Instituto Tecnológico de Oaxaca

Ejercicios Haskell

Unidad 2: Modelo de Programación Funcional.

Docente: Osorio Hernández Luis Eduardo

Alumno: Rojas Pérez Alejandro

Número de control: 20161945

Carrera: ING. en Sistemas Computacionales

Materia: Programación Lógica Y Funcional

7° Semestre

Grupo:” 7SC”

21 de octubre de 2024

Contenido

Funciones Básicas

Ejercicio 1: promedio3
Esta función se utiliza para calcular el promedio de tres números. En el código, se reciben tres números como parámetros: x, y y z. La lógica es simple: se suman los tres números y luego se divide la suma entre 3 para obtener el promedio. La división se realiza entre 3 porque es la cantidad de números que estamos promediando.

Código:
```
promedio3 x y z = (x + y + z) / 3
```

Ejercicio 2: sumaMonedas
Esta función calcula el valor total de varias monedas de diferentes denominaciones. Cada denominación de moneda tiene su variable, por ejemplo, a representa la cantidad de monedas de 1 peso, b representa la cantidad de monedas de 2 pesos, y así sucesivamente. La lógica consiste en multiplicar la cantidad de monedas por su valor correspondiente.

Código:
```
sumaMonedas a b c d e = a * 1 + b * 2 + c * 5 + d * 10 + e * 20
```

Ejercicio 3: volumenEsfera
Esta función se utiliza para calcular el volumen de una esfera dado su radio. La lógica se basa en la fórmula matemática del volumen de una esfera, que es (4/3) * π * r^3. Aquí, r representa el radio de la esfera. La función eleva el radio al cubo (r^3), luego multiplica ese valor por π (que en Haskell se representa como pi), y finalmente multiplica el resultado por 4/3 para obtener el volumen.

Código:
```
volumenEsfera r = (4 / 3) * pi * r^3
```

Ejercicio 4: areaDeCoronaCircular
Esta función calcula el área de una corona circular, que es el área de un anillo formado por dos círculos concéntricos (uno dentro del otro). La lógica se basa en restar el área del círculo interno del área del círculo externo. En la fórmula, r1 es el radio del círculo interno y r2 el radio del círculo externo.

Código:
```
areaDeCoronaCircular r1 r2 = pi * (r2^2 - r1^2)
```

Ejercicio 5: ultimaCifra
Esta función obtiene la última cifra de un número. La lógica utiliza la operación mod, que devuelve el residuo de una división. En este caso, se utiliza x mod 10, lo que significa que se obtiene el residuo de dividir x entre 10. Esta operación es efectiva porque el residuo de dividir por 10 es la última cifra de un número.

Código:
```
ultimaCifra x = x `mod` 10
```

Ejercicio 6: maxTres
La función maxTres encuentra el mayor de tres números dados. Utiliza la función max dos veces: primero para encontrar el mayor entre dos de los números (y y z), y luego compara el resultado con el tercer número (x). De esta forma, la función devuelve el mayor de los tres números.

Código:
maxTres x y z = max x (max y z)

Ejercicio 7: rota1
Esta función mueve el primer elemento de una lista al final. La lógica se basa en tomar todos los elementos excepto el primero usando tail, y luego agregar ese primer elemento al final utilizando [head xs].

Código:
rota1 xs = tail xs ++ [head xs]

Ejercicio 8: rota
Esta función mueve los primeros n elementos de una lista al final. La lógica utiliza las funciones drop y take. La función drop elimina los primeros n elementos, y take toma esos mismos n elementos, que se colocan al final de la lista.

Código:
rota n xs = drop n xs ++ take n xs

Ejercicio 9: rango
La función rango busca el menor y el mayor valor en una lista y los devuelve en una lista con dos elementos. La lógica se basa en las funciones minimum y maximum, que encuentran el valor más pequeño y el más grande, respectivamente.

Código:
rango xs = [minimum xs, maximum xs]

Ejercicio 10: palindromo
Esta función verifica si una lista es un palíndromo, es decir, si se lee igual de izquierda a derecha que de derecha a izquierda. La lógica se basa en comparar la lista original xs con su versión invertida reverse xs. Si ambas son iguales, entonces la lista es un palíndromo.

Código:
palindromo xs = xs == reverse xs

Ejercicio 11: interior
La función interior toma una lista y elimina su primer y último elemento, dejando solo el "interior" de la lista. La lógica utiliza la función init para eliminar el último elemento y tail para eliminar el primero.

