75.31 Teoría de Lenguage
El lenguaje de programación Ruby

28 de mayo de 2012

Diaz, Federico (83568)
Rossi, Federico Martín (92086)

1. Introducción: *Enfoque de la presentación*

La presentación, como así también el presente informe, tienen como fin mostrar a los oyentes este veterano, y a la vez moderno lenguaje de programación desde un punto de vista diferente al que generalmente se acostumbra. Es nuestro objetivo el dar a conocer las fortalezas y debilidades de éste fijando la perspectiva de la presentación en las aplicaciones cotidiantas del lenguaje en el desarrollo de proyectos profesionales. Es decir, sin quitar importancia a tópicos importantes como la sintaxis, nos focalizaremos en detallar aquellas caracteristicas que hacen de Ruby un lenguaje interesante y ávido de nuevos conceptos.

2. Un poco de historia

Ruby fue creado por *Yukihiro "Matz" Matsumoto*. Matz comenzó a trabajar en Ruby en 1993, cuando tenia 28 años, mientras trabajaba para una empresa Open source (netlab.jp). Ya por ese entonces era muy conocido en Japón por tener un alto perfil de evangelista dentro de la comunidad Open source y por trabajar en varios productos Open source. Ruby es el primer software japones altamente conocido fuera de Japón, el cual fue presentado en 1995.

> *"En 1993, yo estaba hablando con un colega acerca de lenguajes de scripting. Estaba muy impresionado por su poder y sus posibilidades. Sentía que el scripting era el camino a seguir.*
>
> *Me pareció que la programación orientada a objetos (POO) era muy adecuada para secuencias de comandos también (como viejo fan de la POO). Luego miré alrededor de la red. Me encontré con que Perl 5, que no se había lanzado todavía, implementaría las características de OO, pero finalmente no resultó lo que esperába. Me di por vencido en creer en Perl como lenguaje de scripts orientado a objetos.*
>
> *Entonces me encontré con Python. Se trataba de un interprete y un lenguaje orientado a objetos. Pero no lo sentía como si fuera un "script"del lenguaje. Además, se trataba de un lenguaje híbrido de la programación de procedural y la programación orientada a objetos.*
>
> *Yo quería un lenguaje que fuera más poderoso que Perl, y más orientado a objetos que Python. Es por eso que me decidí a diseñar mi propio Lenguaje de programación"*
>
> *Yukihiro Matsumoto (a.k.a. "Matz")*

Es así que comenzó a desarrollarlo el 24 de febrero de 1993 y en agosto desarrollo el primer Hello World con ruby. En 1994 se lanzó la primer versión alpha. Matz trabajó solo hasta el año 1996, momento en el cuál se formo la primer comunidad de Ruby.

3. ¿Qué es Ruby?

Es un lenguaje escrito en C y fue diseñado teniendo en mente las capacidades de perl y phyton. [Colocar contenido aquí]

4. El lenguaje y sus Principios

Antes de iniciarnos en la sintaxis del lenguaje creemos necesario entender el por qué es que fue creado Ruby, es decir, entender el impacto que su creador deseaba que tuviera y las razones por las cuales conforma una forma muy particular de realizar aplicaciones.

El lenguaje de programación Ruby es un "*lenguaje de programación interpretado para una rápida y fácil programación orientada a objetos*" - Pero, ¿Qué significa esto? Veámos:

Lenguaje de programación interpretado:

- habilidad para hacer llamadas directas al sistema operativo
- poderozo manejo y operaciones con cadenas y expresiones regulares
- respuestas inmediatas durante el desarrollo

Rápido y fácil:

- innecesarias las declaraciones de variables
- variables no tipadas
- syntaxis simple y consistente
- el manejo de memoria es automático

Programación orientada a objetos:

- todo es un objeto (como SmallTalk)
- clases, métodos, herencia, etc.
- métodos singleton (métodos que pertenecen a un sólo objeto)
- funcionalidad "mixin" por módulo
- iteradores y clausuras

Además:

- números enteros de múltiple precisión
- procesamiento de excepciones
- carga dinámica
- soporte de concurrencia

Sin duda, Ruby fue diseñado para hacer a los programadores mas productivos y felices. Esto es lo que el creador de Ruby nos dice:

