Scripting com Boo

Criando DSLs internas mais expressivas



Fone: (21) 93174772

MSN: me@juanlopes.net

Login DevMedia: juanplopes@gmail.com

**Juan Lopes**

me@juanlopes.net

Arquiteto de soluções pela Living Consultoria. Iniciou a carreira desenvolvendo aplicações biométricas em C++ e Java. Programa principalmente em C# desde 2007, mas desenvolve também em Ruby e Python. É entusiasta da comunidade Microsoft e participa de projetos open source.

*Sugestão de Ilustração: fantasmas, roteiros, quebra-cabeças*

|  |
| --- |
| **De que se trata o artigo** |
| Este artigo trata sobre a criação de mecanísmos de scripting com Boo, uma linguagem criada pelo brasileiro Rodrigo de Oliveira (também conhecido como Rodrigo Bamboo). É uma linguagem que possui sintaxe e paradigma similares aos do Python, com o benefício de ser estaticamente tipada. |
| **Para que serve** |
| Por ser compilada, Boo tira proveito da maior parte das funcionalidades da Common Language Infrastructure o que a torna tão flexível quanto qualquer outra linguagem do .NET. Aliando-se à extensibilidade do compilador e à sintaxe elegante e sucinta vinda do Python, não é difícil compreender porque ela tem se tornado cada vez mais a linguagem preferida para se desenvolver DSLs de forma rápida e confiável. |
| **Em que situação o tema é útil** |
| Esse tema é especialmente útil quando é necessário alcançar uma expressividade maior ao desenvolver uma linguagem de representação para os sistemas. Boo, apesar de compilada, suporta a maior parte das facilidades de linguagens dinâmicas. Além disso, é uma linguagem que evita repetições desnecessárias, o que faz com que sua representação seja quase sempre direta e legível. |

**Scripting com Boo**

Em muitas situações é necessário desenvolver mecanismos de expressão de certos comportamentos do sistema. Existem formas canônicas de realizar esse tipo de tarefa, como declarativamente por XML ou via banco de dados, mas muitas vezes é preciso alcançar uma expressividade maior. Para esse propósito desenvolvem-se DSLs. Este artigo irá tratar de uma das formas de fazê-lo utilizando a linguagem Boo. Este artigo irá tratar sobre como utilizar Boo para realizar tarefas de scripting para expressividade de problemas de domínio, mostrando alguns exemplos práticos de uso da linguagem.

O .NET é uma plataforma que roda sobre uma máquina virtual. Esta máquina virtual tem o nome de Common Language Runtime permite que as diversas linguagens que compilam em para ela possam ser executadas no mesmo ambiente e se comuniquem através de protocolos comuns. Isso estimula que várias linguagens sejam criadas para essa plataforma.

Boo é uma dessas linguagens. Ela existe desde 2003. Apesar de não ser tão nova, Boo ganhou destaque na comunidade recentemente após Oren Eini (também conhecido como Ayende Rahien), um dos maiores contribuidores do Nhibernate, escrever um livro sobre DSLs em Boo.

A estrutura de um código em Boo lembra muito um código em Python. As duas linguagens tem suporte a diversas funcionalidades parecidas, como duck typing, inferência automática de tipos, generators, funções de primeira classe entre outras coisas que permitem uma maior expressividade com a linguagem. Tudo isso sendo uma linguagem estaticamente tipada.

**Nota do Devman**

Duck typing é uma forma de tipagem que verifica a existência de membros num objeto somente no momento em que eles são referenciados em tempo de execução. É uma característica de tipagem dinâmica. Esse nome é oriúndo da famosa frase de James Riley, que é traduzida livremente como “se eu vejo uma ave que anda como um pato, nada como um pato e faz barulho de pato, eu chamo essa ave de pato”. Fazendo alusão de que se um objeto responde às mesmas mensagens de outro, eles podem ser considerados equivalentes em um determinado contexto, mesmo que seus tipos sejam diferentes.

Além disso, Boo tem uma capacidade que a diferencia da maior parte das linguagens. Seu compilador é feito para ser extensível e facilmente embutido em aplicações, o que a torna perfeita para utilização em mecanismos de extensibilidade por scripting.

Scripting

Cada projeto de software precisa permitir níveis de configuração diferentes. Para certas aplicações, um XML de configuração é suficiente. Outras armazenam complexos parâmetros de configuração em banco de dados. Existem aquelas que permitem tal nível de configuração que é possível modificar completamente o comportamento usando plug-ins.

