The boo Programming Language

When and Why

boo was born almost a year ago out of my frustation with existing programming language systems and my growing love for the Common Language Infrastructure and the architectural beauty of the entire .net framework.

I was frustrated mainly because I could not use the language I wanted to use (python at the time) to build the kind of systems I needed to within the technological framework my company has settled on. I had two options: I could either use a different framework (such as the python standard libraries and runtime environment) or a different programming language (C# was the logical choice for such a long time C++ programmer like myself).

I tried both and was completely satisfied by none.

When I was programming in full python mode I missed some of the things I'd normally get from a more statically typed environment such as compile time error checking (good when you're refactoring a large user interface code base such as my company's) but what I missed the most was the well thought out .net architecture and its great support for unicode, globalization and web style applications.

After a not-so-successful python attempt that had put us way behind the schedule I started C# coding like hell. I programmed a lot and by that I mean no xmas or carnival or 6 hour sleep nights. Some pretty good tools came to light during those intense times, Bamboo.Prevalence being just one of them. To make a long story short we finally delivered what we had to using C# and the .net framework. That's the holywood ending. In the alternate ending I was stressed and couldn't avoid those mixed feelings about C# in the light of my previous python experience:

> Nice syntax but do I really have to type in all those casts? What a clean callback and event system design but can't YOU just create that delegate for me, Mr. Compiler? Great, true multidimensional arrays! I need a list with the tasks marked done, couldn't it be a little easier? Thanks for the warning but couldn't I just type in some code and see the results, please? Write a class? What do you mean "Write a class!"?

Imagine those sentences spinning really fast inside your caffeinated ADD brain at 3 AM and you'll start getting a picture.

Now I missed the wrist-friendly python syntax and the ability to easily test my ideas with running code. And I wanted more! I wanted a language I could extend with my own constructs. I wanted a compiler system that could be taught new things, taught how to automagically generate common code for me. I should be able to do that, right? We all should. We are programmers! We're entitled to that, it's all there in the big old book of programmers, page 987 if I recall it correctly... Well, more than anything else, I needed some old-fashioned quality sleep and time to put my head straight.

Being such a hard case of not-invented-here syndrome it all became clear to me: I had to build a new programming language system for the CLI, one that allowed programmers to take advantage of the wonderful .net framework features without getting in their way.

One that could be used, extended and modified by developers according to their specific needs.

I had to build boo.

Goals

A clean and wrist-friendly syntax

Python-like

I got hooked on python the time I wrote my first program with it and that had much to do with the language's syntax. There were some minor things I didn't like and I also had to account for the needs of a statically typed language such as exact method signature declarations and such.

Syntactic sugar for common programming patterns

List, hash and array literals were only some of typing saving things I knew boo should have. Common programming patterns such as object initialization, string formatting and regular expression matching should all be supported as well.

Automatic Variable Declaration

It can be good, it can be bad. But it always worked for me: assignments should be able to introduce new locally scoped variables.

Automatic type inference

Nothing more tiresome than writing the same type name over and over just to make the compiler happy. I'm not talking against static typing in general (boo is statically typed), I'm talking against being coerced by a tyrannical compiler to say things it should already know. Take for instance this simple little program:

```
def one():
	return 1

um = one()
```

The method **one** is clearly returning an integer value. The newly declared variable **um** is also clearly holding an integer value. The compiler should fill in all the blanks. On the other hand, I'm all for programmer's power, the programmer should be able to say what he/she means:

```
def one() as int:
	return 1

uno as int = one()
```

In that case the compiler should just check if the programmer had a good night of sleep and is making sense at all. In other words, a compiler error should be expected for code such as:

```
def one():
	return "1"
```

```
ichi as int = one()
```

Automatic type casting

If a type casting operation could succeed in runtime I don't want to write it. I'll rely on unit tests to make sure my code works.

Classes are not needed

The guys who came up with "**public static void main**" were probably kidding, the problem is that most people didn't get it was a joke. The infamous **HelloWorld** in all its boo glory:

```
print("Hello, world!")
```

"public static void main", that was a good one!

Expressiveness

First class functions

Granted I am no functional programming expert, far from that actually but I certainly can appreciate the expressiveness boost provided by functional composition.

Functions are first class passengers of the boo wagon and as such they can be used as return values:

```
def ignore(item):
	pass

def selectAction(item as int):
	return print if item % 2
	return ignore

for i in range(10):
	selectAction(i)(i)
```

Used as arguments:

```
def x2(item as int):
	return item*2

print(join(map(x2, range(11))))
```

Stored in variables:

```
p = print
p("Hello, world!")
```

Functions as objects:

```
print.Invoke("Hello, world!")
```

With all this power at hand the asynchronous delegate pattern couldn't be any easier:

```
import System

def callback(result as IAsyncResult):
	print("callback")

def run():
	print("executing")

print("started")
```

```
result = run.BeginInvoke(callback, null)
System.Threading.Thread.Sleep(50ms)
run.EndInvoke(result)

print("done")
```

It is also possible to describe function types exactly and explicitly through **callable type definitions** making it feasible to write gems such as:

```
callable Malkovich() as Malkovich

def malkovich() as Malkovich:
	print("Malkovich!")
	return malkovich

malkovich()()()
```

First class generators

From time to time there is the need to represent closed sets such as "the customers in São Paulo" or open sets like "the numbers in the Fibonacci series" in a suscint and memory conservative way. Enter **generators**.

Generators are language constructs capable of producing more than one value when used in a iteration context such as the **for in** loop.

Generator expressions

Generator expressions are defined through the pattern:

```
<expression> for <declarations> in <iterator> [if|unless <condition>]
```

Generator expressions can be used as return values:

```
def GetCompletedTasks():
	return t for t in _tasks if t.IsCompleted
```

Generator expressions can be stored in variables:

```
oddNumbers = i for i in range(10) if i % 2
```

Generator expressions can be used as arguments to functions:

```
print(join(i*2 for i in range(10) if 0 == i % 2))
```

In all cases the evaluation of each inner expression happens only on demand as the generator is consumed by a **for in** loop.

Generator methods (not yet available)

Generator methods are constructed with the **yield** keyword:

```
def fibonacci():
	a, b = 0, 1
	while true:
		yield b
		a, b = b, a+b
```

Given the definition above the following program would print the first five elements of the Fibonacci series:

```
for index as int, element in zip(range(5), fibonacci()):
	print("${index+1}: ${element}")
```

So although the generator definition itself is unbounded (a while true loop) only the necessary elements will be computed, five in this particular case as the **zip** builtin will stop asking for more when the range is exausted.

Generator methods are also a great way of encapsulating iteration logic:

```
def selectElements(element as XmlElement, tagName as string):
	for node as XmlNode in element.ChildNodes:
		if node isa XmlElement and tagName == node.Name:
			yield node
```

Duck Typing

Sometimes it is appropriate to give up the safety net provided by static typing. Maybe you just want to explore an API without worrying too much about method signatures or maybe you're creating code that talks to external components such as COM objects. Either way the choice should be yours not mine:

```
import System.Threading

def CreateInstance(progid):
	type = System.Type.GetTypeFromProgID(progid)
	return type()

ie as duck = CreateInstance("InternetExplorer.Application")
ie.Visible = true
ie.Navigate2("http://www.go-mono.com/monologue/")

Thread.Sleep(50ms) while ie.Busy

document = ie.Document
print("${document.title} is ${document.fileSize} bytes long.")
```

A **duck** typed expression will have all its method calls and property accesses resolved in runtime. So if it looks like a duck and it quacks like a duck...

