Relación de Ejercicios 3

Para realizar estos ejercicios necesitarás crear diferentes ficheros tanto en Haskell como en Java. En cada caso crea un nuevo fichero (con extensión hs para Haskell y java para Java). Añade al principio de tu fichero la siguiente cabecera, reemplazando los datos necesarios:

```
-------------------------------------------------------------------------------
-- Estructuras de Datos. 2º Curso. ETSI Informática. UMA
--
-- (completa y sustituye los siguientes datos)
-- Titulación: Grado en Ingeniería [Informática | del Software | de Computadores].
-- Alumno: APELLIDOS, NOMBRE
-- Fecha de entrega:  DIA | MES | AÑO
--
-- Relación de Ejercicios 3. Ejercicios resueltos: .........
--
-------------------------------------------------------------------------------
import Test.QuickCheck   (Cuando se trate de Haskell)
```

1. Haskell. Escribe una función que permita determinar si una cadena de caracteres está bien balanceada o equilibrada en lo que se refiere a los paréntesis, corchetes y llaves que contiene. El resto de caracteres en la cadena no se tendrán en cuenta. Por ejemplo la cadena "v(hg(jij)hags{ss[dd]dd})" está balanceada correctamente pero no así la cadena "ff(h([sds)sds]ss)hags".

Para solucionar este problema, se utilizará una pila, de forma que, cada vez que aparezca un signo de apertura en la cadena dada, éste se introducirá en la pila y, cada vez que aparezca un signo de cierre en la entrada, se extraerá la cima de la pila, comprobando que éste corresponde con el signo de apertura correspondiente. Si al finalizar de recorrer la cadena la pila está vacía entonces la expresión estará equilibrada.

```haskell
module WellBalanced where

import DataStructures.Stack.LinearStack

wellBalanced  :: String -> Bool
wellBalanced xs = wellBalanced'   xs   S.empty

wellBalanced' :: String -> Stack Char -> Bool
wellBalanced'  [] s = isEmpty s
wellBalanced' (x:xs) s  ...
```

```
*WellBalanced > wellBalanced "vv(hg(jij)hags{ss[dd]dd})"
True
```

2. Java. Escribe un programa Java que tome como argumento una cadena de caracteres y que escriba por pantalla si ésta está equilibrada (usa el algoritmo descrito en el problema anterior)

```java
package wellBalanced;
import dataStructures.stack.*;

public class WellBalanced {
	private final static String OPEN_PARENTHESES ="{[(";
	private final static String CLOSED_PARENTHESES = "}])";

	public static void main(String [] args) {
		...
```

```
        }

        public static boolean wellBalanced(String exp, Stack<Character> stack) {
                ...
        }

        public static boolean isOpenParentheses(char c) {
                return OPEN_PARENTHESES.indexOf(new Character(c).toString()) >= 0;
        }

        public static boolean isClosedParentheses(char c) {
             return CLOSED_PARENTHESES.indexOf(new Character(c).toString()) >= 0;
        }

        public static boolean match(char x, char y) {
                return  OPEN_PARENTHESES.indexOf(new Character(x).toString()) ==
                        CLOSED_PARENTHESES.indexOf(new Character(y).toString());
        }

}
```

3. Java. Teniendo en cuenta la función `postFix` que aparece en las transparencias complementarias de clase en Haskell, escribe un programa en Java que implemente la misma funcionalidad. Para ello, crear las clases `Item`, `Data`, `Operation`, `Add`, `Dif` y `Mul`.
La clase `Item` será abstracta y definirá los siguientes métodos:

```
public boolean isData( ) {
        return false;
}

public boolean isOperation( ) {
        return false;
}

public int getValue() {
        throw new UnsupportedOperationException();
}

public int evaluate(int a1, int a2) {
        throw new UnsupportedOperationException();
}
```

La clase `Data` será subclase de `Item` y permitirá almacenar un valor entero. Su constructor tomará dicho valor. Redefinirá los métodos `isData` (que devolverá true) y `getValue` (que devolverá el entero que almacena)