Uma das alternativas de extensibilidade mais usadas é o scripting. A possibilidade do usuário escrever código que realize tarefas na aplicação. Em geral, é uma técnica aplicada a usuários mais avançados, mas com um pouco de cuidado, é possível desenvolver uma linguagem limpa e expressiva para facilitar a vida de quem tenta customizar a aplicação. Mas essa não é uma tarefa fácil. Quando se fala de scripting, geralmente existem alguns desafios.

O primeiro desafio é a escolha da linguagem a ser usada como base. É possível criar uma linguagem específica para esse propósito ou utilizar uma já existente com as devidas modificações. A grande vantagem de criar sua própria linguagem é a flexibilidade que se ganha, pois é possível criar qualquer estrutura que se deseje. Mas é um trabalho razoavelmente grande, e geralmente não vale o esforço. Usar uma linguagem já existente como base, somente customizando-a para seu próprio fim é uma tarefa bem mais fácil, mas dependendo do nível de customização, pode se tornar um trabalho maior que escrever a linguagem do zero.

O segundo desafio é no campo da performance. Dependendo de como os scripts forem executados, o desempenho pode ser bastante insatisfatório, exigindo níveis de otimização que geralmente não são praticáveis. As escolhas mais naturais para linguagens de script são linguagens interpretadas, que trazem grande flexibidade atrelada ao seu dinamismo. Entretanto, essa flexibilidade tem um custo. Linguagens interpretadas tem naturalmente um desempenho pior que linguagens compiladas.

Já o terceiro desafio é o da legibilidade dos scripts. Se o usuário final vai precisar ser capaz de modificar esses scripts, é preciso que ele saiba entender o que um certo script faz. As linguagens dinâmicas (como Ruby e Python) geralmente exigem muito menos código que linguagens estáticas (como C# e VB).

Tendo esses problemas em mente, a escolha de Boo torna-se mais natural, pois:

* utilizar uma linguagem que compila em CIL permite que ela se comunique com objetos já existentes e utilize estruturas da própria aplicação sem grandes customizações;
* a performance de uma linguagem compilada é quase sempre melhor que a das interpretadas;
* Boo herda o estilo de Python, uma linguagem que preza pela legibilidade e simplicidade;
* a linguagem permite que suas estruturas sejam customizadas através de extensão do compilador.

Estrutura básica do código em Boo

O código em Boo se parece em muito com um código em Python. A linguagem em si tem o objetivo de ser “amigável ao pulso”, numa referência a menor quantidade de caracteres que precisam ser digitados nos programas em Boo.

Um exemplo é que Boo não precisa de uma classe “Program”. Nem de um “public static int Main”. Tal qual no Python, qualquer instrução que não estiver dentro de uma classe será executada no começo do programa. Então, o famoso “Hello World” acaba sendo bastante curto.

print "Hello World"

Para quem não está acostumado com Python, o que chama mais atenção à primeira vista num script em Boo é a inexistência de delimitador de fim de bloco. Ou seja, nada de “End If” ou “}”. Toda a demarcação de bloco é feita através de indentação. Isto é, para definir que um grupo de instruções está no mesmo bloco, basta dar a mesma indentação a elas.

if 2+2==4:

print "Um bloco"

else:

print "Outro bloco"

Boo é uma linguagem estaticamente tipada que confia muito na inferência automática de tipos. Isto é, ao escrever um código como o abaixo, o compilador percebe que o retorno da função dois\_mais\_dois é um inteiro, mesmo que isso não tenha sido declarado em momento algum. E ao tentar invocar o método Substring, um erro de compilação é lançado, informando que Substring não é um método que exista em “int”.

def dois\_mais\_dois:

return 2+2

quatro = dois\_mais\_dois()

print quatro.Substring(2)

Além disso, muitas outras construções são semelhantes às respectivas em Python, como, por exemplo, os generators. No código abaixo, é gerada uma enumeração com todos os pares entre 0 e 1000. Entretanto, ela somente será avaliada quando for necessário. O mais tarde possível. No caso, ao exibir, o código obtem somente os cinco primeiros itens dessa enumeração. É importante perceber também o uso do método Take, do Linq to Objects.

pares = i for i in range(1000) if i % 2 == 0

print join(pares.Take(5))

**Nota do Devman**

Generators são estruturas presentes na linguagem Python (e em algumas outras também) que permitem criar enumerações avaliadas tardiamente. É possível criar um generator utilizando uma sintaxe com expressões “for .. in ... if/unless” ou utilizando generator methods, que possui sintaxe e funcionamento bem parecidos com os iterators do C#.

