

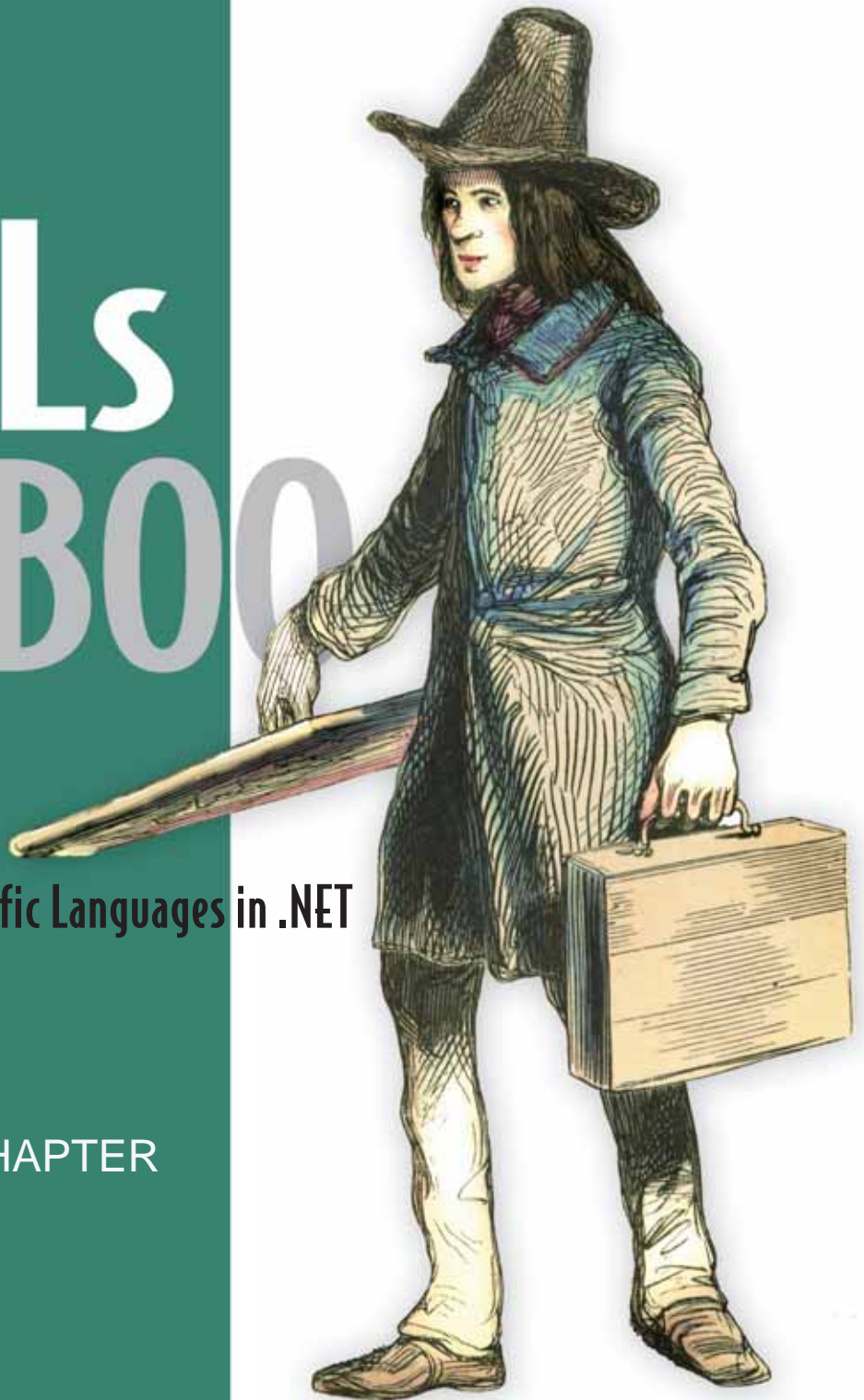
Ayende Rahien

DSLs in BOO

Domain-Specific Languages in .NET

SAMPLE CHAPTER

 MANNING





DSLs in Boo

Oren Eini

writing as Ayende Rahien

Chapter 6

brief contents

- 1 ■ What are domain-specific languages? 1
- 2 ■ An overview of the Boo language 22
- 3 ■ The drive toward DSLs 39
- 4 ■ Building DSLs 63
- 5 ■ Integrating DSLs into your applications 86
- 6 ■ Advanced compiler extensibility approaches 108
- 7 ■ DSL infrastructure with Rhino DSL 134
- 8 ■ Testing DSLs 150
- 9 ■ Versioning DSLs 173
- 10 ■ Creating a professional UI for a DSL 194
- 11 ■ DSLs and documentation 221
- 12 ■ DSL implementation challenges 239
- 13 ■ A real-world DSL implementation 263

Advanced compiler extensibility approaches

In this chapter

- Cracking open the Boo compiler
- Quasi-quotation and meta-methods
- Extending the compiler
- Changing the language

Boo offers a rich set of extensibility mechanisms that you can use. We'll look at them for a couple of reasons: to make sure you understand what is possible, and to expose you to the way they can be used to create easily readable, intuitive, and natural DSLs.

But before we can start talking about compiler extensibility, we need to look at the compiler itself and define some common terms. A lot of Boo's capabilities are exposed in ways that only make sense if you understand these concepts. I'll assume compilers aren't your area of expertise and make sure that you can understand and use the capabilities we'll look at.

You won't need this knowledge on a day-to-day basis. You can build most DSLs without going deep into the compiler, but it's important to understand how things work under the hood. There are some situations when it's easier to go into the

compiler and make a change than work around the problem with the tools that are exposed on the surface.

The basic structure of the Boo compiler is the *pipeline*—it's how the compiler transforms bits of text in a file into executable code.

6.1 The compiler pipeline

When the compiler starts to compile a set of files, it runs a set of steps to produce the final assembly. This series of steps is called the compiler *pipeline*.

Figure 6.1 shows a partial list of the steps in a standard compilation. Right now, a typical Boo compiler pipeline has over 40 steps, each of which performs a distinct task. When you write a step, you only have to focus on that particular task, nothing else.

TIP There are several Boo compiler pipelines that you can choose from, but switching pipelines is fairly advanced stuff, and not something you usually need to do for DSLs.

Let's focus on the first step, parsing the code. In this step, the parser reads the code and outputs a set of objects that represent that code. We've seen that done in chapter 1, when we talked about abstract syntax trees (ASTs). ASTs are the object models that the compiler uses to represent the code.

The parser takes code like this,

```
if 1 == 2:
    print "1 = 2"
else:
    print "1 != 2"
```

and turns it into an object model that looks like figure 6.2.

You can think of AST as the compiler's DOM, such as the XML and HTML DOM that you're probably already familiar with. Like them, the AST can be manipulated to get different results. In fact, this is what most of the compiler steps do. The first step, parsing, takes the code and translates it to AST; all the other steps scan and transform the AST. There are several extension points into the compiler pipeline, from custom compiler steps to meta-methods, and from AST attributes to AST macros. We will discuss all of them in this chapter.