La clase `Operation` será también subclase de `Item` y abstracta, y debe redefinir el método `isOperation` (que devolverá true).
Las clases `Add`, `Dif` y `Mul` serán subclases de `Operation` y deben redefinir el método `evaluate(int a1, int a2)` de manera que en la primera clase se devuelva la suma de `a1` y `a2`, en la segunda la diferencia y en la tercera el producto.
Escribe un programa `PostFix` que incluya una función de clase, `static int evaluate(Item[] exprList)` que toma un array de `Items` que representa una expresión posfija y la evalúa. Una expresión ejemplo puede ser:

```
Item [] sample  = {
        new Data(5),
        new Data(6),
        new Data(2),
        new Dif(),
```

```
        new Data(3),
        new Mul(),
        new Add() };
```

4. Haskell. Consideremos el siguiente módulo para representar expresiones como listas de operandos o items:

```haskell
module Expression (
    Item (..)
  , Expression
  , value
  , showExpr
  , sample1, sample2
  ) where

data Item = Add | Dif | Mul | Value Integer | LeftP | RightP deriving Show

type Expression = [Item]

-- sample1 corresponde con 5 + (6 - 2) * 3
sample1 = [ Value 5, Add, LeftP, Value 6
          , Dif, Value 2, RightP, Mul, Value 3 ]
```

Donde `Add` representa la suma, `Dif` la resta, `Mul` el producto, `Value n` el entero n, `LeftP` el paréntesis izquierdo y `RightP` el paréntesis derecho.

Define un módulo Haskell llamado `InFix` que exporte una función

```haskell
evaluateInFix :: Expression -> Integer
```

que evalúa una expresión que está expresada en notación infija. Para este ejercicio consideraremos que la expresión está completamente parentizada, tal como la siguiente:

```haskell
sample :: Expression
sample = [LeftP, LeftP, Value 4, Mul, Value 5,RightP, Dif, Value 6, RightP]
--  sample se corresponde con la expresión ( (4 * 5) - 6)
```

El algoritmo para la evaluación utilizará dos pilas: en una se guardan los datos enteros que van apareciendo en la lista expresión y en la otra las operaciones. El algoritmo explora la lista. Si en la cabeza aparece un paréntesis de apertura (`LeftP`) se ignora y se explora el resto. Si en la cabeza aparece un paréntesis de cierre (`RightP`), se extrae el último operador de la pila de operadores y los dos últimos datos de la pila de datos, se aplica el operador a estos dos datos (el primero sacado es el segundo argumento) y el resultado se vuelve a colocar en la pila de datos; a continuación, se explora el resto de la lista de ítems. Al acabar la lista de ítems puede ocurrir: (a) la pila de datos contiene un único dato y la pila de operadores está vacía, en ese caso la expresión está bien construida (*parentizada*) y el dato que queda es el resultado; (b) en caso contrario la expresión original no está bien construida, por lo que no es posible calcular su valor.

5. Java. Resuelve el problema anterior en Java completando la jerarquía de clases definida en el problema 3. Para ello define la clase `Parentheses` como subclase de `Item` y las clases `LeftP` y `RightP` como subclases de `Parentheses`. Añade a `Item` el método `isParentheses()` que devuelve false y redefínelo en la clase `Parentheses` para que devuelva true. Tendrás que tener cuidado con la igualdad en la clase `Parentheses`.

```java
package infix;
import dataStructures.stack.Stack;
import dataStructures.stack.ListStack;
```

```java
public class InFix {

	static int evaluate(Item[] exprList) {
		Stack<Integer> stackDatas = new StackList<Integer>();
		Stack<Item> stackOperations = new StackList<Item>();
		for (Item expr : exprList) {
			if (expr.isData()) {
				...
			} else if (expr.isOperation()) {
				...
			} else if (expr.isParentheses()) {
				...
			}
		}
		return stackDatas.top();
	}

	public static void main(String [] args) {
		System.out.println(InFix.evaluate(Item.sample));
	}
}
```

6. Java. Para cualquier implementación de la interface `Stack`, define el constructor de copia. Éste es un constructor que toma como argumento otro `Stack` y copia todos sus valores al stack que se está creando.

7. Java. Implementa la interface `Queue` usando dos pilas. La clase se llamará `TwoStacksQueue` y se incluirá en el paquete `dataStructures.queue`. Para ello, observa la implementación `TwoListsQueue` en las transparencias complementarias de clase en Haskell. Las listas allí utilizadas actúan como pilas.