> *"Para mí, el propósito de la vida es, al menos en parte, tener alegría. Los programadores a menudo sienten alegría cuando se pueden concentrar en el aspecto creativo de la programación, por lo que Ruby está diseñado para que los programadores sean felices. Considero a un lenguaje de programación como una interfaz de usuario, por lo que deben seguir los principios de la interfaz de usuario."*
>
> *Yukihiro Matsumoto (a.k.a. "Matz"), 2000*

¿Cuáles son los principios de una buena interfaz de usuario? Estos son los tres principios con citas de apoyo por parte de Matz:

> **Principio de concisión:** *"Quiero que las computadoras sean mis siervos, no mis maestros. Por lo tanto, me gustaría darles órdenes rápidamente. Un buen siervo debe hacer un montón de trabajo con una breve orden."*
>
> **Principio de consistencia:** *"... un pequeño conjunto de normas cubre la totalidad del lenguaje Ruby. Ruby es un lenguaje relativamente sencillo, pero no es demasiado simple. He tratado de seguir el principio de sin sorpresas. Ruby no es demasiado único, por lo que un programador con conocimientos básicos de otros lenguajes de programación puede aprender muy rápidamente."*
>
> **Principio de flexibilidad:** *"Porque las lenguas son para expresar el pensamiento, una lengua no debe restringir el pensamiento humano, sino que debe ayudarlo. Ruby consiste en un núcleo pequeño inmutable (es decir, sintaxis) y bibliotecas arbitrarias de clases extensibles..."*

5. Sintaxis

Ha llegado el momento de conocer la sintaxis de Ruby. El lector será capaz de notar una simpleza extrema de esta, lo que no significa una deficiencia en su potencial. Es decir, veremos que a partir de un número muy acotado de lineas de código seremos capaces de realizar programas realmente interesantes.

5.1. Tipos

Todos los tipos de ruby heredan de la clase Object, es decir que todos los tipos en ruby son objetos. Existen clases ya construidas que representan los tipos básicos de ruby sobre los cuales se construyen los bloques de todos los programas que se desarrollan en ruby.

5.2. Tipos numéricos

Los numeros en ruby son representados por las clases: Fixnum (entero -2¨30..2¨30) y Bignum. Cuando se crea un objeto numérico se asigna automáticamente al tipo Fixnum, si excede el rango será un Bignum. Los numeros enteros son creados ingresando el número sin coma. El formato de representación particular depende de la base del sistema numérico que se utilice. Rubi soporta las bases numéricas 10, 8, 16 y 2.

Código 1: *"Tipos Numéricos"*

```ruby
#Numeros
123456789 # -123456789 Fixnum
0d123456789 # 1234567890 Fixnum
1234323424231 # 1234323424231 Bignum
0x5C1 # 1473 Hex
01411 # 777 Octal
0b10 # 2 binario
1_90_33 # 19033  Fixnum, se omite el caracter _
1.5 # 1.5 Float
1.0e5 # 100000.0 Float

#Algunos Metodos
-14.abs # 14
6.zero? # false
```

En el *Código 1*, se ven los tipos numércos básicos que encontramos en cualquier lenguaje. Debajo de estos se encuentran sus particulares al ser objetos y no ser simplemente "tipos nativos". Existen otras representaciones numéricas como los Numeros imaginarios (Complex). En las secciones siguientes se veran el detalle el resto de los tipos básicos de ruby: Strings y Collections.

5.3. Strings

Los strings son simples secuencias de bytes que representan una secuencia de caracteres. A continuación se muestran ejemplos de código sobre sus cualidades básicas. En la siguiente sección se muestra el poder de las expresiones regulares.

Código 2: *Representacion de strings"*

```ruby
#Representacion en comillas simples o dobles
"abc"
'abc'

#Caracteres de escape
"nueva\nlinea"
"\tnueva parrafo"
```

Código 3: *Representación de strings"*

```ruby
#Expresion de evaluacion
apellido="Matzumoto"
nombre="Japones"
"#{apellido} #{nombre}"
```

```
i=0
"add one: #{i+=1}"
```

Código 4: *Representacion de strings"*

```
#Concatenacion y repeticion
"Matzumoto" + nombre
apellido * 2

#Extraccion de caracteres
wordFOO = "foo"
wordFOO[-1] #muestra 111 (el codigo ASCII de "o")
```

5.4. Expresiones regulares

Otro punto importante integrado en el lenguaje y siguiendo los buenos resultados que se obtiene en PERL, son las expresiones regulares. Estas permiten comprobar si una cadena satisface un patrón.

