

# Introducción a la Programación Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Introducción a la especificación de problemas

# Definición (Especificación) de un problema

problema *nombre*(*parámetros*) : tipo de dato del resultado {  
  requiere *etiqueta*: { condiciones sobre los parámetros de entrada }  
  asegura *etiqueta*: { condiciones sobre los parámetros de salida }  
}

- ▶ *nombre*: nombre que le damos al problema
  - ▶ será resuelto por una función con ese mismo nombre
- ▶ *parámetros*: lista de parámetros separada por comas, donde cada parámetro contiene:
  - ▶ Nombre del parámetro
  - ▶ Tipo de datos del parámetro
- ▶ *tipo de dato del resultado*: tipo de dato del resultado del problema (inicialmente especificaremos funciones)
  - ▶ En los asegura, podremos referenciar el valor devuelto con el nombre de **res**
- ▶ *etiquetas*: son nombres **opcionales** que nos servirán para nombrar declarativamente a las condiciones de los requiere o asegura.

# Retomando: Tipos de datos

- Un **tipo de datos** es un **conjunto de valores** (el conjunto base del tipo) provisto de una serie de **operaciones** que involucran a esos valores.

# Retomando: Tipos de datos

- ▶ Un **tipo de datos** es un **conjunto de valores** (el conjunto base del tipo) provisto de una serie de **operaciones** que involucran a esos valores.
- ▶ Para hablar de un elemento de un tipo  $T$  en nuestro lenguaje, escribimos un **término** o **expresión**

# Retomando: Tipos de datos

- ▶ Un **tipo de datos** es un conjunto de valores (el conjunto base del tipo) provisto de una serie de operaciones que involucran a esos valores.
- ▶ Para hablar de un elemento de un tipo  $T$  en nuestro lenguaje, escribimos un término o expresión
  - ▶ Variable de tipo  $T$

# Retomando: Tipos de datos

- ▶ Un **tipo de datos** es un **conjunto de valores** (el conjunto base del tipo) provisto de una serie de **operaciones** que involucran a esos valores.
- ▶ Para hablar de un elemento de un tipo  $T$  en nuestro lenguaje, escribimos un **término** o **expresión**
  - ▶ Variable de tipo  $T$  (ejemplos:  $x$ ,  $y$ ,  $z$ , etc)
  - ▶ Constante de tipo  $T$

# Retomando: Tipos de datos

- ▶ Un **tipo de datos** es un **conjunto de valores** (el conjunto base del tipo) provisto de una serie de **operaciones** que involucran a esos valores.
- ▶ Para hablar de un elemento de un tipo  $T$  en nuestro lenguaje, escribimos un **término** o **expresión**
  - ▶ Variable de tipo  $T$  (ejemplos:  $x$ ,  $y$ ,  $z$ , etc)
  - ▶ Constante de tipo  $T$  (ejemplos:  $1$ ,  $-1$ ,  $\frac{1}{5}$ , 'a', etc)
  - ▶ Función (operación) aplicada a otros términos (del tipo  $T$  o de otro tipo)

# Retomando: Tipos de datos

- ▶ Un **tipo de datos** es un **conjunto de valores** (el conjunto base del tipo) provisto de una serie de **operaciones** que involucran a esos valores.
- ▶ Para hablar de un elemento de un tipo  $T$  en nuestro lenguaje, escribimos un **término** o **expresión**
  - ▶ Variable de tipo  $T$  (ejemplos:  $x$ ,  $y$ ,  $z$ , etc)
  - ▶ Constante de tipo  $T$  (ejemplos:  $1$ ,  $-1$ ,  $\frac{1}{5}$ ,  $'a'$ , etc)
  - ▶ Función (operación) aplicada a otros términos (del tipo  $T$  o de otro tipo)
- ▶ Todos los tipos tienen un elemento distinguido:  $\perp$  o Indef



# Tipos de datos de nuestro lenguaje de especificación

- ▶ Básicos
  - ▶ Enteros ( $\mathbb{Z}$ )
  - ▶ Reales ( $\mathbb{R}$ )
  - ▶ Booleanos (Bool)
  - ▶ Caracteres (Char)
- ▶ Enumerados
- ▶ Uplas
- ▶ Secuencias

# Tipo upla (o tupla)

- ▶ Una estructura de datos es una forma particular de organizar la información.
- ▶ Uplas, de dos o más elementos, cada uno de cualquier tipo.
- ▶  $T_0 \times T_1 \times \cdots \times T_k$ : Tipo de las  $k$ -uplas de elementos de tipos  $T_0, T_1, \dots, T_k$ , respectivamente, donde  $k$  es fijo.

# Tipo upla (o tupla)

- ▶ Una estructura de datos es una forma particular de organizar la información.
- ▶ Uplas, de dos o más elementos, cada uno de cualquier tipo.
- ▶  $T_0 \times T_1 \times \cdots \times T_k$ : Tipo de las  $k$ -uplas de elementos de tipos  $T_0, T_1, \dots, T_k$ , respectivamente, donde  $k$  es fijo.
- ▶ Ejemplos:
  - ▶  $\mathbb{Z} \times \mathbb{Z}$  son los pares ordenados de enteros.

# Tipo upla (o tupla)

- ▶ Una estructura de datos es una forma particular de organizar la información.
- ▶ Uplas, de dos o más elementos, cada uno de cualquier tipo.
- ▶  $T_0 \times T_1 \times \cdots \times T_k$ : Tipo de las  $k$ -uplas de elementos de tipos  $T_0, T_1, \dots, T_k$ , respectivamente, donde  $k$  es fijo.
- ▶ Ejemplos:
  - ▶  $\mathbb{Z} \times \mathbb{Z}$  son los pares ordenados de enteros.
  - ▶  $\mathbb{Z} \times \text{Char} \times \text{Bool}$  son las triplas ordenadas con un entero, luego un carácter y luego un valor booleano.

# Tipo upla (o tupla)

- ▶ Una estructura de datos es una forma particular de organizar la información.
- ▶ Uplas, de dos o más elementos, cada uno de cualquier tipo.
- ▶  $T_0 \times T_1 \times \dots \times T_k$ : Tipo de las  $k$ -uplas de elementos de tipos  $T_0, T_1, \dots, T_k$ , respectivamente, donde  $k$  es fijo.
- ▶ Ejemplos:
  - ▶  $\mathbb{Z} \times \mathbb{Z}$  son los pares ordenados de enteros.
  - ▶  $\mathbb{Z} \times \text{Char} \times \text{Bool}$  son las triplas ordenadas con un entero, luego un carácter y luego un valor booleano.
- ▶  $n$ ésimo:  $(a_0, \dots, a_k)_m$  es el valor  $a_m$  en caso de que  $0 \leq m \leq k$ . Si no, está indefinido.
- ▶ Ejemplos:
  - ▶  $(7, 5)_0 = 7$
  - ▶  $('a', \text{DOM}, 78)_2 = 78$

# Secuencias

- **Secuencia:** Varios elementos del mismo tipo  $T$ , posiblemente repetidos, ubicados en un cierto orden.

# Secuencias

- ▶ **Secuencia:** Varios elementos del mismo tipo  $T$ , posiblemente repetidos, ubicados en un cierto orden.
- ▶  $seq\langle T \rangle$  es el tipo de las secuencias cuyos elementos son de tipo  $T$ .

# Secuencias

- ▶ **Secuencia:** Varios elementos del mismo tipo  $T$ , posiblemente repetidos, ubicados en un cierto orden.
- ▶  $seq\langle T \rangle$  es el tipo de las secuencias cuyos elementos son de tipo  $T$ .
- ▶  $T$  es un tipo arbitrario.
  - ▶ Hay secuencias de  $\mathbb{Z}$ , de Bool, de Días, de 5-uplas;



# Secuencias

- ▶ **Secuencia:** Varios elementos del mismo tipo  $T$ , posiblemente repetidos, ubicados en un cierto orden.
- ▶  $\text{seq}\langle T \rangle$  es el tipo de las secuencias cuyos elementos son de tipo  $T$ .
- ▶  $T$  es un tipo arbitrario.
  - ▶ Hay secuencias de  $\mathbb{Z}$ , de Bool, de Días, de 5-uplas;
  - ▶ también hay secuencias de secuencias de  $T$ ;
  - ▶ etcétera.

# Secuencias. Notación

- Una forma de escribir un elemento de tipo  $\text{seq}\langle T \rangle$  es escribir términos de tipo  $T$  separados por comas, entre  $\langle \dots \rangle$ .
  - $\langle 1, 2, 3, 4, 1, 0 \rangle$  es una secuencia de  $\mathbb{Z}$ .

# Secuencias. Notación

- ▶ Una forma de escribir un elemento de tipo  $seq\langle T \rangle$  es escribir términos de tipo  $T$  separados por comas, entre  $\langle \dots \rangle$ .
  - ▶  $\langle 1, 2, 3, 4, 1, 0 \rangle$  es una secuencia de  $\mathbb{Z}$ .
  - ▶  $\langle 1, 1 + 1, 3, 2 * 2, 5 \bmod 2, 0 \rangle$  es otra secuencia de  $\mathbb{Z}$  (igual a la anterior).

# Secuencias. Notación

- ▶ Una forma de escribir un elemento de tipo  $seq\langle T \rangle$  es escribir términos de tipo  $T$  separados por comas, entre  $\langle \dots \rangle$ .
  - ▶  $\langle 1, 2, 3, 4, 1, 0 \rangle$  es una secuencia de  $\mathbb{Z}$ .
  - ▶  $\langle 1, 1 + 1, 3, 2 * 2, 5 \bmod 2, 0 \rangle$  es otra secuencia de  $\mathbb{Z}$  (igual a la anterior).
- ▶ La **secuencia vacía** se escribe  $\langle \rangle$ , cualquiera sea el tipo de los elementos de la secuencia.

# Secuencias. Notación

- ▶ Una forma de escribir un elemento de tipo  $seq\langle T \rangle$  es escribir términos de tipo  $T$  separados por comas, entre  $\langle \dots \rangle$ .
  - ▶  $\langle 1, 2, 3, 4, 1, 0 \rangle$  es una secuencia de  $\mathbb{Z}$ .
  - ▶  $\langle 1, 1 + 1, 3, 2 * 2, 5 \bmod 2, 0 \rangle$  es otra secuencia de  $\mathbb{Z}$  (igual a la anterior).
- ▶ La **secuencia vacía** se escribe  $\langle \rangle$ , cualquiera sea el tipo de los elementos de la secuencia.
- ▶ Se puede formar secuencias de elementos de cualquier tipo.

# Secuencias. Notación

- ▶ Una forma de escribir un elemento de tipo  $seq\langle T \rangle$  es escribir términos de tipo  $T$  separados por comas, entre  $\langle \dots \rangle$ .
  - ▶  $\langle 1, 2, 3, 4, 1, 0 \rangle$  es una secuencia de  $\mathbb{Z}$ .
  - ▶  $\langle 1, 1 + 1, 3, 2 * 2, 5 \bmod 2, 0 \rangle$  es otra secuencia de  $\mathbb{Z}$  (igual a la anterior).
- ▶ La **secuencia vacía** se escribe  $\langle \rangle$ , cualquiera sea el tipo de los elementos de la secuencia.
- ▶ Se puede formar secuencias de elementos de cualquier tipo.
  - ▶ Como  $seq\langle \mathbb{Z} \rangle$  es un tipo, podemos armar secuencias de  $seq\langle \mathbb{Z} \rangle$  (secuencias de secuencias de  $\mathbb{Z}$ , o sea  $seq\langle seq\langle \mathbb{Z} \rangle \rangle$ ).