8. Haskell. Sea la especificación para una cola de prioridades `QueueP`

```haskell
isEmpty :: QueueP a -> Bool
enqueue :: (Ord a) => a -> QueueP a -> QueueP a
dequeue :: QueueP a -> QueueP a
first :: QueueP a -> a
empty :: QueueP a
```

El comportamiento de una cola de prioridades es el mismo que el de una cola, con una sola diferencia: cuando se encola un elemento se coloca detrás de los que son menores o iguales a él, y delante de los que son mayores; así, la operación `first` devolverá el menor elemento de la cola (que no necesariamente debe ser el primero introducido). Implementa el tipo abstracto `QueueP` utilizando una estructura lineal enlazada similar a `LinearQueue`.

9. Haskell. Implementa la siguiente signatura para conjuntos (`Set`):

```haskell
isEmpty :: Set a -> Bool
insert :: Eq a => a -> Set a -> Set a
delete :: Eq a => a -> Set a -> Set a
isElem :: a -> Set a -> Bool
empty :: Set a
```

teniendo en cuenta que en un conjunto no hay elementos repetidos: si insertamos un elemento que ya está incluido, no se hace nada; si se elimina un elemento que no está incluido, no se hace nada.

a) Implementa conjuntos como un tipo de dato algebraico lineal (`LinearSet`)

b) Implementa conjuntos basados en una lista (`ListSet`)

c) Añade las siguientes funciones en cada implementación:

```
union :: Eq a => Set a -> Set a -> Set a
intersection  :: Eq a => Set a -> Set a -> Set a
difference :: Eq a => Set a -> Set a -> Set a
isSubset :: Eq a => Set a -> Set a -> Bool
```

10. Java. Implementa el interfaz Set<T>

```
package set;

public interfaz Set<T> {
       boolean isElem(T el);
       void insert(T el);
       void delete(T el);
       boolean isEmpty();
}
```

Almacenando los elementos del conjunto sin repeticiones y de forma ordenada en:

a) Un array.

b) Una lista de nodos enlazados.

11. Haskell. Un saco (o multiconjunto) es parecido a un conjunto salvo que un elemento puede estar incluido varias veces. Por ejemplo, {'b', 'a', 'd', 'd', 'a', 'c' , b', 'a'} es un saco que incluye tres ocurrencias del carácter 'a' , dos ocurrencias del 'b', una del 'c' y dos del 'd'.

a) Implementa sacos en Haskell usando el siguiente tipo de datos:

```
data Bag a = Empty | Node a Int (Bag a)
```

de forma que en cada nodo aparece, además del saco restante, cada elemento junto con su contador de ocurrencias, o sea, el número de veces que aparece. Para agilizar las operaciones de inserción y borrado en un Bag, interesa que los nodos estén ordenados atendiendo al orden de los elementos a incluir. Además, no deben aparecer elementos con contador nulo (o negativo). Por ejemplo, el saco anterior se representa por:

```
Node 'a' 3 (Node 'b' 2 (Node 'c' 1 (Node 'd' 2 Empty)))
```

La implementación debe incluir las siguientes funciones:

```
empty :: Bag a                -- Devuelve un saco vacío

isEmpty :: Bag a -> Bool      -- Comprueba si un saco está vacío

insert :: (Ord a) => a -> Bag a -> Bag a  --Inserta una nueva ocurrencia

occurrences :: (Ord a) => a -> Bag a -> Int -- Devuelve el número de
-- ocurrencias de un elemento en un saco (0 si el elemento no está) -}

delete :: (Ord a) => a -> Bag a -> Bag a -- Borra una ocurrencia de un
-- elemento de un saco. Devuelve el mismo saco si el elemento no estaba incluido
```

b) Proporciona una especificación de Bag definiendo sus axiomas para las diferentes operaciones y comprueba la implementación realizada con QuickCheck. Para ello, incluye en el módulo la siguiente instancia para generar sacos aleatorios:

```
instance (Ord a, Arbitrary a) => Arbitrary (Bag a) where
  arbitrary =  do
    xs <- listOf arbitrary
    return (foldr insert empty xs)
```

c) Añade al módulo las siguientes funciones para manipular sacos: unión, intersección, diferencia y una función que determine si un saco está contenido en otro. Estas funciones son semejantes a las de los conjuntos pero teniendo en cuenta las ocurrencias de cada elemento.

d) Analiza la complejidad de las diferentes operaciones y justifica las ventajas de mantener los elementos ordenados.

