Tarea 1
Primeros pasos en Haskell

Para la resolución de la tarea, recuerde que

- Todas las funciones que defina a top-level deben ir acompañadas de su tipo. Debe darles siempre el tipo *más general*. La única excepción permitida es para los argumentos y/o resultados de tipo entero, en cuyo caso se puede usar directamente el tipo `Int` en vez de la clase de tipos `Integral`.
- Si una función no tiene su tipo más general, se la penalizará con la mitad del puntaje.
- Cuando defina una función usando pattern matching, trate siempre de combinar los patrones/ecuaciones que sean combinables, como se vio en clase en diversos ejemplos. Esto, para garantizar la definición más simple y concisa posible.
- Cuando defina una función usando pattern matching, y no utilice alguno de los patrones, reemplácelo siempre por el *wildcard* `_`.
- Los mensajes de error que se deban imprimir al usuario, deben ser idénticos a los especificados en los ejemplos.

Ejercicio 1 (Recursión sobre los naturales) — 9 Pt

(a) [4 Pt] Para calcular la potencia b^n (para n entero no-negativo) de una manera más eficiente podemos explotar la siguiente observación:

- Si $n = 2m$ es par, entonces, $b^{2m} = b^m * b^m$;
- Si $n = 2m + 1$ es impar, entonces, $b^{2m+1} = b * b^m * b^m$;

Dé el tipo y defina una función `effPow` que implemente este algoritmo. Al calcular recursivamente b^m, utilice una definición local (via la cláusula `where`), así evita calcularlo dos veces (de manera innecesaria). En caso de que el exponente ingresado sea negativo, debe lanzar un error.

```
> effPow 1.5 3
3.375
> effPow 2 (-5)
*** Exception: exponente negativo
```

(b) [5 Pt] Demostraremos que toda cantidad de plata mayor o igual a 8 pesos puede pagarse con billetes de 3 y 5 pesos. Para 8, 9 y 10 pesos, tenemos que $8 = 3 + 5$, $9 = 3 + 3 + 3$ y $10 = 5 + 5$. Ahora para pagar $n + 3$ pesos (con $n \geq 8$), tomamos los billetes que se necesitan para pagar n pesos, y simplemente le agregamos un billete de 3 pesos.

Dé el tipo y defina una función recursiva `pay` que tome un entero n, y devuelva otro par de enteros, especificando cuántos billetes de 3 y 5 pesos se necesitan para pagar n pesos. Si ingresa un $n < 8$, debe devolver un error (por más que algunos $n < 8$ sí se puedan pagar con billetes de 3 y 5 pesos).

```
> pay 13
(1,2)
> pay 7
*** Exception: no se pueden pagar 7 pesos
```

Ejercicio 2 (Recursión sobre listas) **17 Pt**

(a) [3 Pt] Dé el tipo y defina recursivamente la función `numberOfHits` que dado un predicado y una lista, devuelve la cantidad de elementos de la lista que satisfacen el predicado.

```
> numberOfHits (> 3) [1..7]
4
```

(b) [5 Pt] Dé el tipo y defina recursivamente la función `splitAtFirstHit` que toma un predicado y una lista, y parte la lista en dos, sobre la ocurrencia del primer elemento que satisface el predicado.

```
> splitAtFirstHit (>3) [1..7]
([1,2,3],[4,5,6,7])
> splitAtFirstHit even [2,3,4]
([],[2,3,4])
> splitAtFirstHit even [1,3,5]
*** Exception: no hit in the list
```

(c) [5 Pt] Dé el tipo y defina recursivamente la función `positionsAllHits` que dado un predicado y una lista, devuelve (una lista formada por) las posiciones de los elementos de la lista que satisfacen el predicado. (Asuma que el primer elemento de la lista está en la posición 0.)

```
> positionsAllHits even [4..8]
[0,2,4]
> positionsAllHits even [1,3,5]
[]
```

(d) [4 Pt] Dé el tipo y defina a través de recursión *mutua* las funciones `evens` y `odds` que toman una lista y se quedan sólo con los elementos que están en posición par e impar, respectivamente. (Asuma que el primer elemento de la lista está en la posición 0.)

```
> evens "funcional"
"fninl"
> odds "funcional"
```

```
"ucoa"
```

Ejercicio 3 (Funciones de alto orden) **18 Pt**

(a) [1 Pt] Dé el tipo y defina recursivamente la función `hasSomeHit` que dado un predicado y una lista, determine si algún elemento de la lista satisface el predicado.

```
> hasSomeHit even [2,3]
True
> hasSomeHit even [1,3]
False
```