# Secuencias. Notación

- ▶ Una forma de escribir un elemento de tipo  $seq\langle T \rangle$  es escribir términos de tipo  $T$  separados por comas, entre  $\langle \dots \rangle$ .
  - ▶  $\langle 1, 2, 3, 4, 1, 0 \rangle$  es una secuencia de  $\mathbb{Z}$ .
  - ▶  $\langle 1, 1 + 1, 3, 2 * 2, 5 \bmod 2, 0 \rangle$  es otra secuencia de  $\mathbb{Z}$  (igual a la anterior).
- ▶ La **secuencia vacía** se escribe  $\langle \rangle$ , cualquiera sea el tipo de los elementos de la secuencia.
- ▶ Se puede formar secuencias de elementos de cualquier tipo.
  - ▶ Como  $seq\langle \mathbb{Z} \rangle$  es un tipo, podemos armar secuencias de  $seq\langle \mathbb{Z} \rangle$  (secuencias de secuencias de  $\mathbb{Z}$ , o sea  $seq\langle seq\langle \mathbb{Z} \rangle \rangle$ ).
  - ▶  $\langle \langle 12, 13 \rangle, \langle -3, 9, 0 \rangle, \langle 5 \rangle, \langle \rangle, \langle \rangle, \langle 3 \rangle \rangle$  es un elemento de tipo  $seq\langle seq\langle \mathbb{Z} \rangle \rangle$ .

# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo (*seq* $\langle\mathbb{Z}\rangle$ , *etc...*)

►  $\langle 1, 2, 3, 4, 5 \rangle$ ?



# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $seq\langle\mathbb{Z}\rangle$ , *etc...*)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Tipa como  $seq\langle\mathbb{Z}\rangle$  y  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle 1, true, 3, 4, 5 \rangle$ ?

# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $seq\langle\mathbb{Z}\rangle$ , etc...)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Tipa como  $seq\langle\mathbb{Z}\rangle$  y  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle 1, true, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea (Bool y  $\mathbb{Z}$ )
- ▶  $\langle 'a', 2, 3, 4, 5 \rangle$ ?

# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $seq\langle\mathbb{Z}\rangle$ , *etc...*)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Tipa como  $seq\langle\mathbb{Z}\rangle$  y  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle 1, true, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea (Bool y  $\mathbb{Z}$ )
- ▶  $\langle 'a', 2, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea (*Char* y  $\mathbb{Z}$ )
- ▶  $\langle 'H', 'o', 'l', 'a' \rangle$ ?

# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $seq\langle\mathbb{Z}\rangle$ , etc...)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Típa como  $seq\langle\mathbb{Z}\rangle$  y  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle 1, true, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea (Bool y  $\mathbb{Z}$ )
- ▶  $\langle 'a', 2, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea ( $Char$  y  $\mathbb{Z}$ )
- ▶  $\langle 'H', 'o', 'l', 'a' \rangle$ ? Bien Formada. Típa como  $seq\langle Char \rangle$
- ▶  $\langle true, false, true, true \rangle$ ?

# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $seq\langle\mathbb{Z}\rangle$ , etc...)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Tipa como  $seq\langle\mathbb{Z}\rangle$  y  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle 1, true, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea (Bool y  $\mathbb{Z}$ )
- ▶  $\langle 'a', 2, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea ( $Char$  y  $\mathbb{Z}$ )
- ▶  $\langle 'H', 'o', 'l', 'a' \rangle$ ? Bien Formada. Tipa como  $seq\langle Char \rangle$
- ▶  $\langle true, false, true, true \rangle$ ? Bien Formada. Tipa como  $seq\langle Bool \rangle$
- ▶  $\langle \frac{2}{5}, \pi, e \rangle$ ?

# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $seq\langle\mathbb{Z}\rangle$ , etc...)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Tipa como  $seq\langle\mathbb{Z}\rangle$  y  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle 1, true, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea (Bool y  $\mathbb{Z}$ )
- ▶  $\langle 'a', 2, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea ( $Char$  y  $\mathbb{Z}$ )
- ▶  $\langle 'H', 'o', 'l', 'a' \rangle$ ? Bien Formada. Tipa como  $seq\langle Char \rangle$
- ▶  $\langle true, false, true, true \rangle$ ? Bien Formada. Tipa como  $seq\langle Bool \rangle$
- ▶  $\langle \frac{2}{5}, \pi, e \rangle$ ? Bien Formada. Tipa como  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle \rangle$ ?

# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $seq\langle\mathbb{Z}\rangle$ , etc...)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Típa como  $seq\langle\mathbb{Z}\rangle$  y  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle 1, true, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea (Bool y  $\mathbb{Z}$ )
- ▶  $\langle 'a', 2, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea ( $Char$  y  $\mathbb{Z}$ )
- ▶  $\langle 'H', 'o', 'l', 'a' \rangle$ ? Bien Formada. Típa como  $seq\langle Char \rangle$
- ▶  $\langle true, false, true, true \rangle$ ? Bien Formada. Típa como  $seq\langle Bool \rangle$
- ▶  $\langle \frac{2}{5}, \pi, e \rangle$ ? Bien Formada. Típa como  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle \rangle$ ? Bien formada. Típa como cualquier secuencia  $seq\langle X \rangle$  donde  $X$  es un tipo válido.
- ▶  $\langle \langle \rangle \rangle$ ?

# Secuencias bien formadas

Indicar si las siguientes secuencias están bien formadas. Si están bien formadas, indicar su tipo ( $seq\langle\mathbb{Z}\rangle$ , etc...)

- ▶  $\langle 1, 2, 3, 4, 5 \rangle$ ? Bien Formada. Tipa como  $seq\langle\mathbb{Z}\rangle$  y  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle 1, true, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea (Bool y  $\mathbb{Z}$ )
- ▶  $\langle 'a', 2, 3, 4, 5 \rangle$ ? No está bien formada porque no es homogénea ( $Char$  y  $\mathbb{Z}$ )
- ▶  $\langle 'H', 'o', 'l', 'a' \rangle$ ? Bien Formada. Tipa como  $seq\langle Char \rangle$
- ▶  $\langle true, false, true, true \rangle$ ? Bien Formada. Tipa como  $seq\langle Bool \rangle$
- ▶  $\langle \frac{2}{5}, \pi, e \rangle$ ? Bien Formada. Tipa como  $seq\langle\mathbb{R}\rangle$
- ▶  $\langle \rangle$ ? Bien formada. Tipa como cualquier secuencia  $seq\langle X \rangle$  donde  $X$  es un tipo válido.
- ▶  $\langle \langle \rangle \rangle$ ? Bien formada. Tipa como cualquier secuencia  $seq\langle seq\langle X \rangle \rangle$  donde  $X$  es un tipo válido.



# Funciones sobre secuencias

## Longitud

- ▶ Longitud:  $length(a : seq\langle T \rangle) : \mathbb{Z}$ 
  - ▶ Representa la longitud de la secuencia  $a$ .
  - ▶ Notación:  $length(a)$  se puede escribir como  $|a|$  o como  $a.length$ .
- ▶ Ejemplos:
  - ▶  $|\langle \rangle| =$

# Funciones sobre secuencias

## Longitud

- ▶ Longitud:  $length(a : seq\langle T \rangle) : \mathbb{Z}$ 
  - ▶ Representa la longitud de la secuencia  $a$ .
  - ▶ Notación:  $length(a)$  se puede escribir como  $|a|$  o como  $a.length$ .
- ▶ Ejemplos:
  - ▶  $|\langle \rangle| = 0$
  - ▶  $|\langle 'H', 'o', 'l', 'a' \rangle| =$

# Funciones sobre secuencias

## Longitud

- ▶ Longitud:  $length(a : seq\langle T \rangle) : \mathbb{Z}$ 
  - ▶ Representa la longitud de la secuencia  $a$ .
  - ▶ Notación:  $length(a)$  se puede escribir como  $|a|$  o como  $a.length$ .
- ▶ Ejemplos:
  - ▶  $|\langle \rangle| = 0$
  - ▶  $|\langle 'H', 'o', 'l', 'a' \rangle| = 4$
  - ▶  $|\langle 1, 1, 2 \rangle| =$

# Funciones sobre secuencias

## Longitud

- ▶ Longitud:  $length(a : seq\langle T \rangle) : \mathbb{Z}$ 
  - ▶ Representa la longitud de la secuencia  $a$ .
  - ▶ Notación:  $length(a)$  se puede escribir como  $|a|$  o como  $a.length$ .
- ▶ Ejemplos:
  - ▶  $|\langle \rangle| = 0$
  - ▶  $|\langle 'H', 'o', 'l', 'a' \rangle| = 4$
  - ▶  $|\langle 1, 1, 2 \rangle| = 3$

# Funciones con secuencias

## $i$ -ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se define.

# Funciones con secuencias

## $i$ -ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se define.
- ▶ Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] =$

# Funciones con secuencias

## $i$ -ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se define.
- ▶ Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] = 'H'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[1] =$

# Funciones con secuencias

## *i*-ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se indefine.
- ▶ Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] = 'H'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[1] = 'o'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[2] =$



# Funciones con secuencias

## I-ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se indefine.
- ▶ Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] = 'H'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[1] = 'o'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[2] = 'l'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[3] =$

# Funciones con secuencias

## $i$ -ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se define.
- ▶ Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] = 'H'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[1] = 'o'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[2] = 'l'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[3] = 'a'$
  - ▶  $\langle 1, 1, 1, 1 \rangle[0] =$

# Funciones con secuencias

## $i$ -ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se indefine.
- ▶ Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] = 'H'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[1] = 'o'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[2] = 'l'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[3] = 'a'$
  - ▶  $\langle 1, 1, 1, 1 \rangle[0] = 1$
  - ▶  $\langle \rangle[0] =$

# Funciones con secuencias

## $i$ -ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se indefine.
- ▶ Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] = 'H'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[1] = 'o'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[2] = 'l'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[3] = 'a'$
  - ▶  $\langle 1, 1, 1, 1 \rangle[0] = 1$
  - ▶  $\langle \rangle[0] = \perp$  (Indefinido)
  - ▶  $\langle 1, 1, 1, 1 \rangle[7] =$

# Funciones con secuencias

## I-ésimo elemento

- ▶ Indexación:  $\text{seq}\langle T \rangle[i : \mathbb{Z}] : T$ 
  - ▶ Requiere  $0 \leq i < |a|$ .
  - ▶ Es el elemento en la  $i$ -ésima posición de  $a$ .
  - ▶ La primera posición es la 0.
  - ▶ Notación:  $a[i]$ .
  - ▶ Si no vale  $0 \leq i < |a|$  se indefine.
- ▶ Ejemplos:
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[0] = 'H'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[1] = 'o'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[2] = 'l'$
  - ▶  $\langle 'H', 'o', 'l', 'a' \rangle[3] = 'a'$
  - ▶  $\langle 1, 1, 1, 1 \rangle[0] = 1$
  - ▶  $\langle \rangle[0] = \perp$  (Indefinido)
  - ▶  $\langle 1, 1, 1, 1 \rangle[7] = \perp$  (Indefinido)

# Funciones con secuencias

## Pertenece

- ▶ Pertenece:  $pertenece(x : T, s : seq\langle T \rangle) : Bool$ 
  - ▶ Es **true** sí y solo sí  $x$  es elemento de  $s$ .
  - ▶ Notación:  $pertenece(x, s)$  se puede escribir como  $x \in s$ .
- ▶ Ejemplos:
  - ▶  $(1, MAR) \in \langle (1, LUN), (2, MAR), (3, JUE), (1, MAR) \rangle$  ?

