TP - Influencias del paradigma funciona en OOP

Intro General

La idea de este TP es aplicar ciertos conceptos provenientes de influencias de lenguajes funcionales en el lenguaje Scala.
Así como en el TP anterior priorizamos el uso de mixins sobre otras soluciones como un strategy, a fin de ganar experiencia práctica con Mixins, acá también, vamos a priorizar el uso de estos elementos, como pattern matching, por sobre una solución que ya sabrían diseñar sin pattern-matching, con objetos “puro”.

Léase, no intenten diseñar una solución sólo con objetos. Intenten analizar en qué lugares sería más simple / conveniente el uso de pattern-matching, functiones de orden superior, etc.

Intro al TP

El dominio de este TP es un poco “raro” porque va a ser un lenguaje de programación. O sea.. los elementos de nuestro programa no van a ser un Arma, o Guerrero, etc.. sino que van a ser los conceptos de un lenguaje de programación.
Como si estuviéramos implementado un lenguaje.
Para los que tengan un poquito de conocimientos de teoría de lenguajes, nuestro programa va a ser como un analizador de un AST (AST: es el modelo de objetos o estructura que sale de parsear un programa).

El lenguaje

Nuestro lenguaje va a ser un forma limitada de un lenguaje de programación estructurada. Es decir con variables, asignaciones, operaciones y funciones.
Pero vamos a limitar un poco esto.

Vale aclarar que no tienen que implementar un lenguaje completo. No va a ser algo ejecutable (salvo bonus). Tampoco tienen que implementar un Parser, o cualquier otro elemento de desarrollo de lenguajes.

Entonces cómo se va a usar ?

Como estructuras, piensen que en los tests en lugar de instanciar un Guerrero y pasar la colección de armas.. van a instanciar un “If”, o un “Número”, o una Función.

Ejemplo

```
val suma = new Suma(new Numero(2), new Numero(3))
```

O usando apply

```
val suma = Suma(Numero(2), Numero(3))
```

Si fuera un lenguaje completo entonces también tendríamos un parser y una sintaxis específica, y nuestro usuario escribiría

“2 + 3”

Pero no nos queremos meter en esa problemática.
Entonces hacemos como que el parser ya pasó, y nos instanció a nuestros objetos.

Objetivo del TP

Como objetivos del TP vamos a trabajar en dos funcionalidades sobre los elementos del lenguaje:

- **Checkeos:** su programa va a recorrer y analizar el grafo de objetos y ejecutar reglas de checkeos. Por ejemplo imaginen una Division(Numero(3), Numero(0)).. debería ser analizada y arrojar un error porque no se puede dividir por cero. Ojo, esto no involucra ejecutar la division, ni el programa. Simplemente involucra analizar la estructura del programa (nuestros objetos Suma, Division, Numero, etc) y detectar problemas.
- **Refactors:** con el mismo espíritu de los checkeos, esto se trata de un programa que en lugar de analizar, realiza cambios en nuestro modelo de objetos. Por ejemplo si encuentro If(Booleano(true), … // lado positivo, …. // lado negativo), modifica el programa para eliminar el if, y sólo dejar la parte positiva.
- **Interpretación/Ejecución:** este tercer caso recorre también el grafo de objetos pero no para checkear ni modificar sino para “ejecutar” el programa. Por ejemplo, dada el grafo que representa “3 + 2”, sumara dichos números para retornar el resultado

Vamos a dividir el TP en puntos para hacerlo en forma incremental

1 - Números y operaciones

Vamos a arrancar modelando un *Programa* como una serie de *Elemento's*. Por ahora estos elementos van a ser los siguientes:

Modelar los **Números**, enteros (no vamos a usar doubles u otras variantes, para no complicarla).
Un número se puede:

- **operar**: sumar, restar, dividir, multiplicar.
- **comparar**: mayor, menor, igual, distinto, menorOIgual, mayorOIgual

Modelar estas operaciones también como elementos de nuestro programa (ojo no significa agregarle métodos "sumar", "restar", etc a la clase Numero, sino que significa modelar la idea que un Programa puede tener una Suma, Resta, etc.. son "instrucciones" o "expresiones".

Ejemplo

Programa(Sumar(Numero(2), Numero(3))

Ese es un programa que se representa lo siguiente

2 + 3

a) Checkeos

Codificar un ***Checkeador*** al que le pasemos diferentes programas y evalúe reglas.
El output del checkeador será una lista de *Problema*'s encontrados.
Un problema tendrá:

- una descripción: por ejempleo "No se puede dividir por cero"
- una gravedad: Error / Advertencia
- el elemento del programa que tiene el problema, por ejemplo Division.

