Haskell - Unit Testing usando Hspec

Pre-requisitos

Antes que nada, tener instalado Haskell GHC o similar o bien utilizar Stack. Luego, como vamos a trabajar con un framework de testing llamado Hspec (que se para sobre HUnit) el cual no viene con Haskell, **si vas a trabajar sin stack** tendremos que instalarlo a mano en dos simples pasos:

1. Abrir la consola del sistema operativo.
2. Ejecutar estos comandos comandos[1]:
    a. *cabal update*
    b. *cabal install hspec*

Conceptos básicos

Para usar las funciones de la biblioteca de Hspec, necesitamos importarla a nuestro sistema poniendo al inicio de nuestro código:

```
import Test.Hspec
```

<poco-importante>
Este framework nos otorga 4 funciones muy copadas para crear nuestros tests:

```
hspec :: Spec -> IO ()
```
Es la función principal a la que llamamos para correr toda una pila de tests.

```
describe :: String -> SpecWith a -> SpecWith a
```
Sirve para agrupar tests bajo un mismo nombre al correrlos para una mejor organización.

```
it :: Example a => String -> a -> SpecWith (Arg a)
```
Es la función que usamos para crear cada test; espera el nombre del test y el test en sí, y nos devuelve el output que veremos por pantalla (**Rojo** o **Verde**).

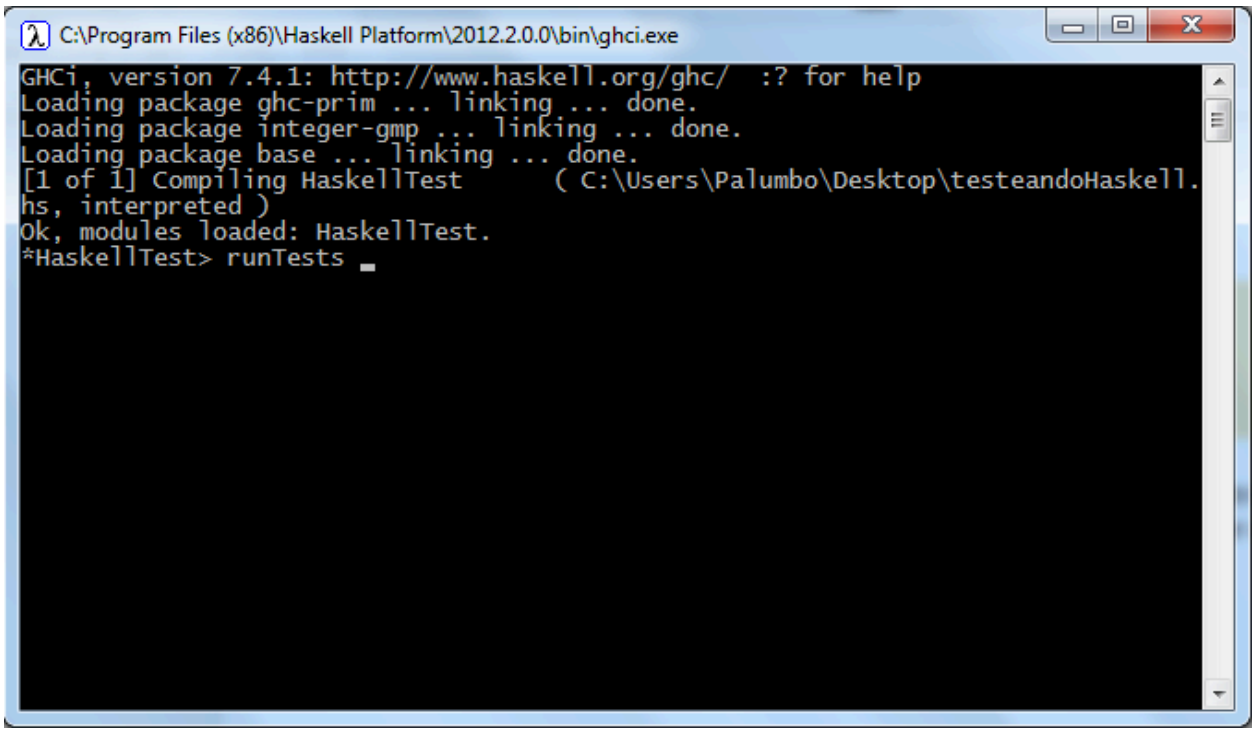
```
shouldBe :: (Show a, Eq a) => a -> a -> Expectation
```
El *assert* básico de igualdad.
</poco-importante>

[1] En los cursos donde se utiliza un proyecto basado en stack este paso no es necesario. Simplemente vayan a la definición del test donde les indiquen. Por ejemplo: `src\Spec.hs`

¡Ya podemos crear nuestro primer test!
Probemos que "1 + 1 es 2" (ese String sería una descripción de lo que se está probando, es importantísimo que sea expresivo, en este caso es difícil pensar en algo que no esté atado a la implementación del test pero podríamos llevarlo a algo como "la suma básica funciona"):

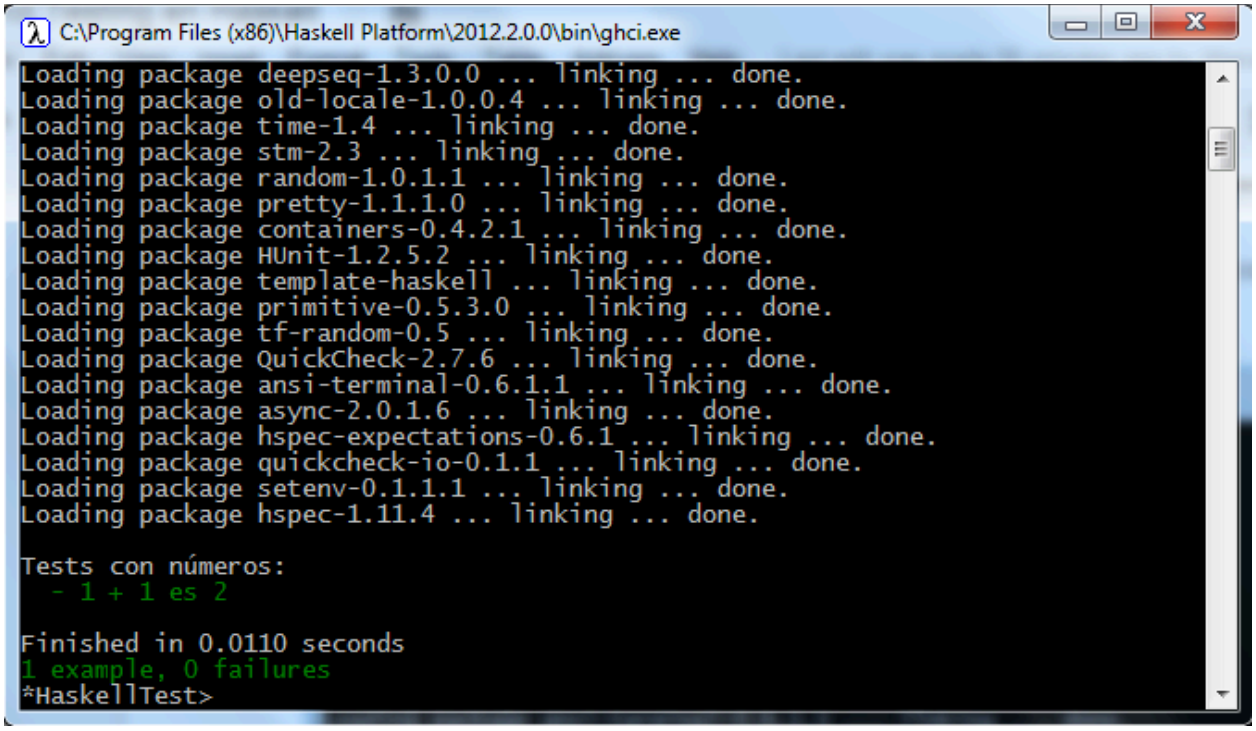
```
runTests[2] = hspec $ do
  describe "Tests con números:" $ do
     it "1 + 1 es 2" $ do
       1 + 1 `shouldBe` 2
```

Así cuando cargamos nuestro programa y verificamos que todo se haya cargado correctamente, podemos llamar a nuestra función *runTests* para correr el test que hicimos:



Al correr una función de *Hspec* por primera vez veremos por consola que se compilan los módulos necesarios. Luego veremos el resultado de nuestro test:

