Pattern Matching

TADP - 2016 C1 - TP Metaprogramación

Descripción del dominio

Pattern matching es la acción de comprobar que un elemento (en nuestro caso, un objeto) posea un conjunto de características determinadas, a las cuales denominaremos patrón, con el fin de poder realizar ciertas operaciones ligadas a dichas características.
Los patrones están fuertemente relacionados con la estructura interna del objeto (por ejemplo, su estado, su comportamiento), y nos permiten trabajar con ella de una manera sencilla, descomponiendo al objeto en los elementos que forman dicha estructura.

El objetivo del trabajo práctico entonces es aplicar los conceptos vistos en clase para construir un framework sencillo en Ruby que provea las herramientas mínimas necesarias para soportar los conceptos antes mencionados.

Entrega Grupal

En esta entrega tenemos como objetivo extender Ruby, implementando las herramientas necesarias para poder comprobar si un objeto cumple un *patrón*, descomponerlo según la estructura del mismo, y realizar ciertas operaciones con los elementos de dicha estructura.

Nota: Además de cumplir con los objetivos descritos, es necesario hacer el mejor uso posible de las herramientas vistas en clase sin descuidar el diseño. Esto incluye:

- Evitar repetir lógica.
- Evitar generar construcciones innecesarias (mantenerlo lo más simple posible).
- Buscar un diseño robusto que pueda adaptarse a nuevos requerimientos.
- Mantener las interfaces lo más limpias posibles.
- Elegir adecuadamente dónde poner la lógica y qué abstracciones modelar.
- Realizar un testeo integral de la aplicación cuidando también el diseño de los mismos.

1. Matcher

Un matcher es una construcción que describe cierta característica sobre un objeto. Al ser evaluada, determinará si el objeto en cuestión cumple o no con la característica definida.

Existen varios tipos de matchers:

a. **de variable:** se cumple siempre. Vendría a ser el matcher *identidad*. Su verdadera utilidad es bindear las variables (más sobre binding en la próxima sección).

```
:a_variable_name.call('anything') #=> true
```

b. **de valor**: se cumple si el valor del objeto es idéntico al indicado.

```
val(5).call(5) #=> true
val(5).call('5') #=> false
val(5).call(4) #=> false
```

c. **de tipo**: se cumple si el objeto es del tipo indicado.

```
type(Integer).call(5) #=> true
type(Symbol).call("Trust me, I'm a Symbol..") #=> false
type(Symbol).call(:a_real_symbol) #=> true
```

d. **de listas**: se cumple si el objeto es una lista, cuyos primeros N elementos coinciden con los indicados; puede además requerir que el tamaño de la lista sea N.

```
an_array = [1, 2, 3, 4]

#list(values, match_size?)
list([1, 2, 3, 4], true).call(an_array) #=> true
list([1, 2, 3, 4], false).call(an_array) #=> true

list([1, 2, 3], true).call(an_array) #=> false
list([1, 2, 3], false).call(an_array) #=> true

list([2, 1, 3, 4], true).call(an_array) #=> false
list([2, 1, 3, 4], false).call(an_array) #=> false

#Si no se especifica, match_size? se considera true
list([1, 2, 3]).call(an_array) #=> false
```

```
#También pueden combinarse con el Matcher de Variables
list([:a, :b, :c, :d]).call(an_array) #=> true
```

e. **duck typing**: se cumple si el objeto entiende una serie de mensajes determinados.

```
psyduck = Object.new
def psyduck.cuack
 'psy..duck?'
end
def psyduck.fly
 '(headache)'
end

class Dragon
 def fly
   'do some flying'
 end
end
a_dragon = Dragon.new

duck(:cuack, :fly).call(psyduck) #=> true
duck(:cuack, :fly).call(a_dragon) #=> false
duck(:fly).call(a_dragon) #=> true
duck(:to_s).call(Object.new) #=> true
```

Nota: solo considerar los mensajes públicos (la interfaz del objeto).

