

# Práctico 2: Especificación, derivación y verificación de programas funcionales

Algoritmos y Estructuras de Datos I  
2<sup>do</sup> cuatrimestre 2025

## Introducción

Esta guía tiene como objetivo obtener las habilidades necesarias para llevar adelante un proceso de derivación o verificación de programas recursivos a partir de especificaciones formales.

Completan esta guía algunos ejercicios para demostrar tanto por inducción sobre naturales, como por inducción **estructural** sobre listas. Se busca mejorar la habilidad para utilizar esta técnica de prueba que es central para la tarea de cálculo de programas recursivos.

Los ejercicios de cálculo de programas tienen una dificultad creciente: en los primeros, la derivación o verificación se obtiene de manera directa a través de una demostración inductiva. Los ejercicios sucesivos son más complejos y requieren el uso de técnicas más avanzadas: modularización y generalización.

Esta guía también incluye ejercicios de **Laboratorio**. Se incluyen secciones específicas de laboratorio a modo de tutorial para introducir nuevos conceptos de programación en Haskell como la definición de nuevos tipos y de funciones polimórficas.

### 1. Dado el programa

$$\frac{\sum \begin{array}{c} sum : [Num] \rightarrow Num \\ sum.[] \doteq 0 \\ sum.(x \triangleright xs) \doteq x + sum.xs \end{array}}{sum : [Num] \rightarrow Num}$$

- a) ¿Qué hace esta función? Escriba en lenguaje natural el **problema** que resuelve.
  - b) Escriba una **especificación** de la función con una expresión cuantificada.
  - c) **Verifique** que esta especificación vale para toda lista  $xs$ .
  - d) **Derive** la definición de la función a partir de su especificación.  
¿Esta derivación es parecida a la demostración en el punto 1c?
2. A partir de las siguientes especificaciones, describir en lenguaje natural qué describe la expresión, dar el tipo de cada función y *derivar* una solución algorítmica para cada caso.
    - a)  $sum\_cuad.xs = \langle \sum i : 0 \leq i < \#xs : xs.i * xs.i \rangle$
    - b)  $iga.e.xs = \langle \forall i : 0 \leq i < \#xs : xs.i = e \rangle$
    - c)  $exp.x.n = x^n$
    - d)  $sum\_par.n = \langle \sum i : 0 \leq i \leq n \wedge par.i : i \rangle$   
donde  $par.i \doteq i \bmod 2 = 0$ .
    - e)  $cuantos.p.xs = \langle \text{N } i : 0 \leq i < \#xs : p.(xs.i) \rangle$
  3. Para todos los ítems del ejercicio anterior, dar un ejemplo de uso de la función, es decir: elegir valores concretos para los parámetros y calcular el resultado usando la solución algorítmica obtenida. Las listas deben tener por lo menos tres elementos.

**Laboratorio 1** Definí en Haskell las funciones derivadas en el ejercicio 2 y evalúalas en los ejemplos utilizados en el ejercicio 3.

---

## Modularización

En algunas ocasiones la solución del problema original requiere la solución de un “sub-problema”. En estos casos suele ser conveniente no atacar ambos problemas simultáneamente, sino “por módulos”, cada uno de los cuales debe ser independiente de los demás. Esta técnica se denomina modularización.

En esta sección trabajaremos fundamentalmente con ejercicios que requieren aplicar la técnica de modularización para poder derivarse.

4. Derivá las siguientes funciones.

- a)  $f: \text{Num} \rightarrow \text{Nat} \rightarrow \text{Num}$  computa la suma de potencias de un número, esto es

$$f.x.n = \langle \sum i : 0 \leq i < n : x^i \rangle .$$

- b)  $pi: \text{Nat} \rightarrow \text{Num}$  computa la aproximación del número  $\pi$

$$pi.n = 4 * \langle \sum i : 0 \leq i < n : (-1)^i / (2 * i + 1) \rangle$$

**Ayuda:** Modularizar dos veces. La segunda con la función *exp* del ejercicio 2c

- c)  $f: \text{Nat} \rightarrow \text{Nat}$  computa el cubo de un número natural  $x$  utilizando únicamente sumas. La especificación es muy simple:  $f.x = x^3$ .