Código:
interior xs = tail (init xs)

Ejercicio 13: segmento
La función segmento extrae una subsección de una lista, dada por dos índices m y n. En primer lugar, la función elimina los primeros m elementos de la lista original utilizando drop. Luego, a partir del m-ésimo elemento, toma n-m+1 elementos usando take. Esto permite obtener un "segmento" o sublista que contiene solo los elementos comprendidos entre las posiciones m y n. La ventaja de esta función es que podemos trabajar con subsecciones de listas sin necesidad de recorrer toda la lista manualmente.

Código:
segmento m n xs = take (n - m + 1) (drop m xs)

Ejercicio 14: extremos
Esta función permite obtener los primeros n y últimos n elementos de una lista xs. La función toma los primeros n elementos usando take y los últimos n usando una combinación de drop y la longitud de la lista. En este proceso, drop elimina los primeros length xs - n elementos, dejando solo los últimos n. Finalmente, se concatenan ambos resultados para formar la nueva lista de extremos. Esta función es útil cuando queremos extraer tanto la "cabeza" como la "cola" de una lista, sin preocuparnos por la longitud exacta de la lista.

Código:
extremos n xs = take n xs ++ drop (length xs - n) xs

Ejercicio 15: mediano
La función mediano devuelve el valor intermedio entre tres números. La lógica es muy sencilla: sumamos los tres números, y luego restamos tanto el valor máximo como el mínimo. Esto nos deja con el valor que no es ni el mayor ni el menor, es decir, el mediano. Esta función es útil cuando queremos encontrar el valor central en un conjunto de tres números, sin necesidad de ordenar el conjunto.

Código:
mediano x y z = x + y + z - maximum [x, y, z] - minimum [x, y, z]

Ejercicio 16: tresIguales
Esta función verifica si tres valores son iguales. Utiliza el operador de igualdad para comparar el primer valor con el segundo, y el segundo con el tercero. Si todas las comparaciones son verdaderas, la función devuelve True; de lo contrario, devuelve False. Esta es una forma sencilla y directa de comprobar si tres elementos son idénticos, útil en casos donde la igualdad exacta de múltiples elementos es importante.

Código:
tresIguales x y z = x == y && y == z

Ejercicio 17: tresDiferentes
En este caso, la función tresDiferentes verifica si tres valores son todos distintos entre sí. Compara cada par de valores utilizando el operador de desigualdad. Si todas las comparaciones resultan verdaderas, la función devuelve True, indicando que los tres valores son diferentes. Si alguno de los valores es igual a otro, la función devuelve False. Esta función es útil cuando se requiere que los elementos de un conjunto sean únicos.

Código:
tresDiferentes x y z = x /= y && y /= z && x /= z

Ejercicio 18: cuatroIguales
Esta función verifica si cuatro valores son iguales. Primero, reutiliza la función tresIguales para comparar los primeros tres valores. Luego, se asegura de que el cuarto valor también sea igual al tercero. Si todas estas comparaciones son correctas, devuelve True, indicando que los cuatro valores son iguales. Esta función simplifica la verificación de igualdad en conjuntos de cuatro elementos.

Código:
```
cuatroIguales x y z u = tresIguales x y z && z == u
```

Guardas y Patrones

Ejercicio 1: divisionSegura
La función divisionSegura realiza una división entre dos números, pero antes verifica si el divisor es cero. Si el divisor es cero, la función devuelve 9999 para indicar que la división no se puede realizar, evitando así errores de división por cero. Si el divisor no es cero, realiza la división de forma normal y devuelve el resultado. Esta función es especialmente útil en casos donde no podemos permitir una excepción por división por cero.

Código:
```
divisionSegura x y
  | y == 0    = 9999
  | otherwise = x / y
```

Ejercicio 2: xor1
Esta función implementa la operación lógica XOR (disyunción excluyente). XOR es verdadera si exactamente uno de los dos valores es verdadero, y falsa en cualquier otro caso. La función define las combinaciones posibles entre dos valores booleanos (True o False) y retorna el valor correspondiente según la tabla de verdad de XOR. Esto es útil para casos donde necesitamos verificar la exclusión mutua entre dos condiciones.