Um dos objetivos principais da linguagem Boo é permitir uma expressividade maior. É escrever menos código repetitivo sem precisar recorrer a trechos ilegíveis de código. Por exemplo, os códigos nas listagens 1 e 2 representam a mesma coisa, porém o primeiro é escrito em C# e o segundo em Boo.

Listagem 1. Classe Cliente em C#

public class Customer

{

string \_fname;

public string FirstName { get { return \_fname; } }

string \_lname;

public string LastName { get { return \_lname; } }

public Customer(string fname, string lname)

{

if (fname == null) throw new ArgumentNullException("fname");

if (lname == null) throw new ArgumentNullException("lname");

\_fname = fname;

\_lname = lname;

}

}

Listagem 2. Classe Customer em Boo

public class Customer:

[getter(FirstName)]

\_fname as string

[getter(LastName)]

\_lname as string

public def constructor([required]fname, [required]lname):

\_fname = fname;

\_lname = lname;

Parece interpretada, mas é compilada

Boo, por ser inspirada em Python, trás muitas das características de uma linguagem dinâmica, apesar de ser estaticamente tipada. Uma dessas características é a duck typing, que só veio aparecer no C# a partir da versão 4.0, conhecida também como dynamic. Em Boo, é possível escrever um código como o abaixo:

duck\_obj as duck = obj

duck\_obj.some\_method()

No exemplo acima, a chamada a some\_method somente será avaliada em tempo de execução, o que permite características bem mais dinâmicas ao programa.

Além disso, Boo permite que o desenvolvedor intercepte chamadas aos métodos e propriedades feitas através de duck typing. Isso é possível através da interface IQuackFu. A Listagem 3 exibe um código de exemplo do uso da interface, o resultado pode ser conferido na Figura 1.

Listagem 3: Exemplo IQuackFu

class QuackExample(IQuackFu):

def QuackGet(name as string, args as (object)):

print "Tentando obter valor de ${name}"

def QuackSet(name as string, args as (object), value):

print "Tentando atribuir ${value} a ${name}"

def QuackInvoke(name as string, args as (object)):

print "${name} foi chamado"

obj = QuackExample()

obj.metodo\_exemplo()

obj.propriedade\_exemplo = 2

print obj.propriedade\_exemplo

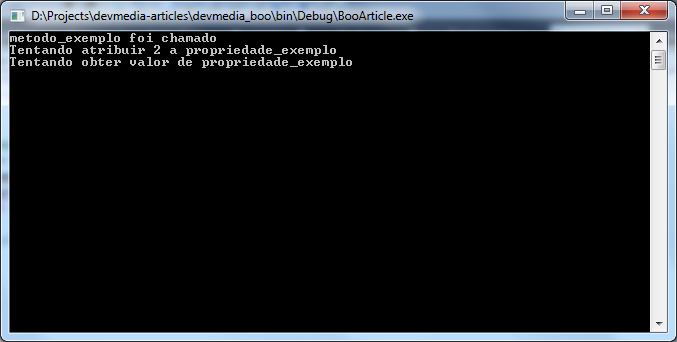


Figura 1. Resultado do uso da interface IQuackFu

Uma linguagem integrável

Um programa em Boo pode ser executado de várias formas: através do compilador da linguagem (“booc”) e executado como uma assembly .NET normal; através do interpretador de Boo (“booi”); interpretado internamente usando a classe InteractiveInterpreter ou compilado internamente através da classe BooCompiler.

Para scripting, a melhor opção é o BooCompiler. Com ele, é possível compilar o código tanto para a memória quanto para um arquivo.

A possibilidade de compilar o código para a memória torna mais fácil obter métodos e tipos em tempo de execução, a partir de simples scripts. Então, apesar de compilada, Boo torna-se uma linguagem simples de ser integrada. Um exemplo simples de uso do compilador para compilar um arquivo para a memória pode ser visto a seguir.

boo = BooCompiler()

boo.Parameters.Input.Add(FileInput("arquivo.boo"))

boo.Parameters.Pipeline = CompileToMemory()

assembly = boo.Run().GeneratedAssembly

**Exemplo: Criando um Formulário usando scripts**

Partindo para um exemplo mais claro, é possível exemplificar a usabilidade de Boo para mecanismos scriptáveis em aplicações. Um dos exemplos mais clássicos talvez seja a construção de um formulário em tempo de execução, usando um script compilado em Boo.

