

# Programación Funcional Avanzada

## Pruebas Automatizadas

Ernesto Hernández-Novich  
<emhn@usb.ve>

Universidad “Simón Bolívar”

Copyright ©2010-2016

# Pruebas Unitarias (Unit Testing)

Porque lo hacen en otros lenguajes, ¿no?

- Metodología para verificar que las funciones hacen lo que se supone deben hacer.
- **Casos de Prueba**
  - Fragmento de código para ejercitar una función.
  - Descripción del resultado esperado.
- Se escriben tantos casos de prueba como sea necesario – cada *bug* genera uno o más casos de prueba.
- Deben verificarse con regularidad para asegurar que nuevo código no rompe nada – *Regression Tests*



# Tasty y HUnit

## Escribiendo pruebas unitarias

- Tasty, HUnit y algunas convenciones adoptadas en cabal permiten coordinar las pruebas.
- Módulo de prueba `test/Tasty.hs` con programa principal

```
module Main where
import Test.Tasty
import Test.Tasty.HUnit as HU
```

- Definir todos los casos de prueba como *asepciones* empleando las funciones provistas por HUnit.
- Compilar y ejecutar.



# Probando árboles binarios

¿Funcionan cómo debe ser?

- Consideremos una librería de árboles binarios

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
              deriving (Show, Eq)
```

- Con una función para insertar

```
tinsert x Leaf          = Node x Leaf Leaf
tinsert x t@(Node z l r) =
  if x == z
  then t
  else if x < z
        then Node z (tinsert x l) r
        else Node z l (tinsert x r)
```

¿Cómo probamos que funciona?



# Escribimos casos de prueba

## Aserciones con nombre

- Un caso de prueba es una función – `TestTree` es un tipo de `Tasty`

```
testInsertOnLeaf :: TestTree
testInsertOnLeaf = HU.testCase "tinsert on Leaf" $
  assertEquals "wrong!" (tinsert 42 Leaf)
                        (Node 42 Leaf Leaf)
```

- `testCase` requiere un nombre para la prueba y la aserción a verificar.
- La aserción se expresa con `assertEquals` indicando el mensaje a emitir en caso de falla y las dos expresiones a evaluar.
- ...y uno escribe montones de pruebas como esa.



# ¿Y que hago con el montón de pruebas?

El flojo debe ser organizado. . .

- Tasty permite crear *grupos de pruebas*

```
allTests :: TestTree
allTests = testGroup "Test all the tests!" [
  testGroup "Test tinsert" [
    testInsertOnLeaf
  ]
]
```



# ¿Y que hago con el montón de pruebas?

El flojo debe ser organizado...

- Tasty permite crear *grupos de pruebas*

```
allTests :: TestTree
allTests = testGroup "Test all the tests!" [
  testGroup "Test tinsert" [
    testInsertOnLeaf
  ]
]
```

- Y el programa de prueba debe ejecutarlas

```
main :: IO ()
main = defaultMain allTests
```

# Try tasting the tasty test testing technique

Sorry, not sorry...

- Se puede ejecutar el programa manualmente

```
$ runhaskell Tasty.hs
Test all the tests!
  Test tinsert
  tinsert on Leaf: OK
```

```
All 1 tests passed (0.00s)
```

- Si el módulo está manejado con cabal

```
$ cabal configure --enable-tests
$ cabal build
$ cabal test
```

y se genera un archivo `Tasty.log` con el reporte.





# Para escribir un poco menos

## Operadores para construir aserciones

- HUnit incluye atajos vía operadores infijos.
- En lugar de

```
assertEqual "bad" checkthis expected
```

se puede escribir

```
checkthis @?= expected
```

o bien

```
expected @?= checkthis
```

- Si algo es un predicado se puede escribir

```
assertBool "bad" predicate
```

o bien

```
predicate @? "bad"
```

# QuickCheck

- Herramienta para asistir en la verificación automática de programas.
- Librería de combinadores Haskell (EDSL).
- El programador provee una *especificación* que describe **propiedades** que deben cumplir las funciones que se desea verificar.
- QuickCheck *genera* casos de prueba y *comprueba* las propiedades.
  - Generación de datos de prueba controlable por el programador.
  - *Distribución* de los datos de prueba controlable por el programador.
  - Si una propiedad no se cumple, se reporta el contraejemplo.
- Aplicable a código funcional puro o código monádico transparente.

Hablaré de QuickCheck 2  
No usen QuickCheck 1



# ¿Qué es una *propiedad*?

Cualquier expresión lógica sobre sus argumentos

