Práctico 1: Expresiones Cuantificadas

Algoritmos y Estructuras de Datos I
2do cuatrimestre 2025

Objetivos

El objetivo de esta guía es trabajar sobre la semántica de cuantificadores generalizados y sus propiedades. Abordaremos ejercicios de cálculo de evaluación de expresiones cuantificadas y de aplicación de sus propiedades para lograr familiaridad en la manipulación simbólica de este tipo de expresiones. Conocer los axiomas y teoremas del cálculo y habilidad en las demostraciones es necesario para poder abordar la tarea de derivación y demostración de programas que encararemos de aquí en adelante.

Semántica de los cuantificadores generalizados

*A los cuntificadores **universial** y **existencial** ya conocidos, se incorporan otros cuantificadores que serán de utilidad para expresar problemas más complejos.*

- *Cuantificador universal* ($\forall$)
- *Cuantificador existencial* ($\exists$)
- *Sumatoria* ($\sum$)
- *Productoria* ($\prod$)
- *Máximo (Max)*
- *Mínimo (Min)*
- *Intersección* ($\bigcap$)
- *Unión* ($\bigcup$)

El objetivo de los siguientes ejercicios es ejercitar la lectura y el uso de dichos cuantificadores generalizados para expresar problemas complejos de manera más simple. Los ejercicios de laboratorio requerirán definir funciones que impementen lo descripto en lenguaje formal.

1. Teniendo en cuenta que xs es una lista de *Figuras* como describimos en el práctico 0, describí el significado de las siguientes expresiones utilizando el lenguaje natural.

   a) $\langle \sum i \ : 0 \leq i < \#xs : \ tam.(xs.i) \rangle$
   b) $\langle \prod \ i \ : 0 \leq i < \#xs : \ tam.(xs.i) \rangle$
   c) $\langle \mathrm{N}\, i \ : 0 \leq i < \#xs : \ rombo.(xs.i) \rangle$
   d) $\langle \mathrm{N}\, i \ : 0 \leq i < \#xs : \ rombo.(xs.i) \wedge rojo.(xs.i) \rangle$

Laboratorio 1 En el ejercicio 1 observamos que todas las expresiones tienen como única variable libre a xs. Es decir, el valor de la expresión, si la queremos evaluar, depende del valor de esa variable. Por lo tanto, para cada expresión podemos definir una función, que tome una lista de figuras y devuelva el valor de esa expresión para esa lista.

a) Definí funciones que calculen el valor de las expresiones en el ejercicio 1.

b) Evaluá las funciones definidas para las listas:

```
[(Circulo, Azul, 40), (Rombo, Rojo, 10)]
[(Circulo, Rojo, 4), (Rombo, Rojo, 10)]
[ (Triangulo, Verdo, 10)]
```

2. Escribí fórmulas para las siguientes expresiones en lenguaje natural.

   a) La suma de los tamaños de todas las figuras de xs es mayor a 10.
   b) Ninguna figura de xs tiene tamaño menor a 7.

Observación: El último ejercicio se puede expresar con el cuantificador Min . Si no lo resolviste así, resolverlo también de esa manera.

Laboratorio 2 De la misma manera que en el ejercicio 1, en el ejercicio 2 observamos que todas las expresiones tienen como única variable libre a xs.

a) Definí funciones que calculen el valor de las expresiones que definiste en el ejercicio 2. Observar que para los casos b) y c) tendrás que definir una función recursiva que calcule el cuantificador, y luego otra función que calcule el valor de la expresión completa. Por ejemplo, si tenemos el predicado [1]

"La suma de los tamaños de las figuras de xs es igual a 22."

definimos el predicado `sumaFig22` de la siguiente manera :

```
sumaFig :: [Figura] -> Int
sumaFig [] = 0
sumaFig (x:xs) = tam x + sumaFig xs

sumaFig22 :: [Figura] -> Bool
sumaFig22 xs = sumaFig xs == 22
```

b) Evaluá las funciones definidas para las listas:

```
[(Circulo, Azul, 40), (Rombo, Rojo, 10)]
[(Circulo, Rojo, 4), (Rombo, Rojo, 10)]
[(Triangulo, Verdo, 10)]
```

3. Considerá una función $sumalista : [Int] \to Int$ que suma todos los elementos de una lista. Por un lado, escribí una especificación que caraterice esta función, utilizando el cuantificador Σ. Por otro lado, escribí un programa recursivo que compute la función.

Laboratorio 3 Implementá en Haskell la función definida en el ejercicio anterior.

