Curso: Programación Lógica y Funcional

Unidad 2: Programación funcional

Sesión 9: Funciones recursivas de tipo menú. De tipo opciones. Intervalos. Operadores.

Docente: Carlos R. P. Tovar

Dudas de la anterior sesión

INICIO
Objetivo de la sesión

- Implementar funciones recursivas que simulen menús interactivos
- Utilizar recursión para manejar opciones y selecciones múltiples
- Trabajar con intervalos y rangos en contextos recursivos
- Aplicar operadores personalizados en funciones recursivas
- Desarrollar programas interactivos con control de flujo recursivo

TRANSFORMACIÓN
Introducción a Funciones Recursivas de Tipo Menú

- **Características principales:**
- Mantiene un ciclo de interacción hasta condición de salida
- Maneja múltiples opciones y caminos de ejecución
- Perfecto para sistemas interactivos y interfaces de usuario
- Combina IO y recursión en Haskell

- **Ventajas:**
- Código claro y mantenible
- Fácil extensión con nuevas opciones
- Control preciso del flujo del programa

Concepto clave: Patrón de diseño donde una función se llama a sí misma para mantener un estado interactivo

Estructura General de un Menú Recursivo

```
menuPrincipal :: IO ()
menuPrincipal = do
  -- Mostrar opciones
  putStrLn "=== MENÚ PRINCIPAL ==="
  putStrLn "1. Opción A"
  putStrLn "2. Opción B"
  putStrLn "3. Salir"

  -- Leer selección
  putStr "Seleccione una opción: "
  opcion <- getLine

  -- Procesar opción (recursión)
  case opcion of
    "1" -> do accionA; menuPrincipal
    "2" -> do accionB; menuPrincipal
    "3" -> putStrLn "¡Hasta pronto!"
    _   -> do putStrLn "Opción inválida";
menuPrincipal
```

Ejemplo Completo: Menú de Operaciones Matemáticas

```
menuMatematicas :: IO ()
menuMatematicas = do
  putStrLn "\n=== CALCULADORA RECURSIVA ==="
  putStrLn "1. Factorial"
  putStrLn "2. Fibonacci"
  putStrLn "3. Sumatoria"
  putStrLn "4. Salir"

  putStr "Opción: "
  opcion <- getLine
```

```
  case opcion of
    "1" -> do calcularFactorial; menuMatematicas
    "2" -> do calcularFibonacci; menuMatematicas
    "3" -> do calcularSumatoria; menuMatematicas
    "4" -> putStrLn "Fin del programa"
    _   -> do putStrLn "Error: opción no válida"; menuMatematicas

calcularFactorial :: IO ()
calcularFactorial = do
  putStr "Ingrese número: "
  n <- readLn
  putStrLn $ "Factorial: " ++ show (factorial n)
```

Funciones con Opciones Múltiples: Patrón para manejar configuraciones y preferencias:

```
data Configuracion = Config {
    modo :: String,
    nivel :: Int,
    activado :: Bool
}


procesarOpciones :: Configuracion -> IO Configuracion
procesarOpciones config = do
    putStrLn "Opciones disponibles:"
    putStrLn "1. Cambiar modo"
    putStrLn "2. Ajustar nivel"
    putStrLn "3. Activar/Desactivar"
    putStrLn "4. Guardar y salir"
    opcion <- getLine
    case opcion of
        "1" -> do nuevoModo <- cambiarModo; procesarOpciones config {modo = nuevoModo}
        "2" -> do nuevoNivel <- ajustarNivel; procesarOpciones config {nivel = nuevoNivel}
        "3" -> procesarOpciones config {activado = not (activado config)}
        "4" -> return config
        _   -> do putStrLn "Opción inválida"; procesarOpciones config
```

Trabajo con Intervalos y Rangos: Generación y procesamiento de intervalos con recursión

```
-- Generar intervalo [a, b]
generarIntervalo :: Int -> Int -> [Int]
generarIntervalo a b
  | a > b    = []
  | otherwise = a : generarIntervalo (a+1) b

-- Sumar valores en intervalo
sumarIntervalo :: Int -> Int -> Int
sumarIntervalo a b
  | a > b    = 0
  | otherwise = a + sumarIntervalo (a+1) b
```

```
-- Filtrar números en intervalo que cumplan condición
filtrarIntervalo :: (Int -> Bool) -> Int -> Int -> [Int]
filtrarIntervalo f a b
  | a > b    = []
  | f a      = a : filtrarIntervalo f (a+1) b
  | otherwise = filtrarIntervalo f (a+1) b
```

Operadores Personalizados en Recursión: Definición y uso de operadores personalizados

```
-- Operador para concatenación recursiva
infixr 5 +++
(+++) :: [a] -> [a] -> [a]
[] +++ ys = ys
(x:xs) +++ ys = x : (xs +++ ys)

-- Operador para composición recursiva
infixr 9 .*
(.*) :: (b -> c) -> (a -> b) -> a -> c
(f .* g) x = f (g x)
```

```
-- Operador para aplicación múltiple
infixl 8 >*>
(>*>) :: a -> (a -> a) -> Int -> a
x >*> _ 0 = x
x >*> f n = f x >*> f (n-1)

-- Ejemplo de uso:
-- doble x = 2 * x
-- 5 >*> doble 3 → 40 (5 * 2 * 2 * 2)
```