(Tabla de Caracteres especiales en expresiones regulares)

Carácter	Descripción
[]	Especificación de rango. (p.e. [a-z] representa una letra en el rango de la a a la z
\w	Letra o dígito; es lo mismo que [0-9A-Za-z]
\W	Ni letra, ni dígito.
\s	Espacio, es lo mismo que [t n r f]
\S	No espacio
\d	Dígito; es lo mismo que [0-9]
\D	No dígito
\b	Backspace (0x08) (sólo si aparece en una especificación de rango)
\b	Límite de palabra (sólo si no aparece en una especificación de rango)
\B	No límite de palabra
*	Cero o más repeticiones de lo que precede
+	Una o más repeticiones de lo que precede
[m,n]	Al menos m y como máximo n de lo que precede
?	Al menos una repetición de lo que precede; es lo mismo que [0,1]
|	Puede coincidir con lo que precede o con lo que sigue
()	Agrupamiento

Código 5: *.Expresiones regulares"*

```
#Expresiones regulares
def chab(s) # contiene la cadena un hexadecimal (entre angulos "<>")
  (s =~ /<0[Xx][\dA-Fa-f]+>/) != nil
end
```

5.5. Colecciones - Rangos

La colección más primitiva de todas es el rango que permite representar una colección secuencial de valores (números o letras secuenciales) Para indicar rangos se pueden utilizar los dos puntos (incluye al primer elemento y al último elemento inclusive) o tres puntos (excluye al último elemento del rango).

Código 6: *Rangos"*

```
#rangos
edadNinos = 4..12
estrellasHotel = 1...6
edadNinos.include?(12) #true
estrellasHotel.include?(6) #false
```

5.6. Colecciones - Arrays

A continuación se muestran ejemplos sobre el uso de arrays. Más adelante, cuando se vean los conceptos de bloques se veran los iteradores sobre esta y el resto de las colecciones.

Código 7: *"Arrays"*

```
#Los array pueden almacenar elementos de distintos tipos
ary = [1, 2, "3"]

#Al igual que las cadenas...
ary + ["foo", "bar"] #[1, 2, "3", "foo", "bar"]
ary * 2 #[1, 2, "3", 1, 2, "3"]
ary[-2] #2

#Uso de indices
ary[0,2] #[1, 2]
ary[-2,2] #[2, "3"]
ary[-2..-1] #[2, "3"]

a = [ "a", "b", "c", "d" ]
a.map! {|x| x + "!" }
a             #=>  [ "a!", "b!", "c!", "d!" ]

a = [1, 2, 3, 4, 5, 0]
a.drop_while {|i| i < 3 }   #=> [3, 4, 5, 0]

a = [1, 2, 3]
a.permutation.to_a     #=> [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

5.7. Colecciones - Hash

Conocidos también como diccionarios (aunque en el mundo ruby se prefiere el nombre hash), a continuación se exponen ejemplos de uso de los mismos.

Código 8: *"Hash"*

```
h = {1 => 2, "2" => "4"} #{"2"=>"4", 1=>2}
h[1] #2
h[5] #nil
h[5] = 10 # anadir un valor
h[1]=nil # borrar un valor
h #{5=>10, "2"=>"4", 1=>nil}

h = { "a" => 100, "b" => 200, "c" => 300 }
h.delete_if {|key, value| key >= "b" }   #=> {"a"=>100}
```

5.8. Variables

En ruby las variables son referencias a valores en memoria. Cuando se asigna una variable a otra, no se duplica el objeto. Para duplicar objetos se puede utilizar los métodos clone o dup que son de Object. Las variables pueden tener alcance global, local (dadas por cualquier bloque: bloque de una estructura de control o bloque de un método) o de clase.

Código 9: *"Variables"*

```
#En ruby las variables son referencias
firstVar="Dogs"
secondVar = firstVar
firstVar.chop!
#ambas variablas terminan
#con una referencia al valor "Dog"
```

5.9. Metodos

Cuando un método es llamado en Ruby, técnicamente no se esta "llamando" un método. Lo que se esta haciendo es enviando un mensaje a un objeto, como diciendo: "Hey, vos tenes este método?" si lo tiene entonces se ejecuta, si no lo tiene se lanza una exception "NoMethodError". Los métodos siempre devuelven un valor, el que corresponde a la última sentencia que ejecuta.

Código 10: *"Declaración metodos"*

```ruby
#Definicion de un metodo
def sayHello(name)
  puts "Hello #{name}!."
end

#Parametros x defecto
def new_method(a = "This", b = "is", c = "fun")
  puts a + ' ' + b + ' ' + c + '.'
end
new_method('Rails') #Rails is fun
```

Código 11: *"Parametros"*

```ruby
#Tamano variable en la lista de parametros
def print_relation(relation, *names)
puts "My #{relation} include: #{names.join(', ')}."
end
print_relation("cousins", "Morgan", "Miles", "Lindsey")

#Se pueden omitir los parametros
puts "Look ma! No parentheses!"