```
prop_assoc x y z = (x + y) + z == x + (y + z)
```



# ¿Qué es una *propiedad*?

Cualquier expresión lógica sobre sus argumentos

```
prop_assoc x y z = (x + y) + z == x + (y + z)
```

Tipos polimórficos permiten reutilizar

```
ghci> quickCheck
      (prop_assoc :: Int -> Int -> Int -> Bool)
+++ OK, passed 100 tests.
ghci> quickCheck
      (prop_assoc :: Float -> Float -> Float -> Bool)
*** Failed! Falsifiable (after 2 tests and 1 shrink):
-2.0
0.7972123
-0.4094756
```



## ¿Qué es una *propiedad*?

Cualquier expresión lógica sobre sus argumentos

```
prop_assoc x y z = (x + y) + z == x + (y + z)
```

Tipos polimórficos permiten reutilizar

```
ghci> quickCheck
      (prop_assoc :: Int -> Int -> Int -> Bool)
+++ OK, passed 100 tests.
ghci> quickCheck
      (prop_assoc :: Float -> Float -> Float -> Bool)
*** Failed! Falsifiable (after 2 tests and 1 shrink):
-2.0
0.7972123
-0.4094756
```

Se acostumbra utilizar el prefijo `prop_` para identificar las propiedades en el código.

# Idea general

## Nuestras funciones de ordenamiento

- Hace algunas semanas escribimos una función para ordenar con el algoritmo *Insertion Sort*.

```
isort :: Ord a => [a] -> [a]
isort = foldr ins []
  where ins x []      = [x]
        ins x (y:ys) = if x < y then (x:y:ys)
                          else y : ins x ys
```

¿Cómo podemos *probar* que funciona bien?



# Idea general

## Nuestras funciones de ordenamiento

- Hace algunas semanas escribimos una función para ordenar con el algoritmo *Insertion Sort*.

```
isort :: Ord a => [a] -> [a]
isort = foldr ins []
  where ins x []      = [x]
        ins x (y:ys) = if x < y then (x:y:ys)
                          else y : ins x ys
```

¿Cómo podemos *probar* que funciona bien?

- Construir propiedades de la función de ordenamiento.



# Idea general

## Nuestras funciones de ordenamiento

- Hace algunas semanas escribimos una función para ordenar con el algoritmo *Insertion Sort*.

```
isort :: Ord a => [a] -> [a]
isort = foldr ins []
  where ins x []      = [x]
        ins x (y:ys) = if x < y then (x:y:ys)
                          else y : ins x ys
```

¿Cómo podemos *probar* que funciona bien?

- Construir propiedades de la función de ordenamiento.
- Generar listas de prueba y aplicar las propiedades.





# Idea general

## Propiedades de la función de ordenamiento

- Idempotente – ordenar lo ordenado es lo mismo que ordenar una vez

```
prop_idempotent xs = isort (isort xs) == isort xs
```



# Idea general

## Propiedades de la función de ordenamiento

- Idempotente – ordenar lo ordenado es lo mismo que ordenar una vez

```
prop_idempotent xs = isort (isort xs) == isort xs
```

- Después de ordenar, la lista debe estar ordenada

```
prop_ordered xs = ordered $ isort xs
  where ordered []      = True
        ordered [x]    = True
        ordered (x:y:ys) = x <= y && ordered (y:ys)
```



# Idea general

## Propiedades de la función de ordenamiento

- Después de ordenar, el primer elemento ha de ser el mínimo

```
prop_minimum xs =  
  not (null xs) ==> head (isort xs) == minimum xs
```



# Idea general

## Propiedades de la función de ordenamiento

- Después de ordenar, el primer elemento ha de ser el mínimo

```
prop_minimum xs =  
  not (null xs) ==> head (isort xs) == minimum xs
```

- Si se ordena una concatenación, el mínimo del resultado es el mínimo entre los mínimos de ambas listas.

