Tecnológico de Costa Rica

IC4700 - Lenguajes de Programación

Proyecto 1: Paradigma Imperativo

Integrantes:

Kevin Núñez Camacho

Yeri Porras Víquez

Roy Silva

Ejercicio 1: Desarrolle una investigación bibliográfica acerca de los siguientes temas, en el lenguaje de programación Haskell:

Árboles: Cómo se implementan

En Haskell, los árboles se definen mediante tipos de datos algebraicos, permitiendo estructuras recursivas, donde cada nodo contiene un valor y la referencia a sus subárboles hijos. Un ejemplo básico:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Explicación de los comandos:

- data: se usa para defionir nuevos tipos de datos personalizados.
- Node: almacena un valor de tipo a y sus dos suibarboles (izq. y der.).
- Empty: representa un nodo vacío, es el equivalente a null en Haskell.
- foldr: aplica insert a cada elemento (función de más adelante).

Ejemplo de un Arbol:

```
arbolEjemplo = Node 5
                    (Node 3
                      (Node 1 Empty Empty)
                      (Node 4 Empty Empty)
                    )
                    (Node 8
                      (Node 6 Empty Empty)
                      Empty
                    )
```

Visualmente se veria algo como:

```
   5
          /    \
        3       8
       / \     /
      1   4   6
```

Para definir una función de insertar en arboles:

```
insert :: Ord a => a -> Tree a -> Tree a
    insert x Empty = Node x Empty Empty  -- Caso base: insertar nodo vacío
    insert x (Node y left right)
     | x < y     = Node y (insert x left) right  -- Insertar a la izquierda
     | otherwise = Node y left (insert x right)  -- Insertar a la derecha
```

Como implementar la función:

```
arbolVacio = Empty
    arbol = foldr insert arbolVacio [5, 3, 8, 1, 4, 6]
```

Maps: Cómo se implementan

Los Map (diccionarios) son estructuras que asocian claves con valores. La implementación más común está en el módulo Data.Map, que utiliza árboles binarios balanceados para garantizar eficiencia. Ejemplo:

```
import qualified Data.Map as Map
    -- Crear un Map vacío, insertar y buscar
    miMap :: Map.Map String Int
    miMap = Map.insert "clave1" 100 (Map.empty)
```

Se basa en árboles balanceados por tamaño (size-balanced trees), donde cada nodo almacena: una clave (key), un valor (value), subárboles izquierdo y derecho, y el tamaño del subárbol.

Definición simplificada:

```
data Map k a = Tip | Bin { size :: Int, key :: k, value :: a, left :: Map k a, right :: Map k a }
```

El map es eficiente ya que opera con árboles de balanceo teniendo un tiempo de O(logn) en operaciones, hacer recorridos ordenados.

Operaciones Principales:

- Map.empty: map vacío.
- Map.insert: insertar clave-valor.
- Map.lookup: busca un valor por clave.
- Map.delete: recibe la clave a eliminar, eliminando la clave-valor.
- Map.fromlist: crear un map con una lista de tuplas, como [("a", 1), ("b", 2)].

Funciones Anónimas: Cómo se implementan.

Son funciones sin nombre, definidas en el momento/bloque para usarlas una vez (ej: con map, filter, etc.). Se definen con la sintaxis **\x -> ...**

Por ejemplo:

```
\lambdaArgumentos -> cuerpo – Formula

\x -> x + 1

-- Duplicar elementos de una lista

listaDuplicada = map (\x -> x * 2) [1, 2, 3] -- [2, 4, 6]
```

```
-- Filtrar números pares

pares = filter (\x -> x `mod` 2 == 0) [1, 2, 3, 4] -- [2, 4]
```

Las lambdas son azúcar sintáctico para funciones anónimas. En Haskell, se traducen directamente a expresiones de función durante la compilación. Por ejemplo, esta función de suma:

```
(\x y -> x + y) 3 5
```

Equivale a:

```
let f x y = x + y in f 3 5
```

Reducciones (Folds): Definición y usos.

Los folds son funciones de orden superior que "colapsan" una lista en un solo valor aplicando una operación binaria recursivamente. También, son un ejemplo común para el uso de funciones anónimas.

Tipos de Folds

Función	Dirección	Evaluación	Uso Recomendado
foldr	Derecha -> izquierda	Útil en listas infinitas	Operaciones que preserven el orden
foldl	Izquierda -> derecha	Evalua toda la lista	Operaciones acumulativas
fold'	Izquierda -> derecha	Evita memory leaks	Evaluaciones muy grandes

foldr: fold right

foldl: fold left

foldl': fold left estricto

Tambien existe foldMap, que combina fold y map para tipos monoid, pero es un poco más compleja.

1. Uso para concatenar strings:

```
palabras = ["Hola", " ", "Mundo"]

frase = foldr (++) "" palabras
```

   Salida Esperada: "Hola" ++ (" " ++ ("Mundo" ++ "")) → "Hola Mundo"

2. Uso para funciones max:

```
maxFoldr :: Ord a => [a] -> a
    maxFoldr = foldr (\x acc -> if x > acc then x else acc) minBound
    maximo = maxFoldr [3, 1, 4, 2]
```

   Salida esperada: 4

3. Uso para funciones min:

```
minFoldl :: Ord a => [a] -> a
    minFoldl = foldl (\acc x -> if x < acc then x else acc) maxBound
    minimo = minFoldl [3, 1, 4, 2]
```

   Salida esperada: 1

4. Foldl y Foldr:
   - Resta: (la suma es igual, pero en vez de – se cambia por un solo +)

```
    restaFoldr = foldr (-) 0 [5, 3, 1]
```

     Salida Foldr: 5 - (3 - (1 - 0)) = 3

restaFoldl = foldl (-) 0 [5, 3, 1]

Salida Foldl: ((0 - 5) - 3) - 1 = -9

- Multiplicación:

  productoFoldr = foldr (*) 1 [2, 3, 4]

  Salida Foldr: 2 * (3 * (4 * 1)) = 24

  productoFoldl = foldl (*) 1 [2, 3, 4]

  Salida Foldl: ((1 * 2) * 3) * 4 = 24

- Strings:

  palabras = ["Hola", " ", "Mundo"]

  frase = foldr (++) "" palabras

  Salida Foldr: "Hola" ++ (" " ++ ("Mundo" ++ "")) → "Hola Mundo"

  fraseFoldl = foldl (++) "" palabras

  Salida Foldl: (("" ++ "Hola") ++ " ") ++ "Mundo" → "Hola Mundo"

Filtros:

Elimina elementos de la lista mientras se cumple un predicado, y retorna el resto de la lista a partir del primer elemento que no lo cumple. Sintax:

filter :: (a -> Bool) -> [a] -> [a]

Ejemplo:

numeros = [1, 2, 3, 4, 5]

pares = filter (\x -> x `mod` 2 == 0) numeros

Salida: [2, 4]

- Any: verifica si al menos un elemento de la lista cumple un predicado.

  lista1 = [1, 3, 5, 7]

  resultado1 = any even lista1

  Salida: False

- takeWhile: toma elementos de la lista mientras se cumple un predicado, y se detiene en el primer elemento que no lo cumple.

  numeros = [1, 2, 3, 4, 1, 2]

  resultado = takeWhile (< 4) numeros

  Salida: [1, 2, 3]

- DropWhile: elimina elementos de la lista mientras se cumple un predicado, y retorna el resto de la lista a partir del primer elemento que no lo cumple.
  numeros = [1, 2, 3, 4, 1, 2]
  resultado = dropWhile (< 3) numeros
  Salida: [3, 4, 1, 2]

Funciones dentro de estructuras de datos:

En Haskell, las funciones son valores de primera clase, o sea, pueden ser entre muchas cosas, almacenadas en estructuras de datos (listas, tuplas, tipos personalizados), que es lo que nos interesa estudiar. Hay 3 formas principales de implementar funciones dentro de estructuras:

1. Listas:
   Las listas pueden contener funciones del mismo tipo. Esto es útil para aplicar múltiples operaciones a un valor.
   - Listas de Funciones Aritméticas:

```
operaciones :: [Int -> Int]
operaciones = [ (+1),   -- Sumar 1
                (*2),   -- Multiplicar por 2
                (\x -> x^2)  -- Elevar al cuadrado (función anónima)
              ]
```

   Aplicamos cada función al número 3

```
resultados :: [Int]
resultados = map (\funcion -> funcion 3) operaciones
```

   Salida: [3+1, 3*2, 3^2] = [4, 6, 9]
   - Listas de Predicados:

```
condiciones :: [Int -> Bool]
condiciones = [ even,        -- ¿Es par?
                (>5),        -- ¿Mayor que 5?
                (\x -> x `mod` 3 == 0)  -- ¿Divisible por 3?
              ]
```