Extensibility

Syntactic Attributes

Very few languages have a good way of automating micro code patterns, those little lines of code we programmers are generally required to write in order to expose fields as properties, properly validate method arguments, check pre/post conditions among other things. Enter **syntactic attributes**:

```
class Person:
	[getter(FirstName)]
	_fname as string

	[getter(LastName)]
	_lname as string

	def constructor([required] fname, [required] lname):
		_fname = fname
		_lname = lname
```

Look carefully at the code above, although **getter** and **required** look just like regular .net custom attributes they would be recognized by the compiler as **syntactic attributes** –

attribute classes that implement the **IAstAttribute** interface. The compiler will treat such attributes in a very special way, it will give them a chance to transform the in memory code representation (the abstract syntax tree, **ast** for short). After this process the code tree would look as if the programmer had actually typed:

```
class Person:

	_fname as string

	_lname as string

	def constructor(fname, lname):
		raise ArgumentNullException("fname") if fname is null
		raise ArgumentNullException("lname") if lname is null
		_fname = fname
		_lname = lname

	FirstName as string:
		get:
			return _fname

	LastName as string:
		get:
			return _lname
```

Anyone can write syntactic attributes just as anyone can write classes that implement interfaces. This opens up for entirely new ways of expression and code composition.

Syntactic Macros

Syntactic macros are pretty much like syntactic attributes in the sense that they are external objects invoked by the compiler to transform the code tree although they serve a different purpose: syntactic macros augment the language with new constructs. Take for instance the following code:

```
using reader=File.OpenText(fname):
	print(reader.ReadLine())
```

using is not a builtin language construct (like in C# for instance), it is a syntactic macro that causes the above code to be interpreted by the compiler as:

```
try:
	reader = File.OpenText(fname)
	print(reader.ReadLine())
ensure:
	if (__disposable__ = (reader as System.IDisposable))
		__disposable__.Dispose()
		__disposable__ = null
	reader = null
```

The good thing is that you don't really have to know that to take advantage of it, you just have to know that **using** makes sure that any listed resources are properly disposed of. And that's the great thing about macros, once you've captured a specific solution to a problem (deterministic disposing of resources) in a code pattern, you can give this code pattern a name (write a macro) and start using that name whenever you need the pattern.

Extensible Compilation Pipeline

An extensible syntax is only part of what I wanted. The compiler, the compilation process itself should be extensible. Programmers should be able to introduce new actions where appropriate to execute and automate a variety of tasks such as producing documentation

and reports, checking coding conventions, applying program transformations to better support debugging or specific execution environments just to cite a few. Programmers should also be able to reuse and/or replace specific compiler components such as the source code parser.

These requirements are realized in boo through an extensible compilation pipeline defined by a set of loosely coupled objects (the steps) communicating through a well defined data structure (the compilation context).

New pipelines and steps can be defined at will the limitating factors being only the programmer's understanding of the compiler inner workings and the feature set exposed by the public API.

The following code defines a new step which checks that every class defined in a program has its name starting with a capital letter, the code also defines an accompanying pipeline as an extension to the standard **CompileToFile** pipeline:

```
namespace StyleChecker

import Boo.Lang.Compiler
import Boo.Lang.Compiler.Ast
import Boo.Lang.Compiler.Steps
import Boo.Lang.Compiler.Pipelines

class StyleCheckerStep(AbstractVisitorCompilerStep):

   override def Run():
     Visit(CompileUnit)

   override def LeaveClassDefinition(node as ClassDefinition):
     if not System.Char.IsUpper(node.Name[0]):
       msg = "Class name '${node.Name}' should start with uppercase letter!"
       Errors.Add(CompilerError(node, msg))

class StyleCheckerPipeline(CompileToFile):

   def constructor():
     self.Insert(1, StyleCheckerStep())
```

The Common Language Infrastructure

A wrist friendly syntax, expressiveness and extensibility. That's not all.

I want my code to play nice with modules written in **other languages.**

I want a rich programming environment with a **well thought out class library**.

I want to be able to run my programs in **multiple platforms**.

I want the CLI.