4. Para cada una de las siguientes funciones escribí una expresión que las describa formalmente. Por otro lado, escribí un programa recursivo que compute la función.
   a) `sumatoria :: [Int] -> Int`, que calcula la suma de todos los elementos de una lista de enteros.
   b) `productoria :: [Int] -> Int`, que calcula el producto de todos los elementos de la lista de enteros.
   c) `factorial :: Int -> Int`, que toma un número n y calcula $n!$.
   d) Utilizá la función `sumatoria` para definir, `promedio :: [Int] -> Int`, que toma una lista de números no vacia y calcula el valor promedio (truncado, usando división entera).

Laboratorio 4 Implementá en Haskell las funciones definidas en el ejercicio anterior.
A continuación mostramos algunos ejemplos del uso de las funciones en `ghci`:

```
$> sumatoria [1, 5, -4]
2
$> productoria [2, 4, 1]
8
$> factorial 5
120
$> promedio [1, 5, -4]
0
```

5. Para cada una de las siguientes fórmulas, describí su significado utilizando el lenguaje natural.
   a) $\langle \prod i : 1 \leq i \leq n : i \rangle$
   b) $\dfrac{\langle \sum i : 0 \leq i < \#xs : xs.i \rangle}{\#xs}$
   c) $\langle \text{Max } i : 0 \leq i < \#xs : xs.i \rangle < \langle \text{Min } i : 0 \leq i < \#ys : ys.i \rangle$
   d) $\langle \exists i, j : (2 \leq i < n) \wedge (2 \leq j < n) : i * j = n \rangle$

6. Para cada uno de los ítems del ejercicio anterior, evaluá respectivamente con los siguientes valores:

   a) $n = 5$.
   b) $xs = [6, 9, 3]$.
   c) $xs = [-3, 9], ys = [6, 7]$.
   d) $n = 5$.

Laboratorio 5 A partir de las expresiones en el ejercicio 5*a*, 5*b* y 5*c*

a) Identificá las variables libres de cada expresión y el tipo de cada una.

b) Definí funciones que tomen como argumento las variables libres identificadas y devuelvan el resultado de la expresión. **Atención:** Tené en cuenta que en algunos casos es necesario definir varias funciones.

c) Evaluá las funciones tomando como argumento los valores señalados en el ejercicio 6.

7. Suponiendo que $f : A \to Bool$ es una función fija cualquiera, y $xs : [A]$, caracterizá con una cuantificación la siguiente función recursiva:

$$algunof : [A] \to Bool$$

$$\begin{aligned} algunof.[\,] &= False \\ algunof.(x \triangleright xs) &= f.x \vee algunof.xs \end{aligned}$$

8. Definí recursivamente una función $todos : [Bool] \to Bool$ que verifica que todos los elementos de una lista son $True$, es decir, que satisface la siguiente especificación:

$$todos.xs \equiv \langle\, \forall\, i \;:\; 0 \leq i < \#xs :\; xs.i \,\rangle$$

9. Escribí fórmulas para describir formalmente las siguientes expresiones en lenguaje natural.

   Preguntas: ¿Qué tipo debe tener cada variable libre para que expresión tenga sentido? ¿Qué tipo tiene cada expresión?

   a) n es potencia de 2.
   b) n es el elemento más grande de xs.
   c) El producto de los elementos pares de xs.
   d) La suma de los elementos en posición par de xs.

Laboratorio 6 A partir de las expresiones del ejercicio anterior b) y c y d)

a) Identificá las variables libres de cada expresión y el tipo de cada una.

b) Definí funciones que tomen como argumento las variables libres identificadas y devuelvan el resultado de la expresión. **Atención:** Tené en cuenta que en algunos casos es necesario definir varias funciones.

10. **Calculá los rangos** de las siguientes cuantificaciones como conjuntos de posibles valores. Tomar $n = 10$, $xs = [-3, 9, 8, 9]$, $ys = [6, 9, 3]$. Usá tuplas cuando haya más de una variable cuantificada.

    a) $\langle \prod\, i \;:\; 1 \leq i \leq n \;\wedge\; i \bmod 3 = 1 :\; i \,\rangle$
    b) $\langle \sum i, j \;:\; 0 \leq i < \#xs \;\wedge\; 0 \leq j < \#ys :\; xs.i * ys.j \,\rangle$
    c) $\langle\, \forall\, i, j \;:\; 0 \leq i < j < \#xs :\; xs.i \neq xs.j \,\rangle$
    d) $\langle \text{Max}\;\; as, bs \;:\; xs = as +\!\!+ bs :\; sum.as \,\rangle$
    e) $\langle \sum i \;:\; 1 \leq i + 1 < \#xs + 1 :\; (x \triangleright xs).(i + 1) \,\rangle$