#Permite recursividad
def sum_integers(a, b)
  return 0 if a > b
  a + sum_integers((a + 1), b)
end

sum_integers(1, 10) #51 Hace la progresion de sumas en el rango dado
```

Código 12: *.Alto orden utilizando metodos"*

```ruby
#Programacion de alto orden
# Implementacion del sig patron:
# def <name>(a, b)
#   return 0 if a > b
#   <term>(a) + <name>(<next>(a), b)
# end
def sum(term, a, the_next, b)
  return 0 if a > b
  term.call(a) + sum(term, the_next.call(a), the_next, b)
end

def termino(x)
  x
end

def inc(x)
 x+=1
end

term = self.method(:termino).to_proc
the_next= self.method(:inc).to_proc
sum(term,1,the_next,10)
```

5.10. Bloques y Func

Los bloques son un concepto muy poderoso e importante en Ruby. En ruby los bloques son objetos que contienen código y todo el contexto necesario para ejecutarse.

Código 13: *"Bloques básicos"*

```ruby
#Bloques Basicos
myarray.each {|element| print "[" + element + "]... " }
#Se esta pasando como parametro, el
#contenido del bloque al metodo each
#para que lo ejecute

myarray.each do |element|
print "[" + element + "]... "
end
```

Un objeto procedimiento nuevo se obtiene utilizando Proc. Se pueden utilizar procedimientos anónimos. Estos objetos preservan el contexto en el cual fueron creados.

Código 14: *"Proc"*

```ruby
#Procs
inthandler = proc{ print "^C ha sido pulsado.\n" }
trap "SIGINT", inthandler

addHandlerToSIGINT = proc{|handler| trap("SIGINT", handler) }
addHandlerToSIGINT.call(inthandler)
```

Código 15: *"Proc anónimos"*

```ruby
#Proc anonimos
trap "SIGINT", proc{ print "^C ha sido pulsado.\n" }
trap "SIGINT", 'print "^C ha sido pulsado.\n"' #Forma mas compacta
```

Código 16: *"Proc y Bloques"*

```ruby
#Procs y Bloques
def make_signal_handler(signal)
  Proc.new{|handler| trap(signal, handler) }
end

SIGINThandlerMaker = make_signal_handler("SIGINT")
SIGINThandlerMaker.call(inthandler)

#Funciones que devuelven funciones
def make_filter(predicate)
  lambda do |list|
    new_list = []
    list.each do |element|
      new_list << element if predicate.call(element)
    end
    new_list
  end
end

filter_odds = make_filter( lambda{|i| i % 2 != 0} )
filter_odds.call(list)
```

Código 17: *"Func lambda"*

```ruby
#funciones lambda
lproc = lambda {|a,b| puts "#{a + b} <- the sum"}
nproc = Proc.new {|a,b| puts "#{a + b} <- the sum"}

nproc.call(1, 2, 3)
# 3
lproc.call(1, 2, 3)
# !ArgumentError (wrong number of arguments (3 for 2))
```

5.11. Estructura de control

Código 18: *."Estructuras de control"*

```ruby
#Case
i=8
case i
  when 1, 2..5
    print "1..5\n"
  when 6..10
    print "6..10\n"
end
#6..10

case 'abcdef'
  when 'aaa', 'bbb'
    print "aaa o bbb\n"
  when /def/
    print "incluye /def/\n"
end
#incluye /def/

#While
i = 0
print "Es cero.\n" if i == 0
print "Es negativo\n" if i < 0
print "#{i+=1}\n" while i < 3
#Es cero.
#1
#2
#3

#For
for elm in [100,-9.6,"pickle"]
  print "#{elm}\t(#{elm.type})\n"
end
#100 (Fixnum)
#-9.6 (Float)
#pickle (String)