**Ayuda:** Usar inducción y modularización varias veces

- d)  $f_xs = \langle \exists i : 0 < i \leq \#xs : \langle \prod j : 0 \leq j < \#(xs \downarrow i) : (xs \downarrow i).j \rangle = xs.(i - 1) \rangle$

**Laboratorio 2** Implementá en Haskell las funciones derivadas en el ejercicio anterior.

5. Especificá formalmente utilizando cuantificadores cada una de las siguientes funciones descriptas informalmente. Luego, derivá soluciones algorítmicas para cada una.

- a)  $iguales: [A] \rightarrow \text{Bool}$ , que determina si los elementos de una lista de tipo  $A$  son todos iguales entre sí. Suponga que el operador  $=$  es la igualdad para el tipo  $A$ .
- b)  $minimo: [\text{Int}] \rightarrow \text{Int}$ , que calcula el mínimo elemento de una lista **no vacía** de enteros.

**Nota:** La función no debe devolver  $\pm\infty$ .

- c)  $creciente: [\text{Int}] \rightarrow \text{Bool}$ , que determina si los elementos de una lista de enteros están ordenados en forma creciente.
- d)  $prod: [\text{Num}] \rightarrow [\text{Num}] \rightarrow \text{Num}$ , que calcula el producto entre pares de elementos en iguales posiciones de las listas y suma estos resultados (producto punto). Si las listas tienen distinto tamaño se opera hasta la última posición de la más chica.

**Laboratorio 3** Implementá en Haskell las funciones derivadas en el ejercicio anterior.

---

## Generalización por abstracción

Una técnica muy poderosa es la generalización por abstracción (por algunos autores llamada inmersión). Aquí la utilizaremos como un medio para resolver una derivación cuando la hipótesis inductiva no puede aplicarse de manera directa. La idea consiste en buscar una especificación más general uno de cuyos casos particulares sea la función en cuestión. Para encontrar la generalización adecuada se introducen parámetros nuevos los cuales servirán para dar cuenta de las subexpresiones que no permiten aplicar la hipótesis inductiva en la derivación original.

6. A partir de las siguientes especificaciones expresá en lenguaje natural qué devuelven las funciones, identificá su tipo y derivalas:

- a)  $psum_xs = \langle \forall i : 0 \leq i \leq \#xs : sum.(xs \uparrow i) \geq 0 \rangle$ , con  $sum$  la función del ejercicio 1.

- b)  $sum\_ant_xs = \langle \exists i : 0 \leq i < \#xs : xs.i = sum.(xs \uparrow i) \rangle$

- c)  $sum8_xs = \langle \exists i : 0 \leq i \leq \#xs : sum.(xs \uparrow i) = 8 \rangle$ .

- d)  $f_xs = \langle \text{Max } i : 0 \leq i < \#xs \wedge sum.(xs \uparrow i) = sum.(xs \downarrow i) : i \rangle$ .

#### Laboratorio 4 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

7. Especificá formalmente utilizando cuantificadores cada una de las siguientes funciones descriptas informalmente. Luego, *derivá* soluciones algorítmicas para cada una.
- $cuad : Nat \rightarrow Bool$ , que dado un natural determina si es el cuadrado de un número.
  - $n8 : [Num] \rightarrow Nat$ , que cuenta la cantidad de segmentos iniciales de una lista cuya suma es igual a 8.

#### Laboratorio 5 Implementá en Haskell las funciones derivadas en el ejercicio anterior.

**Laboratorio 6 Tipos recursivos.** Supongamos que queremos representar una *cola* de deportistas, como aquellas que forman fila para retirar sus credenciales en la villa olímpica. Un deportista llega y se coloca al final de la cola y espera su turno. El orden de atención respeta el orden de llegada, es decir, quien primero llega, es atendido primero. Podemos representar esta situación con el siguiente tipo:

```
data Cola = VaciaC | Encolada Deportista Cola
```

En esta definición, el tipo que estamos definiendo (*Cola*) aparece como un parámetro de uno de sus constructores; por ello se dice que el tipo es *recursivo*. Así una cola o bien está vacía, o bien contiene a una persona encolada, seguida del resto de la cola. Esto nos permite representar colas cuya longitud no conocemos *a priori* y que pueden ser arbitrariamente largas.