# Funciones con secuencias

## Pertenece

- ▶ Pertenece:  $pertenece(x : T, s : seq\langle T \rangle) : Bool$ 
  - ▶ Es **true** sí y solo sí  $x$  es elemento de  $s$ .
  - ▶ Notación:  $pertenece(x, s)$  se puede escribir como  $x \in s$ .
- ▶ Ejemplos:
  - ▶  $(1, MAR) \in \langle (1, LUN), (2, MAR), (3, JUE), (1, MAR) \rangle$  ? **true**
  - ▶  $(1, MAR) \in \langle (1, LUN), (2, MAR), (3, JUE), (3, MAR) \rangle$  ?

# Funciones con secuencias

## Pertenece

- ▶ Pertenece:  $pertenece(x : T, s : seq\langle T \rangle) : Bool$ 
  - ▶ Es **true** sí y solo sí  $x$  es elemento de  $s$ .
  - ▶ Notación:  $pertenece(x, s)$  se puede escribir como  $x \in s$ .
- ▶ Ejemplos:
  - ▶  $(1, MAR) \in \langle (1, LUN), (2, MAR), (3, JUE), (1, MAR) \rangle$  ? **true**
  - ▶  $(1, MAR) \in \langle (1, LUN), (2, MAR), (3, JUE), (3, MAR) \rangle$  ? **false**



# Funciones con secuencias

## Igualdad

Dos secuencias  $s_0$  y  $s_1$  (notación  $s_0 = s_1$ ) son iguales si y sólo si

- ▶ Tienen la misma cantidad de elementos
- ▶ Dada una posición, el elemento contenido en la secuencia  $s_0$  es igual al elemento contenido en la secuencia  $s_1$ .

Ejemplos:

- ▶  $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$  ?

# Funciones con secuencias

## Igualdad

Dos secuencias  $s_0$  y  $s_1$  (notación  $s_0 = s_1$ ) son iguales si y sólo si

- ▶ Tienen la misma cantidad de elementos
- ▶ Dada una posición, el elemento contenido en la secuencia  $s_0$  es igual al elemento contenido en la secuencia  $s_1$ .

Ejemplos:

- ▶  $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$  ? Sí
- ▶  $\langle \rangle = \langle \rangle$  ?

# Funciones con secuencias

## Igualdad

Dos secuencias  $s_0$  y  $s_1$  (notación  $s_0 = s_1$ ) son iguales si y sólo si

- ▶ Tienen la misma cantidad de elementos
- ▶ Dada una posición, el elemento contenido en la secuencia  $s_0$  es igual al elemento contenido en la secuencia  $s_1$ .

Ejemplos:

- ▶  $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$  ? Sí
- ▶  $\langle \rangle = \langle \rangle$  ? Sí
- ▶  $\langle 4, 4, 4 \rangle = \langle 4, 4, 4 \rangle$  ?

# Funciones con secuencias

## Igualdad

Dos secuencias  $s_0$  y  $s_1$  (notación  $s_0 = s_1$ ) son iguales si y sólo si

- ▶ Tienen la misma cantidad de elementos
- ▶ Dada una posición, el elemento contenido en la secuencia  $s_0$  es igual al elemento contenido en la secuencia  $s_1$ .

Ejemplos:

- ▶  $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$  ? Sí
- ▶  $\langle \rangle = \langle \rangle$  ? Sí
- ▶  $\langle 4, 4, 4 \rangle = \langle 4, 4, 4 \rangle$  ? Sí
- ▶  $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 3, 4 \rangle$  ?

# Funciones con secuencias

## Igualdad

Dos secuencias  $s_0$  y  $s_1$  (notación  $s_0 = s_1$ ) son iguales si y sólo si

- ▶ Tienen la misma cantidad de elementos
- ▶ Dada una posición, el elemento contenido en la secuencia  $s_0$  es igual al elemento contenido en la secuencia  $s_1$ .

Ejemplos:

- ▶  $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$  ? Sí
- ▶  $\langle \rangle = \langle \rangle$  ? Sí
- ▶  $\langle 4, 4, 4 \rangle = \langle 4, 4, 4 \rangle$  ? Sí
- ▶  $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 3, 4 \rangle$  ? No
- ▶  $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 4, 5, 6 \rangle$  ?

# Funciones con secuencias

## Igualdad

Dos secuencias  $s_0$  y  $s_1$  (notación  $s_0 = s_1$ ) son iguales si y sólo si

- ▶ Tienen la misma cantidad de elementos
- ▶ Dada una posición, el elemento contenido en la secuencia  $s_0$  es igual al elemento contenido en la secuencia  $s_1$ .

Ejemplos:

- ▶  $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$  ? Sí
- ▶  $\langle \rangle = \langle \rangle$  ? Sí
- ▶  $\langle 4, 4, 4 \rangle = \langle 4, 4, 4 \rangle$  ? Sí
- ▶  $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 3, 4 \rangle$  ? No
- ▶  $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 4, 5, 6 \rangle$  ? No
- ▶  $\langle 1, 2, 3, 5, 4 \rangle = \langle 1, 2, 3, 4, 5 \rangle$  ?

# Funciones con secuencias

## Igualdad

Dos secuencias  $s_0$  y  $s_1$  (notación  $s_0 = s_1$ ) son iguales si y sólo si

- ▶ Tienen la misma cantidad de elementos
- ▶ Dada una posición, el elemento contenido en la secuencia  $s_0$  es igual al elemento contenido en la secuencia  $s_1$ .

Ejemplos:

- ▶  $\langle 1, 2, 3, 4 \rangle = \langle 1, 2, 3, 4 \rangle$  ? Sí
- ▶  $\langle \rangle = \langle \rangle$  ? Sí
- ▶  $\langle 4, 4, 4 \rangle = \langle 4, 4, 4 \rangle$  ? Sí
- ▶  $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 3, 4 \rangle$  ? No
- ▶  $\langle 1, 2, 3, 4, 5 \rangle = \langle 1, 2, 4, 5, 6 \rangle$  ? No
- ▶  $\langle 1, 2, 3, 5, 4 \rangle = \langle 1, 2, 3, 4, 5 \rangle$  ? No

# Funciones con secuencias

## Cabeza o Head

- ▶ Cabeza:  $head(a : seq\langle T \rangle) : T$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es el primer elemento de la secuencia  $a$ .
  - ▶ Es equivalente a la expresión  $a[0]$ .
  - ▶ Si no vale  $|a| > 0$  se indefine.
- ▶ Ejemplos:
  - ▶  $head(\langle 'H', 'o', 'l', 'a' \rangle) =$



# Funciones con secuencias

## Cabeza o Head

- ▶ Cabeza:  $head(a : seq\langle T \rangle) : T$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es el primer elemento de la secuencia  $a$ .
  - ▶ Es equivalente a la expresión  $a[0]$ .
  - ▶ Si no vale  $|a| > 0$  se indefine.
- ▶ Ejemplos:
  - ▶  $head(\langle 'H', 'o', 'l', 'a' \rangle) = 'H'$
  - ▶  $head(\langle 1, 1, 1, 1 \rangle) =$

# Funciones con secuencias

## Cabeza o Head

- ▶ Cabeza:  $head(a : seq\langle T \rangle) : T$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es el primer elemento de la secuencia  $a$ .
  - ▶ Es equivalente a la expresión  $a[0]$ .
  - ▶ Si no vale  $|a| > 0$  se indefine.
- ▶ Ejemplos:
  - ▶  $head(\langle 'H', 'o', 'l', 'a' \rangle) = 'H'$
  - ▶  $head(\langle 1, 1, 1, 1 \rangle) = 1$
  - ▶  $head(\langle \rangle) =$

# Funciones con secuencias

## Cabeza o Head

- ▶ Cabeza:  $head(a : seq\langle T \rangle) : T$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es el primer elemento de la secuencia  $a$ .
  - ▶ Es equivalente a la expresión  $a[0]$ .
  - ▶ Si no vale  $|a| > 0$  se indefine.
- ▶ Ejemplos:
  - ▶  $head(\langle 'H', 'o', 'l', 'a' \rangle) = 'H'$
  - ▶  $head(\langle 1, 1, 1, 1 \rangle) = 1$
  - ▶  $head(\langle \rangle) = \perp$  (Indefinido)

# Funciones con secuencias

## Cola o Tail

- ▶ Cola:  $tail(a : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es la secuencia resultante de eliminar su primer elemento.
  - ▶ Si no vale  $|a| > 0$  se indefin.
- ▶ Ejemplos:
  - ▶  $tail(\langle 'H', 'o', 'l', 'a' \rangle) =$

# Funciones con secuencias

## Cola o Tail

- ▶ Cola:  $tail(a : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es la secuencia resultante de eliminar su primer elemento.
  - ▶ Si no vale  $|a| > 0$  se indefin.
- ▶ Ejemplos:
  - ▶  $tail(\langle 'H', 'o', 'l', 'a' \rangle) = \langle 'o', 'l', 'a' \rangle$
  - ▶  $tail(\langle 1, 1, 1, 1 \rangle) =$

# Funciones con secuencias

## Cola o Tail

- ▶ Cola:  $tail(a : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es la secuencia resultante de eliminar su primer elemento.
  - ▶ Si no vale  $|a| > 0$  se indefin.
- ▶ Ejemplos:
  - ▶  $tail(\langle 'H', 'o', 'l', 'a' \rangle) = \langle 'o', 'l', 'a' \rangle$
  - ▶  $tail(\langle 1, 1, 1, 1 \rangle) = \langle 1, 1, 1 \rangle$
  - ▶  $tail(\langle \rangle) =$

# Funciones con secuencias

## Cola o Tail

- ▶ Cola:  $tail(a : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es la secuencia resultante de eliminar su primer elemento.
  - ▶ Si no vale  $|a| > 0$  se indefin.
- ▶ Ejemplos:
  - ▶  $tail(\langle 'H', 'o', 'l', 'a' \rangle) = \langle 'o', 'l', 'a' \rangle$
  - ▶  $tail(\langle 1, 1, 1, 1 \rangle) = \langle 1, 1, 1 \rangle$
  - ▶  $tail(\langle \rangle) = \perp$  (Indefinido)
  - ▶  $tail(\langle 6 \rangle) =$

# Funciones con secuencias

## Cola o Tail

- ▶ Cola:  $tail(a : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Requiere  $|a| > 0$ .
  - ▶ Es la secuencia resultante de eliminar su primer elemento.
  - ▶ Si no vale  $|a| > 0$  se indefin.
- ▶ Ejemplos:
  - ▶  $tail(\langle 'H', 'o', 'l', 'a' \rangle) = \langle 'o', 'l', 'a' \rangle$
  - ▶  $tail(\langle 1, 1, 1, 1 \rangle) = \langle 1, 1, 1 \rangle$
  - ▶  $tail(\langle \rangle) = \perp$  (Indefinido)
  - ▶  $tail(\langle 6 \rangle) = \langle \rangle$



# Funciones con secuencias

## Agregar al principio o `addFirst`

- ▶ Agregar cabeza:  $addFirst(t : T, a : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , agregándole  $t$  como primer elemento.
  - ▶ Es una función que no se indefine
- ▶ Ejemplos:
  - ▶  $addFirst('x', \langle 'H', 'o', 'l', 'a' \rangle) =$

# Funciones con secuencias

## Agregar al principio o `addFirst`

- ▶ Agregar cabeza:  $addFirst(t : T, a : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , agregándole  $t$  como primer elemento.
  - ▶ Es una función que no se indefine
- ▶ Ejemplos:
  - ▶  $addFirst('x', \langle 'H', 'o', 'l', 'a' \rangle) = \langle 'x', 'H', 'o', 'l', 'a' \rangle$
  - ▶  $addFirst(5, \langle 1, 1, 1, 1 \rangle) =$

# Funciones con secuencias

## Agregar al principio o `addFirst`

- ▶ Agregar cabeza:  $addFirst(t : T, a : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , agregándole  $t$  como primer elemento.
  - ▶ Es una función que no se indefine
- ▶ Ejemplos:
  - ▶  $addFirst('x', \langle 'H', 'o', 'l', 'a' \rangle) = \langle 'x', 'H', 'o', 'l', 'a' \rangle$
  - ▶  $addFirst(5, \langle 1, 1, 1, 1 \rangle) = \langle 5, 1, 1, 1, 1 \rangle$
  - ▶  $addFirst(1, \langle \rangle) =$

# Funciones con secuencias

Agregar al principio o `addFirst`

- ▶ Agregar cabeza:  $\text{addFirst}(t : T, a : \text{seq}\langle T \rangle) : \text{seq}\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , agregándole  $t$  como primer elemento.
  - ▶ Es una función que no se indefine
- ▶ Ejemplos:
  - ▶  $\text{addFirst}('x', \langle 'H', 'o', 'l', 'a' \rangle) = \langle 'x', 'H', 'o', 'l', 'a' \rangle$
  - ▶  $\text{addFirst}(5, \langle 1, 1, 1, 1 \rangle) = \langle 5, 1, 1, 1, 1 \rangle$
  - ▶  $\text{addFirst}(1, \langle \rangle) = \langle 1 \rangle$

# Funciones con secuencias

## Concatenación o concat

- ▶ Concatenación:  $\text{concat}(a : \text{seq}\langle T \rangle, b : \text{seq}\langle T \rangle) : \text{seq}\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , seguidos de los de  $b$ .
  - ▶ Notación:  $\text{concat}(a, b)$  se puede escribir  $a ++ b$ .