# For estilo c o java
for i in coleccion
..
end

# For estilo smalltalk
coleccion.each {|i|
...
}
```

5.12. Convención de nombres

En Ruby se añade, por convención, ! o ? al final de ciertos nombre de métodos. La marca de exclamación (!, pronunciada como un "bang!" sonoro) recalca algo potencialmente peligroso, es decir, algo que puede modificar el valor de lo que toca. El método de la clase cadena chop! afecta directamente a la cadena pero chop sin el signo de exclamación actúa sobre una copia. Los nombres de métodos que finalizan con un signo de interrogación (?, pronunciada a veces como un "huh?" sonoro) indica que el método es un "predicado", aquel que puede devolver o true o false.

5.13. Métodos Singleton

El comportamiento de una instancia es determinado por su clase, pero hay momentos en los que es necesario que una instancia en particular posea un comportamiento especial. En la mayoría de los lenguajes deberíamos adentrarnos en la problematica de definir una nueva clase, la cual será instanciada una sola vez. En cambio, Ruby permite dar a cualquier objeto sus propios métodos.

Código 19: *"Métodos Singleton ejecutado en la consola"*

```
ruby> class SingletonTest
    |   def size
    |     25
    |   end
    | end
   nil
ruby> test1 = SingletonTest.new
   #<SingletonTest:0xbc468>
ruby> test2 = SingletonTest.new
   #<SingletonTest:0xbae20>
ruby> def test2.size
    |    10
    | end
   nil
ruby> test1.size
   25
ruby> test2.size
   10
```

En el *Código 19*, las instancias *test1* y *test2* pertenecen a la misma clase, pero en *test2* se ha redefinido el método *size* por lo que se comportará de forma diferente. Un método que pertenece sólo a un único objeto se denomina *Método Singleton*.

6. Multithreading

Los threads en ruby estan implementados completamente en el interprete de Ruby, esto lo hace portable puesto que no hay dependencia del sistema operativo, pero en contra parte con esto no se obtienen los beneficios de trabajar con threads nativos y por lo tanto se pueden experimentar problemas de innanición (starvation) al utilizar threads en Ruby, asi como en el caso en que se produsca un deadlock en los thread se podria suspender todo el proceso. Tambien en el caso en que un thread que esta en ejecución realiza una llamada a sistema que toma mucho tiempo, hasta que el interprete no tenga nuevamente el control no habria multithreading. El siguiente codigo muestra como se utilizan thread para manejar cada transaccion HTTP para descargar un conjunto de paginas.

Código 20: *"Multithreading"*

```ruby
#Descarga de sitios web
require 'net/http'
pages = ["www.rubycentral.com", "www.awl.com", "www.pragmaticprogrammer.com"]
threads = []

for page in pages
  threads << Thread.new(page) { |myPage|

    h = Net::HTTP.new(myPage, 80)
    puts "Fetching: #{myPage}"
    resp, data = h.get('/', nil )
    puts "Got #{myPage}:  #{resp.message}"
  }
end
threads.each { |aThread|  aThread.join }

#Fetching: www.rubycentral.com
#Fetching: www.awl.com
#Fetching: www.pragmaticprogrammer.com
#Got www.rubycentral.com:  OK
#Got www.pragmaticprogrammer.com:  OK
#Got www.awl.com:  OK
```

Analizando las sutilezas que esconde el codigo podemos decir que los threads son creados haciendo la llamada Thread.new. Esta recibe un bloque que contiene el código que sera ejecutado por el thread. En este caso el bloque utiliza la libreria net/http. Cuando creamos un thread, estamos pasando la pagina web requerida como parametro. Luego este parametro es pasado dentro del bloque como myPage.

6.1. Manipulando threads

En el codigo anterior vemos que un metodo que se invoca de cada thread es join. Cuando un programa en Ruby termina, se matan todos los threads sin tener en cuenta su estado. Join sirve para esperar que termine su ejecución un thread particular, de esta forma el codigo que vimos se asegura que terminen los 3 threads que se lanzaron. Ademas de join hay otros metodos, como por ejemplo Thread.current obitiene el thread actual. Para obtener una lista de todos los threads en ejecucion o parados se ejecuta Thread.list. Para saber el estado de un thread se puede ejecutar thread.status o thread.alive?.

6.2. Variables de threads

Un thread puede acceder a cualquier variable que se encuentre en el alcance donde fue creado. Variables locales del thread (dentro del alcance del bloque de ejecucion del thread) no son compartidas por otros threads. Se puede utilizar a los thread como hash donde se puedan almacenar variables locales a cada thread y ser accedidas desde el el thread principal.

Código 21: *"Variables de threads"*

