Tarea práctica de programación orientada a objetos

Laboratorio de Lenguajes de Programación I

Enero–Marzo 2015

Índice general

La finalidad de esta tarea es usar metodos y conceptos que son aplicados en un lenguaje de programación orientados a objetos, como *duck-typing*, *simple-dispatch*, *double-dispatch* y *mixins*, según convenga o considere el estudiante para resolver los distintos problemas. Cabe destacar que el uso de *introspeccion*, *reflexición*, *variables globales* y *variables de clase* esta penalizado.

La tarea consiste en 3 parte (en cada una deberan usar *mixins*), la primera trata de recorrido sobre arboles (Binarios y Rosas) donde el *mixin* a definir encapsula el recorrido *DFS* de estructuras en general. Además, deberán definir un clase *Node* y extender otras clases existentes. Más adelante se detallarán las estructuras, y se espera que haga uso de la técnica de *despacho doble* para la solución de este problema.

En la segunda parte, se usaran las estructuras de arboles de la primera parte. A diferencia de, que se definirá un nuevo *mixin* que maneje los arboles. El cuál, va a incorporar comportamientos para hacer *fold* y *map*, pero ahora el recorrido será en *BFS*.

Por último, se le pedira al estudiante que simule dos *Type Classes* de Haskell. *Monoid* y *Functor*, e *"instanciar"* ambos *"Type Classes"*. Lo que vendria siendo *extender* las clases con un *mixin*.

1 Arboles (8.25 puntos)

1.1 Estructuras (1 punto)

Considerando las siguiente definición de una clase para representar árboles binarios:

```
class BT
    attr_accessor :n # Objeto que guarda el nodo

    attr_reader :l,  # Hijo izquierdo (BT)
                :r   # Hijo derecho (BT)

    def initialize(n, l=nil, r=nil)
        # ...
    end

    def each &block
        # ...
    end
end
```

Donde el proposito del metodo `each` es *iterar* sobre los los hijos de nodo. Ademas, este metodo recibe un *bloque* de manera implícita.

Por otro lado, se tiene esta clase que representa los árboles rosa (*rose tree*), un atributo con el valor del nodo y un arreglo de sucesores:

...


```ruby
class RT
    attr_accessor :n # Objeto que guarda el nodo
    attr_reader :ss  # Arreglo de hijos

    def initialize(n, *sons)
        # ...
    end

    def each &block
        # ...
    end
end
```

Donde el proposito del metodo `each` es *iterar* sobre los los hijos de nodo, cuando esten definidos. Ademas, este metodo recibe un *bloque* de manera implícita. El constructor de la clase recibe el valor del nodo y de ahi en adelante son los hijos (RT)

1.2 Recorrido DFS (1.5 puntos)

Se espera que implemente un módulo que pueda ser utilizado por ambas clases, haciendo uso de la técnica de *mixins*. El mismo, debera ofrecer los metodos:

- `dfs &block`, que comienza a iterar desde *self*, va retornando los nodos (BT) y en el camino va llamando al bloque con los nodos iterados.
- `dfs! &block`, que comienza a iterar desde *self*, va retornando los nodos (BT) y en el camino va llamando al bloque con los nodos iterados. Pero en este caso, el metodo cambia los valores de los nodos (BT) en su camino.

Para la implementación del *mixin*, solo debera suponer que la clase responde al metodo `each`, para recorrer los hijos del nodo particular y el metodo de acceso a la variable de instancia `n`. Los metodos del *mixin*, no deberá crear variables de instancia o clases adicionales.

1.3 Nodos (1.5 puntos)

En cualquiera de las clases, que se mencionaron previamente, el valor de la variable de instancia `n` donde se almacena un objeto (*valor*). Que solo serán instancias de las clases *Fixnum*, *Symbol* o *Node*. Esta última deberá ser definida por usted, y tendrá una estructura así:


```
class Node
    attr_reader :x, :y

    def initialize x, y
        # ...
    end

    def visitado_por v
        # ...
    end

    def to_s
        # ...
    end
end
```

Podrá definir métodos adicionales, de ser necesario. Ambos atributos serán números (instancias de *Fixnum*). Ahora, las otras posibles instancias que podran habitar el atributo `n` podran ser *extendidas*, solo con el metodo *visitado_por* (de ser necesario).

El metodo *visitado_por*, recibira una instancia de clase (que se detallará más adelante). La cual operará sobre el objeto y devolvera otro objeto de clase *Fixnum*, *Symbol* o *Node* (dependiendo de cual instancia de clase visite).

La idea fundamental es, al momento de tener un objeto de clase arbol, recorrer dicha estructura, que cada nodo sea visitado por un objeto (*visitante*) y este produzca un resultado. Pudiendo cambiar o no el arbol.

1.4 Visitantes (2.75 puntos)

Los posibles visitantes (clases) que pueden existir, serán definidos por usted y tendrán el siguiente esquema:

```
class Visitante; end

class Mirror < Visitante
    # ...
end

class Next < Visitante
    # ...
```


```
end

class Strong < Visitante
    # ...
end
```

Podra notar que la clase *Visitante* no tiene más que su declaración. Porque, sencillamente es una *clase abstracta* (no se instancia).

