

Syntactic Macros

by Toby Miller

Syntactic macros allow new language constructs to be created. See [Macros](#) for a list of existing ones in boo.

We could, for example, mimic VisualBasic's with statement.

Given the following code:

```
fooInstanceWithReallyLongName = Foo()  
fooInstanceWithReallyLongName.f1 = 100  
fooInstanceWithReallyLongName.f2 = "abc"  
fooInstanceWithReallyLongName.DoSomething()
```

If we define a 'with' macro we could rewrite it like this:

```
with fooInstanceWithReallyLongName:  
    _f1 = 100  
    _f2 = "abc"  
    _DoSomething()
```

In boo, macros are CLI objects that implement the `Boo.Lang.Compiler.IAstMacro` interface. It is interesting to note that there is nothing *magic* about these objects. They must simply implement the interface the compiler expects. This implies that boo macros can be written in any CLI language!

When an unknown syntactic structure is encountered at compile time, like the `with` statement above, the compiler will look for the correct `IAstMacro` class, create an instance, and ask that instance to expand the macro. The compiler identifies the class to use via a simple naming convention. The class name must start with the name of the macro and end with 'Macro'. Additionally the [Pascal case](#) naming convention must be used. So in this case the name must be 'WithMacro'.

Here's the code to implement our macro:

```
import Boo.Lang.Compiler  
import Boo.Lang.Compiler.Ast  
import Boo.Lang.Compiler.Ast.Visitors
```

```

class WithMacro(AbstractAstMacro):

    private class
NameExpander(DepthFirstTransformer):

    _inst as ReferenceExpression

    def constructor(inst as
ReferenceExpression):
        _inst = inst

    override def
OnReferenceExpression(node as
ReferenceExpression):
        // if the name of the reference
begins with '_'
        // then convert the reference to
a member reference
        // of the provided instance
        if node.Name.StartsWith('_'):
            // create the new member
reference and set it up
            mre =
MemberReferenceExpression(node.LexicalInfo)
            mre.Name = node.Name[1:]
            mre.Target =
_inst.CloneNode()

            // replace the original
reference in the AST
            // with the new

```

member-reference

ReplaceCurrentNode(mre)

```
override def Expand(macro as
MacroStatement) as Statement:
    assert 1 == macro.Arguments.Count
    assert macro.Arguments[0] isa
ReferenceExpression

    inst = macro.Arguments[0] as
ReferenceExpression

    // convert all _<ref> to inst.<ref>
    block = macro.Block
```

```
ne = NameExpander(inst)
ne.Visit(block)
return block
```

Some explanation is in order. The parsing stage of the compiler pipeline parses a source stream into an abstract syntax tree (AST). A subtree, corresponding to the macro, will be passed to the Expand() method. Expand() is responsible for building an AST that will replace the provided subtree.

The subtree corresponding to a macro statement is embodied by the MacroStatement parameter.

A MacroStatement has a collection of arguments and a block.

In this case we expect a single argument: a reference to an object. We then traverse the block looking for references whose name begins with the '_' character. Whenever we encounter one, we replace it with a reference to a member.

There are two classes related specifically to AST traversal: DepthFirstVisitor and DepthFirstTransformer. Both classes walk an AST invoking appropriate methods for each type of element in the tree. In this case, we subclassed DepthFirstTransformer as a convenient way to find and replace ReferenceExpression nodes in the macro's block.

You can find this plus other macro examples in the [examples/macros directory](#).

A [custom macro syntax](#) is also planned.

Built-in macros

Some other examples of macros already implemented in boo:

```
print "hello!"
```

```
assert x == true
```

```
debug "print debug message"
```

```
using file=File.OpenText(fname): //disposes  
of file when done
```

```
    print(file.ReadLine())
```

```
o1 = object()
```

```
lock o1: //similar to "synchronized" in  
java
```

```
    pass
```