```ruby
count = 0
arr = []
10.times do |i|
  arr[i] = Thread.new {
    sleep(rand(0)/10.0)
    Thread.current["mycount"] = count
    count += 1
  }
end
arr.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"

#8, 0, 3, 7, 2, 1, 6, 5, 4, 9, count = 10
```

6.3. Semáforos

Para controlar el acceso a los recursos compartidos por varios thread (exclusión mutua), ruby implementa en el core los semaforos en la clase Mutex. En las imagenes siguientes se ven la sintaxis y posteriormente un ejemplo de uso.

Código 22: *Üso de mutex"*

```ruby
require 'thread'
semaphore = Mutex.new

a = Thread.new {
  semaphore.synchronize {
    # access shared resource
  }
}

b = Thread.new {
  semaphore.synchronize {
    # access shared resource
  }
}

semaphore.lock

# access shared resource

semaphore.unlock
```

Código 23: *"Ejemplo utilizando mutex"*

```
require 'thread'
mutex = Mutex.new

count1 = count2 = 0
difference = 0
counter = Thread.new do
  loop do
    mutex.synchronize do
      count1 += 1
      count2 += 1
    end
  end
end
spy = Thread.new do
  loop do
    mutex.synchronize do
      difference += (count1 - count2).abs
    end
  end
end
sleep 1
mutex.lock
count1  # 21192
count2  # 21192
difference  # 0
```

7. Manejo de memoria

[Colocar contenido aquí]

8. Recolección de basura

[Colocar contenido aquí]

8.1. El porvenir de Ruby 2.0: *Bitmap Marking GC*

[Colocar contenido aquí]

9. *Ruby On Rails*: un punto fuerte del lenguaje

En la actualidad, Ruby se ha popularizado en el mundo del desarrollo de las aplicaciones webs a través del framework *Ruby On Rails*, o mas comúnmente conocido como *Rails*, escrito en este mismo lenguaje.

Ruby on Rails nace como un framework de desarrollo web especialmente diseñado con un fin en particular: hacer la vida más fácil a las personas que desarrollan aplicaciones destinadas a la web. Muchas de estas personas quiza se sientan frustradas con tecnologias como PHP, Java o .NET, que si bien son buenas, conllevan problemas de carga de recursos y una complejidad innecesaria. Ruby on Rails es simplemente más sencillo.

Rails utiliza el patrón MVC para poder administrar sus recursos. Si bien Java utiliza distintos frameworks también basados en MVC, Rails lleva el concepto mucho mas allá debido a que hay un lugar específico para cada parte del codigo, y cada componente de nuestra aplicación funciona de manera estándar. Es decir, es como si iniciaramos una aplicacion con el esqueleto previamente armado.

Todas estas potenciales caracteristicas hicieron que Ruby on Rails sea una de las razones por las que el lenguaje de programación Ruby haya logrado un importante impulso, siendo cada vez mas reconocido como una buena opción a la hora de elegir un lenguaje con el que llevar a cabo un proyecto.

10. Aplicaciones

Con el paso del tiempo Ruby fue utilizado por un número cada vez mayor de desarrolladores para llevar a cabo proyectos de grande envergadura. A continuación se muestra un listado de aplicaciones[1] realizadas en este

[1]"Véase una lista mas completa de aplicaciones en los siguientes vínculos: http://rubyonrails.org/applications http://www.ruby-lang.org/en/documentation/success-stories"

lenguaje:

Web:

- Twitter (http://www.twitter.com)
- Shopify (http://www.shopify.com)
- Groupon (http://www.groupon.com)
- XING (http://www.xing.com)

Simulaciones:

- *NASA Langley Research Center* usa Ruby para llevar a cabo simulaciones. (http://www.larc.nasa.gov)
- Motorola (http://www.motorola.com)

Modelado 3D:

- Google SketchUp (http://sketchup.google.com)

Telecomunicaciones:

- *Open Domain Server*: permitir a los usuarios el uso de DNS dinámicos (http://ods.org)
- *Lucent*: uso en producto de telefonía wireless 3G (http://www.lucent.com)

11. Un ejemplo práctico de aplicacion

12. Conclusión