```
prop_append xs ys =  
  not (null xs) ==>  
    not (null ys) ==>  
      head (isort (xs++ys)) == min (minimum xs)  
                                   (minimum ys)
```



# QuickCheck en acción

- Las propiedades son funciones que acompañan al módulo.
- Se aprovecha la librería QuickCheck.

```
import Test.QuickCheck
```

- Se carga el módulo en ghci y se prueba manualmente

```
ghci> quickCheck prop_idempotent  
+++ OK, passed 100 tests.  
ghci> quickCheck prop_append  
+++ OK, passed 100 tests.  
ghci> quickCheck prop_minimum  
+++ OK, passed 100 tests.  
ghci> quickCheck prop_ordered  
+++ OK, passed 100 tests.
```



# Controlando QuickCheck

## Párametros de ejecución

- El tipo `Args` se combina con la función `quickCheckWith`.

```
ghci> stdArgs
Args { replay = Nothing, maxSuccess = 100,
      maxDiscard = 500, maxSize = 100 }
ghci> quickCheckWith
      (stdArgs { maxSuccess = 1000 }) prop_ordered
+++ OK, passed 1000 tests.
```

- `Args` controla cuántos casos generar y cuán complejos
  - `replay` – generador de números al azar para pruebas repetibles.
  - `maxSuccess` – cuántos éxitos para concluir.
  - `maxDiscard` – cuántos descartes para concluir.
  - `maxSize` – cuán *grande* cada caso.



# Pruebas en *batch*

- Pruebas manuales.
  - Utilidad limitada a depuración.
  - Requieren un humano – preferiblemente pensante.
  - Cansan, aburren y por eso fracasan.



# Pruebas en *batch*

- Pruebas manuales.
  - Utilidad limitada a depuración.
  - Requieren un humano – preferiblemente pensante.
  - Cansan, aburren y por eso fracasan.
- Pruebas automáticas.
  - Cubrir *todo* el módulo.
  - Todo el día, todos los días, cuantas veces quieras.
  - Asociarlas al *commit* en el VCS.





# Pruebas en *batch*

- Pruebas manuales.
  - Utilidad limitada a depuración.
  - Requieren un humano – preferiblemente pensante.
  - Cansan, aburren y por eso fracasan.
- Pruebas automáticas.
  - Cubrir *todo* el módulo.
  - Todo el día, todos los días, cuantas veces quieras.
  - Asociarlas al *commit* en el VCS.
- ¿Cómo automatizarlas?
  - El programa quickCheck – buscarlo en Hackage.

```
$ runhaskell quickCheck.hs testing.hs
```

- Escribir su propio programa de pruebas.



# Controlando QuickCheck

## Generando datos de prueba

- Propiedades cuantificadas universalmente de forma implícita – ¿cómo generar ese “para todo” de forma práctica?
- Definición de generadores

```
class Arbitrary a where  
  arbitrary :: Gen a
```

- Se proveen instancias predefinidas – Bool, Int, tuplas, listas, ...
- Gen es un Monad State que maneja la generación de números al azar.
- Basta proveer una instancia para generar datos arbitrarios.



# Funciones de apoyo

- Usar `arbitrary` en el contexto `Gen a` – resultados en `IO`.

```
sample' :: Gen a -> IO [a]
```

- Tomar “muestras” de un generador particular.

```
sample' :: Gen a -> IO [a]
```

```
ghci> sample' (arbitrary :: Gen Int)
[-1,1,2,-4,4,-9,12,-10,53,115,212]
```

```
ghci> take 3 'liftM'
      sample' (arbitrary :: Gen (Int,Bool))
[(1,False),(-1,True),(-2,False)]
```



# Funciones de apoyo

- Usar `arbitrary` en el contexto `Gen a` – resultados en `IO`.