# Funciones con secuencias

## Concatenación o concat

- ▶ Concatenación:  $concat(a : seq\langle T \rangle, b : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , seguidos de los de  $b$ .
  - ▶ Notación:  $concat(a, b)$  se puede escribir  $a ++ b$ .
- ▶ Ejemplos:
  - ▶  $concat(\langle 'H', 'o' \rangle, \langle 'l', 'a' \rangle) =$

# Funciones con secuencias

## Concatenación o concat

- ▶ Concatenación:  $concat(a : seq\langle T \rangle, b : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , seguidos de los de  $b$ .
  - ▶ Notación:  $concat(a, b)$  se puede escribir  $a ++ b$ .
- ▶ Ejemplos:
  - ▶  $concat(\langle 'H', 'o' \rangle, \langle 'l', 'a' \rangle) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $concat(\langle 1, 2 \rangle, \langle 3, 4 \rangle) =$

# Funciones con secuencias

## Concatenación o concat

- ▶ Concatenación:  $concat(a : seq\langle T \rangle, b : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , seguidos de los de  $b$ .
  - ▶ Notación:  $concat(a, b)$  se puede escribir  $a ++ b$ .
- ▶ Ejemplos:
  - ▶  $concat(\langle 'H', 'o' \rangle, \langle 'l', 'a' \rangle) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $concat(\langle 1, 2 \rangle, \langle 3, 4 \rangle) = \langle 1, 2, 3, 4 \rangle$
  - ▶  $concat(\langle \rangle, \langle \rangle) =$



# Funciones con secuencias

## Concatenación o concat

- ▶ Concatenación:  $concat(a : seq\langle T \rangle, b : seq\langle T \rangle) : seq\langle T \rangle$ 
  - ▶ Es una secuencia con los elementos de  $a$ , seguidos de los de  $b$ .
  - ▶ Notación:  $concat(a, b)$  se puede escribir  $a ++ b$ .
- ▶ Ejemplos:
  - ▶  $concat(\langle 'H', 'o' \rangle, \langle 'l', 'a' \rangle) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $concat(\langle 1, 2 \rangle, \langle 3, 4 \rangle) = \langle 1, 2, 3, 4 \rangle$
  - ▶  $concat(\langle \rangle, \langle \rangle) = \langle \rangle$
  - ▶  $concat(\langle 2, 3 \rangle, \langle \rangle) =$

# Funciones con secuencias

## Concatenación o concat

- Concatenación:  $concat(a : seq\langle T \rangle, b : seq\langle T \rangle) : seq\langle T \rangle$ 
  - Es una secuencia con los elementos de  $a$ , seguidos de los de  $b$ .
  - Notación:  $concat(a, b)$  se puede escribir  $a ++ b$ .
- Ejemplos:
  - $concat(\langle 'H', 'o' \rangle, \langle 'l', 'a' \rangle) = \langle 'H', 'o', 'l', 'a' \rangle$
  - $concat(\langle 1, 2 \rangle, \langle 3, 4 \rangle) = \langle 1, 2, 3, 4 \rangle$
  - $concat(\langle \rangle, \langle \rangle) = \langle \rangle$
  - $concat(\langle 2, 3 \rangle, \langle \rangle) = \langle 2, 3 \rangle$
  - $concat(\langle \rangle, \langle 5, 7 \rangle) =$

# Funciones con secuencias

## Concatenación o concat

- Concatenación:  $concat(a : seq\langle T \rangle, b : seq\langle T \rangle) : seq\langle T \rangle$ 
  - Es una secuencia con los elementos de  $a$ , seguidos de los de  $b$ .
  - Notación:  $concat(a, b)$  se puede escribir  $a ++ b$ .
- Ejemplos:
  - $concat(\langle 'H', 'o' \rangle, \langle 'l', 'a' \rangle) = \langle 'H', 'o', 'l', 'a' \rangle$
  - $concat(\langle 1, 2 \rangle, \langle 3, 4 \rangle) = \langle 1, 2, 3, 4 \rangle$
  - $concat(\langle \rangle, \langle \rangle) = \langle \rangle$
  - $concat(\langle 2, 3 \rangle, \langle \rangle) = \langle 2, 3 \rangle$
  - $concat(\langle \rangle, \langle 5, 7 \rangle) = \langle 5, 7 \rangle$

# Funciones con secuencias

## Subsecuencia o subseq

- ▶ Subsecuencia:  $subseq(a : seq\langle T \rangle, d, h : \mathbb{Z}) : seq\langle T \rangle$ 
  - ▶ Es una sublista de  $a$  en las posiciones entre  $d$  (inclusive) y  $h$  (exclusive).
  - ▶ Cuando  $0 \leq d = h \leq |a|$ , retorna la secuencia vacía.
  - ▶ Cuando no se cumple  $0 \leq d \leq h \leq |a|$ , se **indefine**!
- ▶ Ejemplos:
  - ▶  $subseq(\langle 'H', 'o', 'l', 'a' \rangle, 0, 1) =$

# Funciones con secuencias

## Subsecuencia o subseq

- ▶ Subsecuencia:  $\text{subseq}(a : \text{seq}\langle T \rangle, d, h : \mathbb{Z}) : \text{seq}\langle T \rangle$ 
  - ▶ Es una sublista de  $a$  en las posiciones entre  $d$  (inclusive) y  $h$  (exclusive).
  - ▶ Cuando  $0 \leq d = h \leq |a|$ , retorna la secuencia vacía.
  - ▶ Cuando no se cumple  $0 \leq d \leq h \leq |a|$ , **se *undefine*!**
- ▶ Ejemplos:
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 1) = \langle 'H' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 4) =$

# Funciones con secuencias

## Subsecuencia o subseq

- ▶ Subsecuencia:  $subseq(a : seq\langle T \rangle, d, h : \mathbb{Z}) : seq\langle T \rangle$ 
  - ▶ Es una sublista de  $a$  en las posiciones entre  $d$  (inclusive) y  $h$  (exclusive).
  - ▶ Cuando  $0 \leq d = h \leq |a|$ , retorna la secuencia vacía.
  - ▶ Cuando no se cumple  $0 \leq d \leq h \leq |a|$ , **se *undefine*!**
- ▶ Ejemplos:
  - ▶  $subseq(\langle 'H', 'o', 'l', 'a' \rangle, 0, 1) = \langle 'H' \rangle$
  - ▶  $subseq(\langle 'H', 'o', 'l', 'a' \rangle, 0, 4) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $subseq(\langle 'H', 'o', 'l', 'a' \rangle, 2, 2) =$

# Funciones con secuencias

## Subsecuencia o subseq

- ▶ Subsecuencia:  $subseq(a : seq\langle T \rangle, d, h : \mathbb{Z}) : seq\langle T \rangle$ 
  - ▶ Es una sublista de  $a$  en las posiciones entre  $d$  (inclusive) y  $h$  (exclusive).
  - ▶ Cuando  $0 \leq d = h \leq |a|$ , retorna la secuencia vacía.
  - ▶ Cuando no se cumple  $0 \leq d \leq h \leq |a|$ , **se *indefine*!**
- ▶ Ejemplos:
  - ▶  $subseq(\langle 'H', 'o', 'l', 'a' \rangle, 0, 1) = \langle 'H' \rangle$
  - ▶  $subseq(\langle 'H', 'o', 'l', 'a' \rangle, 0, 4) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $subseq(\langle 'H', 'o', 'l', 'a' \rangle, 2, 2) = \langle \rangle$
  - ▶  $subseq(\langle 'H', 'o', 'l', 'a' \rangle, -1, 3) =$

# Funciones con secuencias

## Subsecuencia o subseq

- ▶ Subsecuencia:  $\text{subseq}(a : \text{seq}\langle T \rangle, d, h : \mathbb{Z}) : \text{seq}\langle T \rangle$ 
  - ▶ Es una sublista de  $a$  en las posiciones entre  $d$  (inclusive) y  $h$  (exclusive).
  - ▶ Cuando  $0 \leq d = h \leq |a|$ , retorna la secuencia vacía.
  - ▶ Cuando no se cumple  $0 \leq d \leq h \leq |a|$ , **se *undefine*!**
- ▶ Ejemplos:
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 1) = \langle 'H' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 4) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 2, 2) = \langle \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, -1, 3) = \perp$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 10) =$



# Funciones con secuencias

## Subsecuencia o subseq

- ▶ Subsecuencia:  $\text{subseq}(a : \text{seq}\langle T \rangle, d, h : \mathbb{Z}) : \text{seq}\langle T \rangle$ 
  - ▶ Es una sublista de  $a$  en las posiciones entre  $d$  (inclusive) y  $h$  (exclusive).
  - ▶ Cuando  $0 \leq d = h \leq |a|$ , retorna la secuencia vacía.
  - ▶ Cuando no se cumple  $0 \leq d \leq h \leq |a|$ , **se *indefine*!**
- ▶ Ejemplos:
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 1) = \langle 'H' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 4) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 2, 2) = \langle \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, -1, 3) = \perp$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 10) = \perp$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 3, 1) =$

# Funciones con secuencias

## Subsecuencia o subseq

- ▶ Subsecuencia:  $\text{subseq}(a : \text{seq}\langle T \rangle, d, h : \mathbb{Z}) : \text{seq}\langle T \rangle$ 
  - ▶ Es una sublista de  $a$  en las posiciones entre  $d$  (inclusive) y  $h$  (exclusive).
  - ▶ Cuando  $0 \leq d = h \leq |a|$ , retorna la secuencia vacía.
  - ▶ Cuando no se cumple  $0 \leq d \leq h \leq |a|$ , **se *indefine*!**
- ▶ Ejemplos:
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 1) = \langle 'H' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 4) = \langle 'H', 'o', 'l', 'a' \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 2, 2) = \langle \rangle$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, -1, 3) = \perp$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 10) = \perp$
  - ▶  $\text{subseq}(\langle 'H', 'o', 'l', 'a' \rangle, 3, 1) = \perp$

