Ejercicio 1

Programá las siguientes funciones:

- `esCero :: Int -> Bool`, que verifica si un entero es igual a 0.

```
esCero :: Int → Bool
esCero x = x == 0
```

La función recibe un entero, luego devuelve un booleano, está definida como `x == 0`, esta expresión devuelve `True` si `x` es 0, y `False` en caso contrario.

- `esPositivo :: Int -> Bool`, que verifica si un entero es estrictamente mayor a 0.

```
esPositivo :: Int → Bool
esPositivo x = x > 0
```

La función recibe un entero, luego devuelve un booleano, está definida como `x > 0`, esta expresión devuelve `True` si `x` es mayor a 0, y `False` en caso contrario.

- `esVocal :: Char -> Bool`, que verifica si un carácter es una vocal en minúscula.

```
esVocal :: Char → Bool
esVocal x = x elem ['a', 'e', 'i', 'o', 'u']
```

La función recibe un carácter como entrada y devuelve un valor booleano. Toma un argumento x (carácter) luego verifica mediante la función `elem` si el elemento dado está o no en la lista de vocales en minúscula.

- `valorAbsoluto :: Int -> Int`, que devuelve el valor absoluto de un entero ingresado.
  - *Opción 1*, se puede definir a la función utilizando la función `abs` nativa:

```
    valorAbsoluto :: Int → Int
    valorAbsoluto x = abs x
```

  - *Opción 2*, se puede definir la función emulando el funcionamiento de `abs`, haciendo lo siguiente:

```
    valorAbsoluto :: Int → Int
    valorAbsoluto x | x >= 0 = x
                    | otherwise = -x
```

Esta función toma un entero x, luego verifica si este es mayor o igual a 0, retorna el mismo valor de x, en caso contrario devuelve $-x$.

Ejercicio 2

Programá las siguientes funciones usando recursión o composición:

- `paraTodo :: [Bool] -> Bool`, que verifica que *todos* los elementos de una lista sean `True`.

```
paraTodo :: [Bool] → Bool
paraTodo [] = True
paraTodo (x:xs) | (x == True) = paraTodo xs
                | otherwise = False
```

Tipo de la función: `[Bool] -> Bool`. La función toma una lista de valores booleanos y devuelve un valor booleano. La función verifica si todos los elementos de la lista son verdaderos. La función `paraTodo` utiliza pattern matching para manejar dos casos: cuando la lista está vacía (`[]`) y cuando la lista tiene al menos un elemento (`(x:xs)`).

1. En el caso base, si la lista está vacía, la función devuelve `True`. Esto implica que todos los elementos (ninguno en este caso) son verdaderos.
2. En el caso recursivo, se verifica si el primer elemento (`x`) es igual a `True`. Si es así, la función `paraTodo` se llama recursivamente con el resto de la lista (`xs`). Esto se hace para verificar los elementos restantes de la lista.
3. Si el primer elemento no es `True`, la función devuelve `False`, ya que al menos un elemento no cumple con la condición de ser verdadero.

En resumen, la función `paraTodo` verifica si todos los elementos de la lista son `True`, devolviendo `True` solo si la lista está vacía o todos los elementos son `True`.

- `sumatoria :: [Int] -> Int`, que calcula la suma de todos los elementos de una lista de enteros.

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

Tipo de la función: `[Int] -> Int`. La función toma una lista de enteros y devuelve la suma de todos los elementos de la lista. La función `sumatoria` utiliza pattern matching para manejar dos casos: cuando la lista está vacía (`[]`) y cuando la lista tiene al menos un elemento (`(x:xs)`).

1. En el caso base, si la lista está vacía, la función devuelve `0`. Esto representa la suma de una lista vacía, que es por definición cero.
2. En el caso recursivo, la suma se calcula sumando el primer elemento (`x`) al resultado de llamar recursivamente a `sumatoria` con el resto de la lista (`xs`). Esto se hace para sumar todos los elementos de la lista, ya que cada llamada recursiva agrega el siguiente elemento al resultado acumulado.

En resumen, la función `sumatoria` calcula la suma de todos los elementos de la lista utilizando recursión.

- `productoria :: [Int] -> Int`, que calcula el producto de todos los elementos de la lista de enteros.

```
productoria :: [Int] → Int
productoria [] = 1
productoria (x:xs) = x * productoria xs
```

Tipo de la función: `[Int] -> Int`. La función toma una lista de enteros y devuelve el producto de todos los elementos de la lista. La función `productoria` utiliza pattern matching para manejar dos casos: cuando la lista está vacía (`[]`) y cuando la lista tiene al menos un elemento (`(x:xs)`).

1. En el caso base, si la lista está vacía, la función devuelve `1`. Esto representa el producto de una lista vacía, que es por definición uno.
2. En el caso recursivo, el producto se calcula multiplicando el primer elemento (`x`) por el resultado de llamar recursivamente a `productoria` con el resto de la lista (`xs`). Esto se hace para calcular el producto de todos los elementos de la lista, ya que cada llamada recursiva multiplica el siguiente elemento al resultado acumulado.

En resumen, la función `productoria` calcula el producto de todos los elementos de la lista utilizando recursión.

- `factorial :: Int -> Int`, que toma un número n y calcula $n!$.

```
    factorial :: Int → Int
    factorial x | x == 0 = 1
                | x == 1 = 1
                | otherwise = x * (factorial (x-1))
```

Tipo de la función: `Int -> Int`. La función toma un número entero y devuelve otro entero. La función `factorial` utiliza múltiples ecuaciones con guardias para manejar diferentes casos:

1. En el primer bloque, se manejan los casos base cuando `x` es 0 o 1. En estos casos, el factorial es 1.
2. En el segundo bloque, se maneja el caso de error. Si `x` es un número negativo, la función lanza un error indicando que el factorial no está definido para números negativos.
3. En el tercer bloque, se maneja el caso recursivo. Para cualquier otro valor de `x`, el factorial se calcula multiplicando `x` por el factorial de `(x - 1)`.

En resumen, la función `factorial` calcula el factorial de un número entero utilizando recursión y maneja casos especiales para 0, 1 y números negativos.

- Utiliza la función sumatoria para definir, `promedio : [Int] -> Int`, que toma una lista de números no vacía y calcula el valor promedio (truncando, usando división entera).

```
promedio :: [Int] → Int
promedio [] = 0
promedio xs = (sumatoria xs) div (length xs)
```

Tipo de la función: `[Int] -> Int`. La función toma una lista de números enteros y devuelve un número entero. **Cuerpo de la función:** - El primer caso base (`[]`) maneja el escenario donde la lista está vacía, y en ese caso, el promedio es 0. - El segundo caso toma la lista `xs` y calcula el promedio dividiendo la suma de los elementos de `xs` (`sumatoria xs`) por la longitud de `xs` (`length xs`). **Función utilizada:** - La función `sumatoria` se usa para calcular la suma de los elementos de la lista. **Ejemplo de ejecución:** - `promedio [2, 4, 6]` calculará la suma de los elementos (12) y la dividirá por la longitud de la lista (3), dando como resultado 4.

