Curso: Programación Lógica y Funcional

Unidad 2: Programación funcional

Sesión 11: Conceptos de listas y functores.

Docente: Carlos R. P. Tovar

UTP Universidad Tecnológica del Perú

INICIO
Objetivo de la sesión

Al finalizar la sesión, el estudiante será capaz de:

- Comprender la estructura y manipulación de listas en Haskell
- Dominar las funciones de orden superior para procesamiento de listas
- Introducir el concepto de functor y su importancia
- Aplicar functores para transformar datos en contextos
- Implementar operaciones complejas usando composición de functores

Dudas de la anterior sesión

UTILIDAD

¿Por qué aprender listas?

- Son la estructura de datos más usada en Haskell.
- Representan secuencias de valores.
- Base para operaciones funcionales (map, filter, fold).

¿Por qué aprender functores?

- Generalizan la aplicación de funciones sobre estructuras.
- Permiten escribir código más flexible y reutilizable.
- Están en la base de muchos conceptos en Haskell (Maybe, listas, árboles).

TRANSFORMACIÓN
Introducción a las Listas en Haskell

Definición: Estructuras homogéneas de datos

```
-- Listas básicas
numeros = [1, 2, 3, 4, 5]
letras = ['a', 'b', 'c']
vacia = []

-- Notación de rangos
del1al10 = [1..10]
pares = [2, 4..20]
```

Características:

- Homogéneas: todos los elementos del mismo tipo
- Inmutables: no se pueden modificar, solo transformar
- Recursivas: estructura cons (a:as)
- Perezosas: evaluación bajo demanda

Estructura Fundamental de Listas

Pattern matching con listas:

```
-- Descomposición
head (x:xs) = x
tail (x:xs) = xs

-- Funciones básicas
longitud [] = 0
longitud (x:xs) = 1 + longitud xs

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

Operador cons (:):

```
1 : [2, 3, 4]  -- [1, 2, 3, 4]
'a' : "bc"     -- "abc"
```

Funciones de Orden Superior para Listas

Map - Transformación element-wise:

```
map :: (a -> b) -> [a] -> [b]
map (+1) [1, 2, 3]  -- [2, 3, 4]
map toUpper "hola"  -- "HOLA"
```

Filter - Selección por condición:

```
filter :: (a -> Bool) -> [a] -> [a]
filter even [1, 2, 3, 4]  -- [2, 4]
filter (>0) [-2, -1, 0, 1, 2]  -- [1, 2]
```

Reducción de Listas

Foldr (derecha a izquierda):

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (+) 0 [1, 2, 3]  -- 1 + (2 + (3 + 0)) = 6
```

Foldl (izquierda a derecha):

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (+) 0 [1, 2, 3]  -- ((0 + 1) + 2) + 3 = 6
```

Aplicaciones prácticas:

```
-- Concatenar lista de strings
foldr (++) "" ["Hola", " ", "Mundo"]
-- "Hola Mundo"

-- Encontrar máximo
foldr1 max [3, 1, 4, 2]  -- 4
```

List Comprehensions

Sintaxis similar a notación matemática:

```
-- Cuadrados de números pares
[x^2 | x <- [1..10], even x]  -- [4, 16, 36, 64, 100]

-- Producto cartesiano
[(x, y) | x <- [1,2], y <- [3,4]]  -- [(1,3),(1,4),(2,3),(2,4)]

-- Con condiciones múltiples
[x | x <- [1..20], x > 10, x < 15, odd x]  -- [11, 13]
```

Introducción a Functores

Concepto: Tipos que pueden ser mapeados

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

Analogía: Caja que contiene valores

- Preserva la estructura
- Aplica función a valores internos
- Mantiene el contexto

Functor para Listas

Listas como functores:

```
instance Functor [] where
   fmap = map

-- Ejemplos
fmap (+1) [1, 2, 3]  -- [2, 3, 4]
fmap length ["Hola", "Mundo"]  --
[4, 5]
```

Composición de operaciones:

```
fmap (fmap (+1)) [[1,2], [3,4]]  --
[[2,3], [4,5]]
```

Functor para Maybe

Manejo elegante de valores opcionales:

```
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just x) = Just (f x)

-- Ejemplos
fmap (+1) (Just 5)  -- Just 6
fmap (+1) Nothing   -- Nothing
```

Aplicación práctica:

```
-- Procesamiento seguro de datos
fmap toUpper (Just "hola")  -- Just "HOLA"
fmap readMaybe (Just "123") -- Just 123
```

Functor para Either

Manejo de errores con contexto:

```
instance Functor (Either e) where
   fmap _ (Left e) = Left e
   fmap f (Right x) = Right (f x)

-- Ejemplos
fmap (+1) (Right 5)  -- Right 6
fmap (+1) (Left "error")  -- Left "error"
```

Leyes de los Functores

1. **Identidad:**

fmap id = id

2. Composición:

fmap (f . g) = fmap f . fmap g

Importancia:

- Comportamiento predecible
- Compatibilidad con otras abstracciones
- Garantías matemáticas

Aplicaciones Prácticas de Functores

Procesamiento de datos con contexto:

```
-- Transformación segura con Maybe
procesarUsuario :: Maybe Usuario -> Maybe Usuario
procesarUsuario = fmap (\u -> u { edad = edad u + 1 })

-- Procesamiento de listas anidadas
transponer :: [[a]] -> [[a]]
transponer = fmap fmap ...  -- Usando composición de functores
```

Functores Customizados

Creación de tipos functor personalizados:

```
data Arbol a = Vacio | Nodo a (Arbol a) (Arbol a)

instance Functor Arbol where
   fmap _ Vacio = Vacio
   fmap f (Nodo x izq der) = Nodo (f x) (fmap f izq) (fmap f der)

-- Ejemplo
fmap (*2) (Nodo 1 (Nodo 2 Vacio Vacio) (Nodo 3 Vacio Vacio))
-- Nodo 2 (Nodo 4 Vacio Vacio) (Nodo 6 Vacio Vacio)
```

Composición de Functores

Functores anidados:

```
-- Lista de Maybes
fmap (fmap (+1)) [Just 1, Nothing, Just 3]  -- [Just 2, Nothing, Just 4]

-- Maybe de lista
fmap (map (+1)) (Just [1,2,3])  -- Just [2,3,4]
```

Operador <$> (sinónimo de fmap):

```
(+1) <$> [1,2,3]  -- [2,3,4]
toUpper <$> Just "hola"  -- Just "HOLA"
```

PRACTICA
Ejercicio 1 - Procesamiento de Datos

Procesar una lista de transacciones:

```
data Transaccion = Transaccion {
    monto :: Float,
    tipo :: String,
    exitosa :: Bool
}
```

```
-- Calcular montos exitosos
montosExitosos :: [Transaccion] -> [Float]
montosExitosos = map monto . filter exitosa


-- Convertir a dólares
aDolares :: [Transaccion] -> [Float]
aDolares = fmap ((* 3.7) . monto) . filter exitosa
```

Ejercicio 2 - Validación con Functores

Sistema de validación escalable:

```
validarEdad :: Int -> Maybe Int
validarEdad x
    | x < 0 = Nothing
    | x > 120 = Nothing
    | otherwise = Just x
```

```
validarEmail :: String -> Maybe String
validarEmail x
    | '@' `elem` x = Just x
    | otherwise = Nothing


-- Combinar validaciones
validarUsuario :: Int -> String -> Maybe (Int, String)
validarUsuario edad email = (,) <$> validarEdad edad <*> validarEmail email
```

Ejercicio 3 - Transformaciones Anidadas

Procesamiento de datos complejos:

```
type Error = String
type Resultado a = Either Error a

procesarArchivos :: [FilePath] -> [Resultado String]
procesarArchivos = fmap leerArchivo
```

```
leerArchivo :: FilePath -> Resultado String
leerArchivo path
  | ".txt" `isSuffixOf` path = Right "Contenido del archivo"
  | otherwise = Left "Formato no soportado"


-- Filtrar y transformar resultados exitosos
procesarExitosos :: [Resultado String] -> [String]
procesarExitosos = fmap (fmap (map toUpper)) . filter isRight
```

Patrones Avanzados

Functor aplicativo (introducción):

```
-- Secuenciación de operaciones
data Usuario = Usuario {
   nombre :: String,
   edad :: Int
} deriving Show

crearUsuario :: Maybe String -> Maybe Int -> Maybe Usuario
crearUsuario nombre edad = Usuario <$> nombre <*> edad

-- Ejemplo
crearUsuario (Just "Ana") (Just 25)  -- Just (Usuario "Ana" 25)
```

Ejercicio final:

Implementar un procesador de datos que use listas y functores para:

1. Leer múltiples fuentes de datos
2. Aplicar transformaciones consistentes
3. Manejar errores elegantemente
4. Producir resultados estructurados

CIERRE
Conclusiones

- Las listas y functores son pilares fundamentales de la programación funcional en Haskell.

- Las listas proporcionan una forma poderosa de manipular colecciones de datos, mientras que los functores ofrecen una abstracción para transformar valores en contextos.

- La combinación de estos conceptos permite crear código expresivo, seguro y mantenible.

Universidad
Tecnológica
del Perú