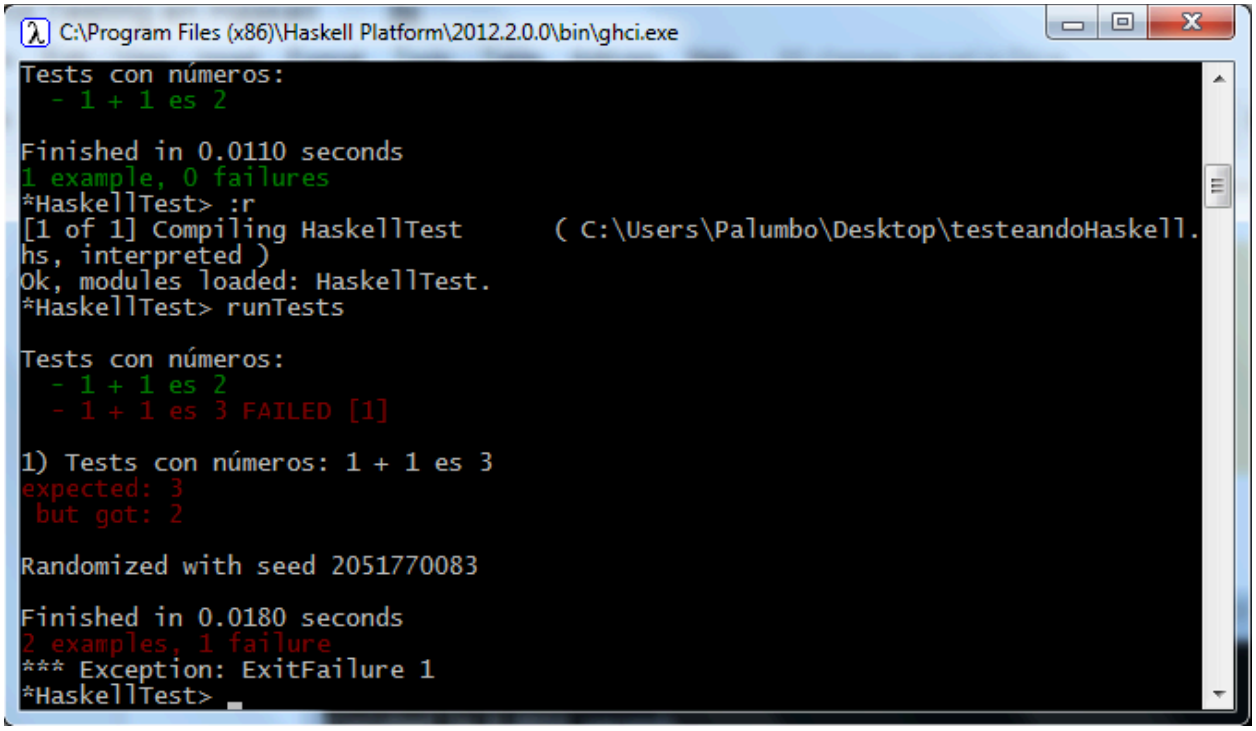
[2] Si estás usando stack test para correr los tests, reemplazá la función *runTests* por *main*



VERDE pasó correctamente! 😎
Bien, ahora hagamos un test falso para ver que todo funciona como esperamos:

```
runTests = hspec $ do
  describe "Tests con números:" $ do
    it "1 + 1 es 2" $ do
      1 + 1 `shouldBe` 2
    it "1 + 1 es 3" $ do
      1 + 1 `shouldBe` 3
```

Recargamos y ejecutamos *runTests*:



ROJO. Podemos apreciar que en caso que un test falle, *Hspec* nos da información copada para saber el por qué. :D

¡¡Ya estamos listos para testear nuestros programas!!