Ejercicio 3

Programa la función pertenece :: Int → [Int] → Bool , que verifica si un número se encuentra en una lista.

```
pertenece :: Int → [Int] → Bool
pertenece _ [] = False
pertenece k (x:xs) = k == x || pertenece k xs
```

Tipo de la función: `Int -> [Int] -> Bool`. La función toma un entero y una lista de enteros, y devuelve un valor booleano.
Cuerpo de la función:

- El primer caso base (`_ []`) maneja el escenario donde la lista está vacía. En este caso, la función siempre devuelve `False`.
- El segundo caso toma la lista no vacía `(x:xs)` y verifica si el elemento actual `x` es igual al entero buscado `k`. Si es así, la función devuelve `True`. Si no es igual, la función se llama recursivamente con el resto de la lista `xs`.

Observaciones:

- La función está diseñada para verificar si un entero `k` está presente en la lista, el operador — — nos asegura que cuando encuentre un elemento que cumpla el predicado devolverá `True` ya que es el absorbente de ——.
- Utiliza la recursividad para explorar la lista hasta encontrar una coincidencia o llegar al final de la lista.

Ejemplo de ejecución:

- `pertenece 3 [1, 2, 3, 4, 5]` devolverá `True` porque el número 3 está en la lista.
- `pertenece 7 [1, 2, 3, 4, 5]` devolverá `False` porque el número 7 no está en la lista.

Ejercicio 4

Programa las siguientes funciones que implementan los cuantificadores generales. Nota que el segundo parámetro de cada función es otra función.

- `paratodo' :: [a] -> (a -> Bool) -> Bool`, dada una lista xs de tipo `[a]` y un predicado `t :: a -> Bool`, determina si todos los elementos de xs satisfacen el predicado t.

```haskell
paraTodo' :: [a] → (a → Bool) → Bool
paraTodo' [] f = True
paraTodo' (x:xs) f = f x && paraTodo' xs f
```

  Tipo de la función: `[a] -> (a -> Bool) -> Bool`. La función toma una lista de elementos de tipo `a` y una función `f` que toma un elemento de tipo `a` y devuelve un booleano. La función devuelve un booleano. **Cuerpo de la función:**

  - El primer caso base (`[]`) maneja el escenario donde la lista está vacía. En este caso, la función siempre devuelve `True`.
  - El segundo caso toma la lista no vacía `(x:xs)` y verifica si la aplicación de la función `f` al elemento actual `x` es verdadera (`f x`). Si es verdadero, la función se llama recursivamente con el resto de la lista `xs` y el mismo predicado `f`. Si la aplicación de `f` a `x` es falsa, la función devuelve `False`.

  Observaciones:

  - La función está diseñada para verificar si la función `f` es verdadera para todos los elementos de la lista.
  - Utiliza la recursividad para aplicar la función a cada elemento de la lista.

  Ejemplo de ejecución:

  - `paraTodo' [2, 4, 6] (\n -> n` mod `2 == 0)` devolverá `True` porque la función `(\n -> n` mod `2 == 0)` es verdadera para todos los elementos de la lista `[2, 4, 6]`.

- `existe' :: [a] -> (a -> Bool) -> Bool`, dada una lista xs de tipo `[a]` y un `predicado t :: a -> Bool`, determina si algún elemento de xs satisface el predicado t.

```haskell
existe' :: [a] → (a → Bool) → Bool
existe' [] f = False
existe' (x:xs) f = f x || existe' xs f
```

  Tipo de la función: `[a] -> (a -> Bool) -> Bool`. La función toma una lista de elementos de tipo `a` y una función `f` que toma un elemento de tipo `a` y devuelve un booleano. La función devuelve un booleano. **Cuerpo de la función:**

  - El primer caso base (`[]`) maneja el escenario donde la lista está vacía. En este caso, la función siempre devuelve `False`.
  - El segundo caso toma la lista no vacía `(x:xs)` y verifica si la aplicación de la función `f` al elemento actual `x` es verdadera (`f x`). Si es verdadero, la función devuelve `True`. Si la aplicación de `f` a `x` es falsa, la función se llama recursivamente con el resto de la lista `xs` y el mismo predicado `f`.

  Observaciones:

  - La función está diseñada para verificar si la función `f` es verdadera para al menos un elemento de la lista.
  - Utiliza la recursividad para buscar a través de la lista y determinar si hay al menos un elemento que satisface la condición.

  Ejemplo de ejecución:

  - `existe' [1, 3, 5] (\n -> n mod 2 == 0)` devolverá `False` porque la función `(\n -> n mod 2 == 0)` es falsa para todos los elementos de la lista `[1, 3, 5]`.

- `sumatoria' :: [a] -> (a -> Int) -> Int`, dada una lista xs de tipo `[a]` y una función `t :: a -> Int` (toma elementos de tipo a y devuelve enteros), calcula la suma de los valores que resultan de la aplicación de t a los elementos de xs.

```
sumatoria' :: [a] → (a → Int) → Int
sumatoria' [] f = 0
sumatoria' (x:xs) f = f x + sumatoria' xs f
```

  Tipo de la función: `[a] -> (a -> Int) -> Int`. La función toma una lista de elementos de tipo `a`, una función `f` que toma un elemento de tipo `a` y devuelve un entero, y devuelve un entero. **Cuerpo de la función:**

  - El primer caso base (`[]`) maneja el escenario donde la lista está vacía. En este caso, la función siempre devuelve `0`.
  - El segundo caso toma la lista no vacía `(x:xs)` y calcula la suma de la aplicación de la función `f` al elemento actual `x` más la sumatoria del resto de la lista `xs` con la misma función `f`.

  Observaciones:

  - La función está diseñada para calcular la suma de los valores resultantes de aplicar la función `f` a cada elemento de la lista.
  - Utiliza la recursividad para aplicar la función a cada elemento de la lista y sumar los resultados.

  Ejemplo de ejecución:

  - `sumatoria' [1, 2, 3] (\n -> n * 2)` devolverá `12` porque la función `(\n -> n * 2)` se aplica a cada elemento de la lista y se suman los resultados: `2 + 4 + 6 = 12`.

