Curso: Programación Lógica y Funcional

Unidad 1: Programación funcional

Sesión 8: Formas diversas de implementar funciones recursivas a nivel de la programación lógica. Ejercicios ilustrativos.

Docente: Carlos R. P. Tovar

Dudas de la anterior sesión

INICIO
Objetivo de la sesión

- Explorar diferentes patrones de recursión en programación funcional
- Implementar funciones recursivas con diversos enfoques y técnicas
- Analizar ventajas y desventajas de cada aproximación recursiva
- Resolver problemas mediante múltiples estrategias recursivas
- Desarrollar habilidades para elegir el enfoque recursivo adecuado

UTILIDAD
¿Por qué Múltiples Enfoques?

Diferentes problemas requieren diferentes estrategias:

- Recursión simple → Claridad conceptual
- Tail recursion → Eficiencia en memoria
- Recursión mutua → Problemas interconectados
- Recursión con acumuladores → Procesamiento incremental
- Recursión sobre múltiples parámetros → Problemas complejos

TRANSFORMACIÓN
Recursión Simple vs. Tail Recursion

Recursión Simple:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Ventaja: Más intuitiva y fácil de entender

Tail Recursion:

```
factorial n = factAux n 1
  where
    factAux 0 acc = acc
    factAux n acc = factAux (n-1) (n * acc)
```

Ventaja: Más eficiente, optimizable

Recursión Mutua

Problemas interconectados:

par 0 = True

par n = impar (n-1)

impar 0 = False

impar n = par (n-1)

Aplicación:

- Análisis sintáctico
- Procesamiento de lenguajes
- Validaciones cruzadas

Recursión con Acumuladores

Procesamiento incremental:

```
sumaLista :: [Int] -> Int
sumaLista lista = sumaAux lista
0
  where
    sumaAux [] acc = acc
    sumaAux (x:xs) acc = sumaAux
xs (x + acc)
```

Ventajas:

- Evita stack overflow
- Más eficiente en memoria
- Ideal para listas grandes

Recursión sobre Múltiples Parámetros

Problemas complejos:

```
combinaciones :: Int -> [a] ->
[[a]]
combinaciones 0 _ = [[]]
combinaciones n lista =
    [x:xs | x:rest <- tails lista, xs <-
combinaciones (n-1) rest]
```

Aplicaciones:

- Generación de subconjuntos
- Problemas combinatorios
- Búsqueda exhaustiva

Recursión con Continuaciones

Style avanzado:

```
factorialCont :: Int -> (Int -> r) -> r
factorialCont 0 cont = cont 1
factorialCont n cont =
factorialCont (n-1) (\result ->
cont (n * result))
```

Usos:

- Manejo de control de flujo
- Implementación de mónadas
- Transformación de programas

Ejercicio 1 - Múltiples Enfoques para Fibonacci

Implementa Fibonacci usando:

- Recursión simple
- Tail recursion
- Recursión con tuplas
- Usando iterate y lazy evaluation

Compara:

- Tiempo de ejecución
- Uso de memoria
- Claridad del código

Ejercicio 2 - Procesamiento de Árboles

Diferentes enfoques para recorrer árboles binarios:

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

-- Recorrido preorden
preorden :: Arbol a -> [a]
preorden Vacio = []
preorden (Nodo valor izq der) =
   [valor] ++ preorden izq ++ preorden
der
```

```
--Recorrido con acumulador

preordenTail :: Arbol a -> [a]
preordenTail arbol = preordenAux
arbol []
 where
   preordenAux Vacio acc = acc
   preordenAux (Nodo v i d) acc =
      v : preordenAux i (preordenAux d
acc)
```

Ejercicio 3 - Transformación de Recursión

Convierte recursión simple a tail recursion:

```
--Versión simple

revertir :: [a] -> [a]
revertir [] = []
revertir (x:xs) = revertir xs ++ [x]
```

```
-- Versión tail recursion
revertirTail :: [a] -> [a]
revertirTail lista = revertirAux lista []
  where
    revertirAux [] acc = acc
    revertirAux (x:xs) acc = revertirAux xs (x:acc)
```

Ejercicio 4 - Recursión Mutual en Validación

Sistema de validación recursivo mutuo:

```
validarExpresion :: String -> Bool
validarExpresion = validarParentesis

validarParentesis :: String -> Bool
validarParentesis [] = True
validarParentesis (c:cs)
  | c == '(' = validarContenido cs
  | otherwise = False

validarContenido :: String -> Bool
validarContenido [] = False
validarContenido (c:cs)
  | c == ')' = validarParentesis cs
  | otherwise = validarContenido cs
```

Patrones de Recursión Comunes

Catálogo de patrones:

- **Recursión estructural**: Sigue la estructura de datos
- **Recursión generativa**: Crea nuevas estructuras
- **Recursión acumulativa**: Acumula resultados
- **Recursión mutual**: Funciones que se llaman entre sí
- **Recursión con continuaciones**: Manejo explícito de control

Análisis de Eficiencia

Comparativa de aproximaciones:

- Complejidad temporal
- Complejidad espacial
- Uso de stack
- Posibilidad de optimización
- Legibilidad y mantenibilidad

PRACTICA
Ejercicio Integrador

Sistema de procesamiento de expresiones:
Implementa un evaluador de expresiones matemáticas usando:

- Recursión simple para parsing
- Tail recursion para evaluación
- Recursión mutual para validación
- Acumuladores para cálculo incremental

CIERRE
Mejores Prácticas

Elección del enfoque recursivo:

- Usa recursión simple para claridad en problemas pequeños
- Prefiere tail recursion para eficiencia en problemas grandes
- Considera recursión mutual para problemas interconectados
- Emplea acumuladores para procesamiento incremental
- Evalúa continuaciones para control de flujo complejo

Preguntas de Discusión

- ¿Qué factores consideras al elegir un enfoque recursivo?
- ¿Cómo decides entre claridad y eficiencia?
- ¿Qué patrones recursivos has encontrado más útiles?
- ¿Cómo manejas el trade-off entre diferentes aproximaciones?

Recursos Adicionales

- "Learn You a Haskell" - Capítulo sobre recursión
- "Purely Functional Data Structures"
- Artículos sobre optimización de recursión
- Casos de estudio de aplicaciones reales

Conclusiones

La diversidad de enfoques recursivos nos permite elegir la herramienta adecuada para cada problema, balanceando claridad, eficiencia y expresividad.

¡La recursión es como una caja de herramientas: cada técnica tiene su propósito ideal!

Universidad
Tecnológica
del Perú