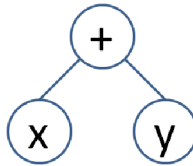


Figure 6.1 A logical representation of an expression that adds two parameters. As you can see, IL is nowhere to be found because you're focusing on the structure of the implementation, not the opcodes to do it.

Notice that you don't ever see

any IL when you use an expression, and frankly, that's a good thing. Knowing intimate details of IL isn't necessarily bad because it can yield great insight into the inner workings of .NET. However, coding in IL can lead to some unintended results if you're not extremely careful. Using expressions is much more natural.

Unfortunately, you can't create an expression at runtime using the "fat arrow" syntax: =>. You'll see more details on the Expression API in this chapter, but the following is a code snippet that does the same thing as the lambda expression syntax—the only difference is it uses the Expression API explicitly:

```
var x = Expression.Parameter(typeof(int));
var y = Expression.Parameter(typeof(int));

return (Expression.Lambda(
    Expression.Add(x, y), x, y)
    .Compile() as Func<int, int, int>)(2, 3);
```

This may seem a little verbose, but look at the next code snippet that does the same thing in IL:

```
var method = new DynamicMethod("m",
    typeof(int), new Type[] { typeof(int), typeof(int) });
var x = method.DefineParameter(
    1, ParameterAttributes.In, "x");
var y = method.DefineParameter(
    1, ParameterAttributes.In, "y");
var generator = method.GetILGenerator();
generator.Emit(OpCodes.Ldarg_0);
generator.Emit(OpCodes.Ldarg_1);
generator.Emit(OpCodes.Add);
generator.Emit(OpCodes.Ret);
return (method.CreateDelegate(
    typeof(Func<int, int, int>))
    as Func<int, int, int>)(2, 3);
```

In our opinion, the first approach is easier to read and more succinct. Imagine if you had to write all your dynamic code this way. Yes, you could do it, but using less code that does the same thing seems like a better approach.

Later in this chapter you'll use this API extensively to create code that will create different code based on that code at runtime. The next section gives an example of a popular .NET assembly that uses expressions to simplify method resolution.

Why can't I use var for my expressions?

You may have noticed that the code snippets in this section use an explicit type declaration for the lambda expression. Here's why. If you typed this

```
var expression = (x, y) => x + y;
```

did you mean this?

```
Func<int, int, int> expression = (x, y) => x + y;
```

or this?

```
Expression<Func<int, int, int>> expression =
    (x, y) => x + y;
```

Without explicit typing, the compiler can't tell which one you mean, and there's a difference. A `Func` or an `Action` type turns into an anonymous method in your code once the compiler is done, but an expression resolves into API calls that you'll see in section 6.2.

To find out more gory details about when the `var` keyword can't be used in C# check out the two articles at <http://mng.bz/IOD0> and <http://mng.bz/56bw>.

6.1.2 Expressions take metaprogramming mainstream

You may be thinking, "Great, expressions are awesome and expressive and wonderful, but where in the world would I use this in my code?" At first glance, it may seem like

you'd never use expressions in your code base (though if you're reading this book, you might). But a lot of .NET-based frameworks out there use expressions to some degree or another. One in particular is Component-based Scalable Logical Architecture (CSLA), which is a business object framework. Our intent isn't to go over every aspect of CSLA in this section; rather, you're going to see how CSLA uses expressions under the hood.

If you've ever used CSLA, you're familiar with the object creation convention through the `DataPortal` class. You don't make an instance of an object directly; you let the portal create it for you. The following code listing demonstrates how you could fetch data for a `Person` object via an identifier typed as a `Guid`.

Listing 6.1 Fetching data for an object in CSLA

```
[Serializable]
public sealed class Person
    : BusinessBase<Person>
{
    private Person()
        : base() { }

    public static Person Fetch(Guid id)
    {
        return DataPortal.Fetch<Person>(id);
    }

    private void DataPortal_Fetch(Guid id)
    {
        // ...
    }
}
```

You usually create a static `Fetch()` method that takes all the values it needs for a successful lookup. You pass those values into the `Fetch()` method from the `DataPortal`. The `DataPortal` creates an instance of the target specified by the generic parameter value and then looks for a `DataPortal_Fetch` method that has a parameter with the right type. In this case, there's a method that takes a `Guid`, so everything will work out.

NOTE If your static `Fetch()` method takes multiple parameters, you have to pack those up into one criteria object. Using a `Tuple` makes that relatively painless.

CSLA is using a bit of reflection to resolve the call. It's going to do something like this:

```
var method = typeof(T).GetMethod(
    "DataPortal_Fetch",
    BindingFlags.Public | BindingFlags.NonPublic |
        BindingFlags.DeclaredOnly | BindingFlags.Instance,
    null,
    new Type[] { criteria.GetType() }, null)
```

The exact implementation in CSLA is more involved than this, but it boils down to a method lookup via reflection. Once CSLA knows the method exists, it then creates

a method dynamically at runtime to figure out the call flow and caches this in memory. In chapter 5 (section 5.5.3), you saw the benefit of keeping dynamic methods around once they're created, which is why CSLA does this. But CSLA doesn't use `DynamicMethod`; it uses the Expression API to create this new method. When CSLA was first created for .NET, it initially used reflection exclusively, and once `DynamicMethod` was introduced, CSLA switched over to that for performance reasons. But the Expressions API produces code with the same performance characteristics without requiring the developer to understand IL, which is why CSLA uses the Expression API for method invocation scenarios like `DataPortal.Fetch`.

TIP CSLA is available in Nuget. Download the source code from www.lhotka.net/cslanet/Download.aspx. If you're interested in seeing how CSLA uses the Expression API, take a look at the `DynamicMethodHandlerFactory` class.

Now that you've had a high-level overview of expressions and where they exist in the .NET world, let's take a look at the mechanics of expressions. You'll start by seeing why expressions exist in a sub-namespace of LINQ.

6.2 *Making dynamic methods with LINQ Expressions*

As you've already seen in this chapter, you can use expressions to create methods on the fly. In this section, you'll dive deeper into this API to see what's possible with expression creation in .NET, including the following:

- Creating and calling methods
- Using mathematical operations
- Adding exception handlers
- Controlling the flow of code

Before we do that, let's start by looking at why LINQ is even in the picture with expressions.

6.2.1 *Understanding LINQ Expressions*

At first glance, for the Expression API to reside in the `System.Linq.Expressions` namespace may seem odd. What does LINQ have to do with expressions anyway? The best way to see how expressions are used in .NET is to write some LINQ, decompile it, and look at the results.

The following listing shows a simple LINQ query that finds objects with property values that contain the character a.

