

Laboratórios de Informática I

2022/2023

Licenciatura em Engenharia Informática

Ficha 5

Testes Unitários em *Haskell*

O desenvolvimento de testes é uma boa prática de programação. Existe mesmo quem programe de forma orientada aos testes, isto é, definindo primeiro os testes que o programa deve passar e só depois desenvolvendo o programa concreto.

A inclusão de testes no processo de desenvolvimento de software tornou-se tão comum que começaram a surgir *frameworks* específicas que lidam exclusivamente com testes.

Nesta aula vamos abordar a *framework* HUnit¹, que permite a especificação de testes unitários para a linguagem *Haskell*.

1 Framework de testes HUnit

Uma metodologia centrada nos testes usada no desenvolvimento de *software* é mais eficiente se os testes forem fáceis de criar, alterar e executar. Foi com este objetivo que foi desenvolvida a plataforma HUnit para a linguagem *Haskell*. A título de curiosidade, as *frameworks* de testes unitários para as demais linguagens de programação são, frequentemente, denominadas por xUnit.

Usando o HUnit, é possível criar testes unitários, atribuir-lhes designações, agrupá-los em suites de teste e ainda executá-los, sendo que o HUnit verifica os seus resultados automaticamente. Um teste unitário permite testar se uma dada unidade do programa (uma função) executa da forma esperada.

Nas próximas secções vamos explicar como obter e instalar o HUnit e ainda como utilizar a ferramenta para construir testes e executá-los.

1.1 Instalar o HUnit

O HUnit está disponível no *hackage* como um pacote da linguagem *Haskell*. Como tal, pode ser obtido de forma simples usando o gestor de pacotes *cabal*. Basta introduzir no terminal os comandos:

```
$ cabal update
$ cabal install --lib HUnit
```

¹hunit.sourceforge.net

1.2 Como escrever testes

Os testes são especificados de forma composicional. As *assertions* são combinadas para fazer um caso de teste e os caso de teste podem ser combinados em suites de teste.

1.2.1 *Assertions*

O bloco de construção básico de um teste é uma *assertion*. Uma *assertion* é uma instrução em que o seu conteúdo pode ser avaliado como **True** ou **False**. Intuitivamente, é possível observar que a construção de um teste vai ser então a construção de uma *assertion* para que esta possa ser verificada: caso seja avaliada como **True**, então o programa passou no teste e, caso seja avaliada como **False**, então o programa falhou nesse teste.

O HUnit disponibiliza três funções que permitem construir *assertions*:

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

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

Devolve uma *assertion* que verifica se a condição booleana que é fornecida é verdadeira. Caso contrário, devolve uma exceção que é descrita pela *string* que é passada como *input*.

`assertString :: String -> Assertion`

Devolve uma exceção que é descrita pela *string* que é passada como *input*.

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

A função `assertEqual` recebe como *input* uma mensagem a ser devolvida caso a *assertion* seja avaliada de forma negativa, um valor esperado e um valor real e compara os dois valores. Caso estes sejam diferentes, devolve uma mensagem de explicação do erro que tem como prefixo a *string* fornecida como *input*.

1.2.2 Testes

Quanto maior for o número de testes especificados, maior é a necessidade de os identificar. Do mesmo modo, é também maior a necessidade de os agrupar em conjuntos de teste. Esta propriedade composicional define um teste como um conjunto de casos de teste. Mais concretamente, um teste é um tipo *Haskell* com a seguinte forma:

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

Deste modo, um teste pode ser um caso de teste isolado ou uma lista de testes. O último construtor deste tipo define um teste que pode ser identificado por uma *label* (*string*).

É igualmente possível contar o número de casos de teste que são especificados usando a função

```
testCaseCount :: Test -> Int
```

Um teste (ou um conjunto de testes) pode ser executado usando a função `runTestTT`.

1.3 Exemplo

No módulo *Haskell* onde os testes vão estar definidos, é necessário importar o módulo `Test.HUnit`.

```
import Test.HUnit
```

Considere-se, a título de exemplo, a seguinte função que soma dois números inteiros:

```
mysum :: Int -> Int -> Int
mysum x y = x+y
```

Vamos, então, definir três testes para esta função. Um teste é definido usando um dos construtores mostrados anteriormente. Neste exemplo, vamos definir testes usando o construtor `TestCase`.

```
test1 = TestCase (assertEqual "for 1+2," 3 (mysum 1 2))
test2 = TestCase (assertEqual "for 5+5," 10 (mysum 5 5))
test3 = TestCase (assertEqual "for 100+100," 100 (mysum 100 100))
```

Note-se que o último caso de teste é claramente falso, já que $100+100 = 200$.

Podemos desde já executar os testes individualmente no `ghci`:

```
*Main> runTestTT test1
Counts {cases = 1, tried = 1, errors = 0, failures = 0}
*Main> runTestTT test2
Counts {cases = 1, tried = 1, errors = 0, failures = 0}
```

Como a função produz o resultado que é esperado, não ocorre nenhuma exceção nem é mostrada nenhuma mensagem de erro. Aquilo que a função `runTestTT` devolve é uma estrutura de dados que contém informação sobre o número de casos de teste, o número de casos testados, o número de erros obtido e o número de falhas. Já que as *assertions* produzidas são verdadeiras, o número de erros e de falhas é zero.