I. Programá las siguientes funciones:

- $atender :: Cola \rightarrow Maybe Cola$ , que elimina de la cola a la persona que está en la primer posición de una cola, por haber sido atendida. Si la cola está vacía, devuelve *Nothing*.
- $encolar :: Deportista \rightarrow Cola \rightarrow Cola$ , que agrega una persona a una cola de deportistas, en la última posición.
- $busca :: Cola \rightarrow Zona \rightarrow Maybe Deportista$ , que devuelve el/la primera futbolista dentro de la cola que juega en la zona que se corresponde con el segundo parámetro. Si no hay futbolistas jugando en esa zona devuelve *Nothing*.

II. ¿A qué otro tipo se parece *Cola*?

**Laboratorio 7** En las páginas de visualización de streaming, normalmente encontramos películas y series. Los videos que se reproducen pueden corresponder a películas o capítulos de series. En algunas plataformas también podríamos tener cola de reproducción de videos, de manera que al terminar de ver uno, comienza el otro.

- Definir el tipo *Video* que consta de dos constructores *Pelicula* y *CapSerie* con los siguientes parámetros:  
El constructor *Pelicula* debe tomar como parámetros el nombre, el director, la duración (en minutos) y el año de estreno. El constructor *CapSerie* debe tomar como parámetros el nombre de la serie, el nro de capítulo, la temporada, el año de estreno de la temporada.
- A partir del tipo definido en el punto anterior, definí los siguientes términos(videos):

```
elPadrino :: Video  
elPadrino = <COMPLETAR>
```

correspondiente a la película “El Padrino” con director “Francis Ford Coppola”, duración 177 minutos y año de estreno 1972.

```
breakingBadS01E01 :: Video  
breakingBadS01E01 = <COMPLETAR>
```

correspondiente al capítulo de serie "Breaking Bad", con capítulo número 1, temporada 1 y año de estreno de temporada 2008.

- c) Definir la función `esPrimerCapitulo :: Video -> Bool` que dado un `Video`, devuelve `True` si el video es el primer capítulo de la primera temporada de una serie, `False` caso contrario.
- d) Definir la función `esEstreno :: Video -> Bool` que dado un `Video`, devuelve `True` si el video es una película cuyo año de estreno es igual a 2024.
- e) Definir la función `duracionPeliMasLarga :: [Video] -> Int` que dada una lista de videos, devuelve la duración de la película más larga. En caso que no haya películas devuelve 0.
- f) Dado el tipo recursivo `ColaVideo` definido de la siguiente manera

```
data ColaVideo = Vacia | Encolada Video ColaVideo deriving Show
```

podemos definir, por ejemplo, una cola de reproducción de la siguiente manera

```
colaReproduccion :: ColaVideo
colaReproduccion = Encolada elPadrino (Encolada breakingBadS01E01 Vacia))
```

Definir la función `pelisDelDirector :: ColaVideo -> String -> ColaVideo` que dada una cola de reproducción de videos q, y el nombre de un director d, devuelve la cola de reproducción que tiene solamente las películas del director d (en el mismo orden que aparecen en q).

**Laboratorio 8** En algunas apps de reproducción de música podemos escuchar lanzamientos musicales de los artistas. Cada lanzamiento puede ser un álbum o un sencillo (o single). También dichas apps disponen de funcionalidad para crear colas de reproducción.

