A Linguagem Ruby

6 de Novembro de 2002

Alan Baronio Menegotto
UNISINOS - Universidade do Vale do Rio dos Sinos
Centro de Ciências Exatas e Técnológicas
São Leopoldo - RS - Brasil
menegotto@terra.com.br
e
Fábio Mierlo
UNISINOS - Universidade do Vale do Rio dos Sinos
Centro de Ciências Exatas e Técnológicas
São Leopoldo - RS - Brasil
fmierlo@yahoo.com

Resumo

Este artigo trata de um resumo das características principais da linguagem Ruby, desde a sua arquitetura até a sua sintaxe. Abordando principalmente temas relativos ao contexto de Linguagens de Programação é possível avaliar uma linguagem de forma correta. Este artigo tenta fazer isto de uma forma neutra, avaliando cada ponto da linguagem Ruby, como seus tipos de dados, forma como variáveis e subprogramas são implementados, etc..., para que o leitor do artigo possa conhecer a linguagem e avaliar o uso da mesma para resolução de algum determinado problema.

Keywords: Linguagem Ruby, Ruby, Linguagens de Scripts

1 INTRODUÇÃO

Ruby é uma linguagem de programação interpretada e orientada à objetos pura, implementada por Yukihiro Matsumoto (matz@netlab.jp). Yukihiro é um programador que trabalha para uma empresa japonesa voltada ao open source, a NetLab. Yukihiro é um dos grandes nomes do software livre no Japão.

A linguagem possui muitas características peculiares, descritas nas próximas seções. A linguagem possui recursos nativos para programação com os mais diversos tipos de aplicativos (processamento de arquivos texto, aplicações multithreaded e aplicações que utilizam sockets são alguns exemplos).

É uma linguagem relativamente nova (atualmente está na versão 1.67), tendo sido implementada originalmente para ambientes *nix, sendo portada posteriormente também para ambientes Windows através do uso de cygwin.

A linguagem Ruby é uma linguagem free, mesmo para uso comercial. A existência de muitas bibliotecas torna a programação simples. Além disso, existe uma compatibilidade entre todas as versões da linguagem, desde o seu início.

Existem interfaces para Python, Perl e Java, ou seja, programas existentes em outras linguagens podem ser reutilizados através de static binding.

Este artigo tem a intenção de demonstrar as principais características da arquitetura da linguagem e de mostrar um overview sobre a sintaxe do Ruby. Na última seção, alguns códigos fontes estão disponíveis para que o leitor deste artigo saiba com o que se parece a sintaxe do Ruby.

Para maiores informações sobre a linguagem, recomendo o site da linguagem, www.ruby-lang.org. Tem muita informação lá, tanto para quem quer começar a desenvolver como os que já possuem um conhecimento aprofundado da linguagem e desejam ajudar no projeto de desenvolvimento da linguagem.

2 CARACTERÍSTICAS DA LINGUAGEM

2.1 Estrutura Geral

- Ruby é uma linguagem interpretada. Sendo assim, não é necessário recompilar os programas para executá-los novamente, basta executá-lo através do interpretador. Isto pode ser bom do ponto de vista da facilidade do programador na hora da implementação e fase de testes, mas diminui a performance de um programa complexo, devido à interpretação do código fonte cada vez que alguma instrução é chamada.

- Como o Ruby é uma linguagem orientada à objetos pura, seu criador decidiu trabalhar somente com objetos, ou seja, tudo na linguagem é tratado como objetos. Inteiros, strings, estruturas de decisão, blocos de controle, loops, tudo é feito através de mensagens entre objetos, assim como na linguagem Smalltalk.

- A sintaxe do Ruby é fortemente influenciada pela sintaxe da linguagem Eiffel. Do ponto de vista do aprendizado, isto facilita muito a vida do programador, pois como a sintaxe é muito simples você escreve programas mais legíveis de uma forma mais elegante e fica fácil de outro programador fazer manutenção de código posteriormente.

- Expressões regulares podem ser utilizadas para o processamento de textos, exatamente como é feito na linguagem Perl. Grandes facilidades neste sentido tornam a vida do programador que não se adaptou às bizarras expressões regulares em Perl trabalhar com elas facilmente utilizando Ruby.