   Verificamos si el número 6 cumple todas las condiciones:

```
checks :: [Bool]
checks = map (\cond -> cond 6) condiciones
```

Salida: (¿6 es par?, ¿6 > 5?, ¿6 es divisible por 3?) = [True, True, True]

2. Tuplas:
   Las tuplas pueden combinar funciones con otros valores.

```
operacionYValor :: (Int -> Int, Int)
operacionYValor = (\x -> x * 3, 4)
```

   La tupla contiene una funcion anonima de multiplicar x*3, y al valor 4\

   resultado = fst operacionYValor (snd operacionYValor)

   Salida: 12

3. Datos Personalizados:
   Podemos definir estructuras que almacenen funciones como campos.
   Dato personalizado que almacena un nombre y una función:

```
data Operacion a = Op {
  nombre :: String,
  funcion :: a -> a
}

duplicar :: Operacion Int
duplicar = Op "Duplicar" (*2)
```

   Aplicar la función de una Operacion:

```
aplicarOperacion :: Operacion a -> a -> a

aplicarOperacion (Op _ f) x = f x


resultado = aplicarOperacion duplicar 5
```

   Salida: 10
   Nota: En este ejemplo, el nombre no tiene un uso practico, pero permite guardar nombres de funciones para desplegar en un menú posterior, y si se deseara cambiar el nombre de la función en todos los lados que aparece sería más fácil modificarlo desde esta estructura y que se aplique en todo el programa.

- Evaluación Perezosa (Lazy Evaluation).
  - Los argumentos son evaluados cuando son necesitados. Explique cómo funciona.
    Haskell utiliza una estrategia de evaluación llamada evaluación perezosa o lazy evaluation. Esto significa que una expresión no se evalúa en el momento en que se define, sino cuando su resultado es necesario para continuar con la ejecución del programa.
    Internamente, Haskell utiliza una estructura llamada thunk, que representa una expresión diferida. Utilizando la memoization que es evaluar una sola vez y guardarlo por si se necesitara reutilizar.

    ```
    infinitosNumeros = [1..]
    primerosCinco = take 5 infinitosNumeros
    ```

- Clases de tipos y métodos de clases: El manejo de clases de tipos es una característica de Haskell que lo distingue de otros lenguajes funcionales. A pesar de que la terminología de clases de tipos se parece a la de la programación OO, hay algunas diferencias. Explique lo siguiente:
  - Ejemplos de clases predefinidas.
    - Eq: para tipos que pueden compararse por igualdad (==, /=).
    - Ord: para tipos que pueden ordenarse (<, >, <=, >=).
    - Show: para convertir valores en cadenas (show).
    - Read: para analizar cadenas y obtener valores (read).
    - Enum: para tipos secuenciales (succ, pred).
    - Bounded: para tipos con un límite mínimo y máximo (minBound, maxBound).
  - Sobrecarga (polimorfismo ad-hoc).

    Esto permite definir funciones que pueden operar sobre múltiples tipos, siempre y cuando esos tipos sean instancias de una clase determinada.

  - Explique cómo funcionan las jerarquías de clases de tipos.
    Esto funciona de la siguiente forma: una clase puede requerir que los tipos que la implementen también implementen otra

clase, por ejemplo, la clase Ord depende de Eq porque hay que comparar los elementos para ordenarlos.

- Para qué sirve Deriving y cuáles clases de Standard Prelude pueden utilizarla.
  En haskell la palabra clave deriving permite generar instancias automáticas para ciertas clases y de esta forma ahorrar trabajo al usuario, las que pueden hacer deriving son Eq, Ord, Show, Read, Enum, Bounded y se pueden usar de esta manera:

  ```
  data Examen = Examen String Int deriving (Eq, Show, Ord)
  ```

Fuentes bibliográficas

- https://wiki.haskell.org/Laziness
- https://learnyouahaskell.com/types-and-typeclasses
- https://hackage.haskell.org/package/base/docs/Prelude.html
- https://www.haskell.org/onlinereport/haskell2010/haskellch6.html
- https://en.wikibooks.org/wiki/Haskell/Classes_and_types
- https://www.cse.chalmers.se/edu/year/2018/course/TDA452/labs/Evaluation/Evaluation.html
- http://book.realworldhaskell.org
- https://teaching.well-typed.com/intro/
- http://learn.hfm.io/
- https://www.haskell.org/documentation/