Código:
```
xor1 True False = True
xor1 False True = True
xor1 _ _ = False
```

Ejercicio 3: mayorRectangulo
La función mayorRectangulo compara las áreas de dos rectángulos, cada uno representado por su base y altura. Para hacer la comparación, se multiplican las bases por las alturas de ambos rectángulos. Si el área del primer rectángulo es mayor o igual a la del segundo, se devuelve el primer rectángulo, de lo contrario, se devuelve el segundo. Esta función es útil cuando queremos elegir el rectángulo con mayor área entre dos opciones.

Código:
```
mayorRectangulo r1@(b1, h1) r2@(b2, h2)
  | b1 * h1 >= b2 * h2 = r1
  | otherwise = r2
```

Ejercicio 4: intercambia
La función intercambia toma un par de valores y devuelve el mismo par, pero con sus elementos en orden inverso. Es decir, el primer valor pasa a ser el segundo, y el segundo pasa a ser el primero. Esta función es útil cuando necesitamos cambiar el orden de los elementos en un par sin modificar los valores.

Código:

intercambia (x, y) = (y, x)

Ejercicio 5: distancia
Esta función calcula la distancia entre dos puntos en un plano bidimensional (coordenadas x1, y1) y (x2, y2). Utiliza la fórmula de distancia euclidiana: la diferencia entre las coordenadas se eleva al cuadrado, se suman ambas diferencias, y se obtiene la raíz cuadrada de la suma. Esto da como resultado la distancia entre los dos puntos, lo que es útil en geometría o en gráficos.

Código:
```
distancia (x1, y1) (x2, y2) = sqrt ((x2 - x1)^2 + (y2 - y1)^2)
```

Ejercicio 6: ciclo
La función ciclo mueve el último elemento de una lista al principio, dejando los demás elementos en el mismo orden. Para lograr esto, se utiliza la función last para obtener el último elemento, y luego se utiliza init para obtener todos los elementos de la lista excepto el último. Finalmente, se coloca el último elemento al principio. Esto es útil cuando se necesita rotar una lista de forma cíclica.

Código:
```
ciclo [] = []
ciclo xs = last xs : init xs
```

Ejercicio 7: numeroMayor
La función numeroMayor toma dos dígitos y forma el mayor número de dos cifras posible, colocando el mayor de los dos en la posición de las decenas y el menor en las unidades. Compara los dos dígitos, y dependiendo de cuál es mayor, construye el número correspondiente. Esta función es útil para formar el mayor número posible con dos cifras dadas.

Código:
```
numeroMayor x y
  | x >= y    = x * 10 + y
  | otherwise = y * 10 + x
```

Ejercicio 8: numeroDeRaices
La función numeroDeRaices determina cuántas raíces reales tiene una ecuación cuadrática basándose en el valor del discriminante. El discriminante se calcula como b^2 - 4 * a * c. Si el discriminante es mayor que cero, hay dos raíces reales; si es igual a cero, hay una raíz real; y si es menor que cero, no hay raíces reales. Esta función es útil para determinar el tipo de soluciones que tiene una ecuación cuadrática sin resolverla completamente.

Código:
```
numeroDeRaices a b c
  | discriminante > 0 = 2
  | discriminante == 0 = 1
  | otherwise = 0
  where discriminante = b^2 - 4 * a * c
```

Ejercicio 9: raices
Esta función calcula las raíces reales de una ecuación cuadrática de la forma ax^2 + bx + c = 0. La lógica se basa en el cálculo del discriminante, que se define como b^2 - 4 * a * c. Si el discriminante es menor que cero, no hay raíces reales y se devuelve una lista vacía. Si el discriminante es igual a cero, se devuelve una lista con una sola raíz, calculada como -b / (2 * a). Si el discriminante es positivo, se calculan las dos raíces utilizando la fórmula cuadrática y se devuelven en una lista. Esta funci...

Código:
```
raices a b c
  | discriminante < 0 = []
  | discriminante == 0 = [-b / (2 * a)]
  | otherwise = [(-b + sqrt discriminante) / (2 * a), (-b - sqrt discriminante) / (2 * a)]
  where discriminante = b^2 - 4 * a * c
```

Ejercicio 10: area
Esta función calcula el área de un triángulo utilizando la fórmula de Herón. La lógica de la función es calcular primero el semiperímetro s, que es la suma de los tres lados dividida entre 2. Luego, se utiliza la fórmula que multiplica s por las diferencias entre s y cada uno de los lados, y se toma la raíz cuadrada de ese producto para obtener el área del triángulo. Esta función es especialmente útil cuando conocemos los tres lados de un triángulo y queremos calcular su área sin usar ángulos.

Código:
```
area a b c = sqrt (s * (s - a) * (s - b) * (s - c))
  where s = (a + b + c) / 2
```

Ejercicio 11: interseccion
La función interseccion encuentra la intersección de dos intervalos representados como listas de dos elementos [a, b] y [c, d]. La lógica se basa en verificar si el valor máximo de los extremos inferiores es menor o igual que el valor mínimo de los extremos superiores. Si se cumple esta condición, significa que hay una intersección, y se devuelve una lista con los límites de la intersección. Si no se cumple, significa que los intervalos no se superponen y se devuelve una lista vacía.

Código:
```
interseccion [] _ = []
interseccion _ [] = []
interseccion [a,b] [c,d]
  | max a c <= min b d = [max a c, min b d]
  | otherwise = []
```

Ejercicio 12: linea
Esta función genera una línea específica de un triángulo aritmético, donde cada línea contiene números consecutivos. La lógica de la función es calcular el punto de inicio de la línea n utilizando la fórmula (n * (n - 1) / 2) + 1, y luego generar los n números consecutivos a partir de ese punto utilizando un rango. Esta función es útil para construir estructuras numéricas de manera organizada.

Código:

```
linea n = [start .. start + n - 1]
  where start = n * (n - 1) `div` 2 + 1
```

Recursividad

Ejercicio 1: potencia

La función potencia calcula la potencia de un número base x elevado a un exponente n de forma recursiva. La lógica de la función se basa en el hecho de que cualquier número elevado a la potencia 0 es igual a 1. Si n es 0, la función devuelve 1. En caso contrario, multiplica el número base x por el resultado de la función potencia con el exponente reducido en 1. Esta definición recursiva permite descomponer el problema en multiplicaciones sucesivas del número base.

Código:

```
potencia _ 0 = 1
potencia x n = x * potencia x (n-1)
```

Ejercicio 2: mcd

Esta función calcula el máximo común divisor (MCD) de dos números naturales a y b utilizando el algoritmo de Euclides. La lógica de este algoritmo es muy eficiente: si b es igual a cero, entonces el MCD es a. De lo contrario, se calcula el MCD de b y el residuo de dividir a entre b. Este proceso se repite recursivamente hasta que b sea cero. El MCD es útil para simplificar fracciones y resolver problemas que requieren divisibilidad.

Código:

```
mcd a 0 = a
mcd a b = mcd b (a `mod` b)
```

Ejercicio 3: pertenece

Esta función verifica si un elemento x está presente en una lista xs. La lógica es simple: si la lista está vacía, devuelve False, ya que no hay elementos en la lista. Si el primer elemento de la lista es igual a x, devuelve True. De lo contrario, recurre a la función pertenece para buscar en el resto de la lista. Esto permite recorrer la lista hasta encontrar el elemento o llegar al final.

Código:

```
pertenece _ [] = False
pertenece x (y:ys) = (x == y) || pertenece x ys
```

Ejercicio 4: tomar

La función tomar recibe un número n y una lista xs, y devuelve una nueva lista con los primeros n elementos de xs. Si n es cero o la lista está vacía, devuelve una lista vacía. De lo contrario, toma el primer elemento y recurre a sí misma para tomar n-1 elementos de la lista restante.

Código:

```
tomar _ [] = []
```

```
tomar 0 _ = []
tomar n (x:xs) = x : tomar (n-1) xs
```

Ejercicio 5: digitosC
Esta función devuelve los dígitos de un número como una lista de enteros. La lógica se basa en convertir el número a una lista de sus dígitos. Por cada iteración, se divide el número entre 10 y se toma el residuo como el último dígito, repitiendo el proceso hasta que el número sea cero.

Código:
```
digitosC 0 = []
digitosC n = digitosC (n `div` 10) ++ [n `mod` 10]
```

Ejercicio 6: sumaDigitosR
La función sumaDigitosR calcula la suma de los dígitos de un número de forma recursiva. Si el número es cero, devuelve cero. De lo contrario, toma el residuo de dividir el número entre 10 (último dígito) y lo suma al resultado de la función sumaDigitosR para el número dividido entre 10 (quitando el último dígito).

Código:
```
sumaDigitosR 0 = 0
sumaDigitosR n = (n `mod` 10) + sumaDigitosR (n `div` 10)
```

Ejercicio 2.1: ordenaRapida
La función ordenaRapida implementa el algoritmo de ordenación rápida (QuickSort) de forma recursiva. Se selecciona el primer elemento de la lista como pivote, se divide la lista en los elementos menores o iguales al pivote y los elementos mayores. Luego, se ordenan ambas partes recursivamente y se unen los resultados junto con el pivote. Este algoritmo es eficiente para listas de gran tamaño.

Código:
```
ordenaRapida [] = []
ordenaRapida (x:xs) = ordenaRapida [y | y <- xs, y <= x] ++ [x] ++ ordenaRapida [y | y <- xs, y > x]
```

Nuevos tipos de datos

Definición del tipo de datos Estudiante
El tipo Estudiante representa a un estudiante con cuatro atributos: el nombre, el apellido, la edad y el número de control. El número de control es un entero de 8 dígitos que sigue el formato del Tecnológico de Oaxaca, comenzando con los dígitos del año de ingreso (por ejemplo, "2021" para 2021). Esta estructura permite almacenar información relevante sobre cada estudiante y acceder a ella de manera clara y organizada.

Código:
```
data Estudiante = Estudiante {
  nombre :: String,
  apellido :: String,
```

```
  edad :: Int,
  numeroControl :: Int
} deriving (Show)
```

Lista de estudiantes

listaEstudiantes es una lista de instancias del tipo Estudiante, donde cada entrada contiene un nombre, apellido, edad y un número de control. Esta lista es útil para realizar operaciones como ordenación, cálculos y búsquedas.

Código:

```
listaEstudiantes = [
  Estudiante "Carlos" "Perez" 20 20210001,
  Estudiante "Ana" "Lopez" 22 20200001,
  Estudiante "Juan" "Gomez" 19 20220003,
  Estudiante "Maria" "Martinez" 21 20210004,
  Estudiante "Pedro" "Hernandez" 20 20200005,
  Estudiante "Sofia" "Diaz" 23 20200006,
  Estudiante "Luis" "Ramirez" 22 20200007,
  Estudiante "Laura" "Garcia" 19 20220008,
  Estudiante "Fernando" "Sanchez" 24 20190009,
  Estudiante "Lucia" "Ortiz" 20 20210010
  ]
```

Función para obtener al estudiante más joven

Esta función organiza una lista de estudiantes en función de su edad, ordenándolos de menor a mayor. La función utiliza recursión para dividir y conquistar el problema, basándose en la técnica de ordenamiento rápido (quick sort). La estructura de la función es simple, pero altamente eficiente para ordenar listas pequeñas.

Caso Base: La función comienza verificando si la lista de entrada está vacía. Este es el caso base de la recursión:

```
ordenarPorEdad [] = []
```

Si la lista está vacía, la función simplemente devuelve una lista vacía, lo que finaliza la recursión. Esto ocurre cuando hemos dividido la lista en fragmentos tan pequeños que ya no hay más elementos por ordenar.

Caso Recursivo: Si la lista contiene al menos un elemento, tomamos el primer elemento de la lista, x, como "pivote" y separamos el resto de los estudiantes en dos listas:

menores: Esta lista contiene a todos los estudiantes que tienen una edad menor o igual a la del estudiante pivote x. Utilizamos una lista por comprensión para crearla:

menores = [y | y <- xs, edad y <= edad x]

Aquí y <- xs toma cada estudiante del resto de la lista (xs), y el filtro edad y <= edad x asegura que solo los estudiantes cuya edad sea menor o igual a la de x se incluyan en la lista menores.

mayores: Similarmente, esta lista contiene a todos los estudiantes cuya edad es mayor que la del pivote:

mayores = [y | y <- xs, edad y > edad x]

Esta lista se construye filtrando solo los estudiantes con edad mayor a la del estudiante pivote.

Concatenación de Resultados: Luego, la función llama a sí misma recursivamente para ordenar las listas menores y mayores. Después, concatena los resultados de la siguiente manera:

ordenarPorEdad menores ++ [x] ++ ordenarPorEdad mayores

Esto significa que primero obtenemos una lista ordenada de los estudiantes menores, luego colocamos el estudiante pivote x en medio, y finalmente colocamos la lista de los estudiantes mayores, también ordenada. