Haskell - Unit Testing usando Hspec

Pre-requisitos

Antes que nada, tener [instalado](http://pdep.com.ar/Home/software/software-haskell) Haskell GHC o similar.

Luego, como vamos a trabajar con un framework de testing llamado [Hspec](https://hackage.haskell.org/package/hspec-0.3.0/docs/Test-Hspec.html) (que se para sobre [HUnit](https://docs.google.com/document/d/19U4YytEH2Rfb7lGqm775umPLz2bjYW8qQLRhk-gfXb4/edit#)) el cual no viene con Haskell, tendremos que instalarlo a mano en dos simples pasos:

1. Abrir la consola del sistema operativo.
2. Ejecutar estos comandos comandos:
   1. *cabal update*
   2. *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** :: [IO](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/System-IO.html#t:IO) [[Spec](https://hackage.haskell.org/package/hspec-0.3.0/docs/Test-Hspec.html#t:Spec)] -> [IO](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/System-IO.html#t:IO) [()](https://hackage.haskell.org/packages/archive/ghc-prim/0.2.0.0/doc/html/GHC-Unit.html#t:-40--41-)

Es la función principal a la que llamamos para correr toda una pila de tests.

**describe** :: [String](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/Data-Char.html#t:String) -> [[IO](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/System-IO.html#t:IO) ([String](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/Data-Char.html#t:String), [Result](https://hackage.haskell.org/package/hspec-0.3.0/docs/Test-Hspec.html#t:Result))] -> [IO](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/System-IO.html#t:IO) [[Spec](https://hackage.haskell.org/package/hspec-0.3.0/docs/Test-Hspec.html#t:Spec)]

Sirve para agrupar tests bajo un mismo nombre al correrlos para una mejor organización.

**it** :: [SpecVerifier](https://hackage.haskell.org/package/hspec-0.3.0/docs/Test-Hspec-Internal.html#t:SpecVerifier) a => [String](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/Data-Char.html#t:String) -> a -> [IO](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/System-IO.html#t:IO) ([String](https://hackage.haskell.org/packages/archive/base/4.3.1.0/doc/html/Data-Char.html#t:String), [Result](https://hackage.haskell.org/package/hspec-0.3.0/docs/Test-Hspec.html#t:Result))

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](https://hackage.haskell.org/packages/archive/base/4.5.0.0/doc/html/Text-Show.html#t:Show) a, [Eq](https://hackage.haskell.org/packages/archive/base/4.5.0.0/doc/html/Data-Eq.html#t:Eq) a) => a -> a -> [Assertion](https://hackage.haskell.org/packages/archive/HUnit/1.2.4.2/doc/html/Test-HUnit-Lang.html#t:Assertion)

El *assert* básico de igualdad.

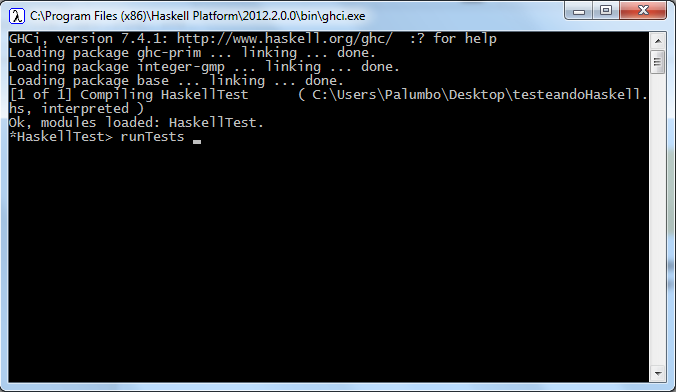
***</poco-importante>***

**¡Ya podemos crear nuestro primer test!**

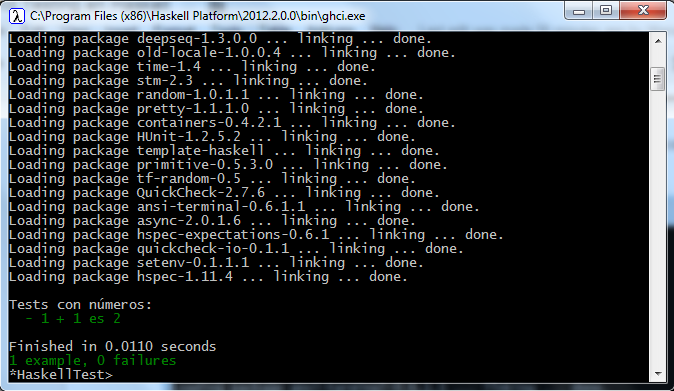
Probemos que “1 + 1 es 2” (ese String sería una descripción de lo que se está probando, es importantísimo que sea expresivo):

|  |
| --- |
| runTests = **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:

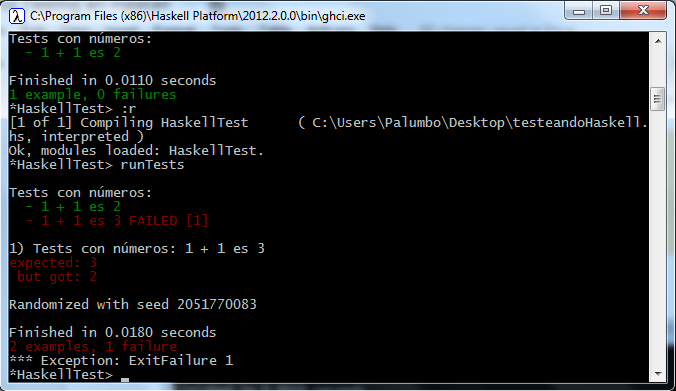


**VERDE** pasó correctamente! :D

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*):

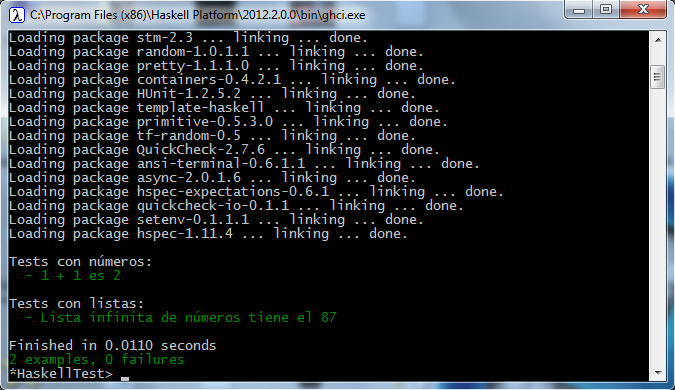
|  |
| --- |
| **describe** "Tests con listas:" $ **do**  **it** "Lista infinita de números tiene el 87" $ **do**  elem 87 [1,2..] `**shouldBe**` True |

Pero sabemos que comparar *Boo*l’s mucho no nos gusta. Para evitar eso podemos usar

**shouldSatisfy** :: [Show](https://hackage.haskell.org/packages/archive/base/4.5.0.0/doc/html/Text-Show.html#t:Show) a => a -> (a -> [Bool](https://hackage.haskell.org/packages/archive/base/4.5.0.0/doc/html/Data-Bool.html#t:Bool)) -> [Assertion](https://hackage.haskell.org/packages/archive/HUnit/1.2.4.2/doc/html/Test-HUnit-Lang.html#t: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 |



Testing avanzado

**<TO-DO>**

A veces vamos a querer verificar que nuestro programa falle.