O objetivo é que, dado um script como o da Figura 2, seja possível construir um formulário que obedeça aos campos e a ordem definida.

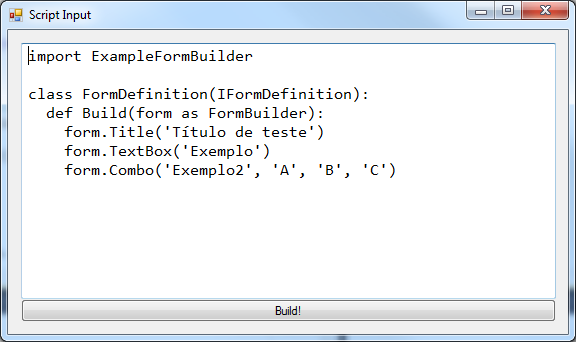


Figura 2. Formulário de entrada do script

É preciso definir, a princípio, o contrato que deverá ser seguido pela classe definida no script. Surge então a necessidade de uma interface IFormDefinition.

public interface IFormDefinition

{

void Build(FormBuilder form);

}

Esta interface define que o script receberá um ponto de entrada na API de construção do formulário. Esse ponto de entrada é a classe FormBuilder, que define métodos especialistas na criação de controles no formulário. Ela é definida como na Listagem 4. Em sua construção, já são criadas instâncias das classes Form e TableLayoutPanel, para preparar a chegada de novos controles. Os métodos de inserção de controles nesta classe são bastante rudimentares, servindo apenas para o propósito de exemplificar uma API mais complexa. Somente os métodos para adicionar text boxes e combo boxes foram criados.

Listagem 4. Classe FormBuilder

public class FormBuilder

{

Form form = new Form();

TableLayoutPanel panel = new TableLayoutPanel()

{

ColumnCount = 1,

Dock = DockStyle.Fill

};

public FormBuilder()

{

form.Controls.Add(panel);

}

public void Title(string title)

{

form.Text = title;

}

public void TextBox(string label)

{

AddLabel(label);

Add(new TextBox());

}

public void Combo(string label, params string[] options)

{

AddLabel(label);

Add(new ComboBox() { DataSource = options });

}

private void AddLabel(string label)

{

Add(new Label()

{

Text = label,

TextAlign = System.Drawing.ContentAlignment.MiddleLeft

});

}

private void Add(Control control)

{

control.Dock = DockStyle.Top;

panel.Controls.Add(control);

}

public void Show()

{

form.Show();

}

}

Juntas, esta classe e esta interface definem a API que será disponibilizada para o script em tempo de execução. É preciso, então definir como o script será compilado. Para esta tarefa, é possível utilizar a classe na Listagem 5.

Esta classe é capaz de receber uma string (o script) e retornar uma instância que implemente IFormDefinition. Para isso, ela instancia o compilador de Boo, a classe BooCompiler, no assembly Boo.Lang.Compiler, utilizando a pipeline CompileToMemory, que significa que todo o processo de compilação será executado, gerando um assembly executável em memória, sem precisar salvá-lo em disco. Além disso, adiciona-se uma referência para o projeto atual, para que o script tenha acesso às classes da API.

No método Compile, basta definir a origem do script de entrada e executar o compilador. Caso haja algum erro de compilação, o método lança a primeira exceção resultante para avisar o formulário de que a compilação falhou. Caso não tenha falhado, basta obter a classe FormDefinition no assembly gerado e criar uma instância dela, fazendo um cast para IFormDefinition.

Listagem 5. Compilador para definições de formulário

public class DefinitionCompiler

{

static BooCompiler boo = GetCompiler();

private static BooCompiler GetCompiler()

{

var boo = new BooCompiler();

boo.Parameters.Pipeline = new CompileToMemory();

boo.Parameters.References.Add(typeof(IFormDefinition).Assembly);

return boo;

}

public IFormDefinition Compile(string script)

{

boo.Parameters.Input.Clear();

boo.Parameters.Input.Add(new StringInput("noname", script));

var compiler = boo.Run();

if (compiler.Errors.Count > 0)

throw compiler.Errors[0];

var assembly = compiler.GeneratedAssembly;

var type = assembly.GetType("FormDefinition");

return (IFormDefinition)Activator.CreateInstance(type);

}

}

Caso todos esses passos sejam executados com sucesso, o resultado é um objeto .NET que corresponde ao script definido como entrada e que implementa a interface IFormDefinition.

Assim, resta apenas instanciar a classe DefinitionCompiler e compilar a definição oriunda da text box no form de entrada. A Listagem 6 exemplifica o uso, que resulta num formulário como o da Figura 3.

Listagem 6. Uso da classe DefinitionCompiler

private void BuildCommand\_Click(object sender, EventArgs e)

{

try

{

var compiler = new DefinitionCompiler();

var definition = compiler.Compile(ScriptInput.Text);

var form = new FormBuilder();

definition.Build(form);

form.Show();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Error",

MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

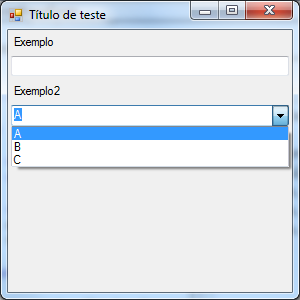


Figura 3. Formulário resultante

Uma linguagem extensível

Boo é uma linguagem feita pensando em extensibilidade. Seu compilador tem muitos pontos de inserção, permitindo que a maior parte do comportamento padrão da linguagem seja modificado. É possível escrever novas macros, atributos ou até mesmo mudar a forma como algumas estruturas são entendidas.

As três principais formas de mudar a linguagem são através de Syntatic Macros, Attributes e, de uma forma mais avançada, através da criação e extensão da pipeline de compilação. As duas primeiras formas são mais fáceis de lidar, pois permitem facilmente estender a linguagem entender exatamente como o compilador funciona.

Um exemplo é o bloco “using”, que no C# é um elemento de linguagem, mas em Boo é apenas uma macro que é expandida em tempo de compilação em “try .. except .. ensure” (try .. catch .. finally, em C#).

Criando uma nova macro

Criar uma macro em Boo é uma tarefa simples. Basta definir uma classe que herde de IAstMacro. O compilador irá automaticamente buscar por convenção esta classe e chamá-la no momento da compilação. Basta que a classe que define a macro esteja no namespace padrão de algum assembly referenciado pelo assembly a ser compilado. Macros podem ser escritas em qualquer linguagem .NET.

As macros tem uma estrutura bem definida. Podem receber parâmetros e blocos. Um exemplo de macro simples seria simplesmente passar exatamente um parâmetro e um bloco. Considere o seguinte código:

bloco "teste":

print "estou dentro do bloco!"

Ao tentar compilar este trecho de código, o compilador avisa que o identificador “bloco” não existe, e que não pode compilar o programa.

Para implementar esta macro, deve-se escrever uma classe como a da Listagem 7. O método Expand desta classe recebe um MacroStatement como parâmetro, que denota a forma como a macro foi chamada no código. Perceba que essa classe é chamada pelo compilador de Boo, logo, a forma de lidar com as estruturas do código original são um pouco peculiares. É preciso navegar pela AST para realizar modificações. Nesta macro, especificamente, é obtido o método WriteLine, da classe Console. Depois, gera-se código para invocar este método com mensagens específicas, que são logo inseridas no bloco, uma no começo e outra no final.

Listagem 7. Macro “bloco”

public class BlocoMacro : AbstractAstMacro

{

public override Statement Expand(MacroStatement macro)

{

var method = typeof(Console).GetMethod("WriteLine",

new[] { typeof(string), typeof(object) });

var call1 = CodeBuilder.CreateMethodInvocation(method,

CodeBuilder.CreateStringLiteral(">>> entrando no bloco {0}"), macro.Arguments[0]);

var call2 = CodeBuilder.CreateMethodInvocation(method,

CodeBuilder.CreateStringLiteral(">>> saindo do bloco {0}"), macro.Arguments[0]);

macro.Body.Insert(0, call1);

macro.Body.Insert(macro.Body.Statements.Count, call2);

return macro.Body;

}

}

**Nota do Devman**

AST é Abstract Syntax Tree, uma estrutura definida pelos compiladores para representar o código-fonte na forma de uma árvore. Esta abordagem simplifica muito a interpretação e tradução do código-fonte em código executável. A AST é gerada após a análise sintática do código-fonte, durante o primeiro passo da compilação e é usada durante todo o processo para denotar as estruturas definidas no código.

