Introducción a la Programación

Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2025

Departamento de Computación - FCEyN - UBA

HUnit: framework de Testing Unitario para Haskell

Formato de un caso de test en HUnit

```
"Nombre del test" ~: <res obtenido> ~?= <res esperado>
```

Donde:

- **res obtenido** es el valor que devuelve la función que queremos testear.
- **res esperado** es el valor que debería devolver la función que queremos testear.

Ejemplos:

```
"El doble de 4 es 8" ~: (doble 4) ~?= 8
"Maximo repetido" ~: (maximo [2,7,3,7,4]) ~?= 7
"esPar de impar" ~: (esPar 5) ~?= False
```

Ejercicio

Crear test unitarios para la función `fib: Int -> Int` que devuelve el i-ésimo número de Fibonacci.
Considerar la siguiente especificación e implementación en Haskell:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

```
problema fib (n: ℤ) : ℤ {
  requiere: { n ≥ 0 }
  asegura: { res = fib(n) }
}
```

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-1)
```

Modularizando el código

- Crear un módulo llamado `MisFunciones` con las funciones que queremos testear.
- El nombre del archivo debe coincidir con el nombre del módulo.

```
module MisFunciones where

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-1)
```

Módulo para los tests

- Crear otro módulo llamado `TestsDeMisFunciones` con los casos de tests.
- Ambos archivos deben estar guardados en la misma carpeta.
- Importar el módulo `Test.HUnit` para poder crear casos de test utilizando HUnit.
- Importar el módulo `MisFunciones` para poder utilizar las funciones allí definidas.

```
module TestsDeMisFunciones where

import Test.HUnit
import MisFunciones

-- Casos de test
```

Agregando casos de test

- Un test para cada caso base.
- Un test para el caso recursivo.

```
module TestsDeMisFunciones where

import Test.HUnit
import MisFunciones

-- Casos de test
run = runTestTT tests

tests = test [
    "Caso base 1: fib 0" ~: (fib 0) ~?= 0,
    "Caso base 2: fib 1" ~: (fib 1) ~?= 1,
    "Caso recursivo 1: fib 2" ~: (fib 2) ~?= 1
 ]
```

Corriendo los casos de test

Para correr los tests en `ghci` debemos:

- Cargar el archivo `TestsDeMisFunciones.hs`.
- Evaluar la función `run`.

```
ghci> run
### Failure in: 2:" Caso recursivo 1: fib 2"
TestsDeMisFunciones.hs:12
expected: 1
 but got: 2
Cases: 3  Tried: 3  Errors: 0  Failures: 1
```

Podemos ver que el ultimo test falló. Por qué?

Corrigiendo el error

Revisemos la implementación de la función `fib`.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-1) -- Aca esta el error
```

Debería restar 2 en lugar de 1. Lo corregimos, guardamos y recargamos.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Volvemos a correr los test.

```
ghci> run
Cases: 3  Tried: 3  Errors: 0  Failures: 0
```

Ahora todos los test son exitosos!

Guía repaso Haskell - Ejercicio 1

Una reconocida empresa de comercio electrónico nos pide desarrollar un sistema de stock de mercadería. El conjunto de mercaderías puede representarse con una secuencia de nombres de los productos, donde puede haber productos repetidos. El stock puede representarse como una secuencia de tuplas de dos elementos, donde el primero es el nombre del producto y el segundo es la cantidad que hay en stock (en este caso no hay nombre de productos repetidos). También se cuenta con una lista de precios de productos representada como una secuencia de tuplas de dos elementos, donde el primero es el nombre del producto y el segundo es el precio. Para implementar este sistema nos enviaron las siguientes especificaciones y nos pidieron que hagamos el desarrollo enteramente en Haskell, utilizando los tipos requeridos y solamente las funciones que se ven en la materia.

Guía repaso Haskell - Ejercicio 1

Implementar la función `generarStock :: [String] -> [(String, Int)]`

`problema generarStock` (mercaderia: $seq\langle String \rangle$) : $seq\langle String \times \mathbb{Z} \rangle$ {
  `requiere:` {True}
  `asegura:` { La longitud de res es igual a la cantidad de productos distintos que hay en $mercaderia$}
  `asegura:` {Para cada $producto$ que pertenece a $mercaderia$ existe un i tal que $0 \leq i < |res|$ y $res[i]_0$=$producto$ y $res[i]_1$ es igual a la cantidad de veces que aparece $producto$ en $mercaderia$}
}

Guía repaso Haskell - Ejercicio 1

Implementar la función `generarStock :: [String] -> [(String, Int)]`

`problema generarStock` (mercaderia: $seq\langle String\rangle$) : $seq\langle String \times \mathbb{Z}\rangle$ {
  `requiere:` {True}
  `asegura:` { La longitud de res es igual a la cantidad de productos distintos que hay en $mercaderia$}
  `asegura:` {Para cada $producto$ que pertenece a $mercaderia$ existe un i tal que $0 \leq i < |res|$ y $res[i]_0$=$producto$ y $res[i]_1$ es igual a la cantidad de veces que aparece $producto$ en $mercaderia$}
}

¿Qué casos tendríamos que testear? Generar HUnit. Algunos casos interesantes...

Guía repaso Haskell - Ejercicio 1

Implementar la función `generarStock :: [String] -> [(String, Int)]`

`problema generarStock` (mercaderia: $seq\langle String\rangle$) : $seq\langle String \times \mathbb{Z}\rangle$ {
  `requiere:` {True}
  `asegura:` { La longitud de res es igual a la cantidad de productos distintos que hay en $mercaderia$}
  `asegura:` {Para cada $producto$ que pertenece a $mercaderia$ existe un i tal que $0 \leq i < |res|$ y $res[i]_0$=$producto$ y $res[i]_1$ es igual a la cantidad de veces que aparece $producto$ en $mercaderia$}
}

¿Qué casos tendríamos que testear? Generar HUnit. Algunos casos interesantes...

- Lista vacía: []
- Muchos elementos distintos: [“clavo”, ”tuerca”, ”tornillo”]
- Un solo elemento repetido: [“clavo”, “clavo”, “clavo”, “clavo”]
- Varios con repeticiones, ordenado: [“clavo”, “clavo”, “clavo”, ”tuerca”, ”tuerca”]
- Varios con repeticiones, desordenado: [“clavo”, ”tuerca”, ”tornillo”, “clavo”, “clavo”, ”tuerca”]