(b) [2.5 Pt] Utilizando la función `hasSomeHit`, dé el tipo y defina la función `isMember` que determina si un elemento está en una lista.

```
> isMember 'a' "funcional"
True
> isMember 5 [1,2,3]
False
```

(c) [3.5 Pt] Utilizando la función `hasSomeHit`, dé el tipo y defina (recursivamente) la función `repeteadElem` que determina si una lista tiene elementos repetidos.

```
> repeteadElem [1,2,3]
False
> repeteadElem [1,2,3,2,5]
True
```

(d) [3 Pt] Dé el tipo y defina recursivamente la función `applyUntil` que aplica repetidamente (cero o más veces) una función a un argumento hasta que el resultado obtenido satisface un predicado.

```
> applyUntil (+2) (>6) 2
8
> applyUntil (+2) (>1) 2
2
```

(e) [4 Pt] Utilizando la función `applyUntil`, dé el tipo y defina la función `leastPow2` que dado un entero no-negativo n devuelve la menor potencia de 2 que es mayor o igual a n. (No puede utilizar la función `log` para resolver el problema. Debe diseñar una solución en términos de `applyUntil`.) En caso de recibir un argumento negativo, `leastPow2` debe devolver un error.

```
leastPow2 11
16
> leastPow2 (-11)
*** Exception: argumento negativo
```

(f) [4 Pt] Utilizando la función `applyUntil`, dé el tipo y defina la función `balancedSufix` que dada una lista de Booleanos, devuelva el mayor sufijo de la lista que tenga la misma cantidad de `True`'s que de `False`'s.

```
> balancedSufix [True, True, True, False, False, True]
[True, False, False, True]
```

Si ningún sufijo de la lista satisface esta propiedad, se permite que la función diverja (i.e. genere una reducción infinita).

Ejercicio 4 (Composición y estilo point-free) **6 Pt**

Parta de la definición de `isMember` que dio en el Ejercicio 3 y transfórmela incrementalmente (como se hizo en la Clase05.pdf, diapositiva 13) hasta obtener una definición equivalente en estilo *point-free* . A la versión point-free, llámela `isMemberPF`. (A una solución que sólo presenta la versión point-free, *sin* la derivación correspondiente, se le asignará 0 puntos.)

Ejercicio 5 (Testing basado en propiedades) **10 Pt**

Usando QuickCheck testee que:

(a) [2.5 Pt] La función `effPow` que definió en el Ejercicio 1 satisface las siguientes propiedades:

$$\forall a, b, n\colon \mathsf{Int}.\ n \geq 0 \implies \mathsf{effPow}\ (a \cdot b)\ n = \mathsf{effPow}\ a\ n\ \cdot\ \mathsf{effPow}\ b\ n$$
$$\forall a, m, n\colon \mathsf{Int}.\ m, n \geq 0 \implies \mathsf{effPow}\ a\ (m + n) = \mathsf{effPow}\ a\ m\ \cdot\ \mathsf{effPow}\ a\ n$$

(b) [1.5 Pt] La función `pay` que definió en el Ejercicio 1 satisface las siguiente propiedad:

$$\forall n\colon \mathsf{Int}.\ n \geq 8 \implies 3a + 5b = n \quad \text{donde} \quad (a, b) := \mathsf{pay}\ n$$

(c) [3.5 Pt] La función `numberOfHits` que definió en el Ejercicio 2 satisface las siguientes propiedades:

$$\begin{aligned}\forall xs, ys\colon [\mathsf{Int}].\ &\mathsf{numberOfHits}\ even\ (xs +\!\!+ ys) = \\ &\mathsf{numberOfHits}\ even\ xs\ +\ \mathsf{numberOfHits}\ even\ ys\\ \forall xs\colon [\mathsf{Int}].\ &\mathsf{numberOfHits}\ true\ xs = |xs|\\ \forall xs\colon [\mathsf{Int}].\ &\mathsf{numberOfHits}\ false\ xs = 0\end{aligned}$$

Notar que aquí *true* y *false* no representan los valores de verdad Booleanos, sino predicados constantes.

(d) [2.5 Pt] Agregue un test a la función `numberOfHits` que modele (de alguna manera) el resultado establecido por el principio de inclusión-exclusión: $|A \cup B| = |A| + |B| - |A \cap B|$.