Buenas prácticas

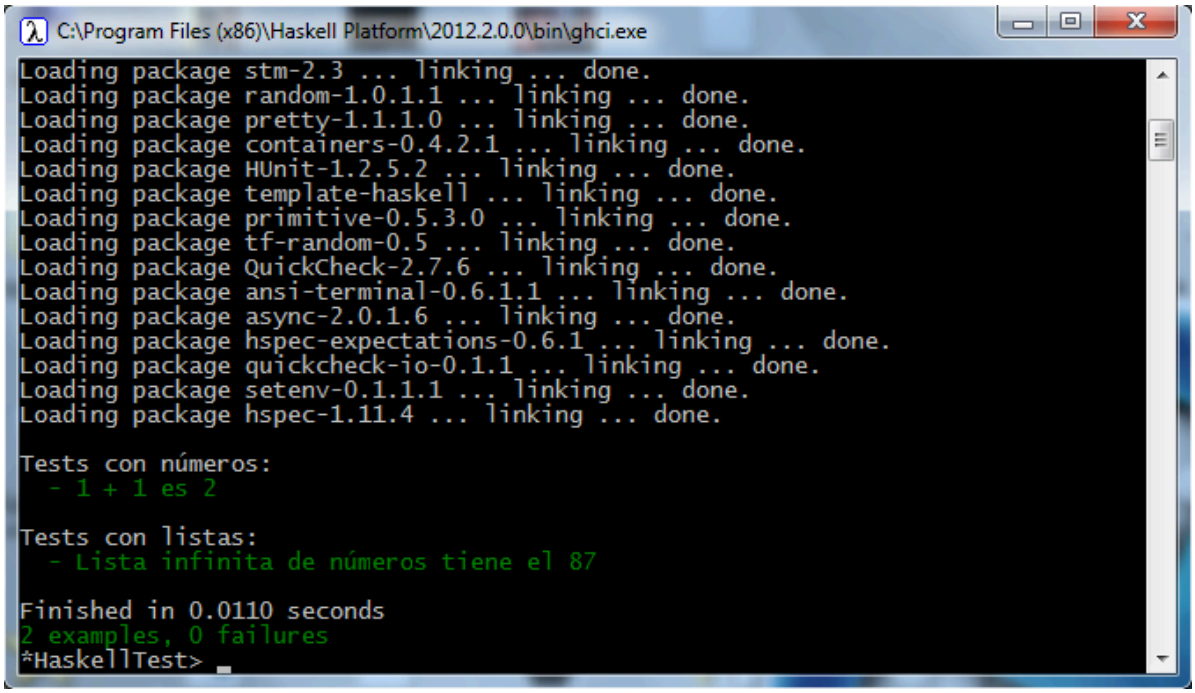
Si queremos testear si una lista infinita de números contiene al 87, por lo que vimos hasta ahora, podríamos hacer (como ahora testeamos listas, podemos escribirlo bajo otro *describe*):

```
describe "Tests con listas:" $ do
  it "Lista infinita de números tiene el 87" $ do
     elem 87 [1,2..] `shouldBe` True
```

Pero sabemos que comparar *Bool*'s mucho no nos gusta. Para evitar eso podemos usar **`shouldSatisfy`** [3]`:: Show a => a -> (a -> Bool) -> Assertion`

Que espera un *a* y una *condición* que tenga que cumplir. Así podemos reescribir nuestro test:

```
runTests = hspec $ do
  describe "Tests con números:" $ do
     it "1 + 1 es 2" $ do
        1 + 1 `shouldBe` 2
  describe "Tests con listas:" $ do
     it "Lista infinita de números tiene el 87" $ do
        [1,2..] `shouldSatisfy` elem 87
```



[3] Pueden encontrar muchos assertions útiles acá. Sobre los tipos, ignoren la parte de HasCallStack.

Testeo de errores

A veces vamos a querer verificar que nuestro programa falle. En ese caso esperamos que haya un error, lo que indica que nuestro test pasa ok:

```
runTests = hspec $ do
  describe "Tests con listas:" $ do
           it "Una lista vacía no tiene primer elemento" $ do
              evaluate (head []) `shouldThrow` anyException
```

```
Tests con listas:
  Una lista vacía no tiene primer elemento

Finished in 0.0014 seconds
1 example, 0 failures
```

En caso de no haber error, el test fallará.

Para poder utilizar la función evaluate, debemos importarla al comienzo de nuestro módulo de test. Como Control.Exception es el paquete que contiene la función evaluate, hay dos formas de hacer el import

```
import Control.Exception (evaluate)
```

importa únicamente la función evaluate. En cambio

```
import Control.Exception
```

importa todas las funciones de ese módulo. En el caso de que necesitemos utilizar más funciones puede ser útil, la desventaja que trae hacer estos imports es que nuestro "ejecutable" (el compilado de los tests y el archivo principal con el código) queda más grande.

Para más información recomendamos que leas este material.