- `productoria' :: [a] -> (a -> Int) -> Int`, dada una lista de xs de tipo `[a]` y una función `t :: a -> Int`, calcula el producto de los valores que resultan de la aplicación de t a los elementos de xs.

```
productoria' :: [a] → (a → Int) → Int
productoria' [] f = 1
productoria' (x:xs) f = f x * productoria' xs f
```

  Tipo de la función: `[a] -> (a -> Int) -> Int`. La función toma una lista de elementos de tipo `a`, una función `f` que toma un elemento de tipo `a` y devuelve un entero, y devuelve un entero. **Cuerpo de la función:**

  - El primer caso base (`[]`) maneja el escenario donde la lista está vacía. En este caso, la función siempre devuelve `1`.
  - El segundo caso toma la lista no vacía `(x:xs)` y calcula el producto de la aplicación de la función `f` al elemento actual `x` por la productoria del resto de la lista `xs` con la misma función `f`.

  Observaciones:

  - La función está diseñada para calcular el producto de los valores resultantes de aplicar la función `f` a cada elemento de la lista.
  - Utiliza la recursividad para aplicar la función a cada elemento de la lista y multiplicar los resultados.

  Ejemplo de ejecución:

  - `productoria' [1, 2, 3] (\n -> n * 2)` devolverá `48` porque la función `(\n -> n * 2)` se aplica a cada elemento de la lista y se multiplican los resultados: `2 * 4 * 6 = 48`.

Ejercicio 5

Definí nuevamente la función paratodo, pero esta vez usando la función paratodo' (sin recursión ni analisis por casos!).

```
esTrue x = x == True
paratodo'' :: [Bool] → Bool
paratodo'' xs = paraTodo' xs esTrue
```

Función esTrue

Tipo de la función: `Bool -> Bool`. La función toma un valor booleano y devuelve un valor booleano.
Cuerpo de la función:

- La función verifica si el valor booleano `x` es igual a `True`. Devuelve `True` si es así, y `False` en caso contrario.

Función paratodo”

Tipo de la función: `[Bool] -> Bool`. La función toma una lista de booleanos y devuelve un booleano.
Cuerpo de la función:

- La función utiliza la función `paraTodo'` con la lista de booleanos `xs` y la función `esTrue` como argumentos.
- Esto significa que `paratodo''` verifica si todos los elementos de la lista son iguales a `True`.

Observaciones:

- `esTrue` se utiliza para transformar el problema de verificar si todos los elementos de la lista son `True` en el problema de verificar si todos los elementos son iguales a `True`.
- `paratodo''` delega la verificación real a la función `paraTodo'`.

Ejemplo de ejecución:

- `paratodo'' [True, True, True]` devolverá `True` porque todos los elementos de la lista son iguales a `True`.
- `paratodo'' [True, False, True]` devolverá `False` porque al menos un elemento de la lista no es igual a `True`.

Ejercicio 6

Utilizando las funciones del ejercicio 4, programá las siguientes funciones por composición, sin usar recursión ni análisis por casos.

- `todosPares :: [Int] -> Bool` verifica que todos los números de una lista sean pares.

  ```
  todosPares :: [Int] → Bool
  todosPares xs = paraTodo' xs even
  ```

  Tipo de la función: `[Int] -> Bool`. La función toma una lista de enteros y devuelve un booleano.
  Cuerpo de la función:

  – La función utiliza la función `paraTodo'` con la lista de enteros `xs` y la función `even` como argumentos.
  – Esto significa que `todosPares` verifica si todos los elementos de la lista son pares.

Observaciones:

- `even` es una función predefinida en Haskell que devuelve `True` si el número es par y `False` si es impar.
- `todosPares` utiliza `paraTodo'` para verificar si todos los elementos de la lista son pares, utilizando la función `even` como criterio.

Ejemplo de ejecución:

- `todosPares [2, 4, 6, 8]` devolverá `True` porque todos los elementos de la lista son pares.
- `todosPares [2, 3, 4, 6]` devolverá `False` porque hay al menos un elemento impar en la lista.

- `hayMultiplo :: Int -> [Int] -> Bool` verifica si existe algun número dentro del segundo parámetro que sea múltiplo del primer parámetro.

```
esMultiplo :: Int → Int → Bool
esMultiplo a b = mod b a == 0
hayMultiplo :: Int → [Int] → Bool
hayMultiplo n xs = existe' xs (esMultiplo n)
```

Tipo de la función: `Int -> Int -> Bool`. La función toma dos enteros y devuelve un booleano. **Cuerpo de la función:**

- La función verifica si `a` es un múltiplo de `b` verificando si el resultado de `mod b a` es igual a `0`.

Tipo de la función: `Int -> [Int] -> Bool`. La función toma un entero `n` y una lista de enteros `xs`, y devuelve un booleano. **Cuerpo de la función:**

- La función utiliza `existe'` con la lista de enteros `xs` y la función `esMultiplo n` como argumentos.
- Esto significa que `hayMultiplo` verifica si hay al menos un elemento en la lista `xs` que es múltiplo de `n`.

Observaciones:

- `esMultiplo` se utiliza como la función de prueba para la existencia de múltiplos en `hayMultiplo`.
- `hayMultiplo` utiliza `existe'` para verificar si al menos un elemento en la lista `xs` es múltiplo de `n`.

Ejemplo de ejecución:

- `hayMultiplo 3 [1, 2, 3, 4, 5]` devolverá `True` porque al menos un elemento (el 3) es múltiplo de 3.
- `hayMultiplo 7 [2, 4, 6, 8]` devolverá `False` porque no hay elementos en la lista que sean múltiplos de 7.

- `sumaCuadrados :: Int -> Int`, dado un número no negativo `n`, calcula la suma de los primeros `n` cuadrados, es decir $\langle \sum i : 0 \leq i < n : i^2 \rangle$.

```
cuadrado :: Int → Int
cuadrado k = k*k
sumaCuadrados :: Int → Int
sumaCuadrados k = sumatoria' [0..(k-1)] cuadrado
```

Función cuadrado

Tipo de la función: `Int -> Int`. La función toma un entero `k` y devuelve el cuadrado de `k`. **Cuerpo de la función:** La función calcula el cuadrado de `k` multiplicando `k` por sí mismo.

Función sumaCuadrados

Tipo de la función: `Int -> Int`. La función toma un entero `k` y devuelve la suma de los cuadrados de los números desde 0 hasta `k-1`.
Cuerpo de la función:

- La función utiliza `sumatoria'` con la lista de números desde 0 hasta `k-1` y la función `cuadrado` como argumentos.
- Esto significa que `sumaCuadrados` calcula la suma de los cuadrados de los números desde 0 hasta `k-1`.