Manipulating the AST takes some getting used to. I have shocked people when I suggested changing `if` statement semantics as one of the options for extending the

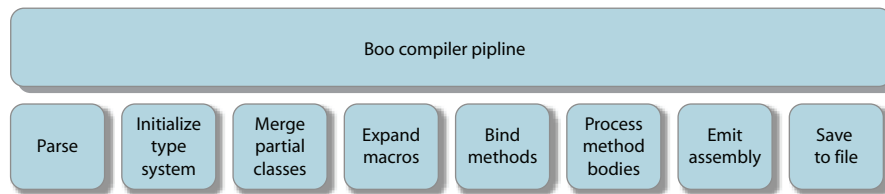


Figure 6.1 A selection of Boo's compiler steps

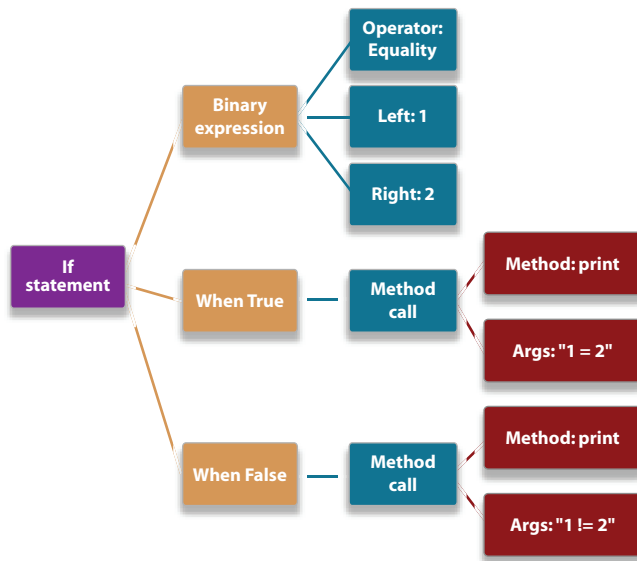


Figure 6.2 Abstract syntax tree of an `if` statement

language. Hopefully, when we go over all the extensibility options in the compiler, it will become clearer. Mind you, you’ll want to exercise some self-control when doing this. Although an `if` loop is an interesting idea, it seems to bother some people.

Now let’s use what we’ve learned about the compiler and look at how it works and how we can modify it.

6.2 *Meta-methods*

The first level of extending the compiler is easy—we covered it in chapter 2 when we talked about duck typing and `IQuackFu`. The next level is when we start working with the compiler’s AST that represents your source code and from which the compiler generates the IL to build assemblies. And the first step in that direction is building meta-methods.

A *meta-method* is a shortcut into the compiler. It’s a method that accepts an AST node and returns an AST node. *AST node* is a generic term for all the types of nodes that compose the AST.

Let’s implement an `assert` method. But, because Boo already has an `assert` statement, we’ll use “`verify`” as the method name. Listing 6.1 shows the method implementation in its entirety.

Listing 6.1 Implementing the `verify` method

```
[Meta]
static def verify(expr as Expression):
    return [
        unless $expr:
            raise $(expr.ToCodeString())
    ]
```

Exploring the AST

We'll do a lot of work in this chapter and the rest of the book that touches the AST directly, and I highly recommend that you explore it yourself. Although I can explain the concepts behind the methods we're going to use, there is no substitute for experience, particularly when it comes to understanding how to use the AST.

The best way to explore it is to ask the Boo compiler to print the resulting AST from a given piece of code. You can do that by including the `PrintAst` step in the compiler pipeline.

Another option is to use quasi-quotation (discussed later in this chapter) and .NET Reflector (<http://www.red-gate.com/products/reflector/>) to get an understanding of how the AST is composed and used. I suggest spending some time familiarizing yourself with the AST. It's important to get a good understanding of how the compiler works so you can fully utilize the compiler's capabilities.

The code in listing 6.1 creates a new keyword in the language. We can use it to ensure that a given condition is true; if it isn't true, an exception will be thrown. We can use `verify` in the following manner:

```
verify arg is not null
```

The `verify` implementation instructs the compiler to output a check and to raise an exception if the check fails. For now, please ignore any syntax you don't understand. I've used quasi-quotation to save some typing; we'll look at quasi-quotation in the next section.

Listing 6.1 shows a static method that's decorated with the `[Meta]` attribute and that accepts an AST expression. This is all you need to create a meta-method.

When you have a meta-method, you can call it like a regular method:

```
verify ( 1 == 2 )
```

A note about the code in this chapter

In this chapter, I will show the compiler manipulation code in Boo and provide several examples in C#. Boo is more suited to writing code for compiler manipulation, but most languages will do.

We'll discuss code management (IDEs, compilation, and the like) in chapter 10 (which focuses on UIs) and in chapter 12 (on managing DSL code).

If you want to use an IDE for the code in this chapter, you will need to use Sharp Develop (<http://www.icsharpcode.net/OpenSource/SD/>).

The code in this chapter was compiled using Boo 0.9.2 and Sharp Develop 3.1.0.4977.

Because Boo supports method invocations without parentheses, you can also call it in the following fashion:

```
verify 1 == 2
```

When the compiler sees a call to a meta-method, it doesn't emit code to call the meta-method at runtime. Instead, the meta-method is executed during compilation. The compiler passes the meta-method the AST of the arguments of the method code (including anonymous blocks), and then we replace this method on the AST with the result of calling the meta-method. Figure 6.3 shows the transformation that is caused by the meta-method.

This process is similar to text-substitution macros in C and C++ (and even more like Lisp and Scheme macros, if you're familiar with them), but this not mere text pre-processing. It's actual code that runs during compilation and outputs any code it wants back into the compilation process. Another difference is that we're dealing directly with the compiler's AST, not just copying lines of text.

A lot of people seem to have a hard time grasping this distinction. The compiler will ask you, at compilation time, what kind of transformation you want to do on the code. It then takes the results of the transformation (the method's return value, which is an AST expression) and replace the method call with the returned expression.

Note the difference between a method returning a value at runtime and what meta-methods are doing. The return value from a meta-method is an AST expression that is replacing the method call. When the compiler emits the final IL, it is the returned expression that will end up in the compiled DLL, not the method call.

The idea of code running at compilation time and modifying the compiled output is the main hurdle to understanding how you can modify the Boo language. I suggest taking a look at the compiled output with Reflector—it usually help clear things up. Figure 6.3 also shows the transformation happening at compile time.

The Boo code in listing 6.1 can also be translated to the C# shown in listing 6.2. This version shows what happens under the covers, and it's a bit more explicit about what is going on. Both method implementations have exactly the same semantics.