A execução do último teste resultaria no seguinte *output* de erro:

```
### Failure:
Desktop/test_hunit.hs:8
for 100+100,
expected: 100
but got: 200
Cases: 1 Tried: 1 Errors: 0 Failures: 1
Counts {cases = 1, tried = 1, errors = 0, failures = 1}
```

Portanto, caso uma *assertion* não seja válida (um teste não produza o *output* esperado), a mensagem de erro descreve a linha onde ocorre o erro, o prefixo da mensagem de erro que foi passado como parâmetro, o valor esperado e o valor obtido.

No entanto, tendo os casos de teste definidos, é possível agrupá-los sob a forma de uma lista usando o construtor `TestList`. Vamos, igualmente, identificá-los usando o construtor `TestLabel`.

```
tests = TestList [
    TestLabel "Teste 1 (1, 2)" test1,
    TestLabel "Teste 2 (5, 5)" test2,
    TestLabel "Teste 3 (100, 100)" test3
]
```

A função `runTestTT` pode também ser usada para executar uma suite de testes. Para o nosso exemplo, a função `runTestTT tests` devolveria:

```
### Failure in: 2:Teste 3 (100, 100)
Desktop/test_hunit.hs:8
for 100+100,
expected: 100
    but got: 200
Cases: 3   Tried: 3   Errors: 0   Failures: 1
Counts {cases = 3, tried = 3, errors = 0, failures = 1}
```

A mensagem de erro mostra onde ocorre o erro, o prefixo da mensagem de erro que foi passado como parâmetro, o valor esperado, o valor obtido, o número de casos de teste, o número de testes efetuados, o número de erros e o número de falhas. Note que a mensagem de erro inicia com a *label* que identifica o teste que falhou.

1.4 Açúcar Sintático

Definido o uso básico desta *framework*, incorpora-se agora o uso de açúcar sintático para facilitar a escrita de testes unitários. Entende-se por açúcar sintático o uso de funções ou definições que tornem a escrita de código mais acessível e simples. Por exemplo, podemos usar o combinador `~:` para produzir um `TestLabel` dado uma `String` e um `Test`. Assim, o exemplo:

```
TestLabel "Teste 1 (1, 2)" test1
```

poderia ser representado por:

```
"Teste 1 (1, 2)" ~: test1
```

Da mesma forma, o combinador `~=?` permite uma definição mais curta do uso da função `assertEquals` em conjunto com o construtor `TestCase`:

```
test1 = TestCase (assertEqual "for 1+2," 3 (mysum 1 2))
```

passa a:

```
test1 = 3 ~=? mysum 1 2
```

Note-se que foi perdido o nome do caso de teste. Podemos acrescentar um nome usando o combinador explicado anteriormente:

```
test1 = "for 1+2," ~: 3 ~=? mysum 1 2
```

Pode-se redefinir o nosso conjunto de testes agora utilizando estas ferramentas, por forma a que cada teste fique representado numa única linha:

```
tests = TestList [
    "Teste 1 (1, 2)"      ~: 3    ~=? mysum 1 2,
    "Teste 2 (5, 5)"      ~: 10   ~=? mysum 5 5,
    "Teste 3 (100, 100)" ~: 100  ~=? mysum 100 100
]
```

Finalmente, pode-se trocar a ocorrência de `TestList` por um uso da função `test`, que irá tentar converter os dados fornecidos em um caso de teste. Neste caso, irá converter uma lista de casos de teste num único teste:

```
tests = test [
    "Teste 1 (1, 2)"      ~: 3    ~=? mysum 1 2,
    "Teste 2 (5, 5)"      ~: 10   ~=? mysum 5 5,
    "Teste 3 (100, 100)" ~: 100  ~=? mysum 100 100
]
```

Existem outras funções e mais açúcar sintático disponível na documentação do `HUnit`². No entanto, as ferramentas explicadas neste documento são suficientes para construir uma *suite* de testes complexa.

1.5 Tarefas

1. Crie uma directoria **Aula4** com a seguinte estrutura:

```
.
|---- src/
|---- doc/
|---- tests/
```

Na subdirectoria `src` vai colocar o código desenvolvido. Na subdirectoria chamada `tests/` vai colocar os testes às funções definidas. Na subdirectoria `doc/` vai colocar a documentação html gerada pelo Haddock.

2. Selecciona algumas questões das fichas das aulas anteriores ou das fichas de Programação Funcional e resolve-as, escrevendo o código num módulo **Aula4**. Comente o código e coloque-o em **Aula4/src/**.
3. Use o Haddock para gerar a documentação em *HTML* em **Aula4/doc/**.
4. Defina testes para as questões resolvidas. Escreva os testes num módulo **Aula4_Spec** e coloque-o em **Aulas4/tests/**. Defina uma suite de testes chamada **testes_Aula4**. Não se esqueça de importar o módulo `Test.HUnit` e o módulo com o código **Aula4**.
5. Teste as funções usando `runTestTT`.
 - Chame o interpretador: `$ ghci -i="src" -i="tests" tests/Aula_Spec.hs`
 - Teste as funções definidas usando: `runTestTT testes_Aula4`

²<https://hackage.haskell.org/package/HUnit-1.6.2.0/docs/Test-HUnit-Base.html>