- Ruby é altamente portável, isto significa que um programa Ruby executa sem nenhuma mudança em qualquer plataforma. É claro que isto somente é verdade se você não estiver utilizando modulos específicos de uma plataforma, como algumas GUIs para UNIX ou WinGKR (Win32 GUI Kit for Ruby). Isto traz como vantagens um menor custo, porque apenas um programa deve ser gerenciado e uma maior distribuição do programa, além dele poder ser executado em várias plataformas.

2.2 Tipos de Dados

- Números: Ruby divide os números em dois tipos: fixnum (inteiros) e bignum(float). Os fixnum podem ser números na faixa de 2^{-62} até 2^{+62}. Qualquer número fora desta faixa é atribuído à classe bignum. Devido à grande faixa de números atribuídos à classe bignum, sua implementação é feita através de uma seqüência de bits dinâmica, permitindo que cálculos com números extremamente grandes possam ser feitos, como por exemplo fatorial(400) (desde que haja memória suficiente para armazenar o número). Para armazenar um número, são gastos 2 longs + 1 int bytes.

- Strings: Strings são seqüências de bytes armazenados dinamicamente sem um limite fixo pré-determinado (somente o limite físico de memória). São gastos 4 long + 1 pointer + tam da string * 1 char para armazenar uma string.

- Arrays: Arrays, em Ruby, são conjuntos de objetos (não necessariamente do mesmo tipo) que podem ser acessados através de um índice inteiro, conforme a inserção no array. Devido à heterogeneidade dos arrays, não existem "structs" em Ruby, somente arrays que podem conter elementos de qualquer tipo. Na seção 3.2 existem maiores explicações sobre arrays em Ruby. São gastos 4 long + 1 pointer + tam do array * 1 pointer para armazenar um array.

- Hashes: Ruby traz uma implementação nativa de tabelas hash, assim como em Perl ou Java. Você pode construir tabelas hash para acelerar o acesso aos seus dados, através da especificação de valores/chaves para os seus dados. Existem algumas restrições que a linguagem faz na construção de tabelas hash. A primeira é que após construída a tabela os valores das chaves não podem ser alterados. Outra restrição é que as chaves sejam todas do mesmo tipo.

- Ranges: A linguagem permite que trabalhemos com faixas de valores como se estivéssemos trabalhando com um objeto qualquer. Isto permite uma grande flexibilidade na hora de codificação. Faixas de valores são, por exemplo, a..z, A..Z, etc...

2.3 Orientação à Objetos

- Características básicas de linguagem orientadas à objetos como o conceito de herança, classes, métodos, polimorfismo e encapsulamento são implementadas em sua totalidade na linguagem Ruby.
- Somente herenças simples podem ser feitas na linguagem Ruby (apesar disto, existe um conceito de compartilhamento de métodos entre classes, permitindo desta maneira uma herença múltipla forçada).
- O encapsulamento é feito através de três construções básicas: private, protected e public. Um método private somente pode ser chamado na sua classe de definição. Um método protected pode ser chamado somente pela sua classe de definição e suas subclasses. Um método public é acessado por qualquer método. Algo que deve ser mencionado aqui é que estas verificações são feitas dinamicamente em tempo de execução. Isto leva programadores menos experientes à run-time errors mais freqüentemente.
- Outro recurso interessante é a possibilidade de dinamicamente expandir os parâmetros formais de um método através de um array. Como arrays são heterogêneos, os parâmetros adicionados dinamicamente podem ser de qualquer tipo. Observe o seguinte trecho de código que exemplifica bem este recurso:

```
def varargs(arg1, *rest)
    "Got #{arg1} and #{rest.join(', ')}"
end
varargs("one")                  »   "Got one only "
varargs("one", "two")           »   "Got one and two"
varargs("one", "two", "three")  »   "Got one and (two, three)"
```

2.4 Métodos

- Métodos são os subprogramas das linguagens orientadas à objetos. Seus parâmetros formais são chamados de protocolos, mas a forma de implementação não difere muito entre os dois paradigmas.
- Na linguagem Ruby, a procura por um método chamado para ser executado é realizada de forma dinâmica. Isto significa que quando um método não é encontrado em sua classe atual, o próximo lugar a ser procurado será a sua classe pai, e assim sucessivamente, até chegarmos na superclasse.
- Os métodos conseguem retornar somente um objeto, não importando que tipo de objeto. Isto significa que um método pode retornar qualquer coisa, devido à capacidade dos arrays serem heterogêneos.