Ejemplo Integrado: Sistema de Configuración

```
data Opcion = Opcion {
  nombre :: String,
  valor :: Int,
  rangoMin :: Int,
  rangoMax :: Int
}

ajustarOpcion :: Opcion -> IO Opcion
ajustarOpcion opcion = do
  putStrLn $ "Opción: " ++ nombre opcion
  putStrLn $ "Valor actual: " ++ show (valor opcion)
  putStrLn $ "Rango permitido: [" ++ show
(rangoMin opcion) ++
        ", " ++ show (rangoMax opcion) ++ "]"
  putStr "Nuevo valor (o 'x' para cancelar): "
  input <- getLine

  case input of
    "x" -> return opcion
    _   -> case reads input of
      [(n, "")] -> if n >= rangoMin opcion && n <=
rangoMax opcion
                   then return opcion {valor = n}
                   else do putStrLn "Valor fuera de rango";
ajustarOpcion opcion
      _ -> do putStrLn "Entrada inválida";
ajustarOpcion opcion
```

Patrones de Diseño para Menús Recursivos

- **Mejores prácticas y patrones comunes:**
- **Separación de concerns:**
  - Lógica de presentación
  - Lógica de negocio
  - Lógica de control

Patrones de Diseño para Menús Recursivos

Manejo elegante de errores:

```
leerEntero :: String -> IO (Maybe Int)
leerEntero prompt = do
    putStr prompt
    input <- getLine
    case reads input of
      [(n, "")] -> return (Just n)
      _         -> return Nothing
```

Recursión con estado:

```
menuConEstado :: Estado -> IO ()
menuConEstado estado = do
    -- mostrar opciones basadas en estado
    -- procesar entrada
    -- llamar recursivamente con nuevo estado
```

Ejercicio 2: Generador de Secuencias

Crear funciones que generen secuencias usando:

- Intervalos personalizados
- Operadores recursivos
- Opciones de configuración

```
-- Ejemplo: generar secuencia aritmética
secuenciaAritmetica :: Int -> Int -> Int -> [Int]
secuenciaAritmetica inicio paso n
    | n <= 0    = []
    | otherwise = inicio : secuenciaAritmetica (inicio + paso) paso (n-1)
```

Ejercicio 3: Sistema de Opciones Anidadas

Implementar menús recursivos anidados:

- Menú principal → Submenús → Acciones
- Cada nivel maneja sus propias opciones
- Posibilidad de volver atrás

```
menuPrincipal :: IO ()
menuPrincipal = do
    -- opciones principales
    -- al seleccionar una, llama a submenu

subMenuConfig :: IO ()
subMenuConfig = do
    -- opciones de configuración
    -- opción para volver al menú principal
```

Manejo de Errores en Patrones Recursivos: Técnicas para manejo robusto de errores

```
menuSeguro :: IO ()
menuSeguro = menuSeguroAux 3  -- 3 intentos máximos

menuSeguroAux :: Int -> IO ()
menuSeguroAux intentos
  | intentos <= 0 = putStrLn "Demasiados intentos fallidos"
  | otherwise = do
    putStrLn "Menú seguro - Intentos restantes: " ++ show intentos
    opcion <- getLine
    case opcion of
      "1" -> do accionValida; menuSeguroAux 3  -- resetear intentos
      "2" -> do accionValida; menuSeguroAux 3
      _   -> do putStrLn "Opción inválida"; menuSeguroAux (intentos-1)
```

Técnicas para manejo robusto de errores:

Técnicas para mejorar el rendimiento:

- Tail recursion para evitar stack overflow
- Memoization de resultados costosos
- Evaluación perezosa para cálculos grandes

```
-- Tail recursion para menús
menuTailRecursivo :: IO ()
menuTailRecursivo = menuLoop
 where
  menuLoop = do
     -- mostrar opciones
     opcion <- getLine
     case opcion of
        "salir" -> return ()
        _       -> do procesarOpcion
opcion; menuLoop
```

PRACTICA
Ejercicio 1: Menú de Conversión de Unidades

Implementar un menú recursivo que permita:

- Convertir entre Celsius y Fahrenheit
- Convertir entre kilómetros y millas
- Convertir entre kilogramos y libras
- Salir del programa

Requisitos:

- Usar recursión para mantener el menú activo
- Validar entradas del usuario
- Mostrar resultados formateados

CIERRE
Conclusiones

- ¿Qué desafíos encontraron al implementar menús recursivos?
- ¿Cómo manejan el estado en programas interactivos?
- ¿Qué ventajas ven en este enfoque versus loops imperativos?
- ¿Cómo probarían funciones recursivas con IO?

Cierre

La recursión no es solo para cálculos matemáticos; es una herramienta poderosa para crear programas interactivos y sistemas complejos de menús.

La combinación de pattern matching, IO y recursión en Haskell permite crear código claro y mantenible para interfaces de usuario y sistemas interactivos.

Universidad
Tecnológica
del Perú