Observaciones:

- `cuadrado` es una función auxiliar que calcula el cuadrado de un número.
- `sumaCuadrados` utiliza `sumatoria'` para calcular la suma de los cuadrados de los números en un rango específico.

Ejemplo de ejecución: `sumaCuadrados 4` calculará la suma de los cuadrados de los números desde 0 hasta 3: `0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14`.

- Programar la función `existeDivisor::Int-> [Int] -> Bool`, que dado en entero `n` y una lista `ls` , devuelve `True` si y solo si, existe algún elemento en `ls` que divida a `n`.

```
esDivisor :: Int -> Int -> Bool
esDivisor n d = n mod d == 0
existeDivisor :: Int -> [Int] -> Bool
existeDivisor n ls = existe' ls (esDivisor n)
```

Función esDivisor

Tipo de la función: `Int -> Int -> Bool`. La función toma dos enteros `n` y `d` y devuelve un booleano.
Cuerpo de la función: La función verifica si `d` es un divisor de `n` comprobando si el resultado de `mod n d` es igual a 0.

Función existeDivisor

Tipo de la función: `Int -> [Int] -> Bool`. La función toma un entero `n` y una lista de enteros `ls`, y devuelve un booleano. **Cuerpo de la función:**

- La función utiliza `existe'` con la lista de enteros `ls` y la función `esDivisor n` como argumentos.
- Esto significa que `existeDivisor` verifica si hay al menos un elemento en la lista `ls` que es divisor de `n`.

Observaciones:

- `esDivisor` se utiliza como la función de prueba para la existencia de divisores en `existeDivisor`.
- `existeDivisor` utiliza `existe'` para verificar si al menos un elemento en la lista `ls` es divisor de `n`.

Ejemplo de ejecución:

- `existeDivisor 7 [2, 3, 4, 5]` devolverá `False` porque ningún elemento de la lista es divisor de 7.
- `existeDivisor 8 [3, 5, 7]` devolverá `True` porque al menos un elemento (el 4) es divisor de 8.

- Utilizando la funcion del apartado anterior, definí la función `esPrimo:: Int -> Bool`, que dado un entero n, devuelve True si y solo si n es primo.

```
esPrimo :: Int → Bool
esPrimo n = n > 1 && not (existeDivisor n [2..(n-1)])
```

Tipo de la función: `Int -> Bool`. La función toma un entero `n` y devuelve un booleano. **Cuerpo de la función:** La función verifica dos condiciones:

1. `n` es mayor que 1.
2. No existe ningún divisor en el rango de 2 a `(n-1)` utilizando la función `existeDivisor`.

Observaciones:

- La función utiliza la función `existeDivisor` para verificar si hay algún divisor en el rango de 2 a `(n-1)`.
- Si ambas condiciones se cumplen, la función devuelve `True`, indicando que `n` es primo. Si alguna de las condiciones no se cumple, devuelve `False`.

Ejemplo de ejecución:

- `esPrimo 7` devolverá `True` porque 7 es un número primo.
- `esPrimo 10` devolverá `False` porque 10 no es un número primo (tiene divisores en el rango de 2 a 9).

- ¿Se te ocurre como redefinir factorial (ej. 2d) para evitar usar recursión?

```
factorial :: Int → Int
factorial x = productoria [1..x]
```

Tipo de la función: `Int -> Int`. La función toma un entero `x` y devuelve un entero. **Cuerpo de la función:**

- La función utiliza la función `productoria` con la lista `[1..x]`. Esta lista contiene todos los números desde 1 hasta `x`.
- La función `productoria` calcula el producto de los elementos en la lista, que es esencialmente el factorial de `x`.

Ejemplo de ejecución:

- `factorial 5` devolverá el mismo resultado que antes, calculando el factorial de 5 utilizando la función `productoria`.

- Programar la función `multiplicaPrimos :: [Int] -> Int` que calcula el producto de todos los números primos de una lista.

```
multiplicaPrimos :: [Int] → Int
multiplicaPrimos xs = productoria' [ x | x <- xs, esPrimo x ] id
```

Tipo de la función: `[Int] -> Int`. La función toma una lista de enteros `xs` y devuelve un entero. **Cuerpo de la función:**

- La función utiliza la función `productoria'` con una lista comprensiva `[x` x ¡- xs, esPrimo x]— y la función identidad `id` como argumentos.
- La lista comprensiva filtra solo los números primos de la lista original `xs`.

Observaciones:

- La función utiliza `esPrimo` para determinar si un número es primo.

  - La función `productoria'` se utiliza para calcular el producto de los números primos en la lista.

  Ejemplo de ejecución: `multiplicaPrimos [2, 3, 4, 5, 6]` devolverá 30 porque 2, 3 y 5 son primos, y su producto es $2 \cdot 3 \cdot 5 = 30$.

- Programar la función `esFib :: Int -> Bool`, que dado un entero `n`, devuelve `True` si y sólo si n está en la sucesión de Fibonacci.

```haskell
fib :: Int → Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

esFib :: Int → Bool
esFib n = pertenece n [fib i | i <- [0..n]]
```

Función fib

Tipo de la función: `Int -> Int`. La función toma un entero `n` y devuelve el n-ésimo número de la secuencia de Fibonacci. **Cuerpo de la función:**

- La función utiliza la recursión para calcular el n-ésimo número de Fibonacci.
- Los casos base son `fib 0 = 0` y `fib 1 = 1`.
- La definición general utiliza la fórmula de Fibonacci: `fib n = fib (n-1) + fib (n-2)`.

Función esFib

Tipo de la función: `Int -> Bool`. La función toma un entero `n` y devuelve un booleano. **Cuerpo de la función:**

- La función crea una lista de los primeros `n+1` números de Fibonacci utilizando la comprensión de listas `[fib i` i ¡- [0..n]]—.
- Luego, verifica si `n` pertenece a esa lista utilizando la función `pertenece`.

Observaciones: La función `esFib` utiliza la función `pertenece` para verificar si un número `n` pertenece a la secuencia de Fibonacci. **Ejemplo de ejecución:** `esFib 5` devolverá `True` porque 5 es parte de la secuencia de Fibonacci.

- Utilizando la función del apartado anterior, definí la función `todosFib :: [Int] -> Bool` que dada una lista xs de enteros, devuelva si todos los elementos de la lista pertenecen (o no) a la sucesión de Fibonacci.

```haskell
todosFib :: [Int] → Bool
todosFib xs = paraTodo' xs esFib
```

Tipo de la función: `[Int] -> Bool`. La función toma una lista de enteros `xs` y devuelve un booleano. **Cuerpo de la función:**

- La función utiliza `paraTodo'` con la lista de enteros `xs` y la función `esFib` como argumentos.
- Esto significa que `todosFib` verifica si todos los elementos de la lista `xs` son números que pertenecen a la secuencia de Fibonacci.