- a) Definir el tipo `Lanzamiento` que consta de dos constructores `Album` y `Sencillo` con los siguientes parámetros: El constructor `Album` debe tomar como parámetros el nombre, el artista, la lista de nombres de temas y el año de estreno. El constructor `Sencillo` debe tomar como parámetros el nombre, el artista, la duración (en segundos) y el año de estreno.
- b) A partir del tipo definido en el punto anterior, definí los siguientes términos(lanzamientos):

```
clicsModernos :: Lanzamiento
clicsModernos = <COMPLETAR>
```

correspondiente al álbum "Clics Modernos" del artista "Charly García", con lista de temas ["Nos siguen pegando abajo", "Dos cero uno", "Nuevos trapos", "Bancate ese defecto", "No me dejan salir", "Los dinosaurios", "Plateado sobre plateado"] y el año de estreno 1983.

```
africa :: Lanzamiento
africa = <COMPLETAR>
```

correspondiente al sencillo "Africa" de "Toto" con duración 260 segundos y año de estreno 1982.

- c) Definir la función `esDelArtista :: Lanzamiento -> String -> Bool` que dado un `Lanzamiento` d y el nombre de un artista a, devuelve `True` si d es un sencillo del artista a, `False` en caso contrario.
- d) Definir la función `esEP :: Lanzamiento -> Bool` que dado un `Lanzamiento`, devuelve `True` si éste es un álbum con cantidad de tracks menor o igual a 4, `False` en caso contrario.
- e) Definir la función `minSencillosArtista :: [Lanzamiento] -> String -> Int` que dada una lista de lanzamientos y el nombre de un artista, devuelve la cantidad de minutos de la suma de la duración de todos los sencillos de ese artista.
- f) Dado el tipo recursivo `ColaLanzamiento` definido de la siguiente manera

```
data ColaLanzamiento = Vacia | Encolada Lanzamiento ColaLanzamiento
deriving Show
```

podemos definir, por ejemplo, una cola de lanzamiento de la siguiente manera

```
colaReproduccion :: ColaLanzamiento
colaReproduccion = Encolada clicsModernos (Encolada africa Vacia)
```

Definir la función `soloSencillos :: ColaLanzamiento -> ColaLanzamiento` que dada una cola de lanzamientos `q`, devuelve la cola de reproducción que tiene solamente los sencillos de la cola `q` (en el mismo orden que aparecen en `q`).

## Segmentos

Dada una lista `xs`, un segmento de ella es una lista cuyos elementos están en `xs`, en el mismo orden y consecutivamente. Por ejemplo: si `xs = [2, 4, 6, 8]`, entonces `[2, 4], [4, 6, 8], []`, son segmentos, mientras que `[4, 2]` o `[2, 6, 8]` no lo son. En problemas como este, suele ser conveniente expresar la lista como una concatenación. Si escribimos `xs = as ++ bs ++ cs`, entonces `as`, `bs` y `cs` son segmentos de `xs`, por lo que podemos usarlos para expresar las condiciones que necesitamos que se satisfagan.

8. **Segmentos de lista:** Para las siguientes expresiones cuantificadas:

- a)  $\langle \forall as, bs : xs = as ++ bs : sum.as \geq 0 \rangle$
- b)  $\langle \text{Min } as, bs, cs : xs = as ++ bs ++ cs : sum.bs \rangle$
- c)  $\langle \text{N } as, b, bs : xs = as ++ (b \triangleright bs) : b > sum.bs \rangle$
- d)  $\langle \text{Max } as, bs, cs : xs = as ++ bs \wedge ys = as ++ cs : \#as \rangle$

- Calculá los rangos como conjuntos de tuplas. Tomar `xs = [9, -5, 1, -3]`, `ys = [9, -5, 3]`.
- Calculá los resultados para las listas dadas.
- Expresá en lenguaje natural qué significa cada expresión.

9. Expresá utilizando cuantificadores las siguientes sentencias del lenguaje natural:

- a) La lista `xs` es un segmento inicial de la lista `ys`.
- b) La lista `xs` es un segmento de la lista `ys`.
- c) La lista `xs` es un segmento final de la lista `ys`.
- d) Las listas `xs` e `ys` tienen en común un segmento no vacío.
- e) La lista `xs` de números enteros tiene la misma cantidad de elementos pares e impares.
- f) La lista `xs` posee un segmento **no** inicial y **no** final cuyos valores son mayores a los valores del resto de la misma.

