L E . P . I . F . Y . O

Lenguaje Para Introducción a Funcional y Objetos

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, funciones de orden superior, etc.

Intro al TP

El dominio de este TP es un poco "raro" porque consiste en modelar un lenguaje de programación. Es decir, los elementos de nuestro programa no van a ser un Arma, o Guerrero, sino que serán los conceptos y estructuras propias 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 similar a un analizador de un AST (AST o *Abstract Syntax Tree*: es el modelo de objetos o estructura que resulta de parsear un programa).

El lenguaje

Nuestro lenguaje va a ser un forma limitada de un lenguaje de programación estructurada. Es decir un lenguaje con variables, asignaciones, operaciones y funciones, excepto que algo limitado.

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 **Numero**, o una **Funcion**.

Ejemplo

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

O usando apply / case clases:

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

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

```
2 + 3
```

Y el parser lo transformaría en la construcción de más arriba. Pero no nos queremos meter en esa problemática. Entonces pretenderemos que el parser ya se ha ejecutado y que nos instanció nuestros objetos.

Objetivo del TP

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

- **Chequeos:** su programa va a recorrer y analizar el grafo de objetos y ejecutar reglas de chequeos. Por ejemplo imaginen una *Division(Numero(3), Numero(0))*. La misma debería ser analizada y arrojar un error porque no se puede dividir por cero. Ojo, esto no involucra ejecutar la división 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 chequeos. Se trata de un programa que en lugar de sólo analizar realizará 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 chequear ni modificar sino para "ejecutar" el programa. Por ejemplo: dado el grafo que representa "3 + 2" sumará 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 *Elementos*. 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

Deberemos modelar estas operaciones también como elementos de nuestro programa. Cuidado: 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(Suma(Numero(2), Numero(3)))
```

Ese es un programa que se representa lo siguiente:

```
2 + 3
```

1a. Chequeos

Codificar un ***Chequeador*** al que le pasemos diferentes programas y evalúe reglas.
El output del Chequeador será una lista de *Problema*s encontrados.

Un problema tendrá:

- una descripción: por ejemplo, “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 redundante:** => ADVERTENCIA
    - restar 0
    - sumar 0
    - multiplicar por 1
    - dividir por 1

- **Comparaciones sin sentido**: => ADVERTENCIA O ERROR (como prefieran)
    - 2 Igual a 2
    - 3 < 4 => siempre da true.
    - 3 > 4 => siempre da false.

Atención!
Queremos que el Chequeador sea “genérico”, es decir que no tenga las reglas “fijas”, en su lugar un chequeador será como un “motor de reglas” que tiene la lógica de recorrer los elementos y evaluar Reglas. Esas reglas se podrán agregar al Chequeador para configurarlo según el conjunto de chequeos que queremos que haga.

Es decir que el Chequeador no deberá tener la lógica de los chequedor sino que delega esto en otros objetos.

1b. Refactors

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

Ejemplo:

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

Deberá ser refactorizado en:

```
Programa(Numero(4))
```

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

```
Programa(MayorA(Numero(3), Numero(1)))
```

Deberá ser refactorizado en:

```
Programa(True)
```

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

1c. Ejecución

Un tercer caso de uso de nuestro lenguaje. Ya tenemos un chequeador y un "motor" de refactors , ahora deberemos crear un *Interprete* de ejecución

El *Interprete* será un objeto al cual le pasamos el programa para que lo ejecute 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 involucrará 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.

```
Variable("edad", Numero(27))
```

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.

Una variable puede utilizarse en otras construcciones de nuestro programa por medio de una *Referencia* que modela justamente el hecho de hacer referencia a una variable por su nombre. En otras palabras: Una variable (por medio de su referencia) podrá ser usada en todas las operaciones y comparaciones que implementamos en el punto 1.

Ejemplo:

```
Variable("anioActual", Numero(2018)),
Variable("edad", Numero(25))
Variable("anioNacimiento", Resta(Referencia("anioActual"), Referencia("edad")))
```

En este caso la *Resta* en lugar de recibir *Numero* vemos que puede recibir una *Referencia* a una variable. Obvio que se pueden combinar valores y variables en una misma expresión:

```
Suma(Referencia("edad"), Numero(1))  // retorna la edad + 1
```

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

Adicionalmente, una variable puede ser creada sin inicializarse.

```
Variable("altura")  // sin inicializar
```

Las variables podrán ser **asignadas**. La asignación será una nueva sentencia de nuestro lenguaje. Para asignarla usaremos las Referencias que antes creamos.

Siguiendo el ejemplo de altura

```
Variable("altura")  // sin inicializar
Asignar(Referencia("altura"), Numero(172))
```

Chequeos

Ahora sí ya podemos implementar nuevos chequeos sobre las variables:

- Detectar cuando una variable está duplicada (dos variables con el mismo nombre)
- Detectar cuando una variable se usa “antes” de su declaración.
- Detectar cuando una variable se declara y nunca se usa.
- Chequear 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”
- Detectar cuando una variable se usa, pero nunca se asigna.

Ejecución

Deberemos hacer que la “ejecución” del lenguaje 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.

3a. 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.

Un ejemplo entonces de una función muy simple:

```
Funcion("miSuma", List(Parametro("a"), Parametro("b")),
    List(
        Retornar(Suma(Referencia("a"), Referencia("b")))
    )
)
```

Esto declara una función con 2 parámetros *"a"* y *"b". El cuerpo de dicha función* es simplemente retornar a + b.

Los componentes de una función 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
}
```

Chequeos:

Ahora sí, sobre esto tendremos los siguientes chequeos:

- Toda función debe retornar algo (debe existir un Retornar).
- No debe haber ninguna instrucción luego del retorno.
- Advertir en caso en que un parámetro no se utilizado dentro de la función.
- Los nombres de los parámetros no se pueden repetir.
- Dentro de la función no se puede referenciar una variable definida fuera de ella. Sólo se puede referenciar a variables locales (definidas en la función misma) o a alguno de sus parámetros. Nótese que ahora Referencia puede referirse tanto a Variables como a Parametros.

Ejecución

Implementar la ejecución de las funciones

3b. Llamado a funciones

Deberemos implementar el llamado a funciones. El mismo 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("resultadoSuma")

val llamada = LlamadaFuncion("miSuma", List(Referencia("uno"), Referencia("tres"))
AsignarVariable("resultadoSuma", 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:

```
var uno = 1
var tres = 3
var resultadoSuma
resultadoSuma = miSuma(uno, tres)
```

Chequeos

Ahora sí podemos implementar los chequeos:

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.

4 - If

Agregar la sentencia *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(Referencia("edad"), Numero(18)),  //condicion
  Asignar(Referencia("mayores", Suma(Referencia("mayores"), Numero(1))), // si mayor
  Asignar(Referencia("menores", Suma(Referencia("menores"), Numero(1))), // si menor
)
```

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

Chequeos

Los chequeos 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 chequeo que se asegura que una función siempre retorna un valor de modo de que el return pueda estar dentro del if. Este programa, por ejemplo, no debería ser valido:

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

Ejecución

Implementar la ejecución del if.