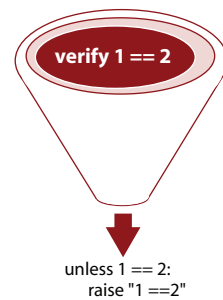


Figure 6.3 The code transformation caused by the `verify` meta-method

Listing 6.2 A C# implementation of the `verify` method

```
[Meta]
public static UnlessStatement verify(Expression expr)
{
    UnlessStatement unless = new UnlessStatement();
    unless.Condition = expr.Clone();
    RaiseStatement raise = new RaiseStatement();
    raise.Exception = new StringLiteralExpression(expr.ToCodeString());
    unless.Statements.Add(raise);
    return unless;
}
```

We've used meta-methods before, when we implemented the `when` keyword for the Scheduling DSL, in chapter 3. Meta-methods are often used in DSLs. When you run into the limits of what the compiler offers out of the box, meta-methods are your next best option.

It's important that you come to grips with the idea of AST manipulation; this is the key to what we'll discuss in the rest of the chapter. Further on in this chapter, we'll talk about AST macros and AST attributes, both of which are similar to meta-methods, and rightfully so. They take the same AST manipulation approach, but they're used differently and generally have more power at their disposal.

But before we get to them, we should take a look at quasi-quotation and why it's useful.

6.3 Quasi-quotation

You've already seen quasi-quotation in the `verify` method (listing 6.1). Quasi-quotation is a way to use the compiler's existing facilities to translate text into code. But in this case, instead of translating text to code, the compiler is translating the original text into code that produces the code. Confusing, isn't it? It will be easier to look at some examples.

Let's say you wanted to create the AST for a trivial `if` statement. The code that you want to generate looks like this:

```
if date.Today.Day == 1:
    print "first of month"
```

Remember the AST diagrams you have seen so far? Let's see what it takes to generate the AST for this statement. We'd need all the code in listing 6.3 to make this work.

Listing 6.3 Generating the AST for a trivial `if` statement

```
ifStmt = IfStatement(
    Condition: BinaryExpression(
        Operator: BinaryOperatorType.Equality,
        Left: AstUtil.CreateReferenceExpression("date.Today.Day"),
        Right: IntegerLiteralExpression(1)
    ))
write = MethodInvocationExpression(
    Target:
        AstUtil.CreateReferenceExpression("System.Console.WriteLine")
)
write.Arguments.Add( StringLiteralExpression('first of month') )
ifStmt.TrueBlock = Block()
ifStmt.TrueBlock.Add(
    write
)
```

If you're like me, you're looking at this code and thinking that programming suddenly seems a lot harder than it used to be. In truth, though, it's not much different than working with the XML or HTML DOM, which you're likely already familiar with. If you've ever done any work using `System.CodeDOM`, for that matter, the code in

listing 6.3 should be instantly familiar to you. Nevertheless, if you've ever done any work with any type of DOM, you'll know that it's extremely tedious. It's easy to get lost in the details when you have even slightly complex scenarios.

You likely won't need to write much AST—usually just wrappers and the like—but this is probably the most tedious and annoying part of having to deal with the compiler. For that reason, we have quasi-quotation, which allows us to produce the AST we want without the pain.

The 15 lines of code in listing 6.3 can be written in 4 lines by using quasi-quotation, as shown in listing 6.4.

Listing 6.4 Generating the AST of a trivial `if` statement by using quasi-quotation

```
ifStmt = [|
    if date.Today.Day == 1:
        System.Console.WriteLine("first of month")
|]
```

Listing 6.4 produces the exact same results as listing 6.3. When the compiler encounters this code, it does the usual parsing of the code, but instead of outputting the IL instructions that would execute the code, it outputs the code to build the required AST, much in the same way we did in listing 6.3.

The fun part is that you aren't limited to generating the AST code statically; you can also generate it dynamically. You can use `$variable` to refer to an external variable or `$(variable.SomeValue)` to refer to a more complex expression, and they will be injected into the generated AST building code. Let's look at an example.

Let's say you want to generate code that will output the date that this code was compiled. You could do it with the code in listing 6.5.

Listing 6.5 Generating code that outputs date this code was compiled

```
currentDate = date.Today.ToString()
whenThisCodeWasCompiled = [|
    System.Console.WriteLine( $currentDate );
|]
```

I believe this is the cue for a light-bulb-over-the-head moment. You aren't limited to using strings to pass to quasi-quotation blocks; you can use anything that's directly translatable to AST. Listing 6.6 shows a more interesting example, if not a particularly useful one.

Listing 6.6 Composing AST using quasi-quotation

```
if compilingOnMono:
    createConnection = [|
        Mono.Data.Sqlite.SqliteConnection()
    |]
else:
    createConnection = [|
```

```

        System.Data.Sqlite.SqliteConnection()
    ]]
connectToDatabase = [|
using con = $createConnection():
    con.ConnectionString = GetConnectionString()
    con.Open();
    # make use of the opened connection...
]|

```

Consider using this code when you want to generate conditional code. As shown in listing 6.6, we choose, at compile time, which library to use to connect to a database, based on the platform we’re running on. Different code will be generated for each platform. This will make those things a piece of cake.

Does this mean that we’re limited to building in Boo?

No, it doesn’t. The Boo AST is composed of standard CLR classes, and it can be used from any CLR language. As such, you can use any CLS-compliant language to interact with the Boo compiler. I have quite a few Boo DSLs in use right now that use C# to manipulate the AST.

But Boo does have facilities to make AST manipulation easier, quasi-quotation being chief among them. You’ll have to decide for yourself what is easier for you.

You can also use the `$()` syntax directly, as in listing 6.7.

Listing 6.7 Using complex expressions in quasi-quotation

```

whenThisCodeWasCompiled = [|
    System.Console.WriteLine( $( date.Now ) );
]|

```

We’ll make heavy use of this technique as we get more involved with AST manipulations.

Now, let’s see how we can use this knowledge to do some really interesting things.

6.4 AST macros

Meta-methods and AST macros differ in several ways:

- An AST macro has full access to the compiler context and the full AST of the code, and it can collaborate with other macros, compiler steps, and AST attributes to produce the final result. A meta-method can only affect what was passed to it via its parameters. Meta-methods can produce the same results as AST macros, but not as easily.
- An AST macro can’t return values, but a meta-method can.
- An AST macro is exposed by importing its namespace; a meta-method must be referenced by namespace and class. This is a minor difference.

Let’s look at a simple example—a macro that will unroll a loop.

Generating blocks with quasi-quotation and macros

When we're using quasi-quotation, we're using the standard Boo parser and compiler to generate the AST code. This is great, because it means that we can use quasi-quotation to generate any type of AST node that we can write on our own (classes, namespaces, properties, method, statements, and expressions). But it means that when the compiler is parsing the code inside the quasi-quotation, it must consider that you can put any type of AST node there.