A continuación las primeras reglas a implementar:

- **División por cero => ERROR**
- **Operación redundantes: => todas ADVERTENCIAS**
    - restar 0
    - sumar 0
    - multiplicar por 1
    - dividir por 1
- **Comparaciones sin sentido**: => ADVERTENCIAS O ERRORES (como prefieran)
    - 2 Igual a 2
    - 3 < 4 => siempre da true !
    - 3 > 4 => siempre da false.

Atención !! queremos que el checkeador sea “genérico”, es decir que no tenga las reglas “fijas”, en su lugar un Checkeador será como un “motor de reglas” que tiene la lógica de recorrer los elementos y evaluar Reglas. Esas reglas se pueden agregar al Checkeador, de modo de configurarlo según el conjunto de checkeos que querramos que haga.
Es decir que el Checkeador no tendrá la lógica de los checkeos sino que delega esto en otros objetos.

b) Refactors

Dada una **operación redundante** (punto anterior) hacer un refactor que la simplifique.
Ejemplo

```
Programa(Suma(Numero(4), Numero(0)))  ===>  Programa(Numero(4))
```

Lo mismo para las **comparaciones sin sentido**.

```
Programa(MayorA(Numero(3), Numero(1)))  ===>  Programa(True)
```

Notarán que para operar con comparaciones necesitamos introducir los Booleanos, True, y False

c) Ejecución

Realizar un tercer caso de uso de nuestro lenguaje (ya tenemos un checkeador, un “motor” de refactors y ahora..), que sea la ejecución, es decir un “Interprete”
Este será n objeto al cual le pasamos el programa, y lo ejecuta, retornando un resultado.

```
val programa = Programa(Suma(Numero(2), Numero(3)))
ejecutador.ejecutar(programa)

// retorna Numero(5)
```

Para eso deberá ir recorriendo cada elemento del programa, y cada elemento involucra diferentes “lógicas”. Por ejemplo la lógica de la Suma sería

```
Suma(Numero(a),Numero(b)) => Numero(a + b)
```

2 - Variables

Vamos a **agregar variables** a nuestro lenguaje
Una variable tiene un nombre, y un valor (que se puede dar al inicializarla, o no)

```
Variable(‘edad’, Numero(27))

Variable(‘altura’)  // sin inicializar
```

Entonces un programa ahora puede tener variables además de números y operaciones.
Para limitar el alcance nuestras variables serán sólo de tipo Número o Booleano

Luego, una variable se puede **asignar**, y esta es una sentencia nueva de nuestro lenguaje. Para asignarla, sin embargo no usamos el mismo objeto Variable, sino otro elemento nuevo en nuestro lenguaje que es una **Referencia.** Esto es, justamente hacer referencia a una variable por su nombre.

Siguiendo el ejemplo de altura

```
Variable(‘altura’)
Asignar(Referencia(‘altura’), Numero(172))
```

Por último podemos utilizar una variable. Dónde ? bueno, en todas las operaciones y comparaciones que implementamos en el punto 1.
Cómo ? utilizando como en este caso “Referencias”

Ejemplo

```
Variable(‘anioActual’),
Variable(‘edad’, Numero(25))

Variable(‘fechaNacimiento’,
     Sumar(Referencia(‘anioActual’), Referencia(‘fechaNacimiento’)
)
```

En este caso la *Suma* en lugar de recibir *Numero*, vemos que puede recibir una *Referencia* a una Variable.
Obvio que se pueden combinar valores y variables

```
Sumar(edad, Numero(1))     // retorna la suma de edad + 1
```

Con esto vemos que es necesario encontrar una abstracción común entre los Numero's y Booleanos' y las Referencias. En particular todos se pueden evaluar a un valor. Podemos decir que son Evaluables, o Valores.

Ahora sí ya podemos implementar checkeos sobre las variables

- a) Detectar cuando una variable está duplicada (dos con el mismo nombre)
- b) Detectar cuando una variable se usa "antes" de su declaración.
- c) Detectar cuando una variable se declara y nunca se usa.
- d) Checkear que una Referencia sea válida, es decir que exista la variable con ese nombre. Ejemplo, si tengo Variable("nombre"), y luego hago Referencia("nome") . Esa referencia debería fallar porque no existe la variable "nome"
- d) Detectar cuando una variable se usa, pero nunca se asigna.
- **Ejecución**
- e) Hacer que la "ejecución" del lenguaje (bonus del punto 1) también incluya la ejecución de las variables.