2.5 Variáveis

- Variáveis são somente referências para objetos. Nada mais. Não existem ponteiros. Apesar desta lacuna, aliases de objetos podem ser feitos, como em Java.
- As variáveis na linguagem Ruby não possuem tipo, assim como em Smalltalk, BASIC Ou Python. Isto significa que conforme as variáveis são utilizadas no programa elas vão tendo o seu tipo atribuído. No contexto de Linguagens de Programação isto é chamado de atribuição de tipos dinâmica. Conseqüentemente, as verificações de tipos de variáveis durante a interpretação do código são minimizadas, tornando assim a linguagem não fortemente tipificada.
- As variáveis não precisam ser declaradas antes de serem utilizadas. Conforme o nome da variável o interpretador sabe qual o escopo da variável utilizada. No contexto de linguagens de programação, estas variáveis são chamadas de variáveis heap-dinâmicas implícitas. Na seção 3.1 você pode ver uma tabela que explica como controlar o escopo de variáveis através do seu nome.
- A atribuição de valores à variáveis é feita através de referências à objetos. Isto significa que ao atribuir um valor (lembre-se de que um valor também é um objeto na linguagem Ruby) à um objeto, este valor não precisa ser copiado e instanciado novamente, somente uma referência para o valor que está sendo atribuído é criada.

2.6 Gerenciamento da Memória

- O gerenciamento de memória do Ruby é todo feito automaticamente pelo próprio interpretador. Periodicamente, a linguagem Ruby aloca pequenos blocos de memória para que nunca falte memória para um script que esteja rodando, automatizando assim o processo de alocação de memória. Um garbage collector encarrega-se de todos objetos que não são referenciados à bastante tempo, facilitando assim a vida do programador, que não precisa desalocar blocos de memórias para objetos.

- Apesar disto, instruções específicas de alocação/desalocação implementadas nativamente possibilitam ao programador gerenciar casos específicos aonde o gerenciamento automático não se aplica.

- Uma grande vantagem do garbage collector é que desaparecem os memory leak, acontecem poucos erros ou travamentos. Isto torna a programação mais rápida e menos complicada, porque não é necessário fazer o gerenciamento de memória manual. Infelizmente perde-se velocidade (aproximadamente 10%), mas como Ruby é uma linguagem interpretada 10% torna-se uma quantia insignificativa.

2. 7 Tratamento de Exceções

- O tratamento de exceções no Ruby funciona da mesma maneira que na linguagem Java.

- Uma classe chamada Exception consegue tratar exceções de duas maneiras distintas, cada qual com sua finalidade específica. Numa maneira você consegue capturar erros mas é obrigado a abandonar a aplicação. Na outra maneira, que funciona através dos comandos catch/throw, você pode tratar exceções em blocos, como no Java.

- Como qualquer outra classe na linguagem Ruby, a classe Exception pode ser alterada para satisfazer necessidades específicas, geralmente através da criação de novas exceções ou do uso do polimorfismo em métodos já existentes.

- A linguagem Ruby permite que o programador "simule" uma exceção, através das chamadas traps.

2. 8 Threads

- Na linguagem Ruby, você pode executar tarefas simultâneas através de múltiplas threads ou através de múltiplos processos. Uma classe threads possui todas as construções para trabalharmos com aplicações multithreaded, sem a necessidade de nenhuma biblioteca externa, como em C por exemplo.

- A linguagem implementa threads totalmente em nível de usuário, o que a torna independente de sistema operacional. Isto traz à linguagem Ruby uma grande portabilidade, mas traz também algumas desvantagens. Uma grande desvantagem de utilizarmos threads em nível de usuário é que elas compartilham o escalonamento do processo ao contrario de threads de sistema que compartilham o processador com os outros processos. Usar threads em ruby nunca faz o programa executar mais rapido e sim mais lento por causa do custo da troca de contexto. Mas permite execução concorrente compartilhando um mesmo espaco de memória.