Listing 6.2 Using LINQ to filter a list of objects

```
public sealed class Container
{
    public string Value { get; set; }
}

// ...
```

```
var containers = new Container[]
{
    new Container { Value = "bag" },
    new Container { Value = "bed" },
    new Container { Value = "car" }
};

var filteredResults =
    from container in containers
    where container.Value.Contains("a")
    select container;
```

There's nothing fancy going on here—you're using a where clause to filter the list. Let's say your filtering was more complex and dynamic based on a user's choices in the application. The user may want, for example, to find the Container objects where the Value property contains a and is between 3 and 10 characters long. You could write this LINQ statement with no issues, but that would be hardcoded at compile-time. There's no way for you to change that query with the familiar LINQ techniques most .NET developers are aware of. But if you're knowledgeable with expressions, you can change the filter on the fly. Let's change the filter from listing 6.2 to use an expression, shown in the following listing.

Listing 6.3 Using a LINQ expression to create the filter

```
var argument = Expression.Parameter(typeof(Container));
var valueProperty = Expression.Property(argument, "Value");
var containsCall = Expression.Call(valueProperty,
    typeof(string).GetMethod(
        "Contains", new Type[] { typeof(string) }),
    Expression.Constant("a", typeof(string)));
var wherePredicate = Expression.Lambda<Func<Container, bool>>(
    containsCall, argument);
var whereCall = Expression.Call(typeof(Queryable), "Where",
    new Type[] { typeof(Container) },
    containers.AsQueryable().Expression, wherePredicate);

var expressionResults = containers.AsQueryable()
    .Provider.CreateQuery<Container>(whereCall);
```

You need an IQueryable object reference to start out, which is what the AsQueryable() extension method is for. Then, you use the Provider property to call the CreateQuery() method. That method takes an expression that can do pretty much whatever you want it to do to the queryable object. The next section explains the details about the creation of this expression in detail, but for now it's sufficient to take away the fact that you can create dynamic LINQ queries at runtime via the Expression API.

At this point, it's time (finally!) to go over the API within System.Linq.Expression. You'll start by going over the expression created in listing 6.3, one part at a time.

Using DynamicQueryable

Buried within the sample that comes with an installation of Visual Studio lies a hidden namespace gem called `System.Linq.Dynamic`, which contains a class called `DynamicQueryable` (among many other interesting classes). This class allows you to write your dynamic query via a small piece of code rather than explicitly using the Expression API. The code in listing 6.3 is reduced to one line of code with `DynamicQueryable`:

```
var dynamicResults = containers.AsQueryable()
    .Where("Value.Contains(\"a\")");
```

For more information on how to use this cool API, see <http://mng.bz/KN7v>.

6.2.2 Generating expressions at runtime

This section examines the classes and methods you use to create dynamic expressions. The API surface is quite extensive, so we focus on the common activities you'll do when creating an expression. You'll also see how you can use other features in the Expression API to create rich implementations in your expressions.

CREATING A SIMPLE LAMBDA EXPRESSION

Listing 6.3 created an expression to perform a dynamic query on a simple data set. The query was the same as writing this line of code:

```
where container.Value.Contains("a")
```

This is the same as writing the following line of code (which is what the C# compiler will do with the previous LINQ statement, more or less):

```
containers.Where(value => value.Value.Contains("a"))
```

Let's go through each line of code in listing 6.3 to see how the expression translates into the exact same line of code that invokes the `Where()` method on the list.

The first thing you need is a parameter to the lambda expression—that's what the `value` parameter is. You do that by creating a `ParameterExpression`:

```
var argument = Expression.Parameter(typeof(Container));
```

You can give an explicit name to the parameter via an override of `Parameter()`, but in this case you only need to specify the type, which is a `Container` type. Note that creating the `ParameterExpression` object requires a static call on the `Expression` class. That's how you'll create all the expression pieces you need. You go through a static factory method on `Expression`.

Now that you have a parameter of type `Container`, you need to use the `Value` property on that parameter. To get it, use the `Property()` method, which returns a `MemberExpression`:

```
var valueProperty = Expression.Property(argument, "Value");
```

As with the Reflection API, there are numerous overloads with many of these factory methods. You can get a `MemberExpression` object for a property by passing a

PropertyInfo object into the `Property()` method, rather than using a string, for example.

The next step is calling `Contains()` on that property. That's what the following line of code does:

```
var containsCall = Expression.Call(valueProperty,
    typeof(string).GetMethod(
        "Contains", new Type[] { typeof(string) }),
    Expression.Constant("a", typeof(string)));
```

The `Call()` method returns a `MethodCallExpression`. You can invoke a method in numerous ways, so you can imagine there are a lot of overloads to `Call()`. In your case, you need to call the method on the `Value` property, which is why that `MemberExpression` is passed in first. Then you specify the method you want to call on the property. Here, the code uses a bit of reflection via `GetMethod()` to look up the `Contains()` method on a `string` with the right signature. The last things `Call()` needs are any argument values. You only need to pass in the literal `"a"` string value, which is what a `ConstantExpression` provides.

You're close to done at this point. You now need a lambda expression that you'll pass into a `Where()` invocation:

```
var wherePredicate = Expression.Lambda<Func<Container, bool>>(
    containsCall, argument);
```

The `Lambda()` call takes an expression to represent the body of the lambda and any arguments the lambda needs.

Finally, you need to invoke the `Where()` method on the queryable object itself:

```
var whereCall = Expression.Call(typeof(Queryable), "Where",
    new Type[] { typeof(Container) },
    containers.AsQueryable().Expression, wherePredicate);
```

That's it. You now have an expression that will invoke the correct method when `CreateQuery()` is invoked.

Let's move on to look at APIs that can help you with adding mathematical capabilities.

INCLUDING MATHEMATICAL OPERATIONS

Let's revisit the code snippet in section 6.1.1. In that expression, you saw how you could create a method that would add two numbers. You did that via the `Add()` method, which returns a `BinaryExpression`. Numerous `Expression` APIs return a `BinaryExpression`. For example, you can make the code in section 6.1.1 perform a subtraction with one change:

```
Expression.Subtract(x, y)
```

If you need the remainder of `x` divided by `y`:

```
Expression.Modulo(x, y)
```

You can also raise the power of `x` by `y` via the `Power()` call:

```
Expression.Power(x, y)
```

What's nice about this function is that you don't have to make a `MethodCallExpression` to `Math.Pow()`. Use this static method. But you do need to make sure that the types of `x` and `y` are defined to be a double type. `Expression.Power()` ends up making a call to `Math.Pow()` for you, so you can't use the `int` type.

You also have the ability to use checked mathematical operators that will raise an `OverflowException`. For example, `AddChecked()` will handle this scenario. Speaking of exceptions, you may wonder whether you can add exception handlers to your expressions. The answer is yes, you can, and that's what the next section is about.

USING EXCEPTION HANDLERS

Let's say you created a dynamic method that uses a checked addition via `AddChecked()`. If someone passes in two values that would cause an overflow, you'd get an exception. Although it may seem odd to catch the `OverflowException` that you want to have raised with `AddChecked()`, let's see how you can do this with expressions in the following listing.

Listing 6.4 Adding a try-catch block to an expression

```
var x = Expression.Parameter(typeof(int));
var y = Expression.Parameter(typeof(int));

var lambda = Expression.Lambda(
    Expression.TryCatch(
        Expression.Block(
            Expression.AddChecked(x, y)),
        Expression.Catch(
            typeof(OverflowException),
            Expression.Constant(0))), x, y);
```

Before 4.0, the Expressions API was limited in what it could do. One of its limitations was in the area of exception handling. There was no way to add a `try...catch` block to your expression body. But in the 4.0 version, you now have that support.

The first thing you need to do is add the exception handler. In listing 6.4, `TryCatch()` is called, which returns a `TryExpression`. Next, take the code that you want in the try block and wrap it in a `BlockExpression`. That's what the `Block()` call does. Finally, with a `try...catch` block, you need to define the code block that will run if an exception occurs, which is what the `Catch()` call performs. Note that in listing 6.4, the catch block is defined to catch exceptions of type `OverflowException`.

With this `try...catch` block in place, the following code would return 5:

```
return (lambda.Compile() as Func<int, int, int>)
    (2, 3);
```

But this code will return 0:

```
return (lambda.Compile() as Func<int, int, int>)
    (int.MaxValue, int.MaxValue);
```

What's nice about the exception handling support in the Expression API is that it's not limited to what you've seen here. You can create `try-finally` handlers, multiple

catch handlers—in fact, there’s a `TryFault()` call that will create a fault handler. Recall from chapter 5’s section 5.1.3 that there’s a fault handler that isn’t supported in either C# or VB. With the Expression API, you can add that support if you want, with relative ease.

ADDING CONTROL FLOW

The Expression API also has numerous ways to support branching and control flow in your code. Let’s look at a simple example of a function that takes a `bool` and returns a 1 if the argument is `true`, and 0 if the argument is `false`.

As before, the first thing you need is an argument:

```
var @switch = Expression.Parameter(typeof(bool));
```

Next, call the `Condition()` method:

```
var conditional = Expression.Condition(@switch,
    Expression.Constant(1),
    Expression.Constant(0));
```

It’s pretty simple. The first expression has to return a `bool` value, which is your argument. If it evaluates to `true`, the expression specified by the second argument is executed. Otherwise, the last expression is run. If you saw this code in C#, it would look something like this:

```
public void AFunction(bool @switch)
{
    if(@switch)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Finally, you compile the expression stuffed into a lambda expression:

```
var function = (Expression.Lambda(conditional, @switch)
    .Compile() as Func<bool, int>);
var result = function(true);
```

The value of the result would be 1 in this case. You can also use `Break()` to move to a specific label in the expression body. There’s even a `Goto()` method if you want “goto” semantics in your expression.

That covers the basics of expressions in .NET. There’s so much more in the Expression API that we haven’t covered—we’ve barely scratched the surface of what you can do with expressions. In fact, you’re not limited by the Expression API compared to what you can do in IL with `DynamicMethod`. Is there any disadvantage to using expressions compared to using a `DynamicMethod`? Why would you use expressions over a `DynamicMethod`? In the next section you’ll see how the two approaches compare.

6.2.3 Comparison with dynamic methods

Chapter 5 introduced you to the `DynamicMethod` class, which was an IL-based way to generate a method at runtime. As you've seen in this chapter, the Expression API provides the same functionality, so the inevitable question is which one should you choose? We'll take two views on this question: abstraction and performance.

Abstraction is all about using an API that's easy to understand, such that a developer can start using it with a minimal learning curve. Both approaches have an API that's consistent, but in our opinion avoiding IL is the preferred approach. Being able to use `Expression.Call()` to invoke a method is easier than trying to figure out the right opcodes to move local variables or arguments on the stack (along with the object for instance methods). Again, at the end of the day, this is a subjective metric, but we've used both approaches, and our preference is the Expression API.

Another thing to take into consideration is whether or not there is overhead in using one over the other. The code samples for this book contain code that compares the performance of the `DynamicMethod` way of creating a generic `ToString()` implementation to using the Expression API. We won't show that code here in the book, as you've already seen how the `DynamicMethod` version works, and the Expression API version is as lengthy. What's far more interesting is to see the comparison between the execution times, which is illustrated in figure 6.2.

As you can see, there's not much difference between the two. There's a slight benefit in execution time with `DynamicMethod`, but the difference is trivial. On average, executing the Expression-based method took about 1.48623 microseconds. The `DynamicMethod` approach was 1.48601 microseconds. Both implementations effectively execute at the same speed. The key difference is that with the Expression API, you don't have to learn IL to use it.

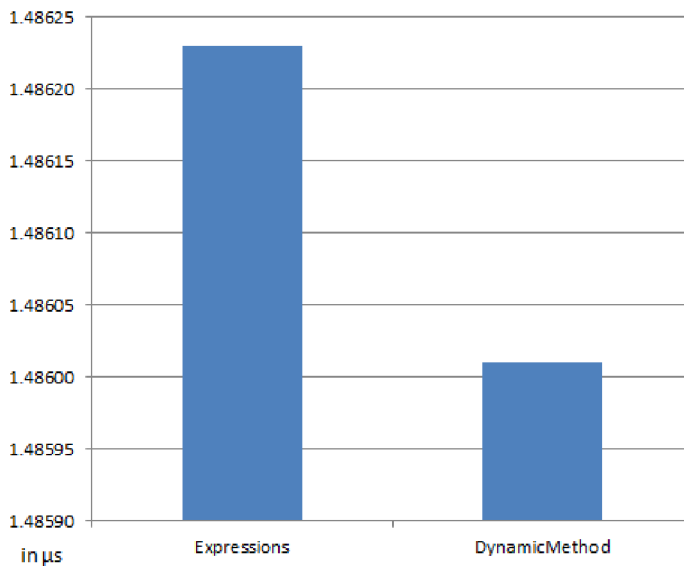


Figure 6.2 Comparing the execution time between an expression and `DynamicMethod`. The difference between the two is virtually the same.

However, the graph in figure 6.2 is a little misleading. The data was gathered using a cached version of both approaches; it doesn't take into consideration the time it takes to create the method.

Let's see how much time it takes to create a method that performs the following calculation:

$$f(x) = ((3 * x) / 2) + 4$$

Here's how it's done with expressions:

```
var parameter = Expression.Parameter(typeof(double));
var method = Expression.Lambda(
    Expression.Add(
        Expression.Divide(
            Expression.Multiply(
                Expression.Constant(3d), parameter),
            Expression.Constant(2d)),
        Expression.Constant(4d)),
    parameter).Compile() as Func<double, double>;
```

Here's how you can do it with `DynamicMethod`:

```
var method = new DynamicMethod("m",
    typeof(double), new Type[] { typeof(double) });
var parameter = method.DefineParameter(
    1, ParameterAttributes.In, "x");
var generator = method.GetILGenerator();
generator.Emit(OpCodes.Ldc_R8, 3d);
generator.Emit(OpCodes.Ldarg_0);
generator.Emit(OpCodes.Mul);
generator.Emit(OpCodes.Ldc_R8, 2d);
generator.Emit(OpCodes.Div);
generator.Emit(OpCodes.Ldc_R8, 4d);
generator.Emit(OpCodes.Add);
generator.Emit(OpCodes.Ret);
var compiledMethod = method.CreateDelegate(
    typeof(Func<double, double>)) as Func<double, double>;
```

If you create 10,000 of each of these and figure out the averages, you get a graph like figure 6.3.

It took 6.8 times longer to use expressions than to use `DynamicMethod`. Keep in mind, though, that creating the method using an expression took, on average, under 1 ms. That may be a performance bottleneck for your application, depending on how fast you need it to execute, but once you create the method, you'll probably cache it so you won't incur any time recreating it. Furthermore, once the method is created, there's no difference in execution time between an expression and a `DynamicMethod`. Although expressions are slower, that creation time may be acceptable for your application. As with any performance results, make sure you do your own evaluation and use your data to come to a conclusion that's right for your code base.

You now have a solid, basic understanding of expressions in .NET. But there's more to expressions than creating methods. You can debug them, use them in `Reflection.Emit`,

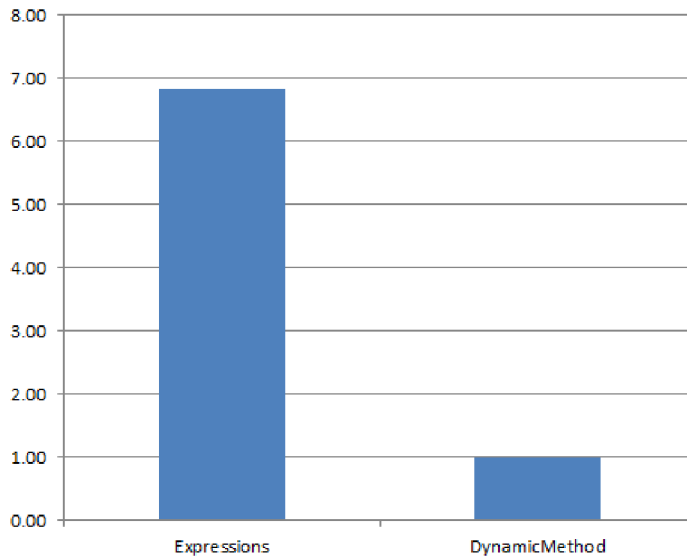


Figure 6.3 Comparing the time it takes to create an expression versus a `DynamicMethod`. The `DynamicMethod` approach is clearly quicker, but requires intimate knowledge of IL.

and mutate them (sort of!). In the next section, you'll see how you can accomplish all three tasks.

6.3 *Using expressions effectively*

Now that you've seen how expressions are created, let's get into areas that will make your expression life easier with debugging, emitting, and mutating. Learning these techniques is important so that you know your expressions do what you think they should do. Let's start by seeing how to add debug information to your expression.

6.3.1 *Debugging expressions*

Using the Expression API is a simpler, cleaner experience than trying to work with IL. That said, any time a developer writes a piece of code, something can go wrong. Fortunately, there are a couple of techniques you can use to debug your expressions. Let's start with the first one: visualizing your expression in the debugger.

VISUALIZING AN EXPRESSION IN VISUAL STUDIO

Whenever you create an expression of any type, you can get a textual visualization of that node in Visual Studio when you run your code under the debugger. You move your mouse pointer over the variable in code, drill down to the Debug View option, and select Text Visualizer. Figure 6.4 shows what the expression from section 6.1 looks like in this visualizer.

You may wonder why the language used in the visualizer doesn't use C# or VB. Expressions can support options that a language may not be able to express (like a fault block), so the designers of the visualizer came up with a different language to show the expression. It's not that hard to follow, and it's a quick and easy way to get a good idea for what your expression looks like at a particular point in its construction.

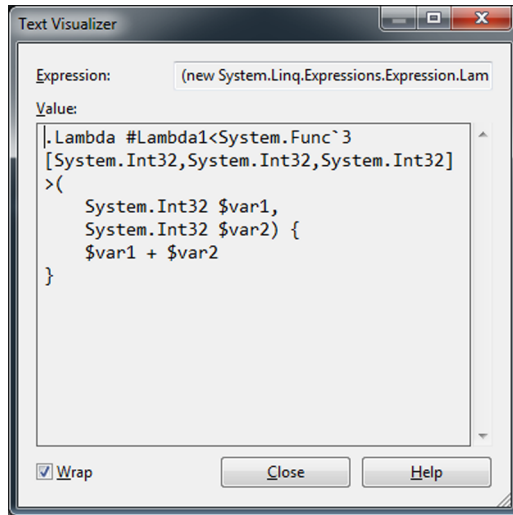


Figure 6.4 Visualizing an expression in Visual Studio. The language may not look like anything you’ve seen before, but the intent is clear.

NOTE For more information on the visualizer, see <http://mng.bz/E6Q7>.

In the next section, you’ll see how you can step into an expression at runtime.

USING REFLECTION.EMIT TO DEBUG EXPRESSIONS

Although a visual representation of an expression is a nice tool to have, sometimes you want to have the debugger dive right into the code. Unfortunately, an expression is a tree that represents your code structure. Or is it? When you compile your method, it’s emitting IL for you, like the code you emitted in chapter 5 that used Reflection.Emit. Surprisingly, there’s a connection between expressions and Reflection.Emit that lets you create debug information for an expression. Let’s see how you can get it to work.

Similar to the exception handler example, you’re going to start with the simple “add two numbers together” code snippet from section 6.1.1. The first thing you need to do is create a bunch of dynamic members from the Reflection.Emit API. Don’t worry, you won’t need to write any IL for debugging purposes; these members act as a host to your expression, as you’ll see in a moment:

```
var name = Guid.NewGuid().ToString("N");
var assembly = AppDomain.CurrentDomain.DefineDynamicAssembly(
    new AssemblyName(name), AssemblyBuilderAccess.Run);
var module = assembly.DefineDynamicModule(name, true);
var type = module.DefineType(
    Guid.NewGuid().ToString("N"), TypeAttributes.Public);
var methodName = Guid.NewGuid().ToString("N");
var method = type.DefineMethod(methodName,
    MethodAttributes.Public | MethodAttributes.Static,
    typeof(int), new Type[] { typeof(int), typeof(int) });
```

All this code is doing is getting an in-memory dynamic assembly set up along with dynamic type and method. As you can see, the names of these members don’t matter

in this case—what does matter is the debugging support. For that, you need a `Debug-InfoGenerator`:

```
var generator = DebugInfoGenerator.CreatePdbGenerator();
var document = Expression.SymbolDocument("AddDebug.txt");
```

The generator created from `CreatePdbGenerator()` will be used when the expression is compiled. You also need to create a symbol document based on a text file. The `Add-Debug.txt` file in this example is the expression expressed in the language used in the expression visualizer demonstrated in the previous section. We won't show the code here, but you'll see what some of it looks like in a figure near the end of this section.

To create debug symbols for sections of the code, you need to wrap a specific node in the expression tree with a `DebugInfoExpression`:

```
var addDebugInfo = Expression.DebugInfo(document,
    6, 9, 6, 22);
var add = Expression.Add(x, y);
var addBlock = Expression.Block(addDebugInfo, add);
```

The section in the document that maps to the expression node being wrapped is defined in the `DebugInfo()` call. This `DebugInfoExpression` object is used in a `Block()` call to wrap the `BinaryExpression` that represents the addition functionality of this expression.

Once you're done defining your expression, save the expression's implementation into the dynamic assembly:

```
var lambda = Expression.Lambda(addBlock, x, y);
lambda.CompileToMethod(method, generator);

var bakedType = type.CreateType();
return (int)bakedType.GetMethod(methodName)
    .Invoke(null, new object[] { a, b });
```

In this case, you use `CompileToMethod()` to specify which method you're implementing in the dynamic assembly along with the debug information related to this method. At this point, when you step into the `Invoke()` call in a debugger, you'll get into the file specified in the `SymbolDocumentInfo` object. Figure 6.5 shows what this looks like when you step into the “expression” language copied to the text file.

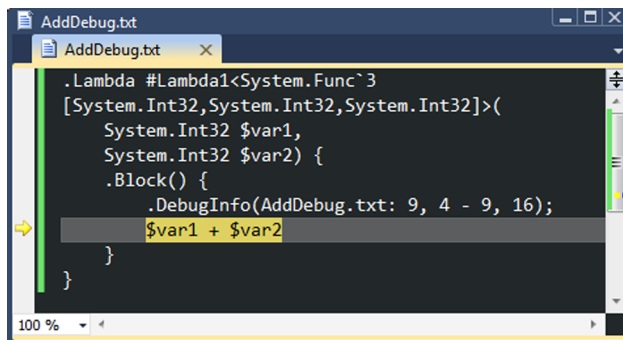


Figure 6.5 Debugging an expression. Even though the language may look a little odd, it's clear that you've stopped at the point where addition occurs in the method.

CAVEAT Using expressions to implement your methods in Reflection.Emit is a nice alternative to IL. But there's one major limitation to this approach: you can only use expressions to define static methods. You can't use an expression for an instance method. For more information on why this is the case, see <http://mng.bz/U254>.

You now know how you can debug your expression.

Let's move on to another topic: immutability. It will definitely affect your design, and that's what the next section is all about.

6.3.2 Mutating expression trees

Section 6.2 contained an overview of the Expression API to create dynamic methods at runtime. In this section, you'll look at expressions, immutability, and how you can create new expressions based on existing ones via the ExpressionVisitor class. Let's start by looking at the reasoning behind having immutable trees.

IMMUTABILITY OF EXPRESSION TREES

To frame the conversation on immutable expressions, let's go back to the expression in section 6.1 that added two integers together:

```
Expression<Func<int, int, int>> add = (x, y) => x + y;
```

Let's say someone wanted to change that expression to make it subtract the two arguments rather than add them. Because Expression<T> is a reference type, the expression you were referencing will now perform a subtraction, not an addition. That's not at all what you want! Immutable data structures are easier to reason about because you know that once the structure is created, it won't change. This also has benefits from a concurrency perspective because these structures are automatically thread-safe.

NOTE For more information on the benefits (and some shortcomings) of programming using immutable values see <http://en.wikipedia.org/wiki/Immutability> and <http://mng.bz/AId8>.

Being able to use a structure as the basis for future changes isn't a bad thing. In the next section, you'll see how you can create a new expression based on an existing one.

CREATING VARIATIONS OF EXPRESSIONS

You now know that expressions are immutable. Let's see how to create a new expression from the contents of an existing expression.

The key class you need is ExpressionVisitor. As the name implies, this class is based on the visitor pattern, which is designed to allow you to traverse complex object structures in a simplistic way by "visiting" specific methods that you care about.

NOTE For more on the visitor pattern, see http://en.wikipedia.org/wiki/Visitor_pattern.

You feed a subclass of ExpressionVisitor the expression that's your baseline and then you overwrite the VisitXYZ() methods you're interested in to create a new

expression. Let's create a custom visitor that will change an add operation to a subtract operation. The following code listing demonstrates what it takes to write such a visitor.

Listing 6.5 Creating an expression visitor

```
internal sealed class AddToSubtractExpressionVisitor
    : ExpressionVisitor
{
    internal Expression Change(Expression expression)
    {
        return this.Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression node)
    {
        return node.NodeType == ExpressionType.Add ?
            Expression.Subtract(
                this.Visit(node.Left), this.Visit(node.Right)) :
                node;
    }
}
```

As you can see, it doesn't take that much code to change an expression. In this case, you override `VisitBinary()` because you're trying to find the mathematical expression node `Add`. If you find one, then you create a new node that subtracts the child nodes. Note that you need to keep visiting the `Left` and `Right` nodes from the given node because they may also contain addition operations. For example, if you didn't do that, an expression like this

$$(x, y) \Rightarrow (((32 * x) / 4) + y) + (x + 4)$$

would turn into this:

$$(x, y) \Rightarrow (((32 * x) / 4) + y) - (x + 4)$$

That's not what you want, because there are still two addition operations in the expression. Visiting the child nodes gives you the right result:

$$(x, y) \Rightarrow (((32 * x) / 4) - y) - (x - 4)$$

NOTE Before .NET 4.0, there wasn't a way in the .NET Framework to visit an expression. The `ExpressionVisitor` class existed in `System.Linq.Expressions`, but it was marked as `internal`, so you couldn't use it. There are a couple of ways to support this technique in .NET 3.5—see <http://mng.bz/09bP> and <http://mng.bz/a3N3> for details on these approaches.

You now know how to debug and change expressions in .NET. In the last main section of this chapter, you'll tie everything together to create programs that better themselves through evolutionary techniques.

6.4 Evolving expression trees

Throughout this chapter, you've seen how to create expressions as well as debug them. Most of the time, the reasons to use expressions are similar to reasons you'd create a `DynamicMethod`. But because of its expressiveness and mutability via the `Expression-Visitor` class, you can use expressions in some amazing programming scenarios. In this section, you'll see how powerful metaprogramming and expressions can be when it comes to creating new programs on the fly. The topic of conversation is genetic programming. Next is an overview of genetic programming.

6.4.1 The essence of genetic programming

Before you start writing code that makes code better by rewriting it, let's start with a definition of what genetic programming is. *Genetic programming* is a technique that uses ideas and operations from the theory of evolution to create new programs based on existing ones. A genetic algorithm (GA) is a more general version of genetic programming. You create a pool of programs, or a population, that changes via an iterative approach. At each step in the process, you evaluate and/or change members of the population based on various rules and conditions. If a program in the population (called a *chromosome*) meets some kind of acceptable criteria, the process stops and that chromosome is selected as the answer.

TIP There's a vast amount of information on genetic programming. We recommend starting with John Koza's web site: www.genetic-programming.com. Koza is considered a pioneer in the world of genetic programming. The problem in section 6.4 is based on one of Koza's first forays into genetic programming. He used Lisp; we use expressions in .NET.

Let's go through an example you'll use in this chapter to create functions that match a given data set. Let's say you were given a data set that looked something like that in figure 6.6. In the real world, the data set would be larger than this, but you get the idea. You're given two columns: the input to a function and the result of that input to the function.

Looking at the data in figure 6.6, you may be able to figure out that the underlying function is this:

$$f(x) = x^3$$

This means *x raised to the power of three*. You'd be right, but the pattern behind the data values you're given may be much harder to find an underlying function. Let's look at how you could use genetic programming to evolve functions that try to produce outputs that match (or are close to) the data set values you're given.

x	f(x)
1	1
2	8
3	27
4	64
5	125

Figure 6.6 Inputs and outputs to a function. Keep in mind that you don't know what the function is.

POPULATION

The first step is to create a population of functions to use for your first iteration. How you create those functions is up to you. They could be a best-guess by someone or generated randomly by a computer. However you do it, let's say you come up with the following set of functions:

```
f(x) = x ^ 2
f(x) = x + 3
f(x) = (x ^ 3) + 1
f(x) = 3 * x
```

Note that your population size will probably be bigger than this, but for our purposes this will suffice. One of these functions is a decent fit for your target data set, and that's okay. You're starting with a bunch of guesses, and you'll improve on them as you move along. The next step is to pick "good" candidates.

SELECTION

Now that you have a population of chromosomes, you need to find those that are a good fit. To determine what a "good fit" is create a fitness function that gives a score for a given chromosome. How you define that fitness function is completely up to you, but hopefully it's a good indicator for how well the chromosome solves the given problem. In this case, we'll use the mean-squared error (MSE) to determine how fit a function is. The smaller the error, the better the function is at matching the given data set.

NOTE For more on MSE, see <http://mng.bz/x12Q> and http://en.wikipedia.org/wiki/Mean_squared_error.

Figure 6.7 shows the average MSE for each of the four functions in the population and an indicator that shows which functions were selected for the next operation.

You may wonder why the third one wasn't selected because it's clearly the best choice. In genetic programming, there are different methods to select the "best" chromosome (such as tournament and roulette), but all the methods have a degree of randomness built in. Therefore, by random selection, you may not get what appears to be the best choice. But this randomness keeps the gene pool (so to speak) dynamic. As you'll see in the next section, it's possible to make something good out of some not-so-good chromosomes.

CROSSOVER

Once you select chromosomes, you'll determine (via random chance) whether you'll perform crossover on the chromosomes. This means you'll take a piece (or pieces) of one chromosome and swap them with a piece (or pieces) from another. Figure 6.8 shows what happens when you swap the constant nodes between the selected functions.



f(x)	MSE Average	Selected
x^2	2528.8	
$x + 3$	3479.4	
$x^3 + 1$	1	
$3 * x$	3027.2	

Figure 6.7 Selecting functions in genetic programming. You want to select good candidates, but you won't always select the best ones.

Sometimes crossover may lead to better solutions, and sometimes it may make things worse. As you see in figure 6.8, the first function now matches perfectly to the data set you started with. But you're not done yet—there's one more operation to cover that can throw new material in the mix.

MUTATION

After crossover is done, you may perform mutation on a given chromosome. Usually the chance of mutation happening in a GA run is low, but it can occur to keep the population dynamic. A mutation may be a good or a bad thing for a chromosome—you never know. Figure 6.9 shows what the fourth function looks like after crossover occurred and the constant value changed into a whole new operation.

Is this “better” or “worse”? We'll leave that calculation up to the interested reader. The point is you now have new genetic material in your population that may (or may not) make things better.

ITERATION

Selection, crossover, and mutation—those are the three key operations that drive genetic programming. You keep iterating over the population with these three operations until one of two things occurs:

- You find a chromosome that exceeds some kind of threshold value (for example, the MSE is under 0.5 percent).
- You exceed the maximum number of iterations that you specified at the beginning of the run.

Hopefully you find a good enough solution before the second condition happens. Fortunately, you can let a GA run go on for hours on a computer and let it find all sorts of interesting solutions. In fact, people who have used GAs on a number of diverse problems have found solutions that they would have never thought of, and that can take them into new areas of research. The searching power of GAs is quite impressive, and the more you take a look into how GAs work...well, you may end up using them on a project of your own in the future to solve a problem in a unique way.

Now that you have an overview of how GAs work, let's see how you can do this in .NET using expression trees.

6.4.2 Applying GAs to expressions

It's one thing to see how GAs work using pictures. But getting them to work and seeing the results are the fun parts. In this section, you'll see how to evolve expressions to find a function to match a given data set. To do this, you'll get code snippets for a program we've written called ExpressionEvolver, which is included with the online

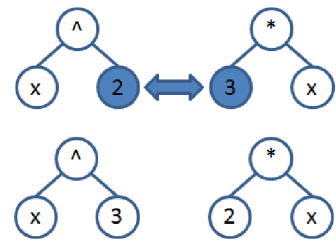


Figure 6.8 Performing crossover on two functions. Note how the first function has changed and how much “better” it is.

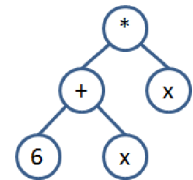


Figure 6.9 The results of mutating a function. The constant has been replaced with a whole new subtree.

code. The code base is quite vast, so we'll focus on the aspects that deal with the GA operators and .NET expressions. Let's start with how you can create random expressions in .NET.

CREATING RANDOM EXPRESSIONS

The first section of code you'll see is the class (`RandomExpressionGenerator`) that handles the creation of expressions. You need this when you create an initial population, or when a mutation occurs. For this example, you'll stick with the basic mathematical operations defined in the `Operators` enum:

```
private enum Operators
{
    Add,
    Subtract,
    Multiply,
    Divide,
    Power
}
```

You'll also need to be able to create constant values, which is handled in the `GetConstant()` method:

```
private ConstantExpression GetConstant()
{
    var value = this.Random.NextDouble() * this.ConstantLimit;
    var constant = value * (this.Random.NextBoolean() ? -1d : 1d);
    return Expression.Constant(constant);
}
```

The value for the `ConstantLimit` property is passed into the constructor—it's a best guess to limit the size of constant values created during a GA run.

Creating a random expression is a bit involved. It's done with the `GetRandomOperation()` method, shown in the following listing.

Listing 6.6 Creating a random operation

```
private void GetRandomOperation(Operators @operator)
{
    var isLeftConstant = this.Random.NextDouble() <
        this.InjectConstantProbabilityValue;
    var isRightConstant = this.Random.NextDouble() <
        this.InjectConstantProbabilityValue;
    var isLeftBody = true;
    var isRightBody = true;

    if(!isLeftConstant && !isRightConstant)
    {
        isLeftBody = this.Random.NextDouble() < 0.5;
        isRightBody = !isLeftBody;
    }
    else if(isLeftConstant && isRightConstant)
    {
        isLeftConstant = this.Random.NextDouble() <
```

```

this.InjectConstantProbabilityValue;
    isRightConstant = !isLeftConstant;
}
else if(@operator == Operators.Divide &&
    !isLeftConstant && !isRightConstant)
{
    isLeftConstant = this.Random.NextBoolean();
    isRightConstant = !isLeftConstant;
}

this.Body = RandomExpressionGenerator.GetExpressionFunction(
    @operator)(
    isLeftConstant ? this.GetConstant() :
    (isLeftBody ? this.Body : this.Parameter),
    isRightConstant ? this.GetConstant() :
    (isRightBody ? this.Body : this.Parameter));
}

```

Most of the function ends up trying to make sure a good mix of constants and parameters are used during the creation of the random expression. The `GetExpressionFunction()` gets a reference to the right function from the `Expression` class based on the value of the operation parameter:

```

private static Func<Expression, Expression, Expression>
    GetExpressionFunction(Operators @operator)
{
    Func<Expression, Expression, Expression> selectedOperation = null;
    switch(@operator)
    {
        case Operators.Add:
            selectedOperation = Expression.Add;
            break;
        case Operators.Subtract:
            selectedOperation = Expression.Subtract;
            break;
        case Operators.Multiply:
            selectedOperation = Expression.Multiply;
            break;
        case Operators.Divide:
            selectedOperation = Expression.Divide;
            break;
        case Operators.Power:
            selectedOperation = Expression.Power;
            break;
        default:
            throw new NotSupportedException(
                string.Format(CultureInfo.CurrentCulture,
                    "Unexpected operator type: {0}", @operator));
    }

    return selectedOperation;
}

```

To create a random expression, the `GetRandomOperation()` method is called in a for loop that builds up the `Body` property to a desired size for the expression tree:

```
private void GenerateBody(int maximumOperationCount)
{
    for(var i = 0; i < maximumOperationCount; i++)
    {
        this.GetRandomOperation(
            (Operators)this.Random.Next((int)Operators.Power + 1));
    }
}
```

Here's how you would use `RandomExpressionGenerator` to create a random expression:

```
var parameter = Expression.Parameter(typeof(double), "a");
var body = new RandomExpressionGenerator(10,
    0.5, 100d, parameter, new SecureRandom()).Body.Compress();
```

You've now seen how to generate random expressions in .NET. Let's move on to the crossover function in a GA and see how to use the `ExpressionVisitor` class to handle that.

HANDLING CROSSOVER WITH EXPRESSIONS

As you saw in the "Creating variations of expressions" subsection, you can use the `ExpressionVisitor` class to create new expressions based on the structure of an existing expression. You'll use this class to handle crossover. You pick a node in two expressions and find where that node exists in the other expression, swapping it for the node in the other expression. You do that with the `ReplacementVisitor` class, which has `ExpressionVisitor` as its base class. Here's how it does a transform:

```
public Expression Transform(Expression source,
    ReadOnlyCollection<ParameterExpression> sourceParameters,
    Expression find, Expression replacement)
{
    this.Find = find;

    if(sourceParameters != null)
    {
        this.Replacement = new ParameterReplacementVisitor()
            .Transform(sourceParameters, replacement);
    }
    else
    {
        this.Replacement = replacement;
    }

    return this.Visit(source);
}
```

The thing you need to keep in mind when you swap components between expressions is that you can't swap parameters. If you move a `ParameterExpression` node from one expression to another, you'll get an exception when you try to use that new expression. That's why the `ParameterReplacementVisitor` class is used. It's a nested class of `ReplacementVisitor`, and all it does is replace any parameters in the target node with the given source parameters.

ReplacementVisitor overrides VisitBinary(), VisitConstant(), and VisitParameter(), all of which call ReplaceNode(), which determines whether the swap should occur:

```
private Expression ReplaceNode(Expression node)
{
    if (node == this.Find)
    {
        return this.Replacement;
    }
    else
    {
        return node;
    }
}
```

You now have implementations in place to generate expressions, along with being able to do crossover on two expressions. The last part is being able to create a data set to test the GA. In the next section, you'll see how you can use a bit of reflection magic to make data generation easy.

GENERATING DATA SETS

When you're evolving expressions, you need a set of data that the GA can use as its target. But nobody likes to create a data set by hand. You'd like to parse an expression in a string like this:

```
"a => a + 3 * Math.Pow(a, 2.5)"
```

Once you have the lambda expression, you could compile it and feed it random input values, capturing the results in the process. Then you'd have your data set that you could give the GA. Unfortunately, there's no parsing functionality in System.Linq.Expressions, but a parser already exists with the C# compiler!

In the ExpressionEvolver solution is a project called ExpressionBaker. This contains a class called Baker, which takes a string and "bakes" it such that an expression is created. This is done in the Bake() method:

```
public Expression<TDelegate> Bake()
{
    var name = string.Format(CultureInfo.CurrentCulture,
        BakerConstants.Name, Guid.NewGuid().ToString("N"));
    var cscFileName = name + ".cs";
    File.WriteAllText(cscFileName,
        string.Format(CultureInfo.CurrentCulture,
            BakerConstants.Program, name, this.GetDelegateType(),
            this.Expression));

    this.CreateAssembly(name, cscFileName);

    return Assembly.LoadFrom(name + ".dll").GetType(name)
        .GetField("func").GetValue(null) as Expression<TDelegate>;
}
```

This method takes the given expression (stored in the Expression property) and puts it into a .cs file, which contains code that looks like this:

```
public const string Program =
    @"using System;using System.Linq.Expressions;
    public static class {0}{{
        public static readonly Expression<{1}> func = {2};}}";
```

Once the file is saved to disk, `CreateAssembly()` creates a new DLL via the C# compiler:

```
private void CreateAssembly(string name, string cscFileName)
{
    var startInformation = new ProcessStartInfo("csc");
    startInformation.CreateNoWindow = true;
    startInformation.Arguments = string.Format(
        CultureInfo.CurrentCulture,
        BakerConstants.CscArguments, name, cscFileName);
    startInformation.RedirectStandardOutput = true;
    startInformation.UseShellExecute = false;

    var csc = Process.Start(startInformation);
    csc.WaitForExit();
}
```

Now that you have an assembly, you can easily find the expression in the read-only field with a little bit of reflection magic, which is what the last line of code in `Bake()` does. It's a little bit of a hack, but hey, if the compiler already has all the logic to parse an expression, why not reuse that code?

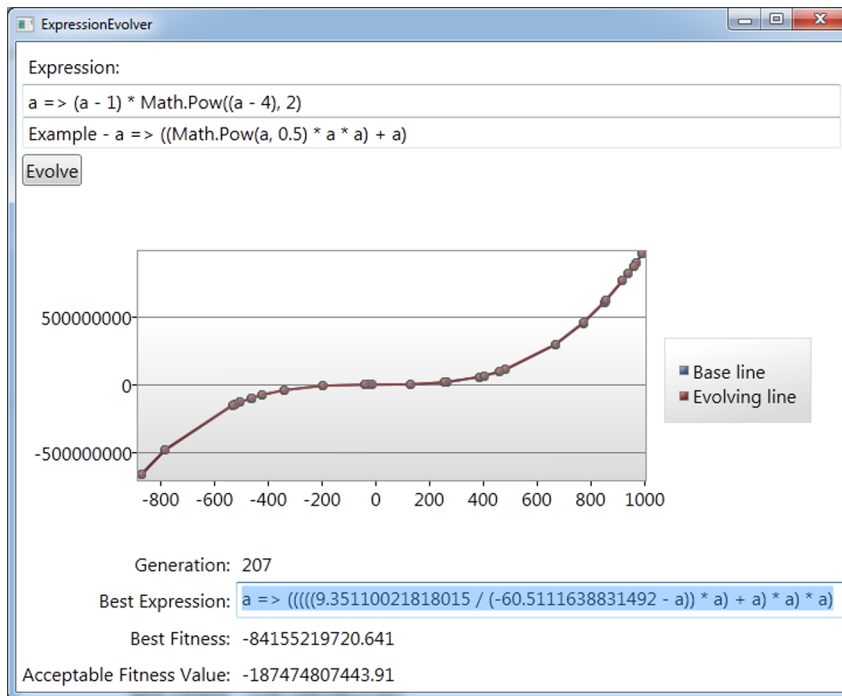


Figure 6.10 Watching expressions evolve. The graph shows the base line with the current best-evolved expression. In this case, you can't see a difference, and that's a good thing!

NOTE Admittedly, although the Baker class provides the functionality needed to compile an expression at runtime, it's clunky and feels hackish. In chapter 10, you'll see how Project Roslyn makes the compiler easily accessible at runtime. You may want to come back to this section and rewrite the Baker class using Project Roslyn's compiler API once you're done with chapter 10.

Again, there's more code to ExpressionEvolver than what's shown in this chapter. But let's see what happens when you run the code.

RUNNING THE CODE

You have everything in place. Mutating expressions, parsing expressions and so on. Now it's time to run the application. Figure 6.10 shows what the application looks like when it runs. You can see two lines: one is the target, and the other is the best-evolved expression in the current generation.

The resulting expression doesn't look much like the given expression, but that's exactly the point. The GA doesn't know what that original expression is, but it's still able to come up with an expression that matches the original line well.

Reducing expressions

One artifact of using tree-based structures in a GA is the notion of *bloat*. Bloat is when the trees start growing out of hand during the evolutionary process. This is something that we've seen in ExpressionEvolver. For example, starting with a data set that was generated from this expression

```
a => (2 * Math.Pow(a, 4)) - (11 * Math.Pow(a, 3)) -
      (6 * Math.Pow(a, 2)) + (64 * a) + 32
```

yielded the following acceptable result:

```
a => (((((-1 * a) - a) * ((-1 * a) - a)) *
      ((-1 * a) - a) - a)) +
      (((((-1 * a) - a) * (-1 * a)) *
      (-1 * a)) * (-1 * a))
```

Even though the graphs are similar, the expressions don't look anything alike, until you do some symbolic reduction on the result:

```
a => (2 * Math.Pow(a, 4)) - (12 * Math.Pow(a, 3))
```

Having an expression that's smaller would help in reducing some of the memory consumption we've seen while executing ExpressionEvolver.

There's a `Reduce()` method in the `Expression` class, but that has nothing to do with mathematical symbolic reduction. It would be nice if there were a library out there to do this kind of reduction on .NET expressions. The closest we've found is WolframAlpha (<http://alpha.wolfram.com>), which is what we used to reduce the previous result. But although an API is available, finding the result would require a web service call. That's not too bad, but we'd have to be careful not to reduce every expression after every generation. Having an engine in-process would be a better alternative.

You've now seen how to use expressions that can create programs better than existing programs. That's quite a powerful example of metaprogramming in action!

6.5 *Summary*

In this chapter, you learned how to create and modify expressions, the differences between expressions and dynamic methods, and how to use expressions to evolve code.

In the next chapter, you'll see how to write succinct code that's augmented after the compiler process. Rather than waiting until runtime to generate code, a parser is used to examine an assembly and change its contents based on the existence of meta-data or a coding convention.

Metaprogramming IN .NET

Hazzard • Bock



When you write programs that create or modify other programs, you are metaprogramming. In .NET, you can use reflection as well as newer concepts like code generation and scriptable software. The emerging Roslyn project exposes the .NET compiler as an interactive API, allowing compile-time code analysis and just-in-time refactoring.

Metaprogramming in .NET is a practical introduction to the use of metaprogramming to improve the performance and maintainability of your code. This book avoids abstract theory and instead teaches you solid practices you'll find useful immediately. It introduces core concepts like code generation and application composition in clear, easy-to-follow language, and then it takes you on a deep dive into the tools and techniques that will help you implement them in your .NET applications.

What's Inside

- Metaprogramming concepts in plain language
- Creating scriptable software
- Code generation techniques
- The Dynamic Language Runtime

Written for readers comfortable with C# and the .NET framework—no prior experience with metaprogramming is required.

Kevin Hazzard is a Microsoft MVP, consultant, teacher, and developer community leader in the mid-Atlantic USA.

Jason Bock is an author, Microsoft MVP, and the leader of the Twin Cities Code Camp.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/Metaprogrammingin.NET

“An excellent way to start fully using the power of metaprogramming.”

—From the Foreword by Rockford Lhotka, Creator of the CSLA .NET Framework

“A thorough and comprehensive distillation of the vast array of code generation options in .NET.”

—Harry Cummings, Softwire

“An extensive collection of *aha!* discoveries on developing applications beyond the mere compiler.”

—William Lee, Qualcomm, Inc.

“An excellent reference ... insightful examples.”

—Arun Noronha
Guardian Protection Services