    Ejemplo: En el segundo ítem tenemos que i y j son independientes entre sí. Por lo tanto tenemos que ver todas las combinaciones posibles con $i \in \{0, 1, 2, 3\}$ y $j \in \{0, 1, 2\}$. En el tercer ítem hay una dependencia de j respecto de i; por lo tanto primero podés calcular los valores posibles de i y luego, para cada posible valor de i, los posibles valores de j.

Cálculo con cuantificadores generalizados

En esta sección comenzaremos a trabajar con las propiedades básicas de los cuantificadores.

11. Simplicá hasta obtener una expresión sin cuantificador las siguientes expresiones aplicando las propiedades de cuantificadores **rango vacío, rango unitario o término constante** y propiedades báscas de la aritmética.

    a) $\langle \exists\, i \ : i = 3 \land i \bmod 2 = 0 : \ 2 * i = 6 \,\rangle$
    b) $\langle \sum i \ : 5 \leq i \land i \leq 5 : \ -2 * i \,\rangle$
    c) $\langle \prod i \ : 0 < i < 1 : \ 34 \,\rangle$
    d) $\langle \text{Min } i \ : i \leq 0 \lor i > 10 : \ n * (i + 2) - n * i \,\rangle$
    e) $\langle \text{Max } a, as \ : a \triangleright as = [\,] : \ \#as \,\rangle$

12. Aplicá **partición de rango** si es que se puede, y si no se puede, explicá porqué.

    a) $\langle \sum i \ : i = 0 \ \lor \ 4 > i \geq 1 : \ n * (i + 1) \,\rangle$
    b) $\langle \forall\, i \ : 3 \leq |i| \leq 4 \ \lor \ 0 < i < 4 : \ \neg f.i \,\rangle$
    c) $\langle \sum i \ : |i| \leq 1 \ \lor \ 0 \leq 2 * i < 7 : \ i * n \,\rangle$
    d) $\langle \prod i \ : 0 \leq i < \#xs \ \land \ (i \bmod 3 = 0 \ \lor \ i \bmod 3 = 1) : \ 2 * xs.i + 1 \,\rangle$ (distribuir primero)

13. Evaluá las expresiones del ejercicio 12 tomando los siguientes valores de las variables libres: $n = 3$, $f.x = |x| < 4$, $xs = [-1, 1, 0, 3]$.

14. Descubrí el error en la siguiente prueba. Es decir, ¿la supuesta propiedad demostrada, vale para todos los valores posibles de xs? ¿Para qué valor o valores de xs falla?

    $\langle \sum i \ : 0 \leq i < \#xs : \ xs.i \,\rangle$
    $=$ { lógica }
    $\langle \sum i \ : i = 0 \ \lor \ 1 \leq i < \#xs : \ xs.i \,\rangle$
    $=$ { partición de rango disjunto }
    $\langle \sum i \ : i = 0 : \ xs.i \,\rangle + \langle \sum i \ : 1 \leq i < \#xs : \ xs.i \,\rangle$
    $=$ { rango unitario }
    $xs.0 + \langle \sum i \ : 1 \leq i < \#xs : \ xs.i \,\rangle$

15. Aplicá **distributividad**, si es que se puede.

    a) $\langle \sum i \ : i = 0 \ \lor \ 4 > i \geq 1 : \ n * (i + 1) \,\rangle$
    b) $\langle \prod i \ : 3 \leq |i| \leq 4 \ \lor \ 0 < i < 4 : \ n + i \,\rangle$
    c) $\langle \forall\, i \ : i = 0 \ \lor \ 4 > i \geq 1 : \ \neg(f.i \ \land \ f.n) \,\rangle$
    d) $\langle \text{Max } i \ : 0 \leq i < \#xs : \ x + xs.i \,\rangle$

16. Calculá los resultados para todos los ítems del ejercicio anterior. Usá $n = 3$, $f.x = (x = 0)$, $x = -1$, $xs = [1, 0, 3]$.