O resultado da execução do programa original pode ser visto na Figura 4.



Figura 4. Executando o programa com a macro “bloco”

É um resultado interessante de um recurso poderoso, entretanto, o código necessário em C# para fazê-lo funciona não é o mais elegante possível. Porém, existe uma forma melhor de fazê-lo.

O compilador escrever a mesma macro de forma muito mais elegante utilizando somente código em Boo. A linguagem Boo disponibiliza uma macro específica para criar novas macros. A Listagem 8 exemplifica seu uso. Este formato mais expressivo é possível graças à capacidade de Boo de transformar qualquer trecho de código em sua AST correspondente. Algo similar ao que o C# faz com o tipo Expression<TDelegate>, mas expandido para todas as estruturas da linguagem.

Listagem 8. Macro “bloco” escrita em Boo.

macro bloco(nome as string):

return [|

print "entrando no bloco ${$nome}"

$(bloco.Body)

print "saindo do bloco ${$nome}"

|]

Criando um novo atributo

Analogamente às macros, os atributos podem ser criados apenas implementando a interface IAstAttribute. Atributos, diferentemente de macros, não possuem o conceito de expansão. Atributos são simplesmente visitados e, nesse momento, é que as modificações podem ser feitas. Considere o código a seguir para fins de exemplo:

def teste\_atributo([exibir]a, [exibir]b):

print "abc"

teste\_atributo(123, null)

Espera-se que ao chamar o método teste\_atributo, os valores de a e b sejam exibidos na tela. Para tanto, é possível criar uma classe ExibirAttribute (Listagem 9) que realiza esse trabalho em tempo de compilação. Esta classe apenas obtem o nome do parâmetro, cria uma referência para o mesmo em código (usando uma ReferenceExpression). Depois, é criado o código que irá exibir o valor desse atributo em tela e inserido no começo do método que contém a declaração do parâmetro. O resultado da execução pode ser visto na Figura 5.

Listagem 9. Classe para o atributo “exibir”

class ExibirAttribute(AbstractAstAttribute):

override def Apply(node as Node):

parameterName = (node as ParameterDeclaration).Name

parameter = ReferenceExpression(parameterName)

exibir = [| print "${$parameterName} = ${$parameter}" |]

(node.ParentNode as Method).Body.Insert(0, exibir)

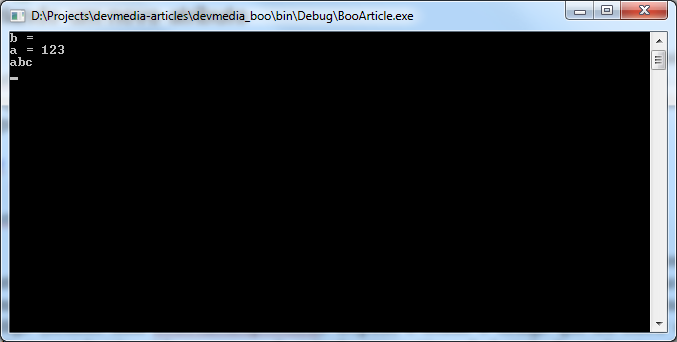


Figura 5. Executando o programa com o atributo “exibir”.

Entendendo a pipeline de compilação em Boo

O processo de compilação de um programa em Boo não é muito diferente do que acontece em outras linguagens. Temos várias fases distintas, começando pela análise léxica e sintática, passando pela análise semântica e culminando na geração de código.

A diferença é que todo o processo de compilação é extensível. Internamente, a compilação é organizada como uma pipeline de processamento, onde vários passos são executados seqüencialmente sobre o contexto de compilação, e vão gerando artefatos que alimentam os próximos passos, até chegar à geração de código executável em si. As pipelines e os passos que elas executam são completamente customizáveis.

Normalmente, o primeiro passo da compilação é análise léxica e sintática (Parse), que é responsável pela transformação do texto em Boo em uma estrutura de dados útil para os outros passos da compilação. Os últimos passos normalmente envolvem a geração de código executável e a posterior persistência do assembly gerado em disco. A pipeline descrita é a mais comum e chama-se CompileToFile. Os passos desta pipeline podem ser agregados de forma genérica em 6 categorias, como demonstra a Figura 6.