2. Combinators

Ahora podemos combinar matchers! La idea es poder definir nuevos matchers como composición de los ya existentes. Cada combinator recibe por parámetro varios matchers y genera uno nuevo:

a. **and**: se cumple si se cumplen todos los matchers de la composición.

```
type(Defensor).and(type(Atacante)).call(una_muralla)#=>false
type(Defensor).and(type(Atacante)).call(un_guerrero)#=> true
duck(:+).and(type(Fixnum), val(5)).call(5) #=> true
```

b. **or**: se cumple si se cumple al menos uno de los matchers de la composición.

```
type(Defensor).or(type(Atacante)).call(una_muralla) #=> true
type(Defensor).or(type(Atacante)).call('un delfín') #=> false
```

c. **not**: genera el matcher opuesto.

```
type(Defensor).not.call(una_muralla) #=> false
type(Defensor).not.call(un_misil) #=> true
```

3. Pattern

Llamaremos patrón a un conjunto determinado de características que un objeto puede tener, es decir, a una colección de ***Matchers***. Como cabe esperar, el patrón puede ser evaluado con un objeto, y determinará si éste cumple con TODAS las características definidas.

Además, cada patrón tiene asociado un bloque donde se permite explotar el objeto, pudiendo acceder a parte de su estructura interna mediante variables.

a. **with**: genera un patrón a partir de un conjunto de ***Matchers*** pasados por parámetro y el bloque a asociar.

```
with(type(Animal), duck(:fly)) { ... }
```

b. **otherwise:** genera el patrón neutro, que siempre se verifica (es decir, no define ninguna característica en particular). No recibe parámetros, solo el bloque a asociar.

c. ***match***: llamamos *match* al evento producido cuando un objeto "encaja" (matchea) con cierto patrón, es decir, cuando cumple con todas las características definidas. El resultado de un *match* es la ejecución del bloque asociado al patrón.

d. **binding:** el *Matcher Variable* tiene una particularidad; ante un *match*, las variables definidas se *bindearán* con los valores correspondientes para poder ser utilizadas dentro del bloque de manera limpia y sencilla.

```
with(type(String), :a_string) { a_string.length }
with(type(Integer), :size) { size }

#Si se aplica sobre [1,2]
with(list[:a, :b]) { a + b } #=> 3
#Si se aplica sobre [1,2,Object.new]
with(list([duck(:+).and(type(Fixnum), :x),
  :y.or(val(4)), duck(:+).not])) { x + y } #=> 3
```

Veremos unos ejemplos completos en la próxima sección.

4. Matches

Ya tenemos matchers, tenemos patrones basados en ellos, y tenemos una forma de explotar los patrones mediante el bindeo de variables. Ahora solo necesitamos definir una sintaxis que permita aplicar una serie de patrones sobre un objeto:

```
matches?(an_object) do
 with(a_pattern) { ... }
 with(other_pattern) { ... }
 ...
 otherwise { ... }
 #Esto nunca se ejecuta!
end
```

Nota 1: Cuando se produce un *match* **no** deben seguirse evaluando los demás patrones.
Nota 2: Debe poder manejarse la situación en que no se produce ningún *match*. Queda a criterio del grupo decidir la mejor forma de implementarla.

Veamos algunos ejemplos concretos de uso:

```
x = [1, 2, 3]
matches?(x) do
 with(list([:a, val(2), duck(:+)])) { a + 2 }
 with(list([1, 2, 3])) { 'acá no llego' }
 otherwise { 'acá no llego' }
end
# => 3

x = Object.new
x.send(:define_singleton_method, :hola) { 'hola' }
matches?(x) do
 with(duck(:hola)) { 'chau!' }
 with(type(Object)) { 'acá no llego' }
end
# => "chau!"
```

```
x = 2
matches?(x) do
 with(type(String)) { a + 2 }
 with(list([1, 2, 3])) { 'acá no llego' }
 otherwise { 'acá si llego' }
end
# => “acá si llego”
```