- A sincronização entre threads cooperantes/competidoras é feita através de seções críticas, implementadas em uma classe chamada Mutex. Além disso, existem atributos próprios da classe Thread que pode ser utilizado para criar "seções críticas" de códigos que precisam ser atômicos, sem a necessidade de utilizar a classe Mutex.

- A criação de novos processos é feita através de métodos da classe kernel, que nada mais são do que chamadas de sistema nativas do ambiente *nix (fork/exec ou system, por exemplo). Com isso você não tem muita portabilidade, mas apesar disso ganha uma grande flexibilidade ao trabalhar com o sistema de processos nativos do *nix.

2.9. GUI

- Apesar da linguagem não possuir nenhuma classe para implementação de GUI nativa, existem várias bibliotecas que podem ser utilizadas junto com o Ruby. Como exemplo poderíamos citar Tk, OpenGL e GTK, entre outros.

2.10 Programação Distribuída

- Através de uma biblioteca criada com os próprios comandos da linguagem Ruby (cerca de 200 linhas, segundo o autor) é possível ter a mesma funcionalidade do RMI utilizando o que é chamado dentro do contexto do Ruby de DistributedRuby ou drb (nome da biblioteca).

2.11 Ferramentas de Desenvolvimento

- A distribuição padrão de Ruby contém ferramentas úteis junto com o interpretador e as bibliotecas padrão: depuradores, "profilers", "irb (espécie de um interpretador interativo)" e um ruby-mode para Emacs. Estas ferramentas ajudam você a depurar e melhorar seus programas Ruby.
- Caso seja necessário acessar banco de dados (como PostgreSQL ou MySQL) ou utilizar um toolkit GUI não padrão como Qt, Gtk, FOX, etc., você pode buscar recursos no Arquivo de Aplicações Ruby (RAA). Neste arquivo podem ser encontrados uma série de programas Ruby, bibliotecas, documentação e pacotes binários compilados para plataformas específicas. Você pode acessar RAA em <http://www.ruby-lang.org/en/raa.html>. RAA continua de longe menor que CPAN da Perl, mas ele está crescendo todos os dias.

3. A SINTAXE DA LINGUAGEM RUBY

3.1 Primeiros Passos

O propósito geral deste artigo é dar uma noção geral sobre a linguagem Ruby, e como não poderia deixar de ser, devemos também abordar como é a sintaxe da linguagem Ruby. Neste mini-tutorial sobre a linguagem já dá pra ter uma noção de como é a sintaxe e o quão fácil é programar em Ruby.

Vamos começar com um simples exemplo. Vamos escrever um método que retorna uma string, adicionando à esta string uma outra string passada como parâmetro.

```
def sayHello(name)
  result = "Hello, " + name
  return result
end

# Hello World!
puts sayGoodnight("World")
```

Antes de tudo, vamos fazer algumas observações. Primeiro, a sintaxe do Ruby é muito simples. Ponto e vírgula como terminadores não são necessários. Comentários são linhas começadas por #.

Métodos são definidos pela palavra reservada def, seguido pelo nome do método e seus parâmetros entre parênteses. O método do exemplo acima é simples e é totalmente compreensível para alguém que possui conhecimento em qualquer linguagem de programação estruturada. Vale observar que não é necessário declarar variáveis.

Métodos não podem ser declarados fora de classes, como no exemplo acima. Entretanto, por motivos de brevidade, não declarei nenhuma classe.

Além disto, você pode chamar o método sayHello de diversas formas, como mostrado abaixo:

```
puts sayHello "World"
puts sayHello("World")
puts(sayHello "World")
puts(sayHello("World"))
```

Este exemplo também mostra como criar um objeto da classe String. Existem muitas maneiras de criar um objeto String, mas provavelmente a mais simples é utilizar uma palavra entre aspas atribuída ao objeto que desejamos criar.

No Ruby, todas aquelas seqüências de caracteres bem conhecidas dos programadores C/C++ que começam com \ são válidas, como por exemplo \n, \t, etc...