```
sample' :: Gen a -> IO [a]
```

- Tomar “muestras” de un generador particular.

```
sample' :: Gen a -> IO [a]
```

```
ghci> sample' (arbitrary :: Gen Int)  
[-1,1,2,-4,4,-9,12,-10,53,115,212]
```

```
ghci> take 3 'liftM'  
      sample' (arbitrary :: Gen (Int,Bool))  
[(1,False),(-1,True),(-2,False)]
```

## Aprovechar que `Gen` es un `Monad`



# Funciones de apoyo

## De valores puros al Monad Gen

- choose – generar valores en un rango específico.

```
choose :: System.Random.Random a => (a,a) -> Gen a
```

```
ghci> sample' $ choose (0,42)  
[30,31,19,41,0,7,33,41,5,7,6]
```

# Funciones de apoyo

## De valores puros al Monad Gen

- `choose` – generar valores en un rango específico.

```
choose :: System.Random.Random a => (a, a) -> Gen a

ghci> sample' $ choose (0,42)
[30,31,19,41,0,7,33,41,5,7,6]
```

- `elements` – algún valor de una lista específica.

```
elements :: [a] -> Gen a

ghci> sample' $ elements "abcde"
"cdbabcdddee"
```



# Combinadores de Generación

Una vez que se tiene `Gen a`

- `listOf` – lista posiblemente vacía.

```
listOf :: Gen a -> Gen [a]
```

```
ghci> sample' $ listOf $ (arbitrary :: Gen Int)
```



# Combinadores de Generación

Una vez que se tiene `Gen a`

- `listOf` – lista posiblemente vacía.

```
listOf :: Gen a -> Gen [a]
```

```
ghci> sample' $ listOf $ (arbitrary :: Gen Int)
```

- `listOf1` – lista no vacía.

```
ghci> sample' $ listOf1 $ (arbitrary :: Gen Int)
```





# Combinadores de Generación

Una vez que se tiene `Gen a`

- `listOf` – lista posiblemente vacía.

```
listOf :: Gen a -> Gen [a]
```

```
ghci> sample' $ listOf $ (arbitrary :: Gen Int)
```

- `listOf1` – lista no vacía.

```
ghci> sample' $ listOf1 $ (arbitrary :: Gen Int)
```

- `vectorOf n` – lista de  $n$  elementos.

```
vectorOf :: Int -> Gen a -> Gen [a]
```

```
ghci> sample' $ vectorOf 3 $ (arbitrary :: Gen Int)
```



# Aún más control

## Generadores a partir de generadores

- `suchThat` – tal que cumpla el predicado.

```
suchThat :: Gen a -> (a -> Bool) -> Gen a
```

```
suchThat (arbitrary :: Gen Int) even
```



# Aún más control

## Generadores a partir de generadores

- `suchThat` – tal que cumpla el predicado.

```
suchThat :: Gen a -> (a -> Bool) -> Gen a
```

```
suchThat (arbitrary :: Gen Int) even
```

- `oneof` – selecciona uno de los generadores con probabilidad uniforme.

```
oneof :: [Gen a] -> Gen a
```

```
oneof [return "foo", return "bar", return "baz"]
```



# Aún más control

## Generadores a partir de generadores

- `suchThat` – tal que cumpla el predicado.

```
suchThat :: Gen a -> (a -> Bool) -> Gen a
```

```
suchThat (arbitrary :: Gen Int) even
```

- `oneof` – selecciona uno de los generadores con probabilidad uniforme.

```
oneof :: [Gen a] -> Gen a
```

```
oneof [return "foo", return "bar", return "baz"]
```

- `frequency` – selecciona con probabilidad relativa según el peso, i.e.

```
frequency :: [(Int, Gen a)] -> Gen a
```

```
frequency [(2, return True), (1, return False)]
```

genera `True` 66% de las veces...



# Aún más control

## Generadores a partir de generadores

- `suchThat` – tal que cumpla el predicado.

```
suchThat :: Gen a -> (a -> Bool) -> Gen a
```

```
suchThat (arbitrary :: Gen Int) even
```

- `oneof` – selecciona uno de los generadores con probabilidad uniforme.