This means that when the compiler encounters the following piece of code, it will output an error:

```
block = [|
    val = 15
    if val == 16:
        return
|]
```

Here the parser believes that we're starting with field declarations because a quasi-quotation can be anything. It can be a class declaration, a method with global parameters, a single expression, or a set of statements. Because of that, the parser needs to guess what the context is. In this particular edge case, the compiler gets it wrong, and we need to help it figure it out.

The problem in the previous example is that we're parsing field declarations (which means that we are in a class declaration context), and then we have an `if` statement in the middle of that class declaration. This is obviously not allowed, and it causes the compiler error.

Here's a simple workaround to avoid this problem:

```
code = [|
    block:
        val = 15
        if val == 16:
            return
|]
block = code.Body
```

Adding `block` in this fashion has forced the parser to consider the previous code as a macro statement, which means it's inside a method where `if` statements are allowed. We're only interested in the content of the block, so we extract the block of code from the macro statement that we fooled the parser into returning.

6.4.1 The unroll macro

Listing 6.8 shows the code using the macro, and listing 6.9 shows the results of compiling this code.

Listing 6.8 Using the unroll macro

```
unroll i, 5:
    print i
```

Listing 6.9 The compiled output of the unroll macro

```

i = 0
print i
i = 1
print i
i = 2
print i
i = 3
print i
i = 4
print i

```

Now look at listing 6. 10, which shows the code for the unroll macro.

Listing 6.10 The unroll macro

```

# Create a class for the macro. The class name is
# meaningful: [macro name]Macro allows us to later refer
# to the macro using [macro name].
# Note that we inherit from AbstractAstMacro
class UnrollMacro(AbstractAstMacro):

    # Perform the compiler manipulation.
    # The compiler hands us a macro statement, and we have
    # to return a statement that will replace it.
    override def Expand(macro as MacroStatement) as Statement:

        # Define a block of code
        block = Block()

        # Extract the second parameter value
        end = cast(IntegerLiteralExpression, macro.Arguments[1]).Value

        for i in range(end):
            # Create assignment statement using the block: trick
            # and add it to the output
            assignmentStatement = [|
                block:
                    $(macro.Arguments[0]) = $i
            |].Body
            block.Add(assignmentStatement)

            # Add the original contents of the macro
            # to the output
            block.Add(macro.Body)

        return block

```

We'll go over listing 6.10 in detail, because AST macros can be confusing the first time you encounter them. AST macros are capable of modifying the compiler object model, causing it to generate different code than what is written in the code file.

First, we define a class that inherits from `AbstractAstMacro`, and then we override the `Expand()` method. When the compiler encounters a macro in the source code, it instantiates an instance of the macro class, calls the `Expand()` method (passing the

The macro class versus the macro statement

It's important to make a distinction between a class that inherits from `AbstractAstMacro` (a macro class) and one that inherits from `MacroStatement` (a macro statement).

The first is the class that implements logic to provide a compile-time transformation. The second is part of the compiler object model, and it's passed to the macro class as an argument for the transformation.

macro statement as an argument), and replaces the original macro statement with the output of the call to `Expand()`.

The `MacroStatement` argument that the compiler passes to the macro class contains both the arguments (the parameters passed after the call to the macro) and the block (the piece of code that's inside the macro statement).

In the case of the unroll macro, we take the second argument to the macro and extract it from the compiler object model. Then we run a `for` loop, and in each iteration we perform four basic actions:

- Generate an assignment of the current value to the first argument to the macro
- Generate a call to the macro body
- Add both statements to the output code
- Return the output of the code to the compiler, which will replace the `MacroStatement` that was in the code

Note that you can return a `null` from the `Expand()` method, in which case the node will be completely removed. This is useful in various advanced scenarios, as we'll see when we discuss correlated macros in section 6.4.5.

6.4.2 Building macros with the `MacroMacro`

The Boo compiler can do a lot of stuff for us, so it should come as no surprise that it can help us with building macros. The `MacroMacro` is an extension to the compiler that makes it simpler to write macros. You don't need to have a class, inherit from `AbstractAstMacro`, override methods, and so on. All you need to do is write the code to handle the macro transformation.

Let's write the unroll macro again, this time using the `MacroMacro`. Listing 6.11 contains the code.

Listing 6.11 Using the `MacroMacro` to write the `UnrollMacro`

```
# Using the MacroMacro, we don't need a class,
# just to define what we want the macro to do
macro Unroll2:

    # extract the second parameter value
    end = cast(IntegerLiteralExpression, Unroll2.Arguments[1]).Value
```

```

for i in range(end):
    # create assignment statement, using the block:
    # trick and add it to
    # the output
    statement = [|
        block:
            $(Unroll2.Arguments[0]) = $i
    |].Body
    yield statement

    # add the original contents of the macro
    # to the output
    yield Unroll2.Body

```

As you can see, the main differences between this and listing 6.10 is that the `MacroMacro` removes the need to create a class. We can also yield the statements directly, instead of gathering them and outputting them in a single batch. Inside the macro block, we can refer to the macro statement using the macro's name. This means that whenever you see `Unroll2` inside the macro in listing 6.11, it refers to the `MacroStatement` instance that's passed in to the macro implementation (the `MacroMacro` creates a `MacroStatement` variable named `Unroll2` behind the scenes).

From the point of view of the code, we've been using macros throughout the book. I just didn't tell you about them until now. Let's take a look at the most common macro:

```
print "hello there, I am a macro"
```

`Print` will translate to calls to `Console.WriteLine`. Let's take a look at the slightly simplified version of this implementation in listing 6.12.

Listing 6.12 Boo's print macro code

```

# This is a helper method that's useful for all sorts of "write line"
# type macros, such as print, debug, warn, etc.
# It accepts the macro and two method invocation expressions: one for
# outputting text and the second for outputting text and a line break.
# This method will take the arguments of the macro and print them all.
def expandPrintMacro(macro as MacroStatement,
                    write as Expression,
                    writeLine as Expression):
    # If the macro is empty, output an empty line
    if len(macro.Arguments) == 0:
        return [| $writeLine() |]

    # Create a block of code that will contain the output
    # methods that will be generated
    block = Block()
    # -1 in Boo's lists means the last element
    last = macro.Arguments[-1]
    for arg in macro.Arguments:
        if arg is last: break
        # Add method call to output a single macro argument
        block.Add([| $write($arg) |].withLexicalInfoFrom(arg))

```

```

        block.Add([ $write(' ') ])
    # Output the last macro argument with a line break
    block.Add([ $writeLine($last) ]).withLexicalInfoFrom(last)
    return block

# The macro, which does a simple redirect
macro print:
    return expandPrintMacro(print,
        [| System.Console.Write |],
        [| System.Console.WriteLine |])

```

Note that the macro will direct you to the `expandPrintMacro` method. This method is also used elsewhere, such as in the `debug` macro, to do most of the work.

Lexical info

You'll note that we're using the `withLexicalInfoFrom(arg)` extension method in listing 6.12. This is a nice way of setting the `LexicalInfo` property of the node.