Observaciones:

- `esFib` se utiliza como la función de prueba para la verificación en `todosFib`.

- `todosFib` utiliza `paraTodo'` para verificar si todos los elementos de la lista `xs` cumplen la propiedad definida por `esFib`.

Ejemplo de ejecución:

- `todosFib [5, 8, 13]` devolverá `True` porque todos los elementos de la lista son parte de la secuencia de Fibonacci.
- `todosFib [5, 8, 14]` devolverá `False` porque el último elemento (14) no es parte de la secuencia de Fibonacci.

Ejercicio 7

Indagá en Hoogle sobre las funciones map y filter. También podes consultar su tipo en ghci con el comando :t .

Función MAP

MAP es una f de orden superior que toma una función (que a su vez ésta toma un a y un b,) y una lista xs y aplica esa función a cada elemento de xs, produciendo una nueva lista.

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Función FILTER

FILTER es una f que toma un predicado y una lista, devolviendo una lista con los elementos que satisfacen el predicado. Es decir, si p x se evalua en True, x es incluido a la lista.

```
filter :: (a → Bool) → [a] → [b]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

¿A qué equivale la expresión map succ [1, -4, 6, 2, -8], donde succ n = n+1?

Equivale a la lista [2,-3,7,3,-7], donde cada elemento es el siguiente de la lista dada.

¿Y la expresión filter esPositivo [1, -4, 6, 2, -8]?

A la lista de los positivos [1,6,2] pertenecientes a la lista dada.

Ejercicio 8

Programá una función que dada una lista de números xs, devuelve la lista que resulta de duplicar cada valor de xs.

- Definila usando recursión:

```
dupLista :: (Num a) => [a] → [a]
dupLista [] = []
dupLista (x:xs) = (2*x) : dupLista xs
```

La función toma una lista de números y devuelve una nueva lista donde cada elemento es el doble del correspondiente elemento en la lista original. La función `dupLista` utiliza patternt matching para manejar dos casos: cuando la lista está vacía (`[]`) y cuando la lista tiene al menos un elemento (`(x:xs)`).

- En el caso base, si la lista está vacía, la función devuelve una lista vacía también.
- En el caso recursivo, se construye una nueva lista donde el primer elemento es el doble de `x`, y el resto de la lista se obtiene llamando recursivamente a `dupLista` con el resto de la lista original (`xs`). Esto se hace para cada elemento de la lista original.

- Definila utilizando la función map:

```
dupLista' :: (Num a) => [a] → [a]
dupLista' xs = map (*2) xs
```

La función `dupLista'` es otra implementación de la función que duplica cada elemento de una lista, pero utiliza la función `map` para lograrlo de una manera más concisa.

- La función `map` toma una función y una lista, y aplica esa función a cada elemento de la lista, devolviendo una nueva lista con los resultados.
- En este caso, la función `(*2)` es la función que se aplica a cada elemento de la lista `xs`. Multiplicar un número por 2 es equivalente a duplicarlo.
- Por lo tanto, `map (*2) xs` produce una lista donde cada elemento es el doble del elemento correspondiente en la lista original.

Ejercicio 9

Programá una función que dada una lista de números xs, calcula una lista que tiene como elementos aquellos números de xs que son primos.

- **a)** Definila usando recursión:

```
    primListas :: [Int] → [Int]
    primListas [] = []
    primListas (x:xs) | esPrimo x = x : primListas xs
                      | otherwise = primListas xs
```

La función `primListas` toma una lista de números enteros y devuelve una nueva lista que contiene solo los números primos de la lista original, utiliza pattern matching para manejar dos casos: cuando la lista está vacía (`[]`) y cuando la lista tiene al menos un elemento (`(x:xs)`).

- En el caso base, si la lista está vacía, la función devuelve una lista vacía también.
- En el caso recursivo, se utiliza la función `esPrimo` para verificar si el primer elemento (`x`) es primo. Si es primo, se agrega a la lista resultante (`x : primListas xs`) y se llama recursivamente con el resto de la lista (`xs`). Si no es primo, simplemente se llama recursivamente con el resto de la lista.

- **b)** Definila usando la función Filter:

```
primListas' :: [Int] → [Int]
primListas' xs = filter esPrimo xs
```

La función `primListas'` es otra implementación de la función que filtra los números primos de una lista, pero utiliza la función `filter` para lograrlo de una manera más concisa.

- La función `filter` toma una función de condición y una lista, y devuelve una nueva lista que contiene solo los elementos que cumplen con la condición.

- En este caso, `esPrimo` es la función de condición que se aplica a cada elemento de la lista `xs`. La función `filter esPrimo xs` devuelve una lista que contiene solo los números primos de la lista original.

- **c)** Revisá tu definicián del ejercicio 6g . ¿Cómo podes mejorarla?

```
multiplicaPrimos' :: [Int] → Int
multiplicaPrimos' xs = product (filter esPrimo xs)
```

La función `multiplicaPrimos'` toma una lista de números enteros y devuelve el producto de los números primos en esa lista.

- `filter esPrimo xs` filtra la lista original `xs` y devuelve una nueva lista que contiene solo los números primos.
- `product` toma una lista de números y devuelve su producto. En este caso, se aplica a la lista de números primos para calcular el producto de esos números.

Por lo tanto, la función `multiplicaPrimos'` primero filtra la lista original para obtener solo los números primos y luego calcula el producto de esos números primos.

Ejercicio 10

La función primIgualesA toma un valor y una lista, y calcula el tramo inicial más largo de la lista cuyos elementos son iguales a ese valor. Por ejemplo:

- primIgualesA 3 [3,3,4,1] = [3,3]
- primIgualesA 3 [4,3,3,4,1] = []
- primIgualesA 3 [] = []
- primIgualesA 'a' "aaadaa" = "aaa"

- **a)** Programá primIguales por recursión.

```
    primIgualesA :: (Eq a) => a → [a] → [a]
    primIgualesA _ []  = []
    primIgualesA k (x:xs) | k == x = x : primIgualesA k xs
                          | otherwise = []
```

La función `primIgualesA` toma un elemento `k` y una lista de elementos, y devuelve una lista que contiene solo los elementos iguales a `k` que están al principio de la lista, utiliza pattern matching para manejar dos casos: cuando la lista está vacía (`[]`) y cuando la lista tiene al menos un elemento (`(x:xs)`).

- En el caso base, si la lista está vacía, la función devuelve una lista vacía también.
- En el caso recursivo, se verifica si el primer elemento (`x`) es igual a `k`. Si es así, se agrega a la lista resultante (`x : primIgualesA k xs`) y se llama recursivamente con el resto de la lista (`xs`). Si el primer elemento no es igual a `k`, la lista resultante será vacía (`[]`).

- **b)** Programá nuevamente la función utilizando takeWhile.