Parse

ExpandMacros

ResolveExpressions

Compile

EmitAssembly

Script.boo

SaveToFile

Figura 6. Pipeline CompileToFile

Cada uma dessas categorias pode conter vários passos. A categoria “Compile”, por exemplo, é composta de 24 passos, entre os quais constam ExpandDuckTypedExtensions e RemoveDeadCode. Qualquer um desses passos pode ser estendido ou até mesmo completamente reescrito, caso seja necessário.

Uma utilização prática da extensão do compilador será vista no próximo exemplo.

Exemplo: Compilador de Funções Matemáticas

Este exemplo irá tratar de uma aplicação que receberá um script que define uma função matemática com incógnitas e irá perguntar ao usuário os valores a serem inseridos para resolver a expressão.

Para tanto, é necessário saber, no script quais são as variáveis referenciadas para pedir ao usuário os valores corretos. Por exemplo:

(-b + Math.Sqrt(b\*\*2 - 4 \* a \* c)) / 2

Na expressão acima, o programa deve ser capaz de reconhecer que as variáveis “a”, “b” e “c” precisam de valores para a expressão poder ser calculada. Para isso, será necessário estender um dos steps de compilação. No caso, o ProcessMethodBodiesWithDuckTyping.

O objetivo é que toda vez que uma referência não resolvida ocorrer dentro do corpo da função de avaliação, o compilador substitua por um valor da entrada, que será definida por um dicionário. O script terá o formato que pode ser observado na Figura 7:

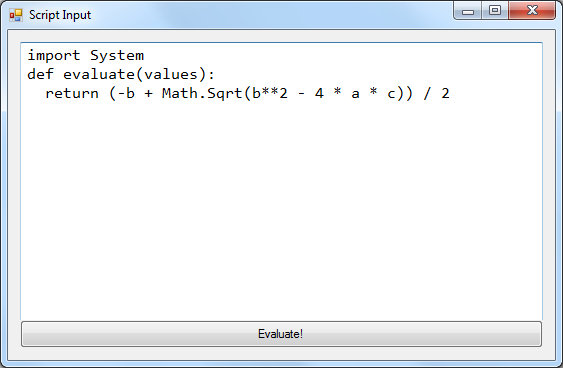


Figura 7. Exemplo de função para avaliação

O que se espera do compilador neste caso, é que ao encontrar uma variável, por exemplo “a”, ele a substitua por uma referência ao dicionário values, com values[“a”]. O código que realiza esta tarefa pode ser visto na Listagem 10.

Listagem 10. Passo de compilação personalizado

public class CustomDuckTypeExpanderStep : ProcessMethodBodiesWithDuckTyping

{

public IEnumerable<string> Variables { get { return variables; } }

HashSet<string> variables = new HashSet<string>();

public override void OnModule(Boo.Lang.Compiler.Ast.Module module)

{

variables.Clear();

base.OnModule(module);

}

public override void OnReferenceExpression(ReferenceExpression node)

{

var entity = this.NameResolutionService.Resolve(node.Name);

if (entity != null)

{

base.OnReferenceExpression(node);

}

else

{

var parameter = \_currentMethod.GetParameters()[0];

var dictionary = TypeSystemServices.Map(typeof(IDictionary<string, object>));

var getter = NameResolutionService.ResolveMethod(dictionary, "get\_Item");

var propCall = CodeBuilder.CreateMethodInvocation(

CodeBuilder.CreateReference(parameter),

getter,

CodeBuilder.CreateStringLiteral(node.Name));

variables.Add(node.Name);

BindExpressionType(propCall, TypeSystemServices.DuckType);

node.ParentNode.Replace(node, propCall);

}

}

}

Esta classe, apesar de parecer complexa, é bastante simples. Ela mantém uma coleção com todas as variáveis que ela já processou. Então, em toda referência que ela encontra no código, inicialmente tenta-se resolver usando o método tradicional. Caso não seja possível, a classe obtém o primeiro parâmetro da função (que deve ser um dicionário), e cria código para chamar nele o método get\_Item, que é equivalente à indexação de um dicionário. Então a variável referenciada é adicionada a lista e a expressão criada substitui a anterior. No final do processamento, o campo variables da classe será uma coleção de todas as variáveis não resolvidas do script.