10. Derivá funciones recursivas a partir de cada una de las especificaciones que escribiste para los ejercicios 9a y 9b.

**Laboratorio 9** Implementá en Haskell las funciones derivadas en el ejercicio anterior.

11. Derivá las funciones especificadas como:

- a)  $sumin.xs = \langle \text{Min } as, bs, cs : xs = as ++ bs ++ cs : sum.bs \rangle$   
(suma mínima de un segmento).
- b)  $f.xs = \langle \text{N } as, bs, cs : xs = as ++ bs ++ cs : 8 = sum.bs \rangle$   
(cantidad de segmentos que suman eso).
- c)  $maxiga.e.xs = \langle \text{Max } as, bs, cs : xs = as ++ bs ++ cs \wedge iga.e.bs : \#bs \rangle$   
(máxima longitud de iguales a `e`)  
donde `iga` es la función del ejercicio 2b.

**Laboratorio 10** Implementá en Haskell las funciones derivadas en el ejercicio anterior.

12. Describí en lenguaje natural y dar el *tipo* de cada una de las siguientes funciones especificadas formalmente:

- $f.xs = \langle N i, j : 0 \leq i < j < \#xs : xs.i = xs.j \rangle$
- $g.xs = \langle \text{Max } p, q : 0 \leq p < q < \#xs : xs.p + xs.q \rangle$
- $h.xs = \langle N k : 0 \leq k < \#xs : \langle \forall i : 0 \leq i < k : xs.i < xs.k \rangle \rangle$
- $k.xs = \langle \forall i, j : 0 \leq i \wedge 0 \leq j \wedge i + j = \#xs - 1 : xs.i = xs.j \rangle$
- $l.xs = \langle \text{Max } p, q : 0 \leq p \leq q < \#xs \wedge \langle \forall i : p \leq i < q : xs.i \geq 0 \rangle : q - p \rangle$

13. Derivá la función definida en el ejercicio 12b.

**Laboratorio 11** Implementá en Haskell la función derivada en el ejercicio anterior.

---

### Ejercicios integrados

14. Expresá utilizando cuantificadores las siguientes sentencias del lenguaje natural:

- El elemento  $e$  ocurre un número par de veces en la lista  $xs$ .
- El elemento  $e$  ocurre en las posiciones pares de la lista  $xs$ .
- El elemento  $e$  ocurre únicamente en las posiciones pares de la lista  $xs$ .
- Si  $e$  ocurre en la lista  $xs$ , entonces  $l$  ocurre en alguna posición anterior en la misma lista.
- Existe un elemento de la lista  $xs$  que es estrictamente mayor a todos los demás.
- En la lista  $xs$  solo ocurren valores que anulan la función  $f$ .

15. Derivá funciones recursivas a partir de cada una de las especificaciones del ejercicio 14.

**Laboratorio 12** Implementá en Haskell las funciones derivadas en el ejercicio anterior.

16. Sea  $fib$  la definición recursiva *estándar* para la función de Fibonacci. Derivá la función de Fibolucci,  $fbl : Nat \rightarrow Nat$ , especificada como:

$$fbl.n = \langle \sum i : 0 \leq i < n : fib.i \times fib.(n - i) \rangle$$

17. Derivá la función recursiva a partir de la especificación del ejercicio 9c.

18. Especificá formalmente utilizando cuantificadores cada una de las siguientes funciones descriptas informalmente. Luego, derivá soluciones algorítmicas para cada una.

- $listas.iguales : [A] \rightarrow [A] \rightarrow Bool$ , que determina si dos listas son iguales, es decir, contienen los mismos elementos en las mismas posiciones respectivamente.
  - $primo? : Nat \rightarrow Bool$ , que determina si un número es primo.
19. Derivá las funciones a partir de las especificaciones en el ejercicio 12.