Laboratorio de Lenguajes de Programación

USB / CI-3661 / Sep-Dic 2016

(Programación Orientada a Objetos – 15 puntos)

Déjà vu plegable

Árboles y arreglos plegables (3 puntos)

Considere la reapertura de la clase `Array` nativa de Ruby

```ruby
class Array
  def foldr e, b
    # Su código aquí
  end
end
```

En la cual el propósito de `foldr` es recibir un caso base y un *bloque* que será utilizado para recorrer la lista haciendo un pliegue de derecha a izquierda aplicando el *bloque* recibido.

Esto es, una llamada de la forma

```ruby
[1,2,3].foldr(z, f)
```

debe comportarse como si se evaluara `f(1, f(2, f(3, z)))`.

Ahora considere la siguiente definición para una clase que representa árboles multicamino (*Rose Trees*)

```ruby
class Rose
  attr_accessor :elem, :children
  def initialize elem, children = []
    @elem = elem
    @children = children
  end
```

```ruby
  def add elem
    @children.push elem
    self
  end

  def foldr e, b
    # Su código aquí
  end
end
```

En la cual el propósito de `foldr` es recibir un caso base y un *bloque* que será utilizado para recorrer el árbol haciendo un pliegue de derecha a izquierda aplicando el *bloque* recibido.

Un ejemplo de un árbol

```ruby
Rose.new(1, [
  Rose.new(2, [ Rose.new(4), Rose.new(5) ]),
  Rose.new(3, [ Rose.new(6) ])
])
```

Plegar es un comportamiento (3 puntos)

Se desea que usted implante una infraestructura de extensión del comportamiento de pliegue con la técinca de *mixins* estudiada en clase. Se espera que su *mixin* ofreza la siguiente interfaz de programación

- **`null?`** que indica si la estructura de datos está vacía (`true`) o contiene elementos (`false`).
- **`foldr1 &block`** que recorre la estructura de manera similar a `foldr`, pero usando el primer elemento como el elemento neutro y hace el recorrido sobre el resto de la estructura. Debe lanzar (`raise`) una excepción si la estructura está vacía.
- **`length`** que indica el tamaño de la estructura de datos, o, mejor dicho, la cantidad de elementos que contiene.
- **`all? &block`** que indica si **todos** los elementos de una estructura cumplen con un predicado.
- **`any? &block`** que indica si **alguno (1 o más)** de los elementos de una estructura cumplen con un predicado.
- **`to_arr`** que construye una *nueva* instancia de la clase `Array` con todos los elementos contenidos, en orden de izquierda a derecha.

- **`elem?  to_find`** que indica si un elemento se encuentra en la estructura, comparando elementos con `==`.

Para la implantación de su *mixin* sólo puede suponer que las clases usuarias disponen del método **`foldr`**, y que para definir un método del *mixin* puede usar **`foldr`** y cualquier otro método previamente definido en el *mixin*.

Este ejercicio está inspirado en la clase de Haskell estudiada en clase, Data.Foldable[1], por lo que puede leer su documentación como orientación también.

Ejemplos

Los siguientes ejemplos ilustran el comportamiento esperado:

```
> [].foldr(0) {|x,s| x + s}
=> 0
> [1,2,3].foldr(0) {|x,s| x + s}
=> 6

> [].null?
=> true
> [1,2,3].null?
=> false

# maximum
> [].foldr1 {|x,y| if x>y then x else y end}
RuntimeError: foldr1: empty structure
> [1,5,3].fodlr1 {|x,y| if x>y then x else y end}
5

> t = Rose.new(0, [ Rose.new(2, [ Rose.new(4), Rose.new(6) ]) ])
=> #<Rose:0x00...>
> t.length
=> 4

> t.elem? 6
=> true
> [1,2,3].elem? 6
=> false

> t.to_arr
=> [0, 2, 4, 6]
> [1,2,3].to_arr
=> [1, 2, 3]
```

```
# all are even
> t.all? {|x| x % 2 == 0}
=> true
> [1,2,3].all? {|x| x % 2 == 0}
=> false

# any is odd
> t.any? {|x| x % 2 != 0}
=> false
> [1,2,3].any? {|x| x % 2 != 0}
=> true
```

Plegar el promedio (1 punto)

Finalmente, escriba un método para calcular el promedio de los valores en un árbol multicamino usando únicamente funciones del *mixin*. Debe escribir la función de manera tal que recorra el árbol **una sola vez** para los cálculos necesarios.

```
class Rose
  def avg
    # Su código aquí
  end
end
```

Para usarlo de la siguiente manera

```
> t = Rose.new(1, [ Rose.new(5, [ Rose.new(10), Rose.new(6) ]) ])
=> #<Rose:0x00...>
> t.avg
5.5
```

Algebra de Intervalos (8 puntos)

En este ejercicio haremos una simple calculadora de intervalos[2], de manera que pueda aplicar los conceptos de despacho doble.