Lexical info is the compiler's term for the location this source code came from, and it's important for things like error reporting and debugging. We're going to do a lot of code transformations, and it's easy to lose lexical info along the way.

Keeping track of the lexical info is simply a matter of carrying it around, either by using `withLexicalInfoFrom(arg)` or by setting the `LexicalInfo` property directly. Put simply, whenever we make any sort of transformation, we must also include the original code's lexical info in the transformation output.

Without lexical info, the compiler couldn't provide the location for an error. Imagine the compiler giving you the following error: "Missing ; at file: unknown, line: unknown, column: unknown." This is why keeping the lexical info around is important.

The `print` macro also passes two methods to the `expandPrintMacro()` helper method: `Write` and `WriteLine`. Those use quasi-quotation again to make it easier to refer to elements in the code. The interesting part, and what makes Boo easy to work with, is that we can still take advantage of things like overloading when we're rewriting the AST. The method resolution happens at a later stage, which makes our work a lot simpler.

The `expandPrintMacro()` method checks whether we passed any arguments to the macro. If we did, it will output all of them using the `write()` method and output the last one using `writeLine()`.

This explanation of how the `print` macro works took a lot more text than the macro itself, which I consider a good thing. We'll take a peek at another well-known macro, the `using` macro, and then we'll write one of our own.

6.4.3 Analyzing the `using` macro

In C#, `using` is a language keyword. In Boo, it's a macro, and it has a tad more functionality. Take a look at listing 6.13, which makes use of Boo's `using` statement.

Listing 6.13 The using statement in Boo, with multiple parameters

```
using file = File.Create("myFile.txt"), reader = StreamWriter(file):
    reader.WriteLine("something")
```

You can specify multiple disposables in Boo's using macro, which often saves nested scoping. When the Boo compiler sees the code in listing 6.13, it attempts to find a macro for any unknown keywords it finds. It will only generate an error if it can't find a matching macro.

The compiler finds relevant macros by searching the imported namespaces. Boo has no problem if several macros have the same name in different namespaces (though there may be some confusion on the part of the user).

The macro in listing 6.13 will generate the AST shown in figure 6.4. This is the `MacroStatement` instance that the using macro implementation will get. As you'll notice, it's split into two major parts: the `Arguments` collection contains everything that was written after the macro name, and the `Block` contains all the code that was written within this macro.

At compilation time, the compiler will create a new instance of the macro class and pass it the `MacroStatement` object. It will also replace the macro statement with the output that was returned from the macro. If this sounds familiar, that's because it's precisely the way meta-methods work.

Listing 6.14 shows the implementation of the using macro.

Listing 6.14 The using macro's implementation

```
macro using:
    # Get the content of the using statement
    expansion = using.Body
    # Iterate over all the macro expressions in reverse order
    for expression as Expression in reversed(using.Arguments):
        # Create a temporary variable and assign it the
        # current expression.
        temp = ReferenceExpression(_
            context.GetUniqueName("using", "disposable"))
        assignment = [|
            $temp = $expression as System.IDisposable
            |].withLexicalInfoFrom(expression)

        # Create a try/ensure block, with the current expansion, and
        # place it in the expansion variable, so it will be wrapped
```

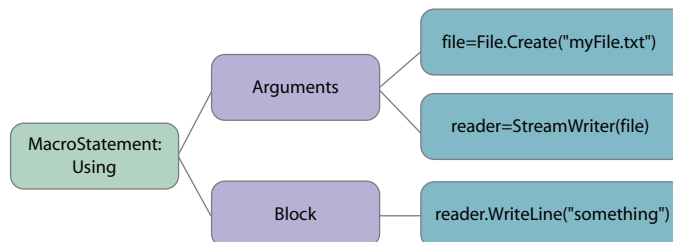


Figure 6.4 A simplified AST representation of a using macro statement

```

# by the next macro argument
expansion = [|
    $assignment
    try:
        $expansion
    ensure:
        if $temp is not null:
            $temp.Dispose()
            $temp = null
|]

return expansion

```

In listing 6.14, we gain access to the `MacroStatement` using the macro name—in this case, `using`. Then we assign the `using.Body` to the expansion variable.

Once that’s done, we iterate over the argument collection in reverse order. On each iteration, we create a new `try ... ensure` statement (the Boo equivalent for C#’s `try ... finally`) and dispose of the argument properly in the `ensure` block. We then take the content of the expansion variable and place it inside the `try` block. We set the newly created `try ... ensure` statement as the new value of the expansion variable, which we carry forward to the next iteration.

The end result of the code in listing 6.14 is shown in listing 6.15.

Listing 6.15 The code outputted by the `using` macro

```

__using2__ = ((file = File.Create('myFile.txt')) as System.IDisposable)
try:
    __using1__ = ((reader = StreamWriter(file)) as System.IDisposable)
    try:
        reader.WriteLine()
    ensure:
        if __using1__ is not null:
            __using1__.Dispose()
            __using1__ = null
ensure:
    if __using2__ is not null:
        __using2__.Dispose()
        __using2__ = null

```

I’m sure you’ll agree that this isn’t pretty code—not with all those underscores and variable names that differ only by a numeral. But it’s what the compiler generates, and you won’t usually see it. I should note that this isn’t code in the style of `Form1.Designer.cs`. This code is generated by the compiler; it’s always regenerated during compilation and never sees the light of day, nor is it saved.

You can see, in the second `try` block, the code that we originally placed in the `using` block: in the `ensure` block we dispose of the reader safely. The inner `try` block is wrapped in another `try` block, which disposes of the file safely.

Now, if you go back to listing 6.14, you’ll see that we’re making use of a strange `Context.GetUniqueName()` method. Where did this come from?

The availability of the context is one of the main differences between macros and meta-methods. The context lets you access the entire state of the compiler. This

means you can call the `GetUniqueName()` method, which lets you create unique variable names, but it also means that you can access more interesting things:

- Compiler services, which include such things as the Type System Service, Boo Code Builder, and Name Resolution Service
- The compilation unit—all the code being compiled at that moment
- The assemblies referenced by the compiled code
- The compiler parameters, which include the compiler pipeline, inputs, and other compilation options
- The errors and warnings collections

Those are useful for more advanced scenarios. But before we make use of them, we'll write a more complex macro, to see how it goes.

6.4.4 Building an SLA macro

Recently I needed to check how long particular operations took and to log warnings if the time violated the service level agreement (SLA)¹ for those operations. The code was in C#, and I couldn't think of any good way of doing this except by manually coding it over and over again. All the ideas I had for solving this problem were uglier than the manual coding, and I didn't want to complicate the code significantly. Let's see how we can handle this with Boo, shall we?

First, we need to define how we want the code using the SLA to look. Listing 6.16 contains the initial draft.

Listing 6.16 Initial syntax of a macro that logs SLA violations

```
limitedTo 200ms:
    PerformLongOperation()
    whenExceeded:
        print "Took more than 200 ms!"
```

Let's start small by creating a macro that will print a warning if the time limit is exceeded. We should note that Boo has built-in support for time spans, so we can say `200ms` (which is equal to `TimeSpan.FromMilliseconds(200)`) and it will create a `TimeSpan` object with 200 milliseconds for us.