O Ruby utiliza uma convenção para variáveis para ajudar na legibilidade do código, assim como em Perl. O primeiro caracter de cada nome de variável indica o seu tipo. Variáveis locais, métodos e seus parâmetros devem começar

com letras minúsculas ou o símbolo _. Variáveis globais são iniciadas por $, enquanto que instâncias de objetos começam com @. Nomes de Classes, módulos e constantes devem começar com letras maiúsculas. Veja a tabela abaixo que clarifica um pouco as coisas com alguns exemplos:

Variáveis Locais	Globais	Instâncias	Classes	Constantes
nome	$debug	@cidadao	@@pc	PI
x_axis	$soma	@prisma	@@objeto	Tamanho
_altura	$i	@objeto	@@linha	Lines
_idade	$arq	@carro	@@estrada	E

3.2 Arrays e Hashes

Arrays e Hashes no Ruby são coleções indexadas de objetos. Ambos armazenam objetos utilizando uma chave como acesso. Nos arrays, as chaves são números inteiros, como em qualquer outra linguagem. Hashes suportam qualquer objeto como chave. Ambos aumentam de tamanho conforme necessário. Uma observação interessante é que tanto arrays quanto hashes podem guardar qualquer tipo de objeto (inclusive de diferentes tipos).

Você pode criar e inicializar arrays declarando-os literalmente - um conjunto de elementos entre chaves, por exemplo. Com o array criado, você pode acessar seus elementos através de índices, como em qualquer outra linguagem, como no exemplo abaixo:

```
a = [ 1, 'cat', 3.14 ]   # array com três elementos
# acessa o primeiro elemento
a[0] » 1
# armazena nil na segunda posição do array
a[2] = nil
# o array após essas operações
a » [1, "cat", nil]
```

Um array vazio pode ser criado de duas maneiras, conforme mostrado abaixo:

```
empty1 = []
empty2 = Array.new
```

Às vezes criar um array de palavras pode ser uma tarefa de hércules, com todas aquelas aspas e vírgulas, Na linguagem Ruby, existe uma outra maneira de isto ser feito:

```
a = %w{ ant bee cat dog elk }
a[0] » "ant"
a[3] » "dog"
```

Hashes são muito parecidas com arrays. Uma declaração literal utiliza chaves ao invés de colchetes, conforme ilustrado abaixo:

```
instSection = {
  'cello'     => 'string',
  'clarinet'  => 'woodwind',
  'drum'      => 'percussion',
  'oboe'      => 'woodwind',
  'trumpet'   => 'brass',
  'violin'    => 'string'
}
```

Para ter acesso aos elementos, utilize a chave declarada:

```
instSection['oboe']    » "woodwind"
instSection['cello']   » "string"
instSection['bassoon'] » nil
```

Como no último exemplo, uma hash retorna nil quando uma chave indexada não é encontrada. Normalmente esta solução não causa muito problema, mas podemos também mudar este valor default. Isto é facilmente feito através do construtor da classe Hash, conforme ilustrado abaixo:

```
histogram = Hash.new(0)
histogram['key1'] » 0
histogram['key1'] = histogram['key1'] + 1
histogram['key1'] » 1
```

Existem uma série de métodos para manipulação destas estruturas de dados já implementados nas classes Array e Hash, mas como esta seção tem como objetivo somente um overall da linguagem, caso necessite algo mais sério pararemos por aqui.

3.3 Estruturas de Controle

A linguagem Ruby possui todas as estruturas de controle padrões das linguagens populares hoje em dia, como ifs e whiles. A única diferença é que o Ruby não usa chaves e sim a palavra reservada end para acabar uma estrutura. Por exemplo:

```
if count > 10
  puts "Tente novamente"
elsif tries == 3
  puts "Você perdeu"
else
  puts "Digite um número"
end
```

Da mesma maneira, um while termina com um end, como no exemplo abaixo:

```
while weight < 100 and numPallets <= 30
  pallet = nextPallet()
  weight += pallet.weight
  numPallets += 1
end
```

Para facilitar a legibilidade do código (ou não) existem as construções podem ser feitas de maneiras um pouco diferentes. Veja no exemplo abaixo o exemplo de um if simples:

```
#uma construção padrão
if radiation > 3000
  puts "Danger, Will Robinson"
end
#agora reescrevendo a construção
puts "Danger, Will Robinson" if radiation > 3000
```

Da mesma forma, um while poderia ser feito da seguinte forma:

```
#da forma padrão
while square < 1000
   square = square*square
end
#modificando a construção
square = square*square while square < 1000
```

Isto é muito comum na linguagem Perl, e como a linguagem Ruby tem como um dos seus objetivos principais o processamento de palavras, estas construções tornam-se úteis no dia-a-dia.