12. Java. Sacos en Java

a) Define la interfaz `Bag` en Java.

b) Define la clase `LinkedBag` que implementa la interfaz `Bag` manteniendo los elementos en una lista enlazada de nodos de manera que cada nodo contiene un elemento y su número de apariciones. Considera los nodos ordenados según sus elementos, que deben ser ordenables. Proporciona un *iterador* para esta estructura de manera que un elemento con n ocurrencias sea devuelto n veces por el iterador.

c) Define una clase `ArrayBag` que proporcione una implementación alternativa de la interfaz `Bag` pero usando un array en lugar de una lista enlazada. Cada celda del array debe mantener un elemento y su correspondiente número de apariciones.

13. Haskell. Consideremos la siguiente función de plegado para el tipo `Stack` vista en clase:

```haskell
module FoldStackQueue where
import qualified DataStructures.Stack.LinearStack as S

foldrStack :: (a -> b -> b) -> b -> S.Stack a -> b
foldrStack f z s
   | S.isEmpty s = z
   | otherwise = S.top s `f` foldrStack f z (S.pop s)
```

a) Utilizando únicamente plegados de listas y de stacks, define las funciones

```haskell
listToStack :: [a]-> S.Stack a
listToStack = foldr …  …
stackToList :: S.Stack a -> [a]
stackToList = foldrStack …  …
```

Por ejemplo:

```
*FoldStackQueue> listToStack [1..10]
LinearStack(1,2,3,4,5,6,7,8,9,10)
*FoldStackQueue> stackToList (listToStack [1..10])
[1,2,3,4,5,6,7,8,9,10]
```

b) (**Complementario**) Observa que `stackToList` es la función inversa a `listToStack`; trata de demostrarlo.

c) Define usando `foldrStack` la función `mapStack` , de forma que la expresión `mapStack f s` aplica la función `f` a todos los elementos del stack `s`:

```haskell
mapStack :: (a -> b) -> S.Stack a -> S.Stack b
```

```
mapStack f = foldrStack  …   …
```

```
*FoldStackQueue > mapStack (100+) $ listToStack [1..6]
LinearStack(101,102,103,104,105,106)
```

d) Utilizando `foldrStack`, define la función que calcula el tamaño de un stack:

```
sizeStack :: Stack a -> Int
sizeStack = foldrStack …  …
```

```
*FoldStackQueue > sizeStack (listToStack [1..200])
200
```

e) (**Complementario**) Imponiendo las condiciones que estimes oportunas, demuestra por inducción sobre el número de operaciones que aparecen en cada stack, la siguiente propiedad:

`sizeStack (mapStack f s) = sizeStack s.`

14. Haskell. Supongamos que añadimos al módulo del ejercicio anterior una función de plegado para el tipo Queue:

```
module FoldStackQueue where
...
import qualified DataStructures.Stack.LinearQueue as S
foldrQueue :: (a -> b -> b) -> b -> Queue a -> b
foldrQueue f z q
        | S.isEmpty q = z
        | otherwise  = f (S.first q) (foldrQueue f z (S.dequeue q))
```

a) Utilizando únicamente plegados de listas y de colas, define las funciones:

```
listToQueue :: [a]-> Queue a
stackToQueue :: Stack a-> Queue a
queueToStack :: Queue a -> Stack a
queueToList :: Queue a -> [a]
```

Por ejemplo:

```
*FoldStackQueue> listToQueue [1..10]
LinearQueue(10,9,8,7,6,5,4,3,2,1)
```

c) ¿Qué hace la función `queueToList . listToQueue` ?

```
*FoldStackQueue> (queueToList . listToQueue) [1..10]
 [10,9,8,7,6,5,4,3,2,1]
*FoldStackQueue> queueToStack (listToQueue [1..10])
 LinearStack(10,9,8,7,6,5,4,3,2,1)
```

d) Define usando `foldrQueue` la función `mapQueue` de forma que la expresión `mapQueue f q` aplica la función f a todos los elementos de la cola q:

```
mapQueue  :: (a -> b) -> Queue  a -> Queue  b
```

Ejercicios Complementarios

15. Haskell. Consideremos el tipo abstracto de datos `Stack a` (pilas LIFO sobre un tipo genérico *a*), con las operaciones y axiomas habituales vistas en clase. Demuestra en el mismo orden en que aparecen las siguientes propiedades

a) Para cualquier n>0,

si s = push x_1 (push x_2 (… (push x_n empty)…)),

entonces pop s = push x_2 (… (push x_n empty)…)

b) Para cualquier n>0,

Si push x_1 (push x_2 (… (push x_n empty)…))

