Programación para la Inteligencia Artificial
Haskell - Práctica 3
Pruebas unitarias en Haskell

Las **pruebas unitarias** consisten en comprobaciones (manuales o automatizadas) que se realizan para verificar que el código correspondiente a un módulo concreto de un sistema software funciona de acuerdo con los requisitos del sistema.

Test-framework es un framework que permite realizar pruebas de software con valores generados mediante `QuickCheck` y casos de prueba `HUnit`.

Para usarlos, debemos tener instalado *Cabal*, y añadir al fichero desde el que hagamos las pruebas:

```
import Test.HUnit
import Test.QuickCheck
```

Pruebas unitarias con `HUnit`

`HUnit` es un framework para pruebas de unidad realizado para Haskell por Dean Herington en 2002, e inspirado por la herramienta de Java: `JUnit`.

Con `HUnit`, como con `JUnit`, podemos fácilmente crear test, nombrarlos, agruparlos y ejecutarlos con el marco de trabajo que valida los resultados automáticamente.

El programador especifica una serie de pruebas del tipo:

`assertEqual` "`nombre_prueba`" `<resultado> <funcion_a_probar>`

Luego se llama a la funcion que ejecuta las pruebas:

`runTestTT <pruebas>`

que muestra por pantalla los resultados de las mismas.

Tipo de dato `Test`

El tipo de dato `Test` es el tipo fundamental del modulo `HUnit`. A partir de él es posible definir los test que se van a realizar. El tipo `Test` está definido de la siguiente manera:

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

Algunos de los constructores de Test son:

- `TestCase` representa una unidad de ejecución de test. Cada uno de estos casos de test son independientes entre sí, por lo que el fallo de uno es independiente del fallo de cualquier otro. Un `TestCase` consiste en uno o varios `Assertion`.

- `TestList` es una lista de `Test` del tipo `TestCase`. Este tipo debe de incluir el conjunto de `TestCase` que se desea ejecutar.

- `TestLabel` nos permite asignar un nombre a un `TestCase` dentro de un `TestList`.

Tipo `Assertion`

En la base del funcionamiento de `HUnit` se encuentran las "aserciones" o *assertions*. Estas se combinan para crear los `testCase` (caso de prueba) que a su vez se agrupan para formar `Test`.

Las aserciones se emplean para asociar a parte de código Haskell un posible valor. En el caso de no cumplirse, se genera un error que mostrará el pertinente mensaje cuando se ejecute la batería de pruebas.

`Assertion` es del tipo `IO()`.

Para indicar cuándo entendemos que se ha producido un fallo, se emplean las siguientes funciones:

- `assertBool :: String -> Bool -> Assertion`

  Mostrará un mensaje de fallo si falla la condición dada.

- `assertString :: String -> Assertion`

  Mostrará un mensaje de fallo si la expresión que le pasamos devuelve algo (no nulo).

- `assertEqual :: (Eq a, Show a) => String -> a -> a -> Assertion`

  Mostrará un mensaje de fallo si la ejecución de la función dada no coincide con los valores proporcionados. Es la más frecuente.

Tipo `Count`

Durante la ejecución de las pruebas, cuatro *counts* de casos de prueba se tienen en cuenta:

```
data Counts = Counts { cases, tried, errors, failures :: Int }
        deriving (Eq, Show, Read)
```

- `cases` es el número de casos de prueba incluidos en el test.
- `tried` es el número de casos de prueba que han sido ejecutados hasta ahora durante la ejecución de los test.
- `errors` es el número de casos de prueba cuya ejecución finalizó con una excepción inesperada.
- `failures` es el número de casos de prueba cuya ejecución termina en fallo.

Ejecución de Test

HUnit posee una función muy sencilla para el uso más común del controlador de test basado en texto:

```
runTestTT :: Test -> IO Counts
```

`runTestTT` invoca a otra función `runTestText`, que devuelve el *counts* final de la ejecución del test.