```
oneof :: [Gen a] -> Gen a
```

```
oneof [return "foo", return "bar", return "baz"]
```

- `frequency` – selecciona con probabilidad relativa según el peso, i.e.

```
frequency :: [(Int, Gen a)] -> Gen a
```

```
frequency [(2, return True), (1, return False)]
```

genera `True` 66% de las veces...

Sólo es necesario construir el generador...

# ¿Cómo construir un Generador?

Hay que definirlos manualmente

- Basta instanciar la clase Arbitrary

```
instance Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

- El programador provee arbitrary usando los combinadores.
- shrink – ¿cómo “simplificar” un contraejemplo?
- La implantación automática de shrink es la lista vacía – hace que contraejemplos sean *fatales*, impidiendo reintentar.
- Es una clase que no puede ser derivada automáticamente.



# Construyendo generadores

## Los casos sencillos

- Para tipos enumerados, es realmente trivial

```
data MetaSyn = Foo | Bar | Baz | Qux
              deriving (Show,Eq)

instance Arbitrary MetaSyn where
  arbitrary = elements [Foo,Bar,Baz,Qux]
```



# Construyendo generadores

## Los casos sencillos

- Para tipos enumerados, es realmente trivial

```
data MetaSyn = Foo | Bar | Baz | Qux
              deriving (Show,Eq)

instance Arbitrary MetaSyn where
  arbitrary = elements [Foo,Bar,Baz,Qux]
```

- Que nos permite hacer

```
ghci> sample' $ (arbitrary :: Gen MetaSyn)
[Bar, Foo, Bar, Qux, Foo, Foo, Baz, Foo, Baz, Baz, Foo]
```

Basta cambiar `elements` por `frequency` si se necesita alterar las probabilidades.





# Construyendo generadores

... con mucho cuidado!

- Un tipo polimórfico para árboles binarios

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
              deriving (Show, Eq)
```



# Construyendo generadores

... con mucho cuidado!

- Un tipo polimórfico para árboles binarios

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
             deriving (Show, Eq)
```

- Primer intento – expresión directa

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
  where tree = oneof [ liftM Leaf arbitrary,
                      liftM3 Branch tree
                      arbitrary
                      tree ]
```



# Construyendo generadores

... con mucho cuidado!

- Un tipo polimórfico para árboles binarios

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
              deriving (Show, Eq)
```

- Primer intento – expresión directa

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
  where tree = oneof [ liftM Leaf arbitrary,
                      liftM3 Branch tree
                      arbitrary
                      tree ]
```

- Probamos

```
ghci> sample' $ (arbitrary :: Gen (Tree Int))
```

Meh... demasiados árboles triviales



# Construyendo un generador

... con mucho cuidado!

- Segundo intento – introducimos un sesgo para árboles “complicados”

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
  where tree = frequency [
      (1, liftM Leaf arbitrary),
      (3, liftM3 Branch tree
          arbitrary
          tree) ]
```



# Construyendo un generador

... con mucho cuidado!

- Segundo intento – introducimos un sesgo para árboles “complicados”

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
  where tree = frequency [
      (1, liftM Leaf arbitrary),
      (3, liftM3 Branch tree
          arbitrary
          tree) ]
```

- Probamos

```
ghci> sample' $ (arbitrary :: Gen (Tree Int))
```



# Construyendo un generador

... con mucho cuidado!

- Segundo intento – introducimos un sesgo para árboles “complicados”

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
  where tree = frequency [
      (1, liftM Leaf arbitrary),
      (3, liftM3 Branch tree
        arbitrary
        tree) ]
```

- Probamos

```
ghci> sample' $ (arbitrary :: Gen (Tree Int))
```

FUUUUUUUUUUUUUUUUUUU ¡Demasiado grandes!  
¿Serán infinitos?



# Construyendo un generador

... con mucho cuidado!