= push y_1 (push y_2 (… (push y_m empty)…))

entonces n = m y además, $\forall i\,.\,1 \leq i \leq n.\ x_i = y_i$

Ayuda: Usa inducción **sobre n**, los axiomas de `Stack` y la propiedad (a) del apartado anterior.

c) Una pila está bien definida si: (1) o bien es empty; (2) o bien es de la forma push x s, siendo s una pila bien definida; (3) o bien es de la forma pop s, donde s es una pila bien definida en la cual aparecen menos operaciones pop que operaciones push, y éstas en número finito. Prueba que cada pila bien definida se puede escribir de forma única como una secuencia

push x_1 (push x_2 (… (push x_n empty)…))

Ayuda: utiliza inducción sobre el total de operaciones pop o push.

16. Haskell. Consideremos el tipo abstracto de datos `Queue a` (colas FIFO sobre un tipo genérico *a*), con las operaciones y axiomas habituales vistas en clase. Demuestra en el mismo orden en que aparecen las siguientes propiedades (utilizamos enq y deq en lugar enqueue y dequeue para abreviar):

a) Para cualquier n>0, si q = enq x_1 (enq x_2 (… (enq x_n empty)…)),
entonces deq q = enq x_1 (… (enq x_{n-1} empty)…)
y además first q = x_n

Ayuda: utiliza inducción sobre n>0, junto a los axiomas.

b) Para cualquier n>0,

Si enq x_1 (… (enq x_n empty)…) = enq y_1 (… (enq y_m empty)…)

entonces n = m y además, $\forall i\,.\,1 \leq i \leq n.\ x_i = y_i$

Ayuda: utiliza la propiedad del apartado anterior e inducción **sobre n**.

c) Una cola está bien definida si: (1) o bien es empty; (2) o bien es de la forma enq x q, siendo q una cola bien definida; (3) o bien es de la forma deq q, donde q es una cola bien definida en la cual aparecen menos operaciones deq que operaciones enq, y éstas en número finito. Prueba que toda cola bien definida se puede escribir de forma única como una secuencia finita de operaciones enq:

enq x_1 (enq x_2 (… (enq x_n empty)…)

AYUDA: usa inducción sobre el total de operaciones enqueue o dequeue.

17. Haskell. Prueba que la implementación lineal `LinearStack` satisface los axiomas Ax1-Ax4.

18. Haskell. Prueba que la implementación lineal `LinearQueue` satisface los axiomas Ax1-Ax6.

19. Haskell. Se considera el TAD de los números enteros con la signatura

```
cero :: Entero
suc :: Entero -> Entero
pre :: Entero -> Entero
```

y un único axioma

(Ax1) `suc (pre x ) = x ,` `pre (suc x ) = x`

a) Demuestra que todo entero "finito" (secuencia finita bien construida) puede escribirse de forma única en una de las formas siguientes:

$\text{cero} \qquad \text{suc}^n\ \text{cero} \qquad \text{pre}^n\ \text{cero}$

b) Consideremos que añadimos la operación suma con la signatura:

```
(+) :: Entero -> Entero -> Entero
```

y los axiomas

(Ax2) `cero + x = x`

(Ax3) `suc x + y = suc ( x + y ), pre x + y = pre (x+y)`

c) Demuestra que entonces se sigue verificando lo anterior, es decir, todo entero "finito" (secuencia finita bien construida mezclando cero, suc, pre y +) puede escribirse de forma única en una de las formas siguientes:

$\text{cero} \qquad \text{suc}^n\ \text{cero} \qquad \text{pre}^n\ \text{cero}$

20. Haskell. Consideremos el siguiente módulo para describir el TADs de los enteros:

```
module Entero  Entero, cero, isCero,  isPos, isNeg, suc, pre) where

import Test.QuickCheck -- necesario para generar enteros arbitrarios

data Entero = Cero | Suc Entero | Pre Entero deriving Eq

instance Show Entero where show = show . enteroToInteger

enteroToInteger :: Entero -> Integer
enteroToInteger Cero    = 0
enteroToInteger (Suc n) = enteroToInteger n  +  1
enteroToInteger (Pre n) = enteroToInteger n  -  1

cero :: Entero
cero  = Cero

suc, pre :: Entero -> Entero
pre (Suc n) = n
pre n       = Pre n

suc (Pre n) = n
```

```
suc n        = Suc n
```