Ejemplo

```
module Main where
import Test.HUnit

-- Funciones que se van a probar
module Main where
import Test.HUnit

-- Funciones que se van a probar
f1 :: Int -> (Int, Int)
f1 x = (1, x)

f2 :: Int ->  (Int, Int)
f2 v = (v+2, v+3)

f3 :: Int ->  Bool
f3 v = (v > 5)

f4 :: Int -> String -> String
f4 n s = if (f3 n) then "" else s

--Creación de los Test
test1,test2,test3,test4 :: Test
test1 = TestCase (assertEqual "Fallo con (f1 3)." (1,3) (f1 3))
test2 = TestCase (assertEqual "Fallo en primer elemento de f2 3." 5 (fst(f2 3)))
test3 = TestCase (assertBool "Fallo de prueba con f3." (f3 (snd(f2 3))))
test4 = TestCase (assertString  (f4 6 "Resultado incorrecto con f4."))

tests :: Test
tests = TestList [TestLabel "test1" test1, TestLabel "test2" test2, TestLabel "test3" test3,TestLabel "test4" test4]

--Ejecución del test
main :: IO Counts
main = runTestTT tests
```

Ejercicio 1

Usando la librería `Pitures.hs`, definir una función

`chessG :: Picture -> Picture -> Picture`

que dibuje un ajedrez de dimensiones fijas, pero que permita introducir como variables las imágenes para las casillas blancas y negras.

Además, definir una función:

`tablero :: Int -> Picture -> Picture -> Picture`

que dibuje un tablero de tipo ajedrez de dimensiones fijas pero con las casillas redimensionadas según un parámetro de entrada. Es decir, `tablero 3 white black`, devolvería un tablero en el que cada casilla sería la composición de "white" o de "black" en una nueva imagen 3×3.

Ejercicio 2

Comprobar las propiedades de las funciones definidas usando `HUnit`. Por ejemplo, añadir test para comprobar:

- Que `tablero 1 white black` devuelve lo mismo que `chessG white black`.
- Que la creación de un tablero con las imágenes invertidas coincide con el resultado de aplicar al tablero completo una inversión de color.

- Que el resultado de dar la vuelta en horizontal o vertical a un tablero es igual a su imagen invertida.
- Cualquier otra comprobación que se te ocurra según las funciones que hayas definido.

Testeo de programas basado en propiedades con `QuickCheck`

Para saber que un programa ha sido escrito correctamente tenemos varias opciones:

- Comprobar que ante ciertas entradas la salida es la esperada. Por ejemplo, podemos esperar que rotar la imagen del caballo en horizontal y luego en vertical, debe devolver la misma imagen que si rotamos primero en vertical y luego en horizontal. Del mismo modo, rotarla dos veces en vertical o dos veces en horizontal, debe devolvernos la imagen original. Esto podemos expresarlo en los siguientes test:

  ```
  test_rotate, test_flipV, test_flipH ::  Bool
  test_rotate = flipV (flipH horse) == flipH (flipV horse)
  test_flipV = flipV (flipV horse) == horse
  test_flipH = flipH (flipH horse) == horse
  ```

  de manera que deben evaluarse a `True` las tres funciones, mientras que

  ```
  test_flipH = flipH (flipV horse) == horse
  ```

  debe evaluarse a `False`.

- Los tests anteriores trabajan sobre casos concretos, y a nosotros puede interesarnos probar una propiedad sobre una colección de entradas. Haskell permite formalizar propiedades como funciones y nos da una herramienta para hacer un test con una batería de entradas generadas de forma aleatoria. De esta forma, podemos poner:

  ```
  prop_rotate, prop_flipV, prop_flipH ::  Picture -> Bool
  prop_rotate pic = flipV (flipH pic) == flipH (flipV pic)
  prop_flipV pic = flipV (flipV pic) == pic
  prop_flipH pic = flipH (flipV pic) == pic
  ```

  Para ejecutar la prueba, podemos poner:

  ```
  *Main> quickCheck prop_rotate
  ```

  de manera que se evaluarán hasta 100 entradas aleatorias, indicando en este caso, que la comprobación es correcta. Esto no ocurriría si evaluamos

  ```
  *Main> quickCheck prop_flipH
  ```

- Una comprobación como la anterior podría no incluir un caso en el que realmente la propiedad no se cumpliera. Para cubrir todos los casos se necesitaría una demostración formal de que la propiedad se cumple. Aunque las demostraciones son posibles en todos los lenguajes de programación, son sustancialmente más fáciles en los lenguajes funcionales que en el resto de los paradigmas.

Ejercicio 3

Basándote en lo anterior, y usando de nuevo la librería `Pictures.hs`, escribe una propiedad para comprobar que el resultado de rotar en vertical la figura obtenida al poner una imagen encima de otra, es el mismo que el que se obtiene cuando se rotan las imágenes y luego se juntan una sobre la otra. Probarlo también para el caso de rotación horizontal.