17. Aplicá el **cambio de variable** indicado, si es que se puede. Explicá porqué puede o no puede aplicarse.

    a) $\langle \sum i \ : |i| < 5 : \ i \ div \ 2 \,\rangle$ con $i \to 2 * i$ (o sea $f.i = 2 * i$)
    b) $\langle \sum i \ : i \bmod 2 = 0 \land |i| < 5 : \ i \ div \ 2 \,\rangle$ con $i \to 2 * i$ (o sea $f.i = 2 * i$)
    c) $\langle \prod i \ : 0 < i \leq \#(x \triangleright xs) : \ (x \triangleright xs).i \,\rangle$ con $i \to i + 1$ (o sea $f.i = i + 1$)
    d) $\langle \text{Max } as \ : as \neq [\,] : \ \#as \,\rangle$ con $(a, as) \to a \triangleright as$ (la función es $f.(a, as) = a \triangleright as$)

18. Simplificá el rango y aplicá alguna de las **reglas para la cuantificación de conteo**:

    a) $\langle N\, a, as \ : a \triangleright as = xs \land xs = [\,] : \ \#as = 1 \,\rangle$
    b) $\langle N\, i \ : i - n = 1 : \ i \bmod 2 = 0 \,\rangle$
    c) $\langle N\, i \ : i = 0 \lor 1 \leq i < \#xs + 1 : \ ((x \triangleright xs).i) \bmod 2 = 0 \,\rangle$

19. Demostrá la siguiente propiedad:
$\langle \sum i \ : 0 \leq i < \#(x \triangleright xs) : \ T.((x \triangleright xs).i) \ \rangle = T.x + \langle \sum i \ : 0 \leq i < \#xs : \ T.(xs.i) \ \rangle$

20. (*) **(Separación de un término)** Demostrá los siguientes teoremas útiles para la materia.
Suponé que $\bigoplus$ es un cuantificador asociado a un operador genérico $\oplus$, que es conmutativo y asociativo (así como el $\forall$ es el cuantificador asociado a la conjunción $\wedge$) y $n : Nat$.
   a) $\langle \bigoplus \ i \ : 0 \leq i < n+1 : \ T.i \ \rangle = \langle \bigoplus \ i \ : 0 \leq i < n : \ T.i \ \rangle \oplus T.n$
   b) $\langle \bigoplus \ i \ : 0 \leq i < n+1 : \ T.i \ \rangle = T.0 \oplus \langle \bigoplus \ i \ : 0 \leq i < n : \ T.(i+1) \ \rangle$

21. **(Rango unitario generalizado)** Sea $\oplus$ un cuantificador asociado a un operador conmutativo y asociativo. Probá la siguiente regla de *rango unitario generalizado* (Z no depende de i ni de j):
$$\langle \bigoplus \ i, j \ : i = Z \wedge R.i.j : \ T.i.j \ \rangle = \langle \bigoplus \ j \ : R.Z.j : \ T.Z.j \ \rangle \ .$$

22. Podemos definir un cuantificador de *conteo* N utilizando la sumatoria:
$$\langle \mathrm{N} \, i \ : R.i : \ T.i \ \rangle \doteq \langle \sum i \ : R.i \wedge T.i : \ 1 \ \rangle$$
Demostrá que $\langle \sum i \ : \ R.i \wedge T.i \ : \ k \rangle = \langle \mathrm{N} i \ : \ R.i \ : \ T.i \rangle * k$

23. (*) Demostrá la siguiente relación entre los cuantificadores de máximo y mínimo cuando R es no vacío:
$$n = \langle \mathrm{Min} \ i \ : \ R.i \ : \ -T.i \rangle \equiv n = -\langle \mathrm{Max} \ i \ : \ R.i \ : \ T.i \rangle$$

24. (*) Demostrá los siguientes teoremas sobre $\forall$, utilizando los axiomas y teoremas del digesto:
   a) *Intercambio de $\forall$ (generalizada)*:
   $\langle \forall i \ : \ R.i \wedge S.i \ : \ T.i \rangle \equiv \langle \forall i \ : \ R.i \ : \ S.i \Rightarrow T.i \rangle$
   b) *Instanciación de $\forall$*:
   $\langle \forall i \ : : \ T.i \rangle \Rightarrow T.x$, cuando x no está cuantificada.
   ¿Como sería la regla de instanciación para $\exists$? Enunciala y demostrala.

Tutorial de laboratorio: Definición de tipos. Clases de tipos.

En esta sección trabajaremos con ejercicios de laboratorio para introducir nuevos conceptos de programación en Haskell. En particular, definiremos nuestros propios tipos de datos. La importancia de poder definir nuevos tipos de datos reside en la facilidad con la que podemos modelar problemas y resolverlos usando las mismas herramientas que para los tipos pre-existentes.

Laboratorio 7 **Tipos enumerados.** Cuando los distintos valores que debemos distinguir en un tipo son finitos, podemos *enumerar* cada uno de los valores del tipo. Por ejemplo, podríamos representar las carreras que se dictan en nuestra facultad definiendo el siguiente tipo:

```
data Carrera = Matematica | Fisica | Computacion | Astronomia
```

Cada uno de estos valores es un *constructor*, ya que al utilizarlos en una expresión, generan un valor del tipo `Carrera`.

a) Implementá el tipo `Carrera` como está definido arriba.

b) Definí la siguiente función, usando *pattern matching*: `titulo :: Carrera -> String` que devuelve el nombre completo de la carrera en forma de *string*. Por ejemplo, para el constructor `Matematica`, debe devolver "Licenciatura en Matemática".

c) Para escribir música se utiliza la denominada *notación musical*, la cual consta de notas (do, re, mi, ...). Además, estas notas pueden presentar algún modificador $\sharp$ (sostenido) o $\flat$ (bemol), por ejemplo $do\sharp$, $si\flat$, etc. Por ahora nos vamos a olvidar de estos modificadores (llamados alteraciones) y vamos a representar las notas básicas.
Definí el tipo `NotaBasica` con constructores `Do`, `Re`, `Mi`, `Fa`, `Sol`, `La` y `Si`

d) El sistema de notación musical anglosajón, también conocido como notación o cifrado americano, relaciona las notas básicas con letras de la A a la G. Este sistema se usa por ejemplo para las tablaturas de guitarra. Programá usando *pattern matching* la función:

```
cifradoAmericano :: NotaBasica -> Char
```

que relaciona las notas `Do`, `Re`, `Mi`, `Fa`, `Sol`, `La` y `Si` con los caracteres `'C'` , `'D'`, `'E'`, `'F'`, `'G'`, `'A'` y `'B'` respectivamente.

Laboratorio 8 **Clases de tipos.** En Haskell usamos el operador `(==)` para comparar valores del mismo tipo:

```
*Main> 4 == 5
False
*Main> 3 == (2 + 1)
True
```

Sin embargo, si intentamos comparar dos valores del tipo `Carrera` veremos que el intérprete nos mostrará un error similar al siguiente:

```
*Main> Matematica == Matematica
• No instance for (Eq Carrera) arising from a use of '=='
```

El problema es que todavía no hemos equipado al tipo nuevo `Carrera` con una noción de igualdad entre sus valores. ¿Cómo logramos eso en Haskell? Debemos garantizar que el tipo `Carrera` sea un miembro de la *clase* `Eq`. Conceptualmente, una clase es un conjunto de tipos que proveen ciertas operaciones especiales:

- Clase `Eq`: tipos que proveen una noción de igualdad (operador ==).
- Clase `Ord`: tipos que proveen una noción de orden (operadores <=, >=, funciones `min`, `max` y más).
- Clase `Bounded`: tipos que proveen una cota superior y una cota inferior para sus valores. Tienen entonces un elemento más grande, definido como la constante `maxBound`, y un elemento más chico, definido como `minBound`.
- Clase `Show`: tipos que proveen una representación en forma de texto (función `show`).
- Muchísimas más.

Podemos indicar al intérprete que infiera automáticamente la definición de una clase para un tipo dado en el momento de su definición, usando `deriving` como se muestra a continuación:

```
data Carrera = Matematica | Fisica | Computacion | Astronomia deriving Eq
```

Ahora es posible comparar carreras:

```
*Main> Matematica == Matematica
True
*Main> Matematica == Computacion
False
```

a) Completá la definición del tipo `NotaBasica` para que las expresiones

```
*Main> Do <= Re
*Main> Fa `min` Sol
```

sean válidas y no generen error. Ayuda: usar *deriving* con múltiples clases.