3.4 Expressões regulares

A maioria das construções do Ruby é familiar para a maioria dos programadores. Expressões regulares, embora sejam construções de aparência bizarra, são uma poderosa ferramenta para o programador que trabalha com processamento de textos.

Livros inteiros tratando somente de expressões regulares foram escritos, portanto não entraremos muito a fundo neste assunto, somente o necessário para que o leitor do artigo tenha uma noção do poder das expressões regulares em Ruby.

Uma expressão regular é somente uma maneira de especificar um padrão de caracteres a ser reconhecido. Na lingaugem Ruby, você tipicamente cria padrões entre barras (/padrão/).

Por exemplo, você poderia escrever um padrão para reconhecimento de uma string que contém as strings Perl ou Python utilizando a seguinte expressão regular:

```
/Perl|Python/
```

As barras externar delimitam o padrão. O caracter pipe (|) indica ou. Outra forma de escrever este padrão seria:

```
/P(erl|ython)/
```

Você também pode especificar repetições nos padrões. Por exemplo, /ab*c/ um a seguido de zero ou mais b´s seguido de um c. /(ab)+c/ permite ou (ab) ou (c).

As mais bizarras construções podem ser feitas, e um programador experiente no uso de expressões regulares pode simplificar muito a sua vida na hora de construir parsers ou outro tipo de processamento de textos.

Para maiores detalhes sobre expressões regulares, consulta a man page do comando egrep ou leia o livro Mastering Regular Expression.

4. CARBONE - UMA MÁQUINA VIRTUAL PARA RUBY

Carbone, baseado em Vmgen, é uma eficiente VM para Ruby. Ela tem como alvo compatibilidade, performance e portabilidade.

Vmgen, de Anton Ertl, é originalmente a ferramenta de construção para a Gforth, a mais rápida e portável implementação de Forth. Ela produz VMs que usam código multi-thread e fornece várias otimizações que fazem a execução de uma máquina virtual menos virtual (i.e. mais rápida).

Existem vários ganhos de flexibilidade usando Vmgen, mas a de maior importância é que instruções VM podem ser codificadas em C.

Os seguintes items já funcionam na Carbone VM:

- sending;
- chamada de funções genéricas de C;
- partes do sistema de objetos do Ruby;
- definir classes;
- definir métodos;
- definir procs;
- chamar métodos e procs;
- instanciar variáveis para classes;
- Fixnum;

Proposta:

- vmgen4rb - Ruby parser para vmgen arquivos .vmb e Ruby writer para estes arquivos (entre algumas estruturas de dados Ruby)

- rb2c tradutor baseado no Carbone (necessário .vmg parser feito em Ruby)
- Garbage Collection - nova geração de frontend para o Boehm Collector. Isto pode ser mais genérico do que visando apenas a Carbone VM; medindo performance de um Copying Collector; Write Barrier;
- Framework para builtins que são escritos em C - eles podem ser fornecidos inline dentro da VM, mas também podem ser bibliotecas carregadas dinâmicamente. Podem ser usados por todos builtins escritos em C, então tudo pode ser "left out" quando se compila a Carbone.

Características:

- código direct thread;
- vmgen (top of stack caching, profiling, disasembling, superinstructions)
- chamadas rápidas de funções genéricas de C;
- cache de class-métodos configuráveis;
- pequeno tamanho da parte não otimizada da VM;
- linguagem intermediária de alto nível (LGram);
- layout de instanciação de memória contínuo;

Características futuras:

- compilador escrito em Ruby (exceto o último passo da compilação: geração das instruções)
- recompilação dinâmica

A linguagem de entrada - primeira linguagem nativa da Carbone - é parecida com Lisp e corresponde a um sub-conjunto de LGram level 1.

Quando o parser de Ruby funcionar e o compilador for escrito em Ruby a linguagem de entrada da Carbone vai ser um tipo de dado interno formatado.