Observa las ecuaciones de las funciones `suc` y `pre` (no las confundas con los constructores `Suc` y `Pre`). Podríamos haber escrito ecuaciones sencillas para cada función:

```
suc = Suc
pre = Pre
```

pero nuestra intención es que cada llamada a las funciones suc y pre conserve la forma normalizada de su argumento; es decir, cada entero resultante se escriba de una de las siguientes formas:

```
Cero     Pre (Pre (...(Pre Cero)...)     Suc (Suc (...(Suc Cero)...))
```

Esta propiedad la llamaremos invariante de la representación (IR).

a) Demuestra que la forma normal Haskell de cualquier secuencia finita de operaciones `pre` y `suc` que empiece con cero está siempre normalizada. Es decir, que una tal secuencia no puede evaluarse a una forma normal tal como `Suc (Pre Cero)`. Para comprobarlo puedes generar la función show en forma automática estructural con la declaración de instancia siguiente:

```
data Entero = Cero | Suc Entero | Pre Entero deriving (Eq, Show)
```

Comprueba y comprende el siguiente diálogo (observa suc y Suc, pre y Pre):

```
*Entero> suc (pre (suc (suc (pre (pre (pre cero))))))
Pre Cero
*Entero> Suc (Pre (Suc (Suc (Pre (Pre (Pre cero))))))
Suc (Pre (Suc (Suc (Pre (Pre (Pre Cero))))))
*Entero> Suc (pre (Suc (suc (Pre (Pre (Pre cero))))))
Suc (Pre (Pre Cero))
```

b) Define ecuaciones para las siguientes funciones del interfaz, teniendo en cuenta que el argumento satisface el IR:

```
isCero, isPos, isNeg :: Entero -> Bool
-- comprueban si un entero es cero, positivo o negativo, respectivamente.
```

c) Consideremos los cuatro axiomas para el TAD `Entero` descritos à la Haskell

```
ax1   n =              suc (pre n) == n
ax1'  n =              pre (suc n) == n
ax2   n = isPos n ==>  isPos (suc n)
ax2'  n = isNeg n ==>  isNeg (pre n)
```

Después de añadirlos al módulo, comprueba con `quickCheck` que se satisfacen; para ello puedes utilizar el siguiente generador de enteros aleatorios:

```
instance Arbitrary Entero where
       arbitrary =  do
         i <- arbitrary
         return (fromInteger i)
```

siendo `fromInteger` la función que figura en la siguiente instancia de `Num`:

```
instance Num  Entero where
       -- fromInteger :: Integer -> Entero
       fromInteger 0              =  Cero
       fromInteger i | i > 0      =  Suc (fromInteger (i-1))
       fromInteger i | otherwise  =  Pre (fromInteger (i+1))
```

d) Consideremos que deseamos que la suma +, diferencia – y producto * satisfagan algunos axiomas que debes completar, tales como

```
(Ax+1)  x  + cero == x
(Ax+2)  x + suc y = suc ( x + y)
(Ax+3)  x + pre y = pre (x + y)
(Ax*1)        x * cero  = cero
(Ax-3)  x – pre y = suc (x – y)
```

Añade a la instancia de Num ecuaciones para el resto de operadores

```
x + Cero      = ??
x + Suc y     = ??
x + Pre y     = ??
x - Cero      = x
. . .
x * Cero      = Cero
. . .
abs (Pre x) = ??
. . .
signum Cero     = 0
. . .
```

y comprueba el funcionamiento con diálogos de la forma:

```
*Entero> 2 * (-2) :: Entero
Pre (Pre (Pre (Pre Cero)))
*Entero> 2 + (-2) :: Entero
Cero
```

NOTA: al escribir las ecuaciones debes tener en cuenta los axiomas y además que los operadores deben conservar el Invariante de la Representación IR: los resultados deben satisfacer IR siempre que lo satisfagan los argumentos.

e) Define ahora otras propiedades de los operadores aritméticos y compruébalos con `quickCheck`; algunas de éstas son:

```
p_neutro      n      =      cero + n == n
p_commu       n m    =      n + m == m + n
p_csigno      n m    =      n - m == - (m  - n)
```