```
primIgualesA' :: (Eq a) => a → [a] → [a]
primIgualesA' n = takeWhile (==n)
```

- `takeWhile` toma una función de condición y una lista, y devuelve la lista de elementos que cumplen con la condición hasta que se encuentre el primer elemento que no la cumple.

- En este caso, `takeWhile (==n)` toma elementos de la lista mientras sean iguales a `n`.

Entonces, la función `primIgualesA'` es equivalente a la función `primIgualesA` en términos de resultados, pero utiliza `takeWhile` para lograrlo de manera más concisa.

Ambas funciones, `primIgualesA` y `primIgualesA'`, hacen lo mismo: devuelven una lista que contiene solo los elementos iguales a `n` que están al principio de la lista original.

Ejercicio 11

La función primIguales toma una lista y devuelve el mayor tramo inicial de la lista cuyos elementos son todos iguales entre sí.

a) Programá primIguales por recursión:

```
primIguales :: (Eq a) => [a] → [a]
primIguales [] = []
primIguales (x:xs) | (x == head xs) = x : primIguales xs
                   | otherwise = x : []
```

La función `primIguales` funciona de la siguiente manera:

- La función toma una lista de elementos de tipo a que es comparable (es decir, a es una instancia de la clase `Eq`) y devuelve otra lista de elementos del mismo tipo.
- La primera ecuación de patrones maneja el caso base cuando se proporciona una lista vacía como entrada. En este caso, devuelve una lista vacía como salida, ya que no hay elementos para comparar.
- La segunda ecuación de patrones maneja el caso en el que la lista tiene al menos un elemento. Divide la lista en la cabeza (`x`) y la cola (`xs`). Luego, verifica si el primer elemento `x` es igual al primer elemento de la cola, `head xs`. Si son iguales, agrega `x` a la lista de salida y llama recursivamente a la función `primIguales` con la cola `xs`.
- La guardia `| otherwise` en la segunda ecuación de patrones significa que si no se cumple la condición anterior (es decir, `x` no es igual a `head xs`), se ejecutará el siguiente bloque.
- Dentro de este bloque, agrega el elemento `x` a una lista unitaria `[x]`. Esto es lo que hace `x : []`, donde `:` se utiliza para agregar un elemento a una lista.

En resumen, la función `primIguales` toma una lista y devuelve otra lista que contiene solo los elementos iniciales que son iguales entre sí. Si hay una secuencia de elementos iguales al principio de la lista, los recoge y los devuelve en una lista separada. Si no hay elementos iguales consecutivos, simplemente devuelve el primer elemento en una lista.

b) Usá cualquier versión de primIgualesA para programar primIguales. Está permitido dividir en casos, pero no usar recursión.

```
primIguales' :: (Eq a) => [a] → [a]
primIguales' (x:xs) = primIgualesA' x (x:xs)
```

Ejercicio 12

Todas las funciones del ejercicio 4 son similares entre sí: cada una aplica la función término t a todos los elementos de una lista, y luego aplica algún operador entre todos ellos, obteniéndose as ı el resultado final. Para el caso de la lista vacía, se devuelve el elemento neutro. De esa manera cada una de ellas computa una cuantificación sobre los elementos de la lista transformados por t:

- $paratodo'.xs.t = \langle \forall i : 0 \leq i < \#xs : t.xs!i \rangle$
- $existe'.xs.t = \langle \exists i : 0 \leq i < \#xs : t.xs!i \rangle$
- $sumatoria'.xs.t = \langle \sum i : 0 \leq i < \#xs : t.xs!i \rangle$
- $productoria'.xs.t = \langle \prod i : 0 \leq i < \#xs : t.xs!i \rangle$

Reescribir todas las funciones del punto 4 utilizando el cuantificador generalizado (sin usar inducción y en una línea por función).

Guiándote por las observaciones anteriores, definí de manera recursiva la función cuantGen (denota la cuantificación generalizada):

```
cuantGen :: (b → b → b) → b → [a] → (a → b) → b
cuantGen op z xs t = foldr op z (map t xs)
```

La función `cuantGen` es una función de orden superior en Haskell que utiliza la función `foldr` para aplicar una operación binaria `op` acumulativamente a los elementos de una lista transformada por la función `t`.

- `map t xs` transforma cada elemento de la lista `xs` aplicando la función de transformación `t`.
- `foldr op z` toma la lista transformada y acumula los elementos utilizando la operación binaria `op`, comenzando con el elemento inicial `z`.

Por lo tanto, la función `cuantGen` toma una operación binaria, un elemento inicial, una lista y una función de transformación. Aplica la operación binaria acumulativamente a los elementos transformados de la lista utilizando `foldr`.

- **a)** $paratodo'.xs.t = \langle \forall i : 0 \leq i < \#xs : t.xs!i \rangle$

```
    paratodo''' :: [a] → (a → Bool) → Bool
    paratodo''' xs t = cuantGen (&&) True xs t
```

Esta función se encarga de verificar si todos los elementos de la lista cumplen con una cierta condición dada por la función `t`.

- `cuantGen (&&) True` utiliza `cuantGen` con la operación lógica AND (`&&`) como la operación binaria y el valor inicial `True`. Esto significa que la función `cuantGen` acumulará los resultados usando la operación lógica AND, y el valor inicial es `True`.
- `xs` es la lista sobre la cual se aplicará la función de condición `t`.
- La función `cuantGen` aplicará la operación lógica AND acumulativamente a los resultados de `t` aplicados a los elementos de la lista. El resultado final será `True` si todos los elementos cumplen con la condición y `False` en caso contrario.

- **b)** $existe'.xs.t = \langle \exists i : 0 \leq i < \#xs : t.xs!i \rangle$

```
    existe''' :: [a] → (a → Bool) → Bool
    existe''' xs t = cuantGen (||) False xs t
```

La función `existe'''` es otra función de orden superior en Haskell que utiliza la función `cuantGen` para verificar si al menos un elemento de la lista cumple con una cierta condición dada por la función `t`.

- `cuantGen (—) False—` utiliza `cuantGen` con la operación lógica OR (`||`) como la operación binaria y el valor inicial `False`. Esto significa que la función `cuantGen` acumulará los resultados usando la operación lógica OR, y el valor inicial es `False`.
- `xs` es la lista sobre la cual se aplicará la función de condición `t`.
- La función `cuantGen` aplicará la operación lógica OR acumulativamente a los resultados de `t` aplicados a los elementos de la lista. El resultado final será `True` si al menos un elemento cumple con la condición y `False` en caso contrario.
- **c)** $sumatoria'.xs.t = \langle \sum i : 0 \leq i < \#xs : t.xs!i \rangle$

```haskell
sumatoria''' :: [a] → (a → Int) → Int
sumatoria''' xs t = cuantGen (+) 0 xs t
```

La función `sumatoria'''` es otra función de orden superior en Haskell que utiliza la función `cuantGen` para calcular la sumatoria de los valores resultantes de aplicar una función `t` a cada elemento de la lista.

- `cuantGen (+) 0` utiliza `cuantGen` con la operación de suma (`(+)`) como la operación binaria y el valor inicial `0`. Esto significa que la función `cuantGen` acumulará los resultados usando la operación de suma, y el valor inicial es `0`.
- `xs` es la lista sobre la cual se aplicará la función de transformación `t`.
- La función `cuantGen` aplicará la operación de suma acumulativamente a los resultados de `t` aplicados a los elementos de la lista. El resultado final será la sumatoria de los valores transformados.
- **d)** $productoria'.xs.t = \langle \prod i : 0 \leq i < \#xs : t.xs!i \rangle$

```haskell
productoria''' :: [a] → (a → Int) → Int
productoria''' xs t = cuantGen (*) 1 xs t
```

La función `productoria'''` es otra función de orden superior en Haskell que utiliza la función `cuantGen` para calcular la productoria de los valores resultantes de aplicar una función `t` a cada elemento de la lista.

- `cuantGen (*) 1` utiliza `cuantGen` con la operación de multiplicación (`(*)`) como la operación binaria y el valor inicial `1`. Esto significa que la función `cuantGen` acumulará los resultados usando la operación de multiplicación, y el valor inicial es `1`.
- `xs` es la lista sobre la cual se aplicará la función de transformación `t`.
- La función `cuantGen` aplicará la operación de multiplicación acumulativamente a los resultados de `t` aplicados a los elementos de la lista. El resultado final será la productoria de los valores transformados.

Ejercicio 13

Definir una función que se denomina distancia de edición. Que toma como entrada dos strings (lista de caracteres).

```haskell
distanciaEdicion :: [Char] → [Char] → Int
```

. La función distanciaEdicion, se comporta de la siguiente manera:

- Si alguna de las listas es vacía, devuelve la longitud de la otra lista.
- Si las dos listas son no vacías, compara los primeros elementos de cada lista:
  - Si `x==y`, no suma y sigue computando la distancia para `xs` e `ys` ,
  - Si `x!=y`, suma 1 y sigue computando la distancia para `xs` e `ys`

La función

```haskell
\begin{haskell}
minTres :: Int → Int → Int → Int
minTres a b c = min a (min b c)
distanciaEdicion :: [Char] → [Char] → Int
distanciaEdicion [] ys = length ys
distanciaEdicion xs [] = length xs
distanciaEdicion (x:xs) (y:ys)
  | x == y = distanciaEdicion xs ys
  | otherwise = 1 + minTres (distanciaEdicion xs (y:ys)) (distanciaEdicion (x:xs)
    ys) (distanciaEdicion xs ys)
```

- La función `minTres` es una función auxiliar que toma tres valores y devuelve el mínimo entre ellos.
- En la función `distanciaEdicion`, se manejan tres casos: si la primera cadena está vacía, la distancia es la longitud de la segunda cadena; si la segunda cadena está vacía, la distancia es la longitud de la primera cadena; y el caso recursivo, donde ambas cadenas tienen al menos un elemento.
- En el caso recursivo, se verifica si los primeros elementos de las cadenas son iguales. Si es así, no se requiere ninguna operación de edición en esta posición, y la función se llama recursivamente con el resto de las cadenas.
- Si los primeros elementos son diferentes, se consideran tres operaciones posibles: insertar y en xs, eliminar x de xs o sustituir x por y en xs. Se utiliza la función `minTres` para determinar cuál de estas operaciones requiere el menor número de ediciones.

Ejercicio 14

Definí una función primeros que cumplen, `primQueCumplen::[a]->(a->Bool)->[a]`, tal que, dada una lista ls y un predicado p, devuelve el tramo inicial de ls que cumple p.

```haskell
primQueCumplen :: [a] → (a → Bool) → [a]
primQueCumplen [] p = []
primQueCumplen (l:ls) p | (p l) = l : primQueCumplen ls p
                        | otherwise = primQueCumplen ls p
```

La función toma una lista `ls` y un predicado `p`, y devuelve el tramo inicial de la lista `ls` que cumple con el predicado `p`. En otras palabras, la función devuelve los primeros elementos de la lista que satisfacen la condición especificada por la función `p`, utiliza pattern matching para manejar dos casos: cuando la lista está vacía (`[]`) y cuando la lista tiene al menos un elemento (`(l:ls)`).

- En el caso base, si la lista está vacía, la función devuelve una lista vacía también.
- En el caso recursivo, se verifica si el primer elemento (`l`) cumple con la condición dada por la función `p`. Si cumple, se agrega a la lista resultante (`l : primQueCumplen ls p`) y se llama recursivamente con el resto de la lista (`ls`). Si el primer elemento no cumple con la condición, simplemente se llama recursivamente con el resto de la lista.

Ejercicio 15

Para cada uno de los siguientes patrones, decidí si están bien tipados, y en tal caso dá los tipos de cada subexpresión. En caso de estar bien tipado, ¿el patrón cubre todos los casos de definición?

- a) **Bien tipado y cubre todos los casos**

```
f :: (a, b) → ...
f (x , y) = ...
```

En el patrón `f (x, y) = ...`, la función `f` toma una tupla como argumento.

1. **Tipo de la función f**:
   - La función `f` toma una tupla como argumento. La tupla tiene dos elementos, `x` y `y`.
   - Por lo tanto, el tipo de la función `f` sería `(a, b) -> ...`, donde `a` y `b` son los tipos de `x` e `y` respectivamente.
2. **Cobertura de casos**:
   - Este patrón cubre todos los casos posibles para tuplas de dos elementos, ya que está desempaquetando ambos elementos `x` e `y` de la tupla.

- b) **No está bien tipado y no cubre adecuadamente los casos**

```
    f :: [(a, b)] → ...
    f (a , b) = ...
```

1. **Tipo de la función f**:
   - La función `f` toma una lista de tuplas como argumento. Por lo tanto, el tipo de la función `f` sería `[(a, b)] -> ...`, donde `a` y `b` son los tipos de los elementos de las tuplas en la lista.
2. **Cobertura de casos**:
   - Este patrón no cubre ningún caso, la función debería tomar una lista de tuplas.

- c) **Bien tipado, pero no cubre todos los casos**

```
    f :: [(a, b)] → ...
    f (x:xs) = ...
```

1. **Tipo de la función f**:
   - La función `f` toma una lista de tuplas como argumento. Por lo tanto, el tipo de la función `f` sería `[(a, b)] -> ...`, donde `a` y `b` son los tipos de los elementos de las tuplas en la lista.
2. **Cobertura de casos**:
   - Este patrón cubre al menos el caso en que la lista no está vacía (`x:xs`). El patrón está utilizando la estructura de cons (`:`) para desempaquetar el primer elemento `x` y el resto de la lista `xs`.

Por lo tanto, el patrón `f (x:xs) = ...` está bien tipado y cubre al menos el caso de una lista no vacía. Sin embargo, se el caso de una lista vacía, y se podría hacer de la forma:

```
f :: [(a, b)] → ...
f [] = ...
f (x:xs) = ...
```

- d) **Bien tipado, pero no cubre todos los casos**

```
    f :: [(a, b)] → ...
    f ((x, y) : ((a, b) : xs)) = ...
```

1. **Tipo de la función f**:
   - La función `f` toma una lista de tuplas como argumento. El tipo de la función debería ser `[(a, b)] -> ...`, donde `a` y `b` son los tipos de los elementos de las tuplas.
2. **Cobertura de casos**:
   - Este patrón solo cubrirá casos en los que la lista tiene al menos dos tuplas. No cubre el caso de una lista vacía ni el caso de una lista con una sola tupla.

Para que cubra todos los casos, la definición podría ser:

```haskell
f :: [(a, b)] -> ...
f [] = ...
f [(x, y)] = ...
f ((x, y) : ((a, b) : xs)) = ...
```

- e) **Bien tipado, pero no cubre todos los casos**

```haskell
f :: [(Int, a)] -> ...
f [(0, a)] = ...
```

1. **Tipo de la función f**:
   - La función `f` toma una lista de tuplas donde el primer elemento es de tipo `Int` y el segundo elemento puede ser de cualquier tipo `a`. El tipo de la función sería `[(Int, a)] -> ...`.
2. **Cobertura de casos**:
   - Este patrón solo cubrirá el caso en el que la lista tenga exactamente un elemento, y ese elemento sea una tupla con el primer elemento igual a `0`. No cubre casos de listas con más de un elemento ni casos de listas vacías.

Para que cubra todos los casos, la definición podría ser:

```haskell
f :: [(Int, a)] -> ...
f [] = ...
f [(0, a)] = ...
f ((x, y):xs) = ...
```

- f) **Mal tipado**

```haskell
f :: [(Int, a)] -> ...
f ((x, 1) : xs) = ...
```

1. **Tipo de la función f**:
   - La función `f` toma una lista de tuplas donde el primer elemento es de tipo `Int` y el segundo elemento es de tipo `a` (pues `1` es un número entero y se puede considerar de cualquier tipo). El tipo de la función sería `[(Int, a)] -> ...`.
2. **Cobertura de casos**:
   - Este patrón solo cubrirá el caso en el que la lista empiece con una tupla donde el primer elemento es de tipo `Int` y el segundo elemento es `1`. Estaría mal tipado ya que `1` no es del tipo `a`.

- g) **Bien tipado y cubre todos los casos**

```
    f :: (Int → Int) → Int → ...
    f a b = ...
```

1. **Tipo de la función `f`**:
   - La función `f` toma una función de tipo `Int -> Int` y un entero. Por lo tanto, el tipo de la función `f` sería `(Int -> Int) -> Int -> ...`.
2. **Cobertura de casos**:
   - Este patrón no tiene un patrón específico de descomposición de datos, ya que toma una función y un entero directamente. Cubre toda posibilidad correctamente.

- h) **Bien tipado, pero no cubre todos los casos**

```
f :: (Int → Int) → Int → ...
f a 3 = ...
```

1. **Tipo de la función `f`**:
   - La función `f` toma una función de tipo `Int -> Int` y un entero. Por lo tanto, el tipo de la función `f` sería `(Int -> Int) -> Int -> ...`.
2. **Cobertura de casos**:
   - Este patrón cubre específicamente el caso en el que el segundo argumento es `3`. No cubre otros casos en los que el segundo argumento puede ser diferente de `3`.

Para que cubra todos los casos, la definición podría ser:

```
f :: (Int → Int) → Int → ...
f a 3 = ...
f a b = ...
```

- i) **Mal tipado**

```
f :: (Int → Int) → Int → ...
f 0 1 2 = ...
```

1. **Tipo de la función `f`**:
   - La función `f` toma una función de tipo `Int -> Int` y un entero. Por lo tanto, el tipo de la función `f` sería `(Int -> Int) -> Int -> ...`.
2. **Cobertura de casos**:
   - Este patrón no cubre ningún caso, la función está mal tipada. `0` no puede tener el tipo `Int -> Int`.

Ejercicio 16

Para las siguientes declaraciones de funciones, da al menos una definición cuando sea posible. ¿Podés dar alguna otra definición alternativa a la que diste en cada caso?

Punto A

```haskell
f :: (a, b) -> b
```

En este caso, no hay otra posibilidad más que devolver el segundo valor de la tupla:

```haskell
f :: (a, b) -> b
f (_,y) = y
```

Punto B

```haskell
f :: (a, b) -> c
```

No podemos construir algo de tipo `c` solo conociendo `a` y `b`, a menos que supongamos que el tipo `c` puede ser una lista con el elemento de tipo `a`, por ejemplo:

```haskell
f :: (a, b) -> c
f (x, y) = [x]
```

Punto C

```haskell
f :: (a -> b) -> a -> b
```

En este caso nos obliga a utilizar una función `t`:

```haskell
f :: (a -> b) -> a -> b
f t x = t x
```

Una alternativa es llamar recursivamente a la función `f`:

```haskell
f :: (a -> b) -> a -> b
f g x = g (f g x)
```

Punto D

```haskell
f :: (a -> b) -> [a] -> [b]
```

En la definición siguiente, se podría utilizar la función `haskell{map}` que cumple con la consigna:

```haskell
f :: (a -> b) -> [a] -> [b]
f t xs = map t xs
```

También se podría utilizar recursión en vez de la función `haskell{map}`.

```haskell
f :: (a -> b) -> [a] -> [b]
f _ [] = []
f t (x:xs) = t x : f t xs
```