Laboratorio 9 **Polimorfismo ad hoc** Recordemos la función `sumatoria` del proyecto anterior:

```
sumatoria :: [Int] -> Int
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs
```

La función suma todos los elementos de una lista. Está claro que el algoritmo que se debe seguir para sumar una lista de números enteros y el algoritmo para sumar una lista de números decimales es idéntico. Ahora, si queremos sumar números decimales de tipo `Float` usando nuestra función:

```
*Main> sumatoria [1.5, 2.7, 0.8 :: Float]
    Couldn't match expected type 'Int' with actual type 'Float'
    (:)
```

El error era previsible ya que `sumatoria` no es polimórfica. Si tratamos de usar polimorfismo paramétrico:

```
sumatoria :: [a] -> a
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs
```

cuando recarguemos la definición de `sumatoria`:

```
*Main> :r
    No instance for (Num a) arising from a use of '+'
    (:)
    No instance for (Num a) arising from the literal '0'
    (:)
```

Esto sucede porque en la definición, la variable de tipo `a` no tiene ninguna restricción, por lo que el tipo no tiene que tener definida necesariamente la suma `(+)` ni la constante `0`. El algoritmo de la función `sumatoria` mientras trabaje con tipos numéricos como `Int`, `Integer`, `Float`, `Double` debería funcionar bien. Todos estos tipos numéricos (y otros más) son justamente los que están en la clase `Num`. Para restringir el polimorfismo de la variable `a` a esa clase de tipo se escribe:

```
sumatoria :: Num a => [a] -> a
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs
```

Este tipo de definiciones se llaman *polimorfismo ad hoc*, ya que no es una definición completamente genérica.

a) Definí usando polimorfismo *ad hoc* la función `minimoElemento` que calcula (de manera recursiva) cuál es el menor valor de una lista de tipo `[a]`. Asegurate que sólo esté definida para listas no vacías.

b) Definí la función `minimoElemento'` de manera tal que el caso base de la recursión sea el de la lista vacía. Para ello revisá la clase `Bounded`. **Ayuda:** Para probar esta función dentro de `ghci` con listas vacías, indicá el tipo concreto con tipos de la clase `Bounded`, por ejemplo: `([1,5,10]::[Int])`, `([]::[Bool])`, etc.

c) Usá la función `minimoElemento` para determinar la nota más grave de la melodía: `[Fa, La, Sol, Re, Fa]`

En las definiciones de los ejercicios siguientes, deben agregar *deriving* sólo cuando sea estrictamente necesario.

Laboratorio 10 **Sinónimo de tipos; constructores con parámetros.** En este ejercicio, introducimos dos conceptos: los sinónimos de tipos y tipos algebraicos cuyos constructores llevan argumentos. Un sinónimo de tipo nos permite definir un nuevo nombre para un tipo ya existente, como el ya conocido tipo `String` que no es otra cosa que un sinónimo para `[Char]`. Por ejemplo, si queremos modelar la altura (en centímetros) de una persona, podemos definir:

```
-- Altura es un sinonimo de tipo.
type Altura = Int
```

Los tipos algebraicos tienen constructores que llevan parámetros. Esos parámetros permiten agregar información, generando potencialmente infinitos valores dentro del tipo. Por ejemplo, si queremos modelar datos sobre

deportistas, podríamos definir los siguientes tipos:

```
-- Sinónimos de tipo
type Altura = Int
type NumCamiseta = Int

-- Tipos algebráicos sin parámetros (aka enumerados)
data Zona = Arco | Defensa | Mediocampo | Delantera
data TipoReves = DosManos | UnaMano
data Modalidad = Carretera | Pista | Monte | BMX
data PiernaHabil = Izquierda | Derecha
-- Sinónimo
type ManoHabil = PiernaHabil

-- Deportista es un tipo algebraico con constructores paramétricos
data Deportista = Ajedrecista                              -- Constructor sin argumentos
                | Ciclista Modalidad                       -- Constructor con un argumento
                | Velocista Altura                         -- Constructor con un argumento
                | Tenista TipoReves ManoHabil Altura       -- Constructor con tres argumentos
                | Futbolista Zona NumCamiseta PiernaHabil Altura  -- Constructor con ...
```

a) Implementá el tipo `Deportista` y todos sus tipos accesorios (`NumCamiseta`, `Altura`, `Zona`, etc) tal como están definidos arriba.

b) ¿Cuál es el tipo del constructor `Ciclista`?

c) Programá la función `contar_velocistas :: [Deportista] -> Int` que dada una lista de deportistas xs, devuelve la cantidad de velocistas que hay dentro de xs. Programar `contar_velocistas` sin usar igualdad, utilizando pattern matching.

d) Programá la función `contar_futbolistas :: [Deportista] -> Zona -> Int` que dada una lista de deportistas xs, y una zona z, devuelve la cantidad de futbolistas incluidos en xs que juegan en la zona z. No usar igualdad, sólo pattern matching.

e) ¿La función anterior usa `filter`? Si no es así, reprogramala para usarla.