# Funciones con secuencias

- ▶ Cambiar una posición:  $setAt(a : seq\langle T \rangle, i : \mathbb{Z}, val : T) : seq\langle T \rangle$ 
  - ▶ Requiere  $0 \leq i < |a|$
  - ▶ Es una secuencia igual a  $a$ , pero con valor  $val$  en la posición  $i$ .
- ▶ Ejemplos:
  - ▶  $setAt(\langle 'H', 'o', 'l', 'a' \rangle, 0, 'X') =$

# Funciones con secuencias

- ▶ Cambiar una posición:  $\text{setAt}(a : \text{seq}\langle T \rangle, i : \mathbb{Z}, \text{val} : T) : \text{seq}\langle T \rangle$ 
  - ▶ Requiere  $0 \leq i < |a|$
  - ▶ Es una secuencia igual a  $a$ , pero con valor  $\text{val}$  en la posición  $i$ .
- ▶ Ejemplos:
  - ▶  $\text{setAt}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 'X') = \langle 'X', 'o', 'l', 'a' \rangle$
  - ▶  $\text{setAt}(\langle 'H', 'o', 'l', 'a' \rangle, 3, 'A') =$

# Funciones con secuencias

- ▶ Cambiar una posición:  $\text{setAt}(a : \text{seq}\langle T \rangle, i : \mathbb{Z}, \text{val} : T) : \text{seq}\langle T \rangle$ 
  - ▶ Requiere  $0 \leq i < |a|$
  - ▶ Es una secuencia igual a  $a$ , pero con valor  $\text{val}$  en la posición  $i$ .
- ▶ Ejemplos:
  - ▶  $\text{setAt}(\langle 'H', 'o', 'l', 'a' \rangle, 0, 'X') = \langle 'X', 'o', 'l', 'a' \rangle$
  - ▶  $\text{setAt}(\langle 'H', 'o', 'l', 'a' \rangle, 3, 'A') = \langle 'H', 'o', 'l', 'A' \rangle$
  - ▶  $\text{setAt}(\langle \rangle, 0, 5) =$

# Funciones con secuencias

- ▶ Cambiar una posición:  $setAt(a : seq\langle T \rangle, i : \mathbb{Z}, val : T) : seq\langle T \rangle$ 
  - ▶ Requiere  $0 \leq i < |a|$
  - ▶ Es una secuencia igual a  $a$ , pero con valor  $val$  en la posición  $i$ .
- ▶ Ejemplos:
  - ▶  $setAt(\langle 'H', 'o', 'l', 'a' \rangle, 0, 'X') = \langle 'X', 'o', 'l', 'a' \rangle$
  - ▶  $setAt(\langle 'H', 'o', 'l', 'a' \rangle, 3, 'A') = \langle 'H', 'o', 'l', 'A' \rangle$
  - ▶  $setAt(\langle \rangle, 0, 5) = \perp$  (Indefinido)

# Operaciones sobre secuencias

- ▶  $length(a : seq\langle T \rangle) : \mathbb{Z}$  (notación  $|a|$ )
- ▶  $pertenece(x : T, s : seq\langle T \rangle) : Bool$  (notación  $x \in s$ )
- ▶ indexación:  $seq\langle T \rangle[i : \mathbb{Z}] : T$
- ▶ igualdad:  $seq\langle T \rangle = seq\langle T \rangle$
- ▶  $head(a : seq\langle T \rangle) : T$
- ▶  $tail(a : seq\langle T \rangle) : seq\langle T \rangle$
- ▶  $addFirst(t : T, a : seq\langle T \rangle) : seq\langle T \rangle$
- ▶  $concat(a : seq\langle T \rangle, b : seq\langle T \rangle) : seq\langle T \rangle$  (notación  $a++b$ )
- ▶  $subseq(a : seq\langle T \rangle, d, h : \mathbb{Z}) : \langle T \rangle$
- ▶  $setAt(a : seq\langle T \rangle, i : \mathbb{Z}, val : T) : seq\langle T \rangle$

# Definición (Especificación) de un problema

## ► *Sobre los requiere*

- Describen todas las condiciones y posibles valores o casuísticas de los parámetros de entrada.
- Puede haber más de un requiere (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
- Evitar contradicciones (un requiere no debería contradecir a otro).

## ► *Sobre los asegura*

- Describen todas las condiciones y posibles valores o casuísticas de los parámetros de salida y entrada/salida en función de los parámetros de entrada.
- Puede haber más de un asegura (recomendamos una condición por renglón). Se asume que valen todos juntos (es una conjunción).
- Evitar contradicciones (un asegura no debería contradecir a otro).



# Problemas comunes de las especificaciones

- ▶ ¿Qué sucede si especifico de menos?
- ▶ ¿Qué sucede si especifico de más?

# Sobre-especificación

# Sobre-especificación

- Consiste en dar una **postcondición más restrictiva** de la que se necesita, o bien dar una **precondición más laxa**.

# Sobre-especificación

- ▶ Consiste en dar una **postcondición más restrictiva** de la que se necesita, o bien dar una **precondición más laxa**.
- ▶ Limita los posibles algoritmos que resuelven el problema, porque impone más condiciones para la salida, o amplía los datos de entrada.

# Sobre-especificación

- Consiste en dar una **postcondición más restrictiva** de la que se necesita, o bien dar una **precondición más laxa**.
- Limita los posibles algoritmos que resuelven el problema, porque impone más condiciones para la salida, o amplía los datos de entrada.

- Ejemplo:

```
problema distinto( $x : \mathbb{Z}$ ) :  $\mathbb{Z}$  {  
  requiere: { True }  
  asegura: {  $res = x + 1$  }  
}
```

- ... en lugar de:

```
problema distinto( $x : \mathbb{Z}$ ) :  $\mathbb{Z}$ {  
  requiere: { True }  
  asegura: {  $res \neq x$  }  
}
```

# Sub-especificación

- Consiste en dar una **precondición más restrictiva** de lo realmente necesario, o bien una **postcondición más débil** de la que se necesita.

# Sub-especificación

- ▶ Consiste en dar una **precondición más restrictiva** de lo realmente necesario, o bien una **postcondición más débil** de la que se necesita.
- ▶ Deja afuera datos de entrada o ignora condiciones necesarias para la salida (permite soluciones no deseadas).

# Sub-especificación

- ▶ Consiste en dar una **precondición más restrictiva** de lo realmente necesario, o bien una **postcondición más débil** de la que se necesita.
- ▶ Deja afuera datos de entrada o ignora condiciones necesarias para la salida (permite soluciones no deseadas).
- ▶ Ejemplo:

```
problema distinto( $x : \mathbb{Z}$ ) :  $\mathbb{Z}$ {  
  requiere:  $\{x > 0\}$   
  asegura:  $\{res \neq x\}$   
}
```

... en vez de:

```
problema distinto( $x : \mathbb{Z}$ ) :  $\mathbb{Z}$ {  
  requiere:  $\{True\}$   
  asegura:  $\{res \neq x\}$   
}
```



# Modularizacion

Partiendo un problema en problemas mas chicos

Dadas dos secuencias, queremos saber si uno es una permutación<sup>1</sup> de la otra secuencia:

¿Cuándo será una secuencia permutación de la otra?

---

<sup>1</sup> mismos elementos y misma cantidad por cada elemento, en un orden potencialmente distinto

# Modularizacion

Partiendo un problema en problemas mas chicos

Dadas dos secuencias, queremos saber si uno es una permutación<sup>1</sup> de la otra secuencia:

¿Cuándo será una secuencia permutación de la otra?

- ▶ Tienen los mismos elementos
- ▶ Cada elemento aparece la misma cantidad de veces en ambas secuencias

---

<sup>1</sup> mismos elementos y misma cantidad por cada elemento, en un orden potencialmente distinto

# Modularización

## Partiendo un problema en problemas mas chicos

Dadas dos secuencias, queremos saber si uno es una permutación<sup>1</sup> de la otra secuencia:

¿Cuándo será una secuencia permutación de la otra?

- ▶ Tienen los mismos elementos
- ▶ Cada elemento aparece la misma cantidad de veces en ambas secuencias

```
problema esPermutacion(s1, s2 : seq<T>) : Bool {  
  asegura: {res = true  $\leftrightarrow$  para cada elemento es cierto que tiene la misma  
  cantidad de apariciones en s1 y s2 }  
}
```

Pero... falta algo...

---

<sup>1</sup> mismos elementos y misma cantidad por cada elemento, en un orden potencialmente distinto

# Modularizacion

Partiendo un problema en problemas mas chicos

Ahora, tenemos que especificar el problema *cantidadDeApariciones*

¿Cómo podemos saber la cantidad de apariciones de un elemento en una lista?

# Modularizacion

Partiendo un problema en problemas mas chicos

Ahora, tenemos que especificar el problema *cantidadDeApariciones*

¿Cómo podemos saber la cantidad de apariciones de un elemento en una lista?

- ▶ Podríamos sumar 1 por cada posición donde el elemento en dicha posición es el que buscamos!
- ▶ Las operaciones de Sumatorias y Productorias también podemos usarlos

# Modularización

Partiendo un problema en problemas mas chicos

Ahora, tenemos que especificar el problema *cantidadDeApariciones*

¿Cómo podemos saber la cantidad de apariciones de un elemento en una lista?

- ▶ Podríamos sumar 1 por cada posición donde el elemento en dicha posición es el que buscamos!
- ▶ Las operaciones de Sumatorias y Productorias también podemos usarlos

problema *cantidadDeApariciones*( $s : \text{seq}\langle T \rangle, e : T$ ) :  $\mathbb{Z}$  {  
  asegura { $res =$  la cantidad de veces que el elemento  $e$  aparece en la lista  $s$  }  
}

# Recapitulando

## Partiendo un problema en problemas mas chicos

Dadas dos secuencias, queremos saber si uno es una permutación<sup>1</sup> de la otra secuencia:

```
problema esPermutacion(s1, s2 : seq⟨T⟩) : Bool {  
  asegura: {res = true ↔ (para cada elemento e de T, se cumple que  
    (cantidadDeApariciones(s1, e) = cantidadDeApariciones(s2, e)))}  
}
```

Donde...

```
problema cantidadDeApariciones(s : seq⟨T⟩, e : T) : ℤ {  
  asegura {res = la cantidad de veces que el elemento e aparece en la lista s }  
}
```

Y así podemos modularizar y descomponer nuestros problemas, partiendolos en problemas más chicos. Y también los podremos reutilizar!

---

<sup>1</sup> mismos elementos y misma cantidad por cada elemento, en un orden potencialmente distinto

# Modularización

O partir el problema en problemas más chicos...

Los conceptos de modularización y encapsulamiento siempre estarán relacionados con los principios de diseño de software. La estrategia se puede resumir en:

- ▶ Descomponer un problema grande en problemas más pequeños (y sencillos)
- ▶ Componerlos y obtener la solución al problema original

Esto favorece muchos aspectos de calidad como:

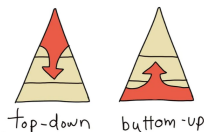
- ▶ La reutilización (una función auxiliar puede ser utilizada en muchos contextos)
- ▶ Es más fácil probar algo chico que algo grande (si cada parte cumple su función correctamente, es más probable que todas juntas también lo haga)
- ▶ La declaratividad (es más fácil entender al ojo humano)



# Modularización

## Top Down versus Bottom Up

También es aplicable a la especificación de problemas:



```
problema esPermutacion(s1, s2 : seq(T)) : Bool {  
  asegura: {res = true  $\leftrightarrow$  (para cada elemento e de T, se cumple que  
    (cantidadDeApariciones(s1, e) = cantidadDeApariciones(s2, e)))}  
}
```

```
problema cantidadDeApariciones(s : seq(T), e : T) :  $\mathbb{Z}$  {  
  asegura {res = la cantidad de veces que el elemento e aparece en la lista s }  
}
```

¿Lo encaramos Top Down o Bottom Up?