Para utilizar este passo de compilação, é necessário substituir o original no compilador. Será criada uma classe para encapsular a complexidade da compilação, tal qual foi feito no primeiro exemplo. Neste caso, substituir-se-á o passo ProcessMethodBodiesWithDuckTyping pelo recém-criado CustomDuckTypeExpanderStep. Isso é feito na classe ExpressionCompiler (Listagem 11).

Listagem 11. Classe compiladora de expressões

public class ExpressionCompiler

{

public IEnumerable<string> LastExpressionParams

{

get { return step.Variables; }

}

BooCompiler boo = new BooCompiler();

CustomDuckTypeExpanderStep step = new CustomDuckTypeExpanderStep();

public ExpressionCompiler()

{

boo.Parameters.Pipeline = new CompileToMemory().Replace(

typeof(ProcessMethodBodiesWithDuckTyping), step);

}

public Func<IDictionary<string, object>, object> Compile(string script)

{

boo.Parameters.Input.Clear();

boo.Parameters.Input.Add(new StringInput("Script", script));

var compiler = boo.Run();

if (compiler.Errors.Count > 0)

throw compiler.Errors[0];

var assembly = compiler.GeneratedAssembly;

var method = assembly.GetType("ScriptModule").GetMethod("evaluate");

return dictionary => method.Invoke(null, new[] { dictionary });

}

}

As maiores diferenças entre esta classe e a anterior, descrita na Listagem 5, são que nesta, a pipeline de compilação é customizada e o método Compile retorna um delegate, em vez de uma instância de uma classe. Além disso, no final da compilação, como visto anteriormente, o valor de step.Variables será igual à lista de todas as variáveis não resolvidas no script.

A utilização desta classe também é bastante comum. Porém, o programa deverá exibir uma caixa de diálogo pedindo o valor das variáveis referenciadas (Figura 8). O código para realizar a compilação pode ser visto na Listagem 12.

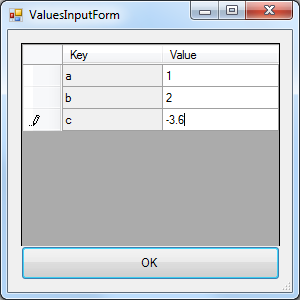


Figura 8. Diálogo de entrada para valores de variável

Listagem 12. Processo de compilação da expressão

private void EvaluateCommand\_Click(object sender, EventArgs e)

{

try

{

var compiler = new ExpressionCompiler();

var function = compiler.Compile(ScriptInput.Text);

var input = compiler.LastExpressionParams.ToArray();

var form = new ValuesInputForm(input);

if (form.ShowDialog() == System.Windows.Forms.DialogResult.OK)

{

var result = function(form.Values);

MessageBox.Show(string.Format("The result is: {0}", result), "Result");

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Error",

MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

Assim, foi possível demonstrar como modificar o comportamento do compilador sem precisar recompilá-lo inteiro, apenas utilizando os pontos de extensão fornecidos pela própria linguagem.

Conclusão

A linguagem Boo vem demonstrando ser bastante versátil para uso em DSLs e até para o dia-a-dia. Apesar de não ser nova, ainda está engatinhando no mundo .NET, mas ultimamente ela vem ganhando cada vez mais espaço na preferência dos desenvolvedores. Hoje em dia é disputa forte com linguagens como IronRuby e IronPython no ramo scripting. Espera-se que no futuro, com apoio da comunidade e – quiçá – da Microsoft, Boo aproxime-se do rol de linguagens consagradas da plataforma .NET.

**Links**

Página da linguagem

<http://boo.codehaus.org/>

Categoria de artigos sobre Boo no blog do Ayende Rahien

<http://ayende.com/Blog/category/498.aspx>

Link da Amazon para o livro DSLs in Boo

<http://www.amazon.com/DSLs-Boo-Domain-Specific-Languages/dp/1933988606>

Tutorial Boo Primer

<http://communitymapbuilder.org/pages/viewpage.action?pageId=37746&navigatingVersions=true>

Twitter do autor

<http://twitter.com/juanplopes>

**Saiba Mais**

Intermediate Language - IL - .Net Magazine 81

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=18960>

Reflection - Artigo .net Magazine 84

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=20409>

Introdução ao desenvolvimento Web com Python

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=6552>

Como o framework .NET reconhece várias linguagens de programação?

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=14830>

Desmitificando delegates - Artigo .net Magazine 84

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=20410>