Quanto à velocidade, o Carbone é capaz de executar até sete vezes mais rápido do que a Ruby Machine original (em um Pentium III). Isto acontece especialmente com códigos que não usam muitos builtins (como Fixnum#+, ...).

Isto porque apesar dos builtins serem feitos em C compilado, eles utilizam código recursivo ou chamadas à outros métodos que são implementados em Ruby. Por uma razão óbvia o fator de melhora mencionado vai para zero quando está se gastando todo o tempo em builtins.

O Carbone atualmente é um pré-release apenas para desenvolvedores. Ainda não existe um compilador para usuários finais. Nem compatibilidade passada e futura está garantida.

Existe somente versão do Carbone para Linux. O autor da Carbone é Markus Liedl <markus.lado@gmx.de> e ela pode ser baixada em http://www.nongnu.org/carbone/.

5. COMPARAĆÃO DO RUBY COM OUTRAS LINGUAGENS INTERPRETADAS

5.1 Fibonacci (n = 32)

Linguagem	CPU (seg)	Mem (kb)	Linhas de Código
Python	19.68	1260	9
Perl	20.64	1156	8
Ruby	24.95	1356	9
Tcl	64.89	1040	10
PHP	72.63	1320	8

5.2 Instanciação de Objetos (n = 1000000)

Linguagem	CPU (seg)	Mem (kb)	Linhas de Código
Ruby	19.98	2204	43
Guile	40.29	1788	32
Python	50.59	1268	42
Perl	90.99	1300	52

5.3 HeapSort (n = 80000)

Linguagem	CPU (seg)	Mem (kb)	Linhas de Código
Perl	9.08	3800	46
Python	10.13	3828	46
Ruby	16.83	3276	43
Tcl	23.34	8924	61
PHP	33.76	8740	46

6. EXEMPLOS DE CÓDIGO EM RUBY

6.1 Fibonacci

```
# calculate Fibonacci(20)
def fib(n)
  if n<2
    n
  else
    fib(n-2)+fib(n-1)
  end
end
print(fib(20), "\n");
```

6.2 Dining Philosophers Problem

```
#
# The Dining Philosophers - thread example
#
require "thread"

srand
N=9                     # number of philosophers
$forks = []
for i in 0..N-1
  $forks[i] = Mutex.new
end
$state = "-o"*N

def wait
  sleep rand(20)/10.0
end

def think(n)
  wait
end
```

```
def eat(n)
  wait
end

def philosopher(n)
  while TRUE
    think n
    $forks[n].lock
    if not $forks[(n+1)%N].try_lock
      $forks[n].unlock          # avoid deadlock
      next
    end
    $state[n*2] = ?|;
    $state[(n+1)%N*2] = ?|;
    $state[n*2+1] = ?*;
    print $state, "\n"
    eat(n)
    $state[n*2] = ?-;
    $state[(n+1)%N*2] = ?-;
    $state[n*2+1] = ?o;
    print $state, "\n"
    $forks[n].unlock
    $forks[(n+1)%N].unlock
  end
end

for n in 0..N-1
  Thread.start(n){|i| philosopher(i)}
  sleep 0.1
end

sleep
```

6.3 Calcula PI

```
k, a, b, a1, b1 = 2, 4, 1, 12, 4

while TRUE
  # Next approximation
  p, q, k = k*k, 2*k+1, k+1
  a, b, a1, b1 = a1, b1, p*a+q*a1, p*b+q*b1
  # Print common digits
  d = a / b
  d1 = a1 / b1
  while d == d1
    print d
    $stdout.flush
    a, a1 = 10*(a%b), 10*(a1%b1)
    d, d1 = a/b, a1/b1
  end
end
```

Referências

[1] BAGLEI, D., T*he Great Language Computer Shootout* (http://www.bagley.org/~doug/shootout/), 2002

[2] MATSUMOTO, Y., *Programming Ruby - The Pragmatic Programmers Guide* Segunda Edi ćão, Addison Wesley LongMan Inc., 2001

[3] Joshua, D. D., *Programming in the Ruby Language*, IBM DeveloperWorks (http://www-106.ibm.com/developerworks/linux/library/l-ruby1.html), 2001

[4] MATSUMOTO, Y., *The Ruby FAQ* (http://www.ruby-lang.org/en/), 2001