Guía de Laboratorio – Sesión 14
Recursividad con Listas y Aplicaciones

Docente: Carlos R. P. Tovar

Ciclo 2025-2

Objetivos

- Comprender la aplicación de la recursividad en listas.
- Implementar funciones recursivas para operaciones básicas.
- Resolver problemas prácticos utilizando listas en Haskell.
- Analizar la complejidad de funciones recursivas.

Requisitos Previos

- Instalación de GHC/GHCi (The Glasgow Haskell Compiler)
- Conocimiento básico de sintaxis de Haskell y pattern matching
- Comprensión de tipos de datos básicos y listas
- Familiaridad con el concepto de recursividad simple

Conceptos Clave

- Una función recursiva sobre listas siempre tiene:
  - Caso base (lista vacía `[]`)
  - Llamada recursiva (procesa la cola de la lista `xs`)
- La recursividad reemplaza los bucles tradicionales
- Esquema general:

```
funcion :: [a] → b
funcion []     = ...       -- Caso base
funcion (x:xs) = ... x ... funcion xs ...  -- Caso recursivo
```

Ejercicios Resueltos

1. Suma de elementos de una lista

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

-- Pruebas
-- suma [1,2,3,4]  => 10
-- suma []         => 0
```

2. Conteo de elementos de una lista

```
contar :: [a] → Int
contar []     = 0
contar (_:xs) = 1 + contar xs

-- Pruebas
-- contar [5,10,15]      => 3
-- contar ["a","b","c","d"] => 4
```

3. Buscar un elemento en una lista

```
buscar :: Eq a => a → [a] → Bool
buscar _ [] = False
buscar e (x:xs)
    | e == x    = True
    | otherwise = buscar e xs

-- Pruebas
-- buscar 3 [1,2,3,4] => True
-- buscar 5 [1,2,3,4] => False
```

4. Invertir una lista

```haskell
invertir :: [a] → [a]
invertir []     = []
invertir (x:xs) = invertir xs ++ [x]

-- Pruebas
-- invertir [1,2,3] => [3,2,1]
-- invertir "Haskell" => "lleksaH"
```

Nota: Esta implementación es ineficiente ($O(n^2)$). Versión más eficiente:

```haskell
invertirEficiente :: [a] → [a]
invertirEficiente xs = invertirAux xs []
  where
    invertirAux []     acum = acum
    invertirAux (y:ys) acum = invertirAux ys (y:acum)
```

5. Máximo de una lista

```haskell
maximo :: Ord a => [a] → a
maximo [x]    = x
maximo (x:xs) = max x (maximo xs)

-- Pruebas
-- maximo [2,7,5,9,3] => 9
-- maximo [42]        => 42
```

Nota: Esta función falla con lista vacía. Versión segura:

```haskell
maximoSeguro :: Ord a => [a] → Maybe a
maximoSeguro []     = Nothing
maximoSeguro [x]    = Just x
maximoSeguro (x:xs) = case maximoSeguro xs of
                        Just y  → Just (max x y)
                        Nothing → Just x
```

6. Filtrar elementos pares

```haskell
pares :: [Int] → [Int]
pares [] = []
pares (x:xs)
    | even x    = x : pares xs
    | otherwise = pares xs

-- Pruebas
-- pares [1,2,3,4,5,6] => [2,4,6]
```

7. Concatenar lista de listas

```haskell
concatenar :: [[a]] → [a]
concatenar [] = []
concatenar (xs:xss) = xs ++ concatenar xss

-- Pruebas
-- concatenar [[1,2],[3,4],[5]] => [1,2,3,4,5]
```

8. Aplanar lista de listas (usando foldr)

```haskell
aplanar :: [[a]] → [a]
aplanar = foldr (++) []

-- Pruebas
-- aplanar [[1,2],[],[3,4]] => [1,2,3,4]
```

Análisis de Complejidad

Ejercicios Propuestos

- Implementar una función recursiva que elimine todas las ocurrencias de un número en una lista.
- Definir una función que cuente cuántos números pares hay en una lista.
- Crear una función que calcule el producto de todos los elementos de una lista.
- Escribir una función que obtenga el mínimo de una lista de enteros.
- Implementar una función que sume únicamente los números impares de una lista.

Función	Complejidad Temporal	Complejidad Espacial
`suma`	O(n)	O(n)
`contar`	O(n)	O(n)
`buscar`	O(n)	O(n)
`invertir`	$O(n^2)$	O(n)
`invertirEficiente`	O(n)	O(1)
`maximo`	O(n)	O(n)
`pares`	O(n)	O(n)
`concatenar`	O(m)	O(m)
`aplanar`	O(m)	O(1)

Cuadro 1: Complejidad de las funciones implementadas (n = tamaño lista, m = total elementos)

- Crear una función que verifique si una lista está ordenada ascendentemente.
- Implementar una función que devuelva los primeros n elementos de una lista.
- Escribir una función que elimine elementos duplicados consecutivos.
- Implementar la función `takeWhile` de forma recursiva.
- Crear una función que aplique una función a cada elemento de una lista (como `map`).

Depuración de Funciones Recursivas

- **Usar `Debug.Trace`:**

```haskell
import Debug.Trace

sumaDebug :: [Int] → Int
sumaDebug [] = trace "Caso base: []" 0
sumaDebug (x:xs) = trace ("Procesando: " ++ show x) (x +
   sumaDebug xs)
```

- **Verificar siempre el caso base**
- **Probar con casos límite:**
  - Lista vacía `[]`
  - Lista con un elemento `[x]`
  - Lista con dos elementos `[x,y]`
- **Usar tipos de datos apropiados** (`Maybe` para posibles fallos)

Soluciones Sugeridas para Ejercicios Propuestos

Eliminar ocurrencias de un elemento

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

-- Prueba: eliminar 3 [1,3,2,3,4] => [1,2,4]
```

Contar números pares

```
contarPares :: [Int] → Int
contarPares [] = 0
contarPares (x:xs)
  | even x    = 1 + contarPares xs
  | otherwise = contarPares xs

-- Prueba: contarPares [1,2,3,4,5,6] => 3
```

Producto de elementos

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

-- Prueba: producto [1,2,3,4] => 24
```

Mínimo de una lista

```
minimo :: Ord a => [a] → Maybe a
minimo [] = Nothing
minimo [x] = Just x
minimo (x:xs) = case minimo xs of
                  Just y  → Just (min x y)
                  Nothing → Just x

-- Prueba: minimo [3,1,4,2] => Just 1
```

Implementación de map

```haskell
miMap :: (a → b) → [a] → [b]
miMap _ []     = []
miMap f (x:xs) = f x : miMap f xs

-- Prueba: miMap (*2) [1,2,3] => [2,4,6]
```

Formato de Entrega

- Subir archivo `Sesion13.hs` con todas las funciones implementadas
- Incluir capturas de pantalla con pruebas en GHCi
- Comentar el código explicando cada caso (base y recursivo)
- Incluir análisis de complejidad para cada función

Conclusiones

- La recursividad es la herramienta fundamental para manipular listas en Haskell
- Permite definir funciones expresivas y declarativas sin efectos secundarios
- Refuerza la abstracción y prepara para estructuras de datos más complejas
- El análisis de complejidad ayuda a optimizar el código
- La práctica con casos límite mejora la robustez de las implementaciones

Próxima Sesión

- **Tema:** Evaluación perezosa y listas infinitas
- **Preparación:** Investigar sobre `map`, `filter` y `fold`
- **Ejercicio avanzado:** Implementar `takeWhile` de forma recursiva