- Tercer intento – sesgo y control de tamaño

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
  where tree = sized tree'
        tree' 0          = liftM Leaf arbitrary
        tree' n | n > 0 = frequency [
            (1, liftM Leaf arbitrary),
            (3, liftM3 Branch subtree
              arbitrary
              subtree) ]
        where subtree = tree' (n `div` 2)
```



# Construyendo un generador

... con mucho cuidado!

- Tercer intento – sesgo y control de tamaño

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
  where tree = sized tree'
        tree' 0          = liftM Leaf arbitrary
        tree' n | n > 0 = frequency [
            (1, liftM Leaf arbitrary),
            (3, liftM3 Branch subtree
              arbitrary
              subtree) ]
        where subtree = tree' (n `div` 2)
```

- sized :: (Int -> Gen a) -> Gen a – tamaño según stdArgs.
- Garantizamos terminación – tamaño eventualmente es cero.
- Mejoramos distribución – sesgo hacia árboles “complicados”.





# Un ejemplo más variado

```
data RegExp = Lambda
  | Symbol Char
  | Concat RegExp RegExp
  | Union RegExp RegExp
  | Star RegExp
  deriving (Show, Eq)
```

- ¡Expresiones regulares!
- Enfatizar el uso de casos base – Lambda y Symbol
- Controlar Star (Star (Star ...))



# Un ejemplo más variado

```
re :: Int -> Int -> Gen RegExp
re stars 0 = frequency [
  (1, return Lambda),
  (3, Symbol 'liftM' choose ('a','z'))
]

re stars n = frequency [
  (3, return Lambda),
  (8, Symbol 'liftM' choose ('a','z')),
  (4, liftM2 Concat (re stars (n `div` 2))
                    (re stars (n `div` 2))),
  (4, liftM2 Union  (re stars (n `div` 2))
                    (re stars (n `div` 2))),
  (stars, Star 'liftM' (re (stars-1) (n `div` 2)))
]

instance Arbitrary RegExp where
  arbitrary = sized (re 1)
```

# Clasificando las pruebas

Por nombre

¿Habrá muchos casos “triviales”?



# Clasificando las pruebas

## Por nombre

¿Habrá muchos casos “triviales”?

```
prop_idempotent' xs =  
  classify (null xs)          "trivial" $  
  classify (not (null xs)) "mejor"   $  
  isort (isort xs) == isort xs
```



# Clasificando las pruebas

## Por nombre

¿Habr  muchos casos "triviales"?

```
prop_idempotent' xs =  
  classify (null xs)          "trivial" $  
  classify (not (null xs)) "mejor"   $  
  isort (isort xs) == isort xs
```

... al ejecutarlo

```
ghci> quickCheck prop_idempotent'  
+++ OK, passed 100 tests:  
95% mejor  
5% trivial
```

Justificaci n para ajustar distribuciones.

# Clasificando las pruebas

## Por distribución

¿Qué longitudes tienen las listas de prueba?



# Clasificando las pruebas

## Por distribución

¿Qué longitudes tienen las listas de prueba?

```
prop_idempotent'' xs = collect (length xs 'div' 10) $  
  isort (isort xs) == isort xs
```



# Clasificando las pruebas

## Por distribución

¿Qué longitudes tienen las listas de prueba?

```
prop_idempotent'' xs = collect (length xs `div` 10) $  
  isort (isort xs) == isort xs
```

... al ejecutarlo

```
ghci> quickCheck prop_idempotent''  
+++ OK, passed 100 tests:  
39% 0  
16% 1  
14% 2  
12% 3  
 7% 4  
 5% 6  
 3% 9  
 3% 8  
 1% 7
```



# ¿Y si quiero ver *todas* las pruebas?

```
prop_sick x = collect x $ ...
```



¿Y si quiero ver *todas* las pruebas?

```
prop_sick x = collect x $ ...
```

Busca ayuda profesional...



# Cobertura de Ejecución

¿Qué es lo que estamos probando exactamente?

- El código de un programa puede ser
  - **Inalcanzable** – cuando no es utilizado, bien sea por que es una parte no utilizada de una librería o porque el flujo de ejecución jamás podrá pasar por allí.
  - **Alcanzable** – cuando, en principio, el programa puede pasar por allí.
  - Los compiladores son capaces de eliminar el primero.
  - Probar lo que quede es responsabilidad del programador.