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra, Papel y Tijera?... ¿Cómo podríamos especificarlo?

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra, Papel y Tijera?... ¿Cómo podríamos especificarlo?

- ▶ ¿Cómo lo podemos modelar?
- ▶ ¿Qué tipo de datos podríamos utilizar?

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra, Papel y Tijera?... ¿Cómo podríamos especificarlo?

- ▶ ¿Cómo lo podemos modelar?
- ▶ ¿Qué tipo de datos podríamos utilizar?

Opciones:

- ▶ Números?
- ▶ Letras?
- ▶ Un enumerado?

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}
```

- ¿Qué problemas tenemos que resolver?

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}
```

- ▶ ¿Qué problemas tenemos que resolver?
  - ▶ En cada jugada... cada jugador elige jugar con: Piedra, Papel o Tijera
    - ▶ Si ambos jugadores eligen lo mismo, empatan
    - ▶ Piedra le gana a la Tijera
    - ▶ Tijera le gana al Papel
    - ▶ Papel le gana a la Piedra
  - ▶ Una partida es al mejor de 3 jugadas...

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}
```



# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}
```

```
problema determinarGanadorJugada(j1 : PPT, j2 : PPT) :  $\mathbb{Z}$  {  
    requiere: {...}  
    asegura: {...}  
}
```

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}  
problema determinarGanadorJugada(j1 : PPT, j2 : PPT) :  $\mathbb{Z}$  {  
    asegura: {res = 1  $\leftrightarrow$  esGanador(j1, j2)}  
    asegura: {res = 2  $\leftrightarrow$  esGanador(j2, j1)}  
    asegura: {res = 0  $\leftrightarrow$  j1 es igual j2}  
}
```

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Todos sabemos jugar al ¿Piedra Papel y Tijera?... ¿Cómo podríamos especificarlo?

```
enum PPT {  
    PIEDRA, PAPEL, TIJERA  
}  
problema determinarGanadorJugada(j1 : PPT, j2 : PPT) :  $\mathbb{Z}$  {  
    asegura: {res = 1  $\leftrightarrow$  esGanador(j1, j2)}  
    asegura: {res = 2  $\leftrightarrow$  esGanador(j2, j1)}  
    asegura: {res = 0  $\leftrightarrow$  j1 es igual j2}  
}  
problema esGanador(j1 : PPT, j2 : PPT) : Bool {  
    asegura: {res = true  $\leftrightarrow$  ((j1 = PIEDRA  $\wedge$  j2 = TIJERA)  $\vee$  (j1 =  
    TIJERA  $\wedge$  j2 = PAPEL)  $\vee$  (j1 = PAPEL  $\wedge$  j2 = PIEDRA))}  
}
```

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Ya tenemos una jugada... ¿cómo sería una partida?

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Ya tenemos una jugada... ¿cómo sería una partida?

problema *determinarGanadorPartida*(jugadas : seq⟨(PPTxPPT)⟩) :  $\mathbb{Z}$  {

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Ya tenemos una jugada... ¿cómo sería una partida?

problema *determinarGanadorPartida*(jugadas : seq⟨(PPT×PPT)⟩) : ℤ {  
  requiere: {|jugadas| = 3}

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Ya tenemos una jugada... ¿cómo sería una partida?

problema *determinarGanadorPartida*(jugadas : seq<((PPTxPPT))>) :  $\mathbb{Z}$  {  
  requiere:  $\{|jugadas| = 3\}$   
  asegura:  $\{res = 1 \leftrightarrow cantidadJugadasGanadas1(jugadas) >$   
     $cantidadJugadasGanadas2(jugadas)\}$   
  asegura:  $\{res = 2 \leftrightarrow cantidadJugadasGanadas1(jugadas) <$   
     $cantidadJugadasGanadas2(jugadas)\}$   
  asegura:  $\{res = 0 \leftrightarrow cantidadJugadasGanadas1(jugadas) =$   
     $cantidadJugadasGanadas2(jugadas)\}$   
}

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Ya tenemos una jugada... ¿cómo sería una partida?

problema *determinarGanadorPartida*(jugadas : seq⟨(PPTxPPT)⟩) :  $\mathbb{Z}$  {  
  requiere:  $\{|jugadas| = 3\}$   
  asegura:  $\{res = 1 \leftrightarrow cantidadJugadasGanadas1(jugadas) > cantidadJugadasGanadas2(jugadas)\}$   
  asegura:  $\{res = 2 \leftrightarrow cantidadJugadasGanadas1(jugadas) < cantidadJugadasGanadas2(jugadas)\}$   
  asegura:  $\{res = 0 \leftrightarrow cantidadJugadasGanadas1(jugadas) = cantidadJugadasGanadas2(jugadas)\}$   
}

problema *cantidadJugadasGanadas1*(jugadas : seq⟨(PPTxPPT)⟩) :  $\mathbb{Z}$  {



# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Ya tenemos una jugada... ¿cómo sería una partida?

problema *determinarGanadorPartida*(jugadas : seq⟨(PPTxPPT)⟩) :  $\mathbb{Z}$  {  
  requiere:  $\{|jugadas| = 3\}$   
  asegura:  $\{res = 1 \leftrightarrow cantidadJugadasGanadas1(jugadas) >$   
  *cantidadJugadasGanadas2(jugadas)* $\}$   
  asegura:  $\{res = 2 \leftrightarrow cantidadJugadasGanadas1(jugadas) <$   
  *cantidadJugadasGanadas2(jugadas)* $\}$   
  asegura:  $\{res = 0 \leftrightarrow cantidadJugadasGanadas1(jugadas) =$   
  *cantidadJugadasGanadas2(jugadas)* $\}$   
}

problema *cantidadJugadasGanadas1*(jugadas : seq⟨(PPTxPPT)⟩) :  $\mathbb{Z}$  {  
  asegura:  $\{res \text{ es la cantidad de jugadas } j \text{ de la lista } jugadas \text{ tales que}$   
  *determinarGanadorJugada}(j\_0, j\_1) = 1\}  
}*

# Una que sepamos todo...

¿Cómo describiríamos el Piedra, Papel y Tijera?

Ya tenemos una jugada... ¿cómo sería una partida?

problema *determinarGanadorPartida*(jugadas : seq((PPTxPPT))) :  $\mathbb{Z}$  {  
  requiere:  $\{|jugadas| = 3\}$   
  asegura:  $\{res = 1 \leftrightarrow cantidadJugadasGanadas1(jugadas) > cantidadJugadasGanadas2(jugadas)\}$   
  asegura:  $\{res = 2 \leftrightarrow cantidadJugadasGanadas1(jugadas) < cantidadJugadasGanadas2(jugadas)\}$   
  asegura:  $\{res = 0 \leftrightarrow cantidadJugadasGanadas1(jugadas) = cantidadJugadasGanadas2(jugadas)\}$   
}

problema *cantidadJugadasGanadas1*(jugadas : seq((PPTxPPT))) :  $\mathbb{Z}$  {  
  asegura:  $\{res \text{ es la cantidad de jugadas } j \text{ de la lista } jugadas \text{ tales que } determinarGanadorJugada(j_0, j_1) = 1\}$   
}

problema *cantidadJugadasGanadas2*(jugadas : seq((PPTxPPT))) :  $\mathbb{Z}$  {  
  asegura:  $\{res \text{ es la cantidad de jugadas } j \text{ de la lista } jugadas \text{ tales que } determinarGanadorJugada(j_0, j_1) = 2\}$   
}

## Siguiente paso: Algoritmos y programas

- ▶ Hasta ahora estudiamos lógica y aprendimos a **especificar** problemas
- ▶ El objetivo es ahora escribir un **algoritmo** que cumpla esa especificación
  - ▶ Secuencia de pasos que pueden llevarse a cabo mecánicamente
- ▶ Puede haber varios algoritmos que cumplan una misma especificación

# Siguiente paso: Algoritmos y programas

- ▶ Hasta ahora estudiamos lógica y aprendimos a **especificar** problemas
- ▶ El objetivo es ahora escribir un **algoritmo** que cumpla esa especificación
  - ▶ Secuencia de pasos que pueden llevarse a cabo mecánicamente
- ▶ Puede haber varios algoritmos que cumplan una misma especificación
- ▶ Una vez que se tiene el algoritmo, se escribe el **programa** que implementa el algoritmo
  - ▶ Expresión formal de un algoritmo
  - ▶ Lenguajes de programación
    - ▶ sintaxis definida
    - ▶ semántica definida
    - ▶ qué hace una computadora cuando recibe ese programa
    - ▶ qué especificaciones cumple
    - ▶ ejemplos: Haskell, C, C++, C#, Python, Java, Smalltalk, Prolog, etc.
- ▶ A partir de un algoritmo van a existir múltiples programas que implementan dicho algoritmo.

# Paradigmas de Programación

- ▶ Existen distintos paradigmas de programación
  - ▶ Formas de pensar un algoritmo que cumpla una especificación
  - ▶ Cada uno tiene asociado un conjunto de lenguajes
  - ▶ Nos llevan a encarar la programación según ese paradigma

# Paradigmas de Programación

- ▶ Existen distintos paradigmas de programación
  - ▶ Formas de pensar un algoritmo que cumpla una especificación
  - ▶ Cada uno tiene asociado un conjunto de lenguajes
  - ▶ Nos llevan a encarar la programación según ese paradigma
- ▶ Haskell pertenece al paradigma de programación funcional
  - ▶ programa = colección de funciones
    - ▶ Transforman datos de entrada en un resultado
  - ▶ Los lenguajes funcionales nos dan herramientas para explicarle a la computadora cómo computar esas funciones

# Programación funcional

- Un **programa** en un language funcional es un **conjunto de ecuaciones orientadas** que definen una o más funciones.

Por ejemplo:

```
doble x = x + x
```

# Programación funcional

- Un **programa** en un language funcional es un **conjunto de ecuaciones orientadas** que definen una o más funciones.

Por ejemplo:

```
doble x = x + x
```

- La **ejecución** de un programa en este caso corresponde a la **evaluación de una expresión**, habitualmente solicitada desde la consola del entorno de programación.

```
Prelude> doble 10  
20
```



# Programación funcional

- Un **programa** en un language funcional es un **conjunto de ecuaciones orientadas** que definen una o más funciones.

Por ejemplo:

```
doble x = x + x
```

- La **ejecución** de un programa en este caso corresponde a la **evaluación de una expresión**, habitualmente solicitada desde la consola del entorno de programación.

```
Prelude> doble 10  
20
```

- La expresión se evalúa usando las ecuaciones definidas en el programa, hasta llegar a un resultado.

# Programación funcional

- Un **programa** en un language funcional es un **conjunto de ecuaciones orientadas** que definen una o más funciones.

Por ejemplo:

```
doble x = x + x
```

- La **ejecución** de un programa en este caso corresponde a la **evaluación de una expresión**, habitualmente solicitada desde la consola del entorno de programación.