Las operaciones (6 puntos)

Utilizaremos la case `Interval` para representar la noción de un intervalo de números flotantes, y teniendo las siguiente subclases para ésta

- `Literal` (2 puntos) es la clase que representa los intervalos desde entre **dos** puntos finitos. Tiene cuatro variantes:
  - (a, b) para intervalos desde a hasta b, excluyéndolos (i.e. $x \in (a, b) \Rightarrow a < x < b$).
  - $[a, b)$ para intervalos desde a, incluyéndolo, hasta b, excluyéndolo (i.e. $x \in [a, b) \Rightarrow a \leq x < b$).
  - $(a, b]$ para intervalos desde a, excluyéndolo, hasta b, incluyéndolo (i.e. $x \in (a, b] \Rightarrow a < x \leq b$).
  - $[a, b]$ para intervalos desde a, incluyéndolo, hasta b, incluyéndolo (i.e. $x \in [a, b] \Rightarrow a \leq x \leq b$).
- `RightInfinite` (1 punto) es la clase que representa los intervalos desde **un** punto. Tiene dos variantes:
  - $(a, +\infty)$ para intervalos desde a, excluyéndolo (i.e. $x \in (a, +\infty) \Rightarrow a < x < +\infty$).
  - $[a, +\infty)$ para intervalos desde a, incluyéndolo (i.e. $x \in [a, +\infty) \Rightarrow a \leq x < +\infty$).
- `LeftInfinite` (1 punto) es la clase que representa los intervalos hasta **un** punto. Tiene dos variantes:
  - $(-\infty, a)$ para intervalos hasta a, excluyéndolo (i.e. $x \in (-\infty, a) \Rightarrow -\infty < x < a$).
  - $[-\infty, a]$ para intervalos hasta a, incluyéndolo (i.e. $x \in [-\infty, a) \Rightarrow -\infty < x \leq a$).
- `AllReals` (1 punto) es la clase que representa el intervalo de todos los números reales. Se denota $(-\infty, +\infty)$.
- `Empty` (1 punto) es la clase que representa el intervalo vacío. Se denota $\emptyset$

`AllReals` y `Empty` representan un conjunto específico cada uno, por lo que ambos deben ser singletones[3], es decir, debe existir una sola instancia en el programa para cada una de estas clases.

Debe escribir la clase `Interval` y todas sus subclases. La propiedad de si un extremo es o no incluyente debe ser un atributo de cada instancia.

Todas las subclases deben implantar los métodos:

- **`to_s`** para mostrar el invocante como un `String`, usando paréntesis `()` para extremos excluyentes y corchetes `[]` para extremos incluyentes.
- **`intersection other`** que retorna un nuevo intervalo que representa la intersección del intervalo invocante y el argumento `other`. La intersección entre dos intervalos **siempre** retorna un intervalo.

- `union other` que retorna un nuevo intervalo que representa la unión del intervalo invocante y el argumento `other`. Nótese que la unión de dos intervalos existe si y sólo si tienen una intersección no vacía o un extremo excluyente de uno de los intervalos tiene el mismo valor y es incluyente en el otro intervalo (e.g. $(a, b) \cup [b, c] = (a, c]$), si el intervalo no existe debe arrojar (`raise`) una excepción explicando el error.

La calculadora (2 puntos)

Iniciando la calculadora (1 punto)

Su programa debe recibir por argumentos de línea de comandos un archivo para leer y evaluar expresiones con el formato `«variable» «operador de comparación» «literal flotante»` separadas por conjunciones (`&`) o disyunciones (`|` o saltos de línea). La conjunción tiene mayor precedencia que la disynción. Por ejemplo:

```
x <= 42 & x >= 20 | x < 50
y > 10 & y < 20
z < 100 & z > 101
y > 5
w <= 0
a >= 0 | a <= 10
```

Que debe interpretarse como `((x <= 42 and x >= 20) or x < 50) or (y > 10 and y < 20) or (z < 100 and z > 101) or (y > 5) or (w <= 0) or (a >= 0 or a <= 10)` y generar la salida por pantalla:

```
x in [20,50)
y in (5,20)
z in empty
w in (,0]
a in (,)
```

Note que los extremos infinitos no se imprimen (e.g. `w in (,0]`, `a in (,)`).

Para cada nueva variable encontrada, se debe iniciar con un intervalo de `AllReals` e ir aplicando cada intersección o unión según sea el caso.

Interacción con el usuario (1 punto)

Finalmente, una vez cargados los intervalos para cada variable, el programa debe esperar por algun comando del usuario. El usuario puede:

- Introducir operaciones como `«variable» «operador» «variable»` para ver el resultado de la intersección (`&`) o unión (`|`) entre dos variables. Estas operaciones no alteran el estado de las variables, sólo deben imprimir su resultado en pantalla.
- Introducir `exit` para finalizar el programa.

Detalles de la Entrega

La entrega se hará en un archivo `pR-<G>.tar.gz`, donde `<G>` debe ser sustituido por el número de grupo que le fue asignado. El archivo *debe* estar en formato TAR comprimido con GZIP – ignoraré, sin derecho a pataleo, cualquier otro formato que yo puedo descomprimir pero que *no quiero* recibir.

Ese archivo, al expandirlo, debe producir un *directorio* que *sólo* contenga:

- El archivo `foldable.rb` con la solución a la sección ***Déjà vu* plegable**
- El archivo `algebra.rb` con la solución a la sección **Algebra de Intervalos**

El proyecto debe ser entregado por correo electrónico a la dirección de contacto del profesor Matteo Ferrando, a más tardar el domingo 2016-11-27 a las 23:59. Cada minuto de retraso en la entrega le restará un (1) punto de la calificación final.

Referencias

1: Data.Foldable en Hackage

2: Intervals (mathematics) en Wikipedia

3: Singleton Pattern en Wikipedia