# Cobertura de Ejecución

¿Qué es lo que estamos probando exactamente?

- El código de un programa puede ser
  - **Inalcanzable** – cuando no es utilizado, bien sea por que es una parte no utilizada de una librería o porque el flujo de ejecución jamás podrá pasar por allí.
  - **Alcanzable** – cuando, en principio, el programa puede pasar por allí.
  - Los compiladores son capaces de eliminar el primero.
  - Probar lo que quede es responsabilidad del programador.
- En el curso de una corrida, el código alcanzable puede ser
  - **Cubierto** – cuando en efecto se ejecuta.
  - **No cubierto** – cuando no se ejecuta.

Código **no cubierto** es código no probado.  
Danger, Will Robinson!



# Análisis de cobertura

- **HPC** (*Haskell Program Coverage*) – herramienta integrada con el compilador `ghc` capaz de identificar código no cubierto.
- Usarlo durante las fases de desarrollo y prueba.
  - Compilar el programa con el flag `-fhpc`.
  - Ejecutar el programa – mientras más veces, mejor.
  - `hpc report` – reporte breve.
  - `hpc markup` – reporte HTML con anotaciones sobre el *código fuente*.
- El reporte de HPC
  - Presenta código no cubierto hasta el nivel de **subexpresiones**.
  - Resalta condicionales que han sido siempre ciertos o siempre falsos – se denominan *brazos inertes* y son oportunidad de mejora.

# Así se hace en la práctica...

Escribimos nuestro programa de prueba

```
main = do
  quickCheck
    (prop_idempotent :: [Int] -> Bool)
  quickCheck
    (prop_minimum :: [Int] -> Property)
  quickCheck
    (prop_ordered :: [Int] -> Bool)
  quickCheck
    (prop_append :: [Int] -> [Int] -> Property)
```

Las firmas son necesarias si no se pueden inferir del contexto.



# En la práctica

## Compilar para usar HPC

- Compilamos con las opciones necesarias

```
$ ghc -fhpc -fforce-recomp --make testing.hs
```

- Corremos el programa tantas veces como se desee.
- Archivo `.tix` *acumula* las medidas de cobertura – eliminarlo para comenzar desde cero.

¡No olvide recompilar **sin** `-fhpc`  
antes de pasar a producción!



# Evaluando los resultados

Desde la línea de comandos

```
$ hpc report testing
53% expressions used (77/144)
50% boolean coverage (1/2)
    0% guards (0/1), 1 unevaluated
    100% 'if' conditions (1/1)
    100% qualifiers (0/0)
77% alternatives used (7/9)
40% local declarations used (2/5)
42% top-level declarations used (6/14)
```














# Evaluando los resultados

A través de un navegador

Generar los documentos HTML resumen.

```
$ hpc markup testing
Writing: Main.hs.html
Writing: hpc_index.html
Writing: hpc_index_fun.html
Writing: hpc_index_alt.html
Writing: hpc_index_exp.html
```

El índice `hpc_index.html` muestra un resumen.

module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module <a href="#">Main</a>	42%	6/14		77%	7/9		53%	77/144	
<b>Program Coverage Total</b>	42%	6/14		77%	7/9		53%	77/144	

# Mejores prácticas

- Escriba pruebas para todas las funciones del módulo.
- Escriba un programa que ejercite todas las pruebas del módulo de manera automática.
- Ejecute el programa de prueba bajo HPC.
- Intente alcanzar 100 % de cobertura – puede ser muy difícil.
- Los contraejemplos encontrados por QuickCheck ayudan a identificar defectos en la implantación
  - En ocasiones son difíciles de comprender.
  - “Encogerlos” sintácticamente – `shrink :: a -> [a]`.
  - Ejemplos simples primero, complejos después.

## Quiero saber más...

- Documentación de Tasty
- Documentación de QuickCheck
- Introduction to QuickCheck2 en Haskell Wiki
- Haskell Program Coverage

QuickCheck 2  
(se lee “dos”)