```
Prelude> doble 10  
20
```

- La expresión se evalúa usando las ecuaciones definidas en el programa, hasta llegar a un resultado.
- Las ecuaciones orientadas junto con el mecanismo de reducción describen **algoritmos**.

# Ecuaciones

Para determinar el valor de la aplicación de una función se reemplaza cada expresión por otra, según las ecuaciones.

- ▶ Este proceso puede no terminar, aún con ecuaciones bien definidas.
- ▶ Por ejemplo, consideremos la expresión:

`doble (1 + 1)`

Si reemplazamos `1 + 1` por `doble 1` obtenemos `doble (doble 1)`

Y ahora podemos reemplazar `doble 1` por `1 + 1`

Volvimos a empezar...

`doble (1 + 1)  $\rightsquigarrow$  doble (doble 1)  $\rightsquigarrow$  doble (1 + 1)  $\rightsquigarrow$  ...`

# Ecuaciones orientadas

- ▶ Lado **izquierdo**: expresión a definir
- ▶ Lado **derecho**: definición
- ▶ Cálculo del valor de una expresión : reemplazamos las subexpresiones que sean lado izquierdo de una ecuación por su lado derecho

# Ecuaciones orientadas

- ▶ Lado **izquierdo**: expresión a definir
- ▶ Lado **derecho**: definición
- ▶ Cálculo del valor de una expresión : reemplazamos las subexpresiones que sean lado izquierdo de una ecuación por su lado derecho

Ejemplo: `doble x = x + x`

`doble (1 + 1)  $\rightsquigarrow$  (1 + 1) + (1 + 1)  $\rightsquigarrow$  2 + (1 + 1)  $\rightsquigarrow$  2 + 2  $\rightsquigarrow$  4`

# Ecuaciones orientadas

- ▶ Lado **izquierdo**: expresión a definir
- ▶ Lado **derecho**: definición
- ▶ Cálculo del valor de una expresión : reemplazamos las subexpresiones que sean lado izquierdo de una ecuación por su lado derecho

Ejemplo: `doble x = x + x`

`doble (1 + 1)  $\rightsquigarrow$  (1 + 1) + (1 + 1)  $\rightsquigarrow$  2 + (1 + 1)  $\rightsquigarrow$  2 + 2  $\rightsquigarrow$  4`

También podría ser:

`doble (1 + 1)  $\rightsquigarrow$  doble 2  $\rightsquigarrow$  2 + 2  $\rightsquigarrow$  4`

Más adelante veremos cómo funciona Haskell en particular.

# Transparencia referencial

Es la propiedad de un lenguaje que garantiza que el valor de una expresión depende exclusivamente de sus subexpresiones.

Por lo tanto,

- ▶ Cada expresión del lenguaje representa siempre el mismo valor en cualquier lugar de un programa
- ▶ Es una propiedad muy importante en el paradigma de la programación funcional.
  - ▶ En otros paradigmas el significado de una expresión depende del contexto
- ▶ Es muy útil para verificar correctitud (demostrar que se cumple la especificación)
  - ▶ Podemos usar propiedades ya probadas para sub expresiones
  - ▶ El valor no depende de la historia
  - ▶ Valen en cualquier contexto

# Formación de expresiones

- ▶ Expresiones **atómicas**
  - ▶ También se llaman **formas normales**
  - ▶ Son las más simples, no se puede **reducir** más.
  - ▶ Son la forma más intuitiva de representar un valor
  - ▶ Ejemplos:
    - ▶ 2
    - ▶ False
    - ▶ (3, True)



# Formación de expresiones

- ▶ Expresiones **atómicas**
  - ▶ También se llaman **formas normales**
  - ▶ Son las más simples, no se puede **reducir** más.
  - ▶ Son la forma más intuitiva de representar un valor
  - ▶ Ejemplos:
    - ▶ 2
    - ▶ False
    - ▶ (3, True)
  - ▶ Es común llamarlas “valores” aunque no son un valor, *denotan* un valor, como las demás expresiones

# Formación de expresiones

- ▶ Expresiones **atómicas**
  - ▶ También se llaman **formas normales**
  - ▶ Son las más simples, no se puede **reducir** más.
  - ▶ Son la forma más intuitiva de representar un valor
  - ▶ Ejemplos:
    - ▶ 2
    - ▶ False
    - ▶ (3, True)
  - ▶ Es común llamarlas “valores” aunque no son un valor, *denotan* un valor, como las demás expresiones
- ▶ Expresiones **compuestas**
  - ▶ Se construyen combinando expresiones atómicas con operaciones
  - ▶ Ejemplos:
    - ▶ 1+1
    - ▶ 1==2
    - ▶ (4-1, True || False)

# Formación de expresiones

- ▶ Algunas cadenas de símbolos no forman expresiones
  - ▶ por problemas sintácticos:
    - ▶ `+*1-`
    - ▶ `(True`
    - ▶ `('a',)`
  - ▶ o por error de tipos:
    - ▶ `2 + False`
    - ▶ `2 || 'a'`
    - ▶ `4 * 'b'`
- ▶ Para saber si una expresión está bien formada, aplicamos
  - ▶ Reglas sintácticas
  - ▶ Reglas de asignación o inferencia de tipos (algoritmo de Hindley-Milner)
- ▶ En Haskell toda expresión denota un valor, y ese valor pertenece a un tipo de datos y no se puede usar como si fuera de otro tipo distinto.
  - ▶ Haskell es un lenguaje **fuertemente tipado**

# ¿Cómo ejecuta Haskell?

¿Qué sucede en Haskell cuando escribo una expresión? ¿Cómo se transforma esa expresión en un resultado?

# ¿Cómo ejecuta Haskell?

¿Qué sucede en Haskell cuando escribo una expresión? ¿Cómo se transforma esa expresión en un resultado?

- ▶ Dado el siguiente programa:

```
resta x y = x - y
```

```
suma x y = x + y
```

```
negar x = -x
```

- ▶ ¿Qué sucede al evaluar la expresión `suma (resta 2 (negar 42)) 4`

# Reducción

```
suma (resta 2 (negar 42)) 4
```

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

# Reducción

```
suma (resta 2 (negar 42)) 4
```

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.

# Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).

► Buscamos un redex: suma  $\underbrace{(\text{resta } 2 \text{ (negar } 42)})}_{\text{redex}} 4$



# Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).

► Buscamos un redex: suma  $\underbrace{(\text{resta } 2 \text{ (negar } 42)})}_{\text{redex}}$  4

3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.

►  $\text{resta } x \ y = x - y$   
►  $x \leftarrow 2$   
►  $y \leftarrow (\text{negar } 42)$

# Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).

► Buscamos un redex: suma  $\underbrace{(\text{resta } 2 \text{ (negar } 42))}_{\text{redex}}$  4

3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.

► resta x y = x - y  
► x  $\leftarrow$  2  
► y  $\leftarrow$  (negar 42)

4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.

► suma (resta 2 (negar 42)) 4  $\rightsquigarrow$  suma (2 - (negar 42)) 4

# Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).

► Buscamos un redex: suma (**resta 2 (negar 42)**) 4  
redex

3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.

► resta x y = x - y  
► x ← 2  
► y ← (negar 42)

4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.

► suma (**resta 2 (negar 42)**) 4  $\rightsquigarrow$  suma (**2 - (negar 42)**) 4

5. Si la expresión resultante aún puede reducirse, volvemos al paso 1, sino llegamos a una expresión atómica (forma normal) y ese es el resultado del cómputo.

# Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).

► Buscamos un redex: suma (**resta 2 (negar 42)**) 4  
redex

3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.

► resta x y = x - y  
► x ← 2  
► y ← (negar 42)

4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.

► suma (**resta 2 (negar 42)**) 4  $\rightsquigarrow$  suma (2 - (negar 42)) 4

5. Si la expresión resultante aún puede reducirse, volvemos al paso 1, sino llegamos a una expresión atómica (forma normal) y ese es el resultado del cómputo.

suma (2 - (negar 42)) 4  $\rightsquigarrow$  suma (2 - (- 42)) 4

# Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).

► Buscamos un redex: suma  $\underbrace{(\text{resta } 2 \text{ (negar } 42)})}_{\text{redex}} 4$

3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.

► resta x y = x - y  
► x  $\leftarrow$  2  
► y  $\leftarrow$  (negar 42)

4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.

► suma (resta 2 (negar 42)) 4  $\rightsquigarrow$  suma (2 - (negar 42)) 4

5. Si la expresión resultante aún puede reducirse, volvemos al paso 1, sino llegamos a una expresión atómica (forma normal) y ese es el resultado del cómputo.

suma (2 - (negar 42)) 4  $\rightsquigarrow$  suma (2 - (- 42)) 4  $\rightsquigarrow$  suma (44) 4

# Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).

► Buscamos un redex: suma (**resta 2 (negar 42)**) 4  
redex

3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.

► resta x y = x - y  
► x  $\leftarrow$  2  
► y  $\leftarrow$  (negar 42)

4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.

► suma (**resta 2 (negar 42)**) 4  $\rightsquigarrow$  suma (2 - (negar 42)) 4

5. Si la expresión resultante aún puede reducirse, volvemos al paso 1, sino llegamos a una expresión atómica (forma normal) y ese es el resultado del cómputo.

suma (2 - (negar 42)) 4  $\rightsquigarrow$  suma (2 - (- 42)) 4  $\rightsquigarrow$  suma (44) 4  $\rightsquigarrow$  44 + 4

# Reducción

suma (resta 2 (negar 42)) 4

El mecanismo de evaluación en un lenguaje funcional es la **reducción**:

1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
2. La subexpresión a reemplazar es alguna **instancia** del lado izquierdo de alguna ecuación orientada del programa, y se la llama **radical** o **redex** (*reducible expression*).

► Buscamos un redex: suma (**resta 2 (negar 42)**) 4  
redex

3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.

► resta x y = x - y  
► x ← 2  
► y ← (negar 42)

4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.

► suma (**resta 2 (negar 42)**) 4  $\rightsquigarrow$  suma (2 - (negar 42)) 4

5. Si la expresión resultante aún puede reducirse, volvemos al paso 1, sino llegamos a una expresión atómica (forma normal) y ese es el resultado del cómputo.

suma (2 - (negar 42)) 4  $\rightsquigarrow$  suma (2 - (- 42)) 4  $\rightsquigarrow$  suma (44) 4  $\rightsquigarrow$  44 + 4  $\rightsquigarrow$  48

# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
suma (3+4) (suc (2*3))
```



# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
suma (3+4) (suc (2*3))
```

```
↪ (3+4) + (suc (2*3))
```

# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
suma (3+4) (suc (2*3))
```

```
↪ (3+4) + (suc (2*3))
```

```
↪ 7 + (suc (2*3))
```

# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
suma (3+4) (suc (2*3))
```

```
↪ (3+4) + (suc (2*3))
```

```
↪ 7 + (suc (2*3))
```

```
↪ 7 + ((2*3) + 1)
```

# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
    suma (3+4) (suc (2*3))  
  ~> (3+4) + (suc (2*3))  
  ~> 7 + (suc (2*3))  
  ~> 7 + ((2*3) + 1)  
  ~> 7 + (6 + 1)
```

# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
suma (3+4) (suc (2*3))  
~> (3+4) + (suc (2*3))  
~> 7 + (suc (2*3))  
~> 7 + ((2*3) + 1)  
~> 7 + (6 + 1)  
~> 7 + 7
```

# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
suma (3+4) (suc (2*3))  
~> (3+4) + (suc (2*3))  
~> 7 + (suc (2*3))  
~> 7 + ((2*3) + 1)  
~> 7 + (6 + 1)  
~> 7 + 7  
~> 14
```

# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
suma (3+4) (suc (2*3))  
~> (3+4) + (suc (2*3))  
~> 7 + (suc (2*3))  
~> 7 + ((2*3) + 1)  
~> 7 + (6 + 1)  
~> 7 + 7  
~> 14
```

Otros lenguajes de programación (C, C++, Pascal, Java) tienen un orden de **evaluación eager** (ansioso): primero se evalúan los argumentos y después la función.

# Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).



# Indefinición

- ▶ Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).
- ▶ ¿Cómo podemos clasificar las funciones?

# Indefinición

- ▶ Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).
- ▶ ¿Cómo podemos clasificar las funciones?
  - ▶ Funciones **totales**: nunca se indefinen.

# Indefinición

- ▶ Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).
- ▶ ¿Cómo podemos clasificar las funciones?
  - ▶ Funciones **totales**: nunca se indefinen.  
`suc x = x + 1`

# Indefinición

- ▶ Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).
- ▶ ¿Cómo podemos clasificar las funciones?
  - ▶ Funciones **totales**: nunca se indefinen.  
`suc x = x + 1`
  - ▶ Funciones **parciales**: hay argumentos para los cuales se indefinen.

# Indefinición

- ▶ Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).
- ▶ ¿Cómo podemos clasificar las funciones?
  - ▶ Funciones **totales**: nunca se indefinen.  
`suc x = x + 1`
  - ▶ Funciones **parciales**: hay argumentos para los cuales se indefinen.  
`division x y = div x y`

# Indefinición

- ▶ Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).
- ▶ ¿Cómo podemos clasificar las funciones?
  - ▶ Funciones **totales**: nunca se indefinen.  
`suc x = x + 1`
  - ▶ Funciones **parciales**: hay argumentos para los cuales se indefinen.  
`division x y = div x y`

¿Qué pasa al reducir las siguientes expresiones en Haskell?

- ▶ `(division 1 1 == 0) && (division 1 0 == 1)`
- ▶ `(division 1 1 == 1) && (division 1 0 == 1)`
- ▶ `(division 1 0 == 1) && (division 1 1 == 1)`

# Indefinición

- ▶ Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están **indefinidas** ( $\perp$ ).
- ▶ ¿Cómo podemos clasificar las funciones?
  - ▶ Funciones **totales**: nunca se indefinen.  
`suc x = x + 1`
  - ▶ Funciones **parciales**: hay argumentos para los cuales se indefinen.  
`division x y = div x y`

¿Qué pasa al reducir las siguientes expresiones en Haskell?

- ▶ `(division 1 1 == 0) && (division 1 0 == 1)`
- ▶ `(division 1 1 == 1) && (division 1 0 == 1)`
- ▶ `(division 1 0 == 1) && (division 1 1 == 1)`

¿Y si hiciéramos una evaluación eager o ansiosa?

# Definiciones de funciones por casos

Podemos usar **guardas** para definir funciones por casos:

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1  
    | n /= 0 = 0
```

Palabra clave “si no”.

```
f n | n == 0 = 1  
    | otherwise = 0
```



# La función signo

$$\text{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

```
signo n | n > 0 = 1  
        | n == 0 = 0  
        | n < 0 = -1
```

```
signo n | n > 0 = 1  
        | n == 0 = 0  
        | otherwise = -1
```

# La función máximo

```
maximo x y | x >= y = x  
          | otherwise = y
```

## ¿Qué hacen las siguientes funciones?

```
f1 n | n >= 3 = 5
```

```
f2 n | n >= 3 = 5  
    | n <= 1 = 8
```

```
f3 n | n >= 3 = 5  
    | n == 2 = undefined  
    | otherwise = 8
```

## ¿Qué hacen las siguientes funciones?

```
f4 n | n >= 3 = 5  
      | n <= 9 = 7
```

```
f5 n | n <= 9 = 7  
      | n >= 3 = 5
```

**Prestar atención al orden de las guardas.** ¡Cuando las condiciones se solapan, el orden de las guardas cambia el comportamiento de la función!

## Otra posibilidad usando *pattern matching*

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{si no} \end{cases}$$

```
f n | n == 0 = 1  
    | n /= 0 = 0
```

También se puede hacer:

```
f 0 = 1  
f n = 0
```

## Otra posibilidad usando *pattern matching*

$$\text{signo}(n) = \begin{cases} 1 & \text{si } n > 0 \\ 0 & \text{si } n = 0 \\ -1 & \text{si } n < 0 \end{cases}$$

```
signo n | n > 0 = 1  
       | n == 0 = 0  
       | n < 0 = -1
```

También se puede hacer:

```
signo 0 = 0  
signo n | n > 0 = 1  
       | otherwise = -1
```

## Un ejemplo con especificación

Dados tres números  $a$ ,  $b$  y  $c$ , calcular la cantidad de soluciones reales de la ecuación cuadrática:  $aX^2 + bX + c = 0$ .

# Un ejemplo con especificación

Dados tres números  $a$ ,  $b$  y  $c$ , calcular la cantidad de soluciones reales de la ecuación cuadrática:  $aX^2 + bX + c = 0$ .

problema *cantidadDeSoluciones*( $a : \mathbb{Z}, b : \mathbb{Z}, c : \mathbb{Z}$ ) :  $\mathbb{Z}$  {

  requiere:  $\{a \neq 0\}$

  asegura:  $\{res = 2 \leftrightarrow discriminante(a, b, c) > 0\}$

  asegura:  $\{res = 1 \leftrightarrow discriminante(a, b, c) = 0\}$

  asegura:  $\{res = 0 \leftrightarrow discriminante(a, b, c) < 0\}$

}

problema *discriminante*( $a : \mathbb{Z}, b : \mathbb{Z}, c : \mathbb{Z}$ ) :  $\mathbb{Z}$  {

  requiere:  $\{a \neq 0\}$

  asegura:  $\{res = b^2 - 4 * a * c\}$

}



# Un ejemplo con especificación

Dados tres números  $a$ ,  $b$  y  $c$ , calcular la cantidad de soluciones reales de la ecuación cuadrática:  $aX^2 + bX + c = 0$ .

```
problema cantidadDeSoluciones(a : ℤ, b : ℤ, c : ℤ) : ℤ {  
  requiere: {a ≠ 0}  
  asegura: {res = 2 ↔ discriminante(a, b, c) > 0}  
  asegura: {res = 1 ↔ discriminante(a, b, c) = 0}  
  asegura: {res = 0 ↔ discriminante(a, b, c) < 0}  
}  
problema discriminante(a : ℤ, b : ℤ, c : ℤ) : ℤ {  
  requiere: {a ≠ 0}  
  asegura: {res = b2 - 4 * a * c}  
}
```

```
cantidadDeSoluciones a b c | b2 - 4*a*c > 0 = 2  
                           | b2 - 4*a*c == 0 = 1  
                           | otherwise = 0
```

Otra posibilidad:

```
cantidadDeSoluciones a b c | discriminante > 0 = 2  
                           | discriminante == 0 = 1  
                           | otherwise = 0  
                           where discriminante = b2 - 4*a*c
```

# Tipos de datos

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

## Ejemplos:

1.  $\text{Int} = (\mathbb{Z}, \{+, -, *, \text{div}, \text{mod}\})$  es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
2.  $\text{Float} = (\mathbb{Q}, \{+, -, *, /\})$  es el tipo de datos que representa a los racionales, con la aritmética de **punto flotante**.
3.  $\text{Char} = (\{'a', 'A', '1', '?'\}, \{\text{ord}, \text{chr}, \text{isUpper}, \text{toUpper}\})$  es el tipo de datos que representan los caracteres.
4.  $\text{Bool} = (\{\text{True}, \text{False}\}, \{\&\&, ||, \text{not}\})$  representa a los valores lógicos.

# Tipos de datos

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

## Ejemplos:

1.  $\text{Int} = (\mathbb{Z}, \{+, -, *, \text{div}, \text{mod}\})$  es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
  2.  $\text{Float} = (\mathbb{Q}, \{+, -, *, /\})$  es el tipo de datos que representa a los racionales, con la aritmética de **punto flotante**.
  3.  $\text{Char} = (\{'a', 'A', '1', '?'\}, \{\text{ord}, \text{chr}, \text{isUpper}, \text{toUpper}\})$  es el tipo de datos que representan los caracteres.
  4.  $\text{Bool} = (\{\text{True}, \text{False}\}, \{\&\&, ||, \text{not}\})$  representa a los valores lógicos.
- Podemos declarar explícitamente el tipo de datos del *dominio* y *codominio* de las funciones. A esto lo llamamos dar la **signatura** de la función.
  - No es estrictamente necesario hacerlo (Haskell puede inferir el tipo), pero suele ser una buena práctica (y **¡nosotros lo vamos a pedir!**).

# Aplicación de funciones

En programación funcional (como en matemática) las funciones son elementos (valores).

Notación  $f :: T1 \rightarrow T2 \rightarrow T3 \rightarrow \dots \rightarrow T_n$

- Una función es un valor

# Aplicación de funciones

En programación funcional (como en matemática) las funciones son elementos (valores).

Notación  $f :: T1 \rightarrow T2 \rightarrow T3 \rightarrow \dots \rightarrow T_n$

- ▶ Una función es un valor
- ▶ la operación básica que podemos realizar con ese valor es la **aplicación**
  - ▶ Aplicar la función a un elemento para obtener un resultado

# Aplicación de funciones

En programación funcional (como en matemática) las funciones son elementos (valores).

Notación  $f :: T1 \rightarrow T2 \rightarrow T3 \rightarrow \dots \rightarrow Tn$

- ▶ Una función es un valor
- ▶ la operación básica que podemos realizar con ese valor es la **aplicación**
  - ▶ Aplicar la función a un elemento para obtener un resultado
- ▶ Sintácticamente, la aplicación se escribe como una yuxtaposición (la función seguida de su parámetro).
- ▶ Por ejemplo: sea  $f :: T1 \rightarrow T2$ , y  $e$  de tipo  $T1$  entonces  $f\ e$  es una expresión de tipo  $T2$ .  
Sea `doble :: Int -> Int`, entonces `doble 2` representa un número entero.

# Ejemplos de funciones con la signatura

```
maximo :: Int -> Int -> Int
maximo x y | x >= y = x
           | otherwise = y
```

```
maximoRac :: Float -> Float -> Float
maximoRac x y | x >= y = x
              | otherwise = y
```

```
esMayorA9 :: Int -> Bool
esMayorA9 n | n > 9 = True
            | otherwise = False
```

```
esPar :: Int -> Bool
esPar n | mod n 2 == 0 = True
        | otherwise = False
```

```
esPar2 :: Int -> Bool
esPar2 n = mod n 2 == 0
```

## Otro ejemplo más raro:

```
funcionRara :: Float -> Float -> Bool -> Bool
funcionRara x y z = (x >= y) || z
```

Otras posibilidades, usando *pattern matching*:

```
funcionRara :: Float -> Float -> Bool -> Bool
funcionRara x y True = True
funcionRara x y False = x >= y
```

```
funcionRara :: Float -> Float -> Bool -> Bool
funcionRara _ _ True = True
funcionRara x y False = x >= y
```



# Nueva familia de tipos: Tuplas

## Tuplas

- Dados tipos  $A_1, \dots, A_k$ , el **tipo  $k$ -upla**  $(A_1, \dots, A_k)$  es el conjunto de las  $k$ -uplas  $(v_1, \dots, v_k)$  donde  $v_i$  es de tipo  $A_i$

```
(1, 2)           :: (Int, Int)
(1.1, 3.2, 5.0)  :: (Float, Float, Float)
(True, (1, 2))   :: (Bool, (Int, Int))
(True, 1, 2)     :: (Bool, Int, Int)
```

- En Haskell hay infinitos tipos de tuplas

## Funciones de acceso a los valores de un par en **Prelude**

- `fst` ::  $(a, b) \rightarrow a$       Ejemplo: `fst (1 + 4, 2)  $\rightsquigarrow$  5`
- `snd` ::  $(a, b) \rightarrow b$       Ejemplo: `snd (1, (2, 3))  $\rightsquigarrow$  (2, 3)`

Ejemplo: suma de vectores en  $\mathbb{R}^2$

```
suma :: (Float, Float) -> (Float, Float) -> (Float, Float)
suma v w = ((fst v) + (fst w), (snd v) + (snd w))
```