Bonus

3 - Funciones

Agregaremos la idea de funciones.
Esto tiene dos partes, por un lado la declaración de una función, y por el otro su uso.

a) Declaración de funciones

Para declarar una función especificamos su

- nombre
- nombre de sus parámetros
- cuerpo de la función

Junto a la función aparece una instrucción nueva del lenguaje, el "return" para cortar la ejecución y retornar un valor

Ejemplo entonces de una función muy simple

```
Funcion('miSuma', List(Parametro('a'), Parametro('b')),
       List(
            Retornar(Sumar(Referencia('a'),Referencia('b')))
       )
)
```

Esto declara una función con 2 parámetros “a” y “b”.
El cuerpo de la función es simplemente retornar a + b.

Los componentes de Funcion serían:

```
Funcion(nombre, listaDeParametros, listaDeInstrucciones)
```

Si tuviéramos un parser y una “sintaxis” para nuestro lenguaje, esto se podría escribir así

```
funcion miSuma(a, b) {
    retornar a + b
}
```

Checkeos

Ahora sí, sobre esto tendremos los siguientes checkeos

i) La función debe retornar algo. (debe existir un Retornar)
ii) No debe haber ninguna instrucción luego del retorno
iii) Levantar un warning en caso en que un parámetro no se utiliza dentro de la función.
iv) Los nombres de los parámetros no se pueden repetir
v) Dentro de la función no se puede referenciar una variable definida fuera de ella. O sea, sólo se pueden utilizar variables locales (definidas en la función misma) o alguno de sus parámetros. Nótese que ahora Referencia puede referirse tanto a Variable’s como a Parametro’s.

Ejecución

Implementar la ejecución de las funciones

b) Llamado a funciones

Implementar el llamado a funciones que será una expresión, es decir es algo que “genera un valor”.
Llamar a una función puede involucrar pasarle parámetros
Para nuestro ejemplo anterior un uso válido sería:

```
Variable(‘uno’, Numero(1))
Variable(‘tres’, Numero(3))
Variable(‘suma’)
val llamada = LlamadaFuncion(‘miSuma’, List(
                  Referencia(‘uno’),
```

```
                    Referencia('tres')
                ))
 AsignarVariable('suma', llamada)
```

Este programa declara dos variables uno y tres con esos valores numéricos. Con ellos llama a la función y luego asigna el resultado a otra variable nueva llamada “suma”.
Es conceptualmente similar a:

```
variable uno = 1
variable tres = 3
variable suma
suma = miSuma(uno, tres)
```

Checkeos

Ahora sí podemos implementar los checkeos:

1. Que la función con ese nombre exista.
2. Que el número de argumentos que pasamos a la función sea el mismo que el que declara la función como argumentos. O sea, que sea correcto.

4 - If

Agregar un “If” a nuestro lenguaje.
El if tiene 3 partes:

- la condición: una expresión booleana
- las sentencias por el lado positivo
- las sentencias por el negativo

Ejemplo

```
Variable('mayores', Numero(0) )
Variable('menores', Numero(0) )

Variable('edad', … )
If ( MayorA(Ref('edad'), Numero(18)),   // condicion
      // positivo
      Asignar(Ref('mayores'), Sumar(Ref('mayores'), Numero(1))),
      // negativo
      Asignar(Ref('menores'), Sumar(Ref('menores'), Numero(1)))
 )
```

Este programa, dada una “edad” si es > 18 incrementa la variable “mayores”, sino, la “menores”.

(achicamos el nombre Referencia -> Ref para que se vea mejor)

A) Checkeos

Los checkeos que deberán agregar son:

- El elemento que se le pasa como condición al IF debe ser de tipo Booleano. Ya sea una comparación (operación booleana) o bien una referencia a una variable Booleana)
- Cuando se usa dentro de una función, hay que modificar el checkeo para asegurarse que una función siempre retorna un valor, de modo de que el return pueda estar dentro del if. Así por ejemplo este programa no sería válido

```
if (condicion)
     hagoAlgo
     return 2
else
   hagoOtraCosaPeroNoRetorno
```

B) Ejecución

Implementar la ejecución del if.