Listing 6.17 shows an initial implementation.

Listing 6.17 Initial implementation of the `limitedTo` macro

```
macro limitedTo:
    # Get the expected duration from the macro arguments
    expectedDuration = limitedTo.Arguments[0]
    # Generate a unique variable name to hold the duration
    durationName = ReferenceExpression(Context.GetUniqueName("duration"))
    # Generate a unique variable name to hold the start time
    startName = ReferenceExpression(Context.GetUniqueName("start"))
```

¹ SLA (service level agreement) refers to the contracted delivery time of the service or its performance.

```

# Use quasi-quotation to generate the code to write
#a warning message
actionToPerform = [|
    block:
        print "took too long"
|].Body

# Generate the code to mark the start time, execute the code, get
# the total duration the code has run, and then execute the action
# required if the duration was more than the expected duration
block = [|
    block:
        $startName = date.Now
        $(limitedTo.Body)
        $durationName = date.Now - $startName
        if $durationName > $expectedDuration:
            $actionToPerform
|].Body

return block

```

We define the macro itself. Quasi-quotation is useful, but it has its limitations, so we use the `block:` trick to force the compiler to think that we’re compiling a macro statement, from which we can extract the block of code that we’re actually interested in.

Now, if we write this code, it will print “took too long”:

```

limitedTo 200ms:
    PerformLongOperation()

```

We aren’t done yet. We still need to implement the `whenExceeded` part, and this leads us to the idea of nested macros.

6.4.5 Using nested macros

When we talk about nested macros, we’re talking about more than nesting a print macro in a using macro. Nested macros are two (or more) macros that work together to provide a feature. The `limitedTo` and `whenExceeded` macros in the previous section are good examples of nested macros.

Before we get to the implementation of nested macros, we need to understand how Boo processes macros. The code in listing 6.18 will produce the AST in figure 6.5.

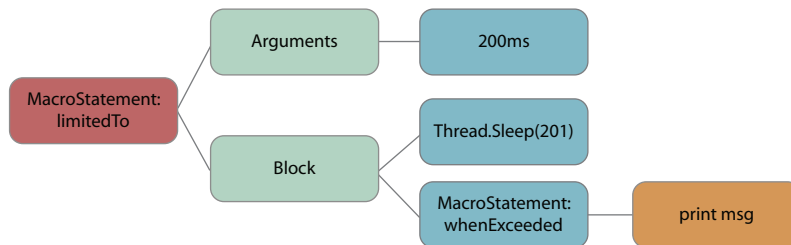


Figure 6.5
The AST
generated from
listing 6.18

Listing 6.18 Using the `limitedTo` and the nested `whenExceeded` macros

```
limitedTo 200ms:
  Thread.Sleep(201);
  whenExceeded:
    print "Took more than 200 ms!"
```

Macros are processed in a depth-first order, so the `whenExceeded` macro will be evaluated before the `limitedTo` macro. Keeping this in mind, we can now write the `whenExceeded` macro, as shown in listing 6.19.

Listing 6.19 The implementation of the nested `whenExceeded` macro

```
macro limitedTo:
  macro whenExceeded:
    limitedTo["actionToPerform"] = whenExceeded.Block
```

The `whenExceeded` macro is decidedly simple. It merely puts the block of code that was under the `whenExceeded` macro in a well-known location in that `limitedTo` dictionary. This ability to annotate AST nodes is useful, because it allows different parts of the compilation process to communicate by adding information to the relevant node, as we did here.

To finish the implementation, we need to modify listing 6.17 and change the `actionToPerform` initialization as follows:

```
actionToPerform as Block = limitedTo["actionToPerform"]
```

Because `whenExceeded` is evaluated before `limitedTo`, we'll have the block of code that was passed to `whenExceeded` in the `limitedTo` dictionary by the time we execute the `limitedTo` macro. This pattern is common with nested macros: the nested macros pass their state to the parent macros, and the parent macros do all the work. The only thing we need to do now is add some error handling and we're ready to go.

Correlated macros

What would happen if we didn't want to nest the `whenExceeded` macro? What if we wanted this code instead:

```
limitedTo 200ms:
  Thread.Sleep(201);
whenExceeded:
  print "Took more than 200 ms!"
```

This looks like a language feature, similar to the `try ... except` statement. There's no nesting, and the `limitedTo` macro is evaluated before the `whenExceeded` macro. We use the `limitedTo` macro to find the `whenExceeded` macro (by going to the parent node and finding the next node, which is the `whenExceeded` macro) and move the `limitedTo` state to it.

Because `whenExceeded` is the macro that's evaluated last, it will be in charge of producing the final code. This is a simple extension of what we did with the nested macros.

One thing that’s worth remembering is that quasi-quotation doesn’t generally work with macros; you won’t be able to use a macro inside a quasi-quotation expression. Usually, you can unpack them manually and it’s rarely a bother.

Macros are one of the key extensibility points of the compiler. They can be used inside a method or inside a class definition (to modify fields, methods, and properties),² but they can’t modify the class itself. You will occasionally want to do this, and there’s a solution for that: AST attributes.

6.5 *AST attributes*

An AST attribute applies the same concepts that we’ve looked at so far, but it does so on a grander scale. Instead of dealing with a few parameters and maybe a block of code, you can use an AST attribute anywhere you can use a typical attribute. The difference between an AST attribute and a standard attribute is that AST attributes take an active part in the compilation process.

We’ve already covered the basics of AST manipulation, so we’ll go directly to the code, and then discuss how it works. Listing 6.20 shows an example of adding post conditions on the class level that apply to all the class’s methods.

Listing 6.20 Sample usage of an AST attribute

```
[Ensure(name is not null)]
class Customer:
    name as string
    def constructor(name as string):
        self.name = name
    def SetName(newName as string):
        name = newName
```

This attribute ensures that if we call the `Customer.SetName` method with a null, it will throw an exception.

But what does it take to make this happen? Not much, as it turns out. Take a look at listing 6.21.

Listing 6.21 Implementing an AST attribute

```
# AST attributes inherit from AbstractAstAttribute
class EnsureAttribute(AbstractAstAttribute):
    # expr is the expression that we were supplied by the compiler
    # during the compilation process
    expr as Expression
    # Store the expression in a field
    def constructor(expr as Expression):
        self.expr = expr
```

² For more information about using macros for changing class definitions, see the “Boo 0.9—Introducing Type Member Macros” entry in the Bamboozled blog: http://blogs.codehaus.org/people/bamboo/archives/001750_boo_09_introducing_type_member_macros.html.

```

# Make the changes that we want
def Apply(target as Node):

    # Cast the target to a ClassDefinition and
    # start iterating over all its members.
    type as ClassDefinition = target

    for member in type.Members:
        method = member as Method
        # We do not support properties to
        # keep the example short
        continue if method is null

        # If the member is a method, modify
        # it to include a try/ensure block, which
        # asserts that the expression must
        # be true. Then override the method body with
        # this new implementation.
        methodBody = method.Body

        method.Body = [|
            try:
                $ methodBody
            ensure:
                assert $expr
        |]

```

We first inherit from `AbstractAstAttribute`; this is the key. We can accept expressions in the constructor, which is helpful, because we can use them to pass the rule that we want to validate.

The bulk of the work is done in the `Apply` method, which is called on the node that the attribute was decorating. We assume that this is a class definition node and start iterating over all its members. If the type member is a method (which includes constructors and destructors), we apply our transformation.

The transformation in this example is simple. We take all the code in the method and wrap it up in a `try ... ensure` block. When the `ensure` block is run, we assert that that expression is valid.

In short, we got the node the attribute was decorating, we applied a simple transformation, and we're done. *Vidi, vicissitude, vici—I saw, I transformed, I conquered* (pardon my Latin).

We can apply AST attributes to almost anything in Boo: classes, methods, properties, enums, parameters, fields, and so on. Boo has several interesting AST attributes, as listed in table 6.1.

AST attributes are a great way to package functionality, particularly for avoiding repetitive or sensitive coding. Implementing the Singleton pattern, the Disposable pattern, or validation post conditions are all good examples.

However good AST attributes are, they only apply to a single node. This can be useful when you want specialized behavior, but it's a pain when you want to apply some cross-cutting concerns, such as adding a method to all classes in a project or changing the default base class.

Table 6.1 Interesting attributes in Boo

Attribute	Description	Sample
[property]	Applies to fields and will generate a property from a field.	<code>[property(Name)] name as string</code>
[required]	Applies to parameters and will verify that a non-null reference was passed to the method.	<code>def Foo([required] obj): pass</code>
[required(arg!= 0)]	Applies to parameters and will verify that the parameter matches the given constraint.	<code>def Foo([required] i as int): pass</code>
[default]	Applies to parameters and will set the parameter to a default value if null was passed.	<code>def Foo([default("novice")] level as string): pass</code>
[once]	Applies to methods and properties. It will ensure that a method only executes once; any future call to this method will get the cached value.	<code>[once] def ExpensiveCall(): pass</code>
[singleton]	Applies to classes and ensures the correct implementation of the Singleton pattern.	<code>[singleton] class MySingleton: pass</code>

As you probably have guessed, Boo has a solution for that as well. Welcome to the world of compiler steps.

AST attributes with DSL

I have used AST attributes with a DSL several times. Mostly, it was to supply additional functionality to methods that were defined in the DSL or to specific parameters of those arguments.

Here's a trivial example:

```
[HtmlEncode]
def OutputFile( \
    [requires(File.Exists(file)] file as string ):
    return File.ReadAllText(file)
```

6.6 *Compiler steps*

At the beginning of this chapter, we looked at the compiler pipeline and saw that it was composed of a lot of steps executed in order to create the final executable assembly. Most modern compilers work in this fashion,³ but most modern compilers don't

³ I have quite a bit of respect for older compilers, which had to do everything in a single pass. There was some amazing coding involved in them.

boast an extensible architecture. Because Boo does have an extensible compiler architecture, you shouldn't be surprised that you can extend the compiler pipeline as well.

6.6.1 Compiler structure

Let's look again at how the compiler is structured. Figure 6.6 shows a list of compiler steps, along with a custom step that you can write and inject into the compiler pipeline. But what, I hear you asking, is a compiler step in the first place? That's a good question. A *compiler step* is a class that implements `ICompilerStep` and is registered on the compiler pipeline. When a compiler step is run, it has the chance to inspect and modify the AST of the current compilation. The entire Boo compiler is implemented using a set of compiler steps, which should give you an idea of how powerful compiler steps can be.

We can do anything we want in a compiler step. I've seen some interesting compiler steps that do the following things:

- Introduce an implicit base class for any code that isn't already in a class
- Transform unknown references to calls to a parameters collection
- Modify the `if` statement to have clearer semantics for the domain at hand (to be done with caution)
- Extend the compiler naming convention so it will automatically translate `apply_discount` to `ApplyDiscount`
- Perform code analysis (imagine `FxCop` integrated into the compiler)
- Perform compile-time code generation, doing things like inspecting the structure of a database and creating code that matches that structure

As you can imagine, this is a powerful technique, and it's useful when you want to build a more complex DSL.

Up until now, everything that we built could be executed directly by the compiler, because we used the well-known extension paths. Compiler steps are a bit different; you need to add them to the compiler pipeline explicitly.

There are two ways to do that. The first involves creating your own pipeline and inserting the compiler step into the list of steps in the appropriate location. Then you can ask the compiler to use that custom pipeline. The second approach involves setting up the compiler context yourself, and directly modifying the pipeline.

We'll see both approaches shortly. First, though, because implicit base classes are so useful, let's take a look at how we can implement them.

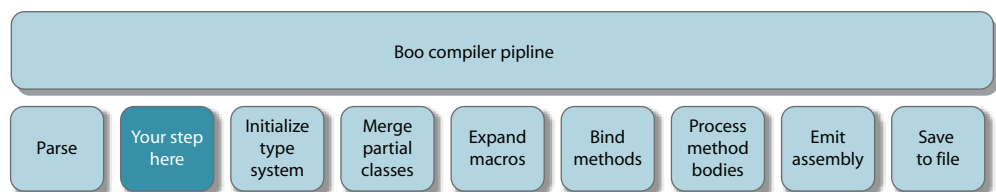


Figure 6.6 The structure of the Boo compiler pipeline, shown with a custom step

TIP The Rhino DSL project already contains a generic implementation of the implicit base class compiler step, and the `DslFactory` introduced in chapter 2 makes it easy to use. The example in section 6.6.2 shows how to build a simple compiler step. I suggest you take advantage of the Rhino DSL project for such common operations. The steps included in Rhino DSL are much more robust than the simple implementation that we will build shortly.

6.6.2 *Building the implicit base class compiler step*

The first thing we need to do when we create an implicit base class is create the base class. Listing 6.22 contains a trivial example. Instead of using one of the DSLs that we introduced in chapter 4, we will use a trivial DSL. In this case, we want to talk only about the mechanics of modifying the compiler, not about a specific DSL.

Listing 6.22 A trivial base class for a trivial DSL

```
abstract class MyDsl:
  [getter(Name)]
  name as string
  abstract def Build():
    pass
```

Listing 6.23 contains the compiler step to perform the transformation from a free-standing script to one with an implicit base class.