Cada una de las clases operará de manera particular sobre los nodos, descrita de la siguiente manera:

- `Mirror`:
  - Con un nodo *Fixnum*, retorna el negativo.
  - Con un nodo *Symbol*, retorna un simbolo que sea palindromo. Ej. :ola -> :olaalo
  - Con un nodo *Node*, retorna un nodo con los atributos negados.
- `Next`:
  - Con un nodo *Fixnum*, retorna el proximo número.
  - Con un nodo *Symbol*, retorna un simbolo donde cada caracter tendra su sucesor. Ej. :nkz -> :ola
  - Con un nodo *Node*, retorna un nodo donde cada atributos tendra su sucesor respectivamente.
- `Strong`:
  - Con un nodo *Fixnum*, retorna el numero por 100.
  - Con un nodo *Symbol*, retorna un simbolo en mayúsculas. Ej. :ola -> :OLA
  - Con un nodo *Node*, retorna un nodo donde cada atributos tendra 100 veces su valor.

Se le sugiere hacer uso de *despacho doble* y *mixins* para la resolución del problema. Evitando cualquier uso de *introspección* y *reflexión*.

1.5 Foldable (1.5 puntos)

Para esta segunda parte, el estudiante deberá implementar un módulo que pueda ser usado por las clases *BT* y *RT*, usando la técnica de *mixins*. Pero en esta ocasión habrá un recorrido con *BFS*. Se espera que su *mixin* provea los siguientes métodos:

- `fold(b, &block)` que parte con un recorrido *BFS* desde el objeto *self*. Llamando al bloque con cada valor de los nodos que se encuentren en el arbol y tomando *b*, como valor base de la operación. Cabe destacar que el bloque recibe 2 argumentos (acumulador y valor del nodo).
- `map(&block)` que parte con un recorrido *BFS* desde el objeto *self*. Llamando al bloque con cada valor de los nodos que se encuentren en el árbol.
- `map!(&block)` que parte con un recorrido *BFS* desde el objeto *self*. Llamando al bloque con cada valor de los nodos que se encuentren en el arbol. A diferencia del metodo anterior, este cambia el valor de los nodos en el árbol.

Para la implantación de su mixin sólo puede suponer que las clases disponen del método de acceso al valor almacenado del nodo, y del método `each` para recorrer todos los hijos de un nodo particular. Los métodos de su mixin no deben crear variables de instancia, ni clases adicionales. Posiblemente necesite métodos adicionales dentro de su mixin para la implantación de su recorrido BFS.

2 Haskell *Type Classes* (1.75 pts.)

Como ultima pregunta, se le pide que defina (simule) dos clases de Haskell. Las clases *Monoid* y *Functor*, que en el caso de *Ruby* seran modulos (*mixins*).

Su modulo debe seguir los siguientes aspectos:

```
module Monoid
    #Minimal implementation
        # mempty  :: a
        # mappend :: a -> a -> a

    # mconcat :: [a] -> a
    def mconcat(as)
        # ...
    end
end

module Functor
    #Minimal implementation
        # fmap :: (a -> b) -> f a -> f b

    # (<$) :: a -> f b -> f a
```


```
    def inj a, fb
        # ...
    end
end
```

Donde toda clase que se extienda de estos modulos, debe cumplir con los *minimal implementation*, y asi poder usar *mconcat* o *inj* de *Monoid* y *Functor* respectivamente.

Y debera extender algunas clases con los modulos, de la siguiente manera:

```
# All
class TrueClass
    extend Monoid

    # minimal implementation
end

# Any
class FalseClass
    extend Monoid

    # minimal implementation
end

# ...

# Functor Instances

class String
    extend Functor

    # minimal implementation
end

# ...
```

3 Condiciones de entrega

Este proyecto debe ser realizado por cada alumno de CI3661 de manera individual. Debe entregar su solución en un archivo llamado `t4-XX-XXXXX.pdf` (donde `XX-XXXXX` sera

sustituido por el número de carné del estudiante), enviado adjunto a un correo electrónico titulado *[CI3661] Tarea Práctica 2* a las direcciones de *todos* los encargados del curso:

- Manuel Gómez manuel.gomez.ch@gmail.com
- David Lilue dvdalilue@gmail.com
- Ricardo Monascal rmonascal@gmail.com
- Wilmer Pereira wpereira@usb.ve

Debe enviar su solución antes de medianoche entre el domingo 2015-06-07 y el lunes 2015-06-08 en hora legal de Venezuela.

Deberá entregar un archivo .tar.gz o .tar.bz2, dentro del cual se encuentren solamente los siguientes archivos y siguiendo estos lineamientos:

- Los archivos `trees.rb`, `node_dd.rb`, `mod_dfs.rb`, `mod_fold.rb`, `mod_hs.rb` conteniendo el código fuente Ruby para implantar las clases y mixins correspondientes a cada sección.
- Se evaluará el estilo y buenas practicas de programación. Debe emplear una indentación adecuada y consistente en cualquier editor.
- El archivo debe estar correctamente documentado.
- Su programa será corregido usando Ruby 2.1, usando las librerías estándar. No está permitido utilizar librerías externas ni gemas.
- Puede escribir métodos adicionales si los necesita, pero estos no pueden ser públicos.
- Valor de Evaluación. Diez (10) puntos.

4 Referencias

- `Mixin`: Mixin.
- `Breadth-first Search`: Breadth-first Search.
- `Depth-first Search`: Depth-first Search.
- `Rose Tree`: Rose Tree.
- `Double Dispatch`: Double Dispatch.
- `Data-Monoid`: Data-Monoid.
- `Data-Functor`: Data-Functor.