Tipos de Datos Algebraicos Definidos en la Clase
Primero, definimos todos los tipos de datos presentados en las diapositivas para que los siguientes bloques de código funcionen correctamente.
Haskell
-- Tipos de Datos Básicos y Recursivos (Diapositivas 4, 8, 9, 10)

[cite_start]-- Ejemplo de Registro (Producto) [cite: 152, 153, 154, 155]
data Persona = P String Int String -- Nombre, Edad, DNI
  deriving (Show)

[cite_start]-- Ejemplo Recursivo Lineal [cite: 68]
data Pizza = Prepizza | Capa Ingrediente Pizza
type Ingrediente = String

[cite_start]-- Ejemplo Recursivo de Árbol (Dungeon) [cite: 86, 87]
data Dungeon = Armario | Habitacion Objeto Dungeon Dungeon
type Objeto = String

[cite_start]-- Tipo de Árbol Binario Genérico [cite: 98, 99, 172, 186, 199]
data Tree a = EmptyT | NodeT a (Tree a) (Tree a)
  deriving (Show)

[cite_start]-- Tipos para Navegación en Árboles [cite: 189, 190]
data Opcion = Izq | Der
  deriving (Show, Eq)
type Posicion = [Opcion]

[cite_start]-- Tipos Complejos (Nave) [cite: 203, 204, 205, 206, 207]
data Componente = LanzaTorpedos | Motor Int | Almacen [Barril]
type Barril = String
data Sector = S SectorId [Componente] [Tripulante]
type SectorId = String
type Tripulante = String
data Nave = N (Tree Sector)

[cite_start]-- Tipos Complejos (Manada de Lobos) [cite: 215, 216, 217]
type Nombre = String
type Presa = String
type Territorio = String
data Lobo = Cazador Nombre [Presa] Lobo Lobo Lobo
      | Explorador Nombre [Territorio] Lobo Lobo
      | Cria Nombre
data Manada = M Lobo

Implementaciones de Funciones sobre Listas
Estas son las funciones de ejemplo que operan sobre listas.
Funciones tomar
Haskell
[cite_start]-- toma los primeros n elementos de una lista [cite: 108]

```
-- PRECOND: n >= 0
tomarHasta :: Int -> [a] -> [a]
tomarHasta 0 _     = []
tomarHasta _ []    = []
tomarHasta n (x:xs) = x : tomarHasta (n-1) xs

[cite_start]-- descarta los primeros n elementos de una lista [cite: 115]
-- PRECOND: n >= 0
tomarDesde :: Int -> [a] -> [a]
tomarDesde 0 xs    = xs
tomarDesde _ []    = []
tomarDesde n (x:xs) = tomarDesde (n-1) xs

[cite_start]-- toma un fragmento de la lista entre dos índices [cite: 119]
-- PRECOND: j >= i >= 0
tomarEntre :: Int -> Int -> [a] -> [a]
tomarEntre i j xs = tomarHasta (j-i+1) (tomarDesde i xs)
```

Funciones de Indexación y Conteo

```
Haskell
-- cuenta las apariciones de cada elemento en una lista
-- Nota: La firma en la diapositiva es ambigua. Esta implementación
[cite_start]-- coincide con el ejemplo "acbaac" -> [('a',3),('c',2),('b',1)] [cite: 127, 129]
apariciones :: Eq a => [a] -> [(a, Int)]
apariciones [] = []
apariciones (x:xs) = agregarOcurrencia x (apariciones xs)

agregarOcurrencia :: Eq a => a -> [(a, Int)] -> [(a, Int)]
agregarOcurrencia x [] = [(x, 1)]
agregarOcurrencia x ((y,n):ys)
  | x == y    = (y, n+1) : ys
  | otherwise = (y, n) : agregarOcurrencia x ys

[cite_start]-- asocia a cada elemento de la lista su índice (posición) [cite: 130]
indexar :: [a] -> [(Int, a)]
indexar xs = indexarDesde 0 xs

[cite_start]-- asocia a cada elemento su índice, comenzando desde un n dado [cite: 132]
indexarDesde :: Int -> [a] -> [(Int, a)]
indexarDesde _ [] = []
indexarDesde n (x:xs) = (n, x) : indexarDesde (n+1) xs
```

Otras Funciones de Listas

```
Haskell
-- inserta un elemento en una lista ordenada
insertar :: Ord a => a -> [a] -> [a]
insertar x [] = [x]
insertar x (y:ys)
  | x <= y    = x : y : ys
  | otherwise = y : insertar x ys

[cite_start]-- ordena una lista utilizando el método de inserción [cite: 141]
```

```
ordenar :: Ord a => [a] -> [a]
ordenar [] = []
ordenar (x:xs) = insertar x (ordenar xs)

[cite_start]-- verifica si un elemento pertenece a una lista [cite: 143]
pertenece :: Eq a => a -> [a] -> Bool
pertenece _ [] = False
pertenece e (x:xs) = e == x || pertenece e xs
```

Funciones sobre [Persona]

Haskell

```
-- constante para la mayoría de edad
mayorDeEdad :: Int
mayorDeEdad = 18

-- extrae la edad de una Persona
edad :: Persona -> Int
edad (P _ e _) = e

[cite_start]-- indica si hay alguna persona mayor de edad en la lista [cite: 157, 159]
hayMayorDeEdad :: [Persona] -> Bool
hayMayorDeEdad [] = False
hayMayorDeEdad (p:ps) = edad p >= mayorDeEdad || hayMayorDeEdad ps

[cite_start]-- indica cuántas personas hay hasta encontrar un mayor de edad [cite: 164, 165]
-- PRECOND: hay al menos un mayor de edad
cantidadHastaMayorDeEdad :: [Persona] -> Int
cantidadHastaMayorDeEdad [] = 0 -- Por completitud, aunque la precondición lo evita
cantidadHastaMayorDeEdad (p:ps)
  | edad p >= mayorDeEdad = 0
  | otherwise          = 1 + cantidadHastaMayorDeEdad ps

[cite_start]-- suma las edades de las personas entre las posiciones dadas [cite: 166, 168]
-- PRECOND: i <= j
sumaDeEdadesEntre :: Int -> Int -> [Persona] -> Int
sumaDeEdadesEntre i j personas =
  let personasEnRango = tomarEntre i j personas
  in sumarEdades personasEnRango

sumarEdades :: [Persona] -> Int
sumarEdades [] = 0
sumarEdades (p:ps) = edad p + sumarEdades ps
```

Implementaciones de Funciones sobre Árboles

Estas son las funciones de ejemplo que operan sobre el tipo Tree a.

Haskell

```
[cite_start]-- lista los nodos de un nivel específico del árbol [cite: 176, 179, 180]
levelN :: Int -> Tree a -> [a]
levelN _ EmptyT = []
levelN 0 (NodeT x _ _) = [x]
levelN n (NodeT _ t1 t2) = levelN (n-1) t1 ++ levelN (n-1) t2
```

```
[cite_start]-- lista todos los niveles del árbol, como una lista de listas [cite: 181, 182]
listPerLevel :: Tree a -> [[a]]
listPerLevel t = listPerLevelDesde 0 t

listPerLevelDesde :: Int -> Tree a -> [[a]]
listPerLevelDesde n t =
  let nivel = levelN n t
  in if null nivel
    then []
    else nivel : listPerLevelDesde (n+1) t

[cite_start]-- describe el elemento en la posición dada [cite: 192, 195]
[cite_start]-- PRECOND: la posición es válida dentro del árbol [cite: 196]
elementoEn :: Posicion -> Tree a -> a
elementoEn _ EmptyT = error "Posición inválida en árbol vacío."
elementoEn [] (NodeT x _ _) = x
elementoEn (p:ps) (NodeT _ t1 t2) =
  case p of
   Izq -> elementoEn ps t1
   Der -> elementoEn ps t2

[cite_start]-- describe las posiciones donde se encuentra un elemento dado [cite: 197, 198]
posicionesDe :: Eq a => a -> Tree a -> [Posicion]
posicionesDe _ EmptyT = []
posicionesDe e (NodeT x t1 t2) =
  let posEnSubarboles = map (Izq:) (posicionesDe e t1) ++ map (Der:) (posicionesDe e t2)
  in if e == x
    then [] : posEnSubarboles -- [] representa la raíz
    else posEnSubarboles
```