Listing 6.23 A compiler step for creating implicit base classes

```
# Inheriting from AbstractTransformerCompilerStep gives us a lot of
# convenience methods
class MyDslAsImplicitBaseClassStep(AbstractTransformerCompilerStep):

  # This is the method that the compiler pipeline will execute
  override def Run():
    # The Visitor pattern makes it easy to pick and choose
    # what we want to process
    super.Visit(CompileUnit)

  # Choose to process all the modules
  override def OnModule(node as Module):
    # Use quasi-quotation to generate a class definition
    # with all the code that was in the module inside it.
    # Use the same name as the module and inherit from MyDsl
    baseClass = [|
      class $(node.Name) (MyDsl):
        override def Build():
          $(node.Globals)
    |]
    # Clear the module's globals and add the newly
    # created class to the node's members.
    node.Globals = Block()
    node.Members.Add(baseClass)
```

We start off by inheriting from the `AbstractTransformerCompilerStep` class, which makes it easier to surgically get the information we want by overriding the appropriate `On[NodeType]` method and transforming only that node.

In this case, we call `Visit(CompileUnit)` when the compiler step is executed. This uses the Visitor pattern to walk through the entire AST node and call the appropriate methods, which makes navigating the AST much easier than it would be otherwise.

In the `OnModule()` method, we create a new class that inherits from the `MyDsl` class. We take the contents of the module's `Globals` section and stick it in the `Build()` method of the newly created class. We also give the class the module name. This is usually the name of the file that originated this module.

Finally, we clear the module's `Globals` section and add the new class to the module's members.

Now we need to understand how to plug this into the compiler. That also turns out to be fairly uncomplicated. We start by creating a new pipeline that incorporates our new compiler step, as shown in listing 6.24.

Listing 6.24 A compiler pipeline that incorporates our custom steps

```
class WithMyDslStep(CompileToFile):
  def constructor():
    super()
    Insert( 1, MyDslAsImplicitBaseClassStep() )
```

This compiler pipeline will add our new compiler step as the second step in the pipeline. The first step in the pipeline is the parsing of the code and the building of the AST, and we want to get the code directly after that.

Now let's compile this code. We can do it from the command line, because the compiler accepts a parameter that sets the compiler pipeline. Here's how it's done:

```
booc "-p:WithMyDslStep, MyDsl" -r:MyDsl.dll -type:library test.boo
```

Assuming that the `test.boo` file contains this line,

```
name = "testing implicit base class"
```

we'll get a library that contains a single type, `test`. If you examine this type in .NET Reflector, you'll see that its `Build()` method contains the code that was in the file, and that the `name` variable is set to the text "testing implicit base class".

Where to register the compiler step?

In general, the earlier that you can execute your compiler step, the better off you'll be. The code you move or create will still need to go through all the other compiler steps for successful compilation.

You want the code that you add to benefit from full compiler processing, so it's best to add the compiler step as early as possible.

TIP .NET Reflector is a tool that allows you to decompile IL to source code. You can find out more at <http://www.red-gate.com/products/reflector/>.

When you're building a DSL, though, you'll usually want to control the compilation process yourself, and not run it from the command line. For that, you need to handle the compilation programmatically, which listing 6.25 demonstrates.

Listing 6.25 Adding a compiler step to a pipeline and compiling programmatically

```
compiler = BooCompiler();
compiler.Parameters.Pipeline = new CompileToFile();
AddInputs(compiler.Parameters) # Add files to be compiled
compiler.Parameters.Pipeline.Insert(1, MyDslAsImplicitBaseClassStep() )
compilerContext = compiler.Run()
```

The `DslFactory` and `DslEngine` in the Rhino DSL project already handle most of this, so we'll usually not need to deal with setting the pipeline or building the compiler context manually. Knowing how to do both is important, but for building DSLs I would recommend using Rhino DSL instead of writing all of that yourself.

6.7 Summary

In this chapter, you've seen most of the extensibility mechanisms of Boo, both from the language perspective and in terms of the compiler's extensibility features. You have seen how to use the language to extend the syntax so you can expose a readable DSL to your end users.

Meta-methods, AST macros, AST attributes, and compiler steps compose a rich set of extension points into which you can hook your own code and modify the language to fit the domain that you're working on at the moment. But those aren't the only tools you have available when you build a language; the standard features of the Boo language (covered in chapter 2) make for a flexible syntax that can easily be made more readable. You can often build full-fledged DSLs without dipping into the more advanced features of compiler extensibility.

You've also seen that features such as quasi-quotation make extending the compiler a simple matter. It's not much harder to extend the compiler without using quasi-quotation, but this is somewhat tedious. Using quasi-quotation also requires far less knowledge about the AST, because you can utilize the compiler to build the AST. This isn't entirely a good thing, though, and I encourage you to learn more about the AST and how to work with it. All abstractions are leaky, and it's good to understand how to plug the leaks on your own. An easy way to do this is to look at the code that Boo generates for quasi-quotations using .NET Reflector. You'll see the AST building code that way.

So far, we've dealt only with the mechanics of extending the language. The features we've explored are powerful, but we're still missing something: how to take the language extensibility features and build a real-world DSL. There is much more to that

than extending the language. Concerns such as ease of use, readability, maintainability, versioning, documentation, and testing all need to be considered when you design and implement a DSL.

We've also talked a bit about how Rhino DSL can take care of a lot of the common things that we need to do. I keep saying that we'll get to that in a bit. The next chapter is dedicated to exploring what we can shuffle off for Rhino DSL to handle.

DSLs in BOO

Ayende Rahien



A general purpose language like C# is designed to handle all programming tasks. By contrast, the structure and syntax of a Domain-Specific Language are designed to match a particular applications area. A DSL is designed for readability and easy programming of repeating problems. Using the innovative Boo language, it's a breeze to create a DSL for your application domain that works on .NET and does not sacrifice performance.

DSLs in Boo shows you how to design, extend, and evolve DSLs for .NET by focusing on approaches and patterns. You learn to define an app in terms that match the domain, and to use Boo to build DSLs that generate efficient executables. And you won't deal with the awkward XML-laden syntax many DSLs require. The book concentrates on writing internal (textual) DSLs that allow easy extensibility of the application and framework. And if you don't know Boo, don't worry—you'll learn right here all the techniques you need.

What's Inside

- Introduction to DSLs, including common patterns
- A fast-paced Boo tutorial
- Dozens of practical examples and tips
- An entertaining, easy-to-follow style

A leader in the .NET community, **Ayende Rahien**, whose real name is **Oren Eini**, contributes to numerous open-source projects including NHibernate, Castle, and Rhino Mocks. He blogs and speaks on architecture, data access, testing, and other topics.

For online access to the author, and a free ebook for owners of this book, go to manning.com/DSLsinBoo

“A great gateway into Boo and Domain-Specific Languages.”

—Justin Chase, Microsoft

“Useful, readable, and empowering—really captures the Boo spirit.”

—Avishay Lavie, Contributor to *The Boo Programming Language*

“Goes way beyond Boo particulars into universally applicable guidance.”

—Mark Seemann, Safewhere

“... will erase any doubts you may have about writing your own DSL.”

—Garabed “Garo” Yeriazarian Baker Hughes, Inc.

