1. Tipos de Datos Pizza e Ingrediente
Estos son los tipos de datos utilizados en el primer ejemplo de recursión lineal.
Haskell
```
-- Definición del tipo Ingrediente
data Ingrediente = Salsa
                 | Queso
                 | Aceitunas Int
                 | Anchoas
                 | Anana
                 | Roquefort
                 deriving (Show)

-- Definición del tipo algebraico recursivo Pizza
data Pizza = Prepizza
           | Capa Ingrediente Pizza
           deriving (Show)
```

---

2. Funciones sobre Pizza
Implementación de la función cantQueso con su función auxiliar unoSiQueso.
Haskell
```
-- Función auxiliar para contar si un ingrediente es Queso
unoSiQueso :: Ingrediente -> Int
unoSiQueso Queso = 1
unoSiQueso _     = 0

-- Función principal para contar la cantidad de capas de queso en una pizza
cantQueso :: Pizza -> Int
cantQueso Prepizza = 0 -- Caso base
cantQueso (Capa ing p) = unoSiQueso ing + cantQueso p -- Caso recursivo
```

---

3. Tipos de Datos Dungeon y Objeto
Estos tipos de datos se usan para el ejemplo de recursión en árboles (un calabozo).
Haskell
```
-- Definición de los objetos que se pueden encontrar en el dungeon
data Objeto = Armadura
            | Escudo
            | Maza
            | Oro
            deriving (Show, Eq)

-- Definición del tipo recursivo Dungeon, que tiene una estructura de árbol binario
data Dungeon = Armario
             | Habitacion Objeto Dungeon Dungeon
             deriving (Show)
```

---

4. Funciones sobre Dungeon
Aquí se implementan las funciones para operar sobre la estructura Dungeon.
Haskell
```
-- Función auxiliar para verificar si un objeto es Oro
unoSiEsOro :: Objeto -> Int
```

```
unoSiEsOro Oro = 1
unoSiEsOro _   = 0

-- Calcula la cantidad total de Oro en un dungeon
cantidadDeOro :: Dungeon -> Int
cantidadDeOro Armario = 0 -- Caso base
cantidadDeOro (Habitacion obj d1 d2) = unoSiEsOro obj + cantidadDeOro d1 + cantidadDeOro d2 --
Caso recursivo

-- Devuelve una lista con todos los objetos del dungeon en orden
objetos :: Dungeon -> [Objeto]
objetos Armario = []
objetos (Habitacion obj d1 d2) = objetos d1 ++ [obj] ++ objetos d2

-- Calcula la profundidad del dungeon (el camino más largo)
profundidad :: Dungeon -> Int
profundidad Armario = 0
profundidad (Habitacion _ d1 d2) = 1 + max (profundidad d1) (profundidad d2)

-- Reemplaza todas las Mazas por Oro
cambiarMazasPorOro :: Dungeon -> Dungeon
cambiarMazasPorOro Armario = Armario
cambiarMazasPorOro (Habitacion obj d1 d2) =
  let nuevoObj = if obj == Maza then Oro else obj
  in Habitacion nuevoObj (cambiarMazasPorOro d1) (cambiarMazasPorOro d2)

-- Función auxiliar para elegir la lista más larga
elegirEntre :: [a] -> [a] -> [a]
elegirEntre lista1 lista2 = if length lista1 > length lista2
                    then lista1
                    else lista2

-- Devuelve los objetos que se encuentran en el camino más largo del dungeon
objsDelCaminoMasLargo :: Dungeon -> [Objeto]
objsDelCaminoMasLargo Armario = [] -- Caso base [cite: 1043, 1044]
objsDelCaminoMasLargo (Habitacion obj d1 d2) =
  obj : elegirEntre (objsDelCaminoMasLargo d1) (objsDelCaminoMasLargo d2) -- Caso recursivo [cite:
1045]
```

---

5. Tipo de Dato Tree Genérico

Esta es la definición de un árbol binario genérico, que no depende de un tipo de dato específico como Objeto.

Haskell

```
-- Un árbol binario genérico que puede contener cualquier tipo de dato 'a'
data Tree a = EmptyT
        | NodeT a (Tree a) (Tree a)
        deriving (Show)
```

---

6. Funciones sobre Tree

Ejemplo de una función que opera sobre el árbol genérico Tree.

Haskell

```
-- Convierte un árbol genérico de Objetos en una estructura de Dungeon
armarDungeon :: Tree Objeto -> Dungeon
armarDungeon EmptyT = Armario -- Caso base
armarDungeon (NodeT obj t1 t2) = Habitacion obj (armarDungeon t1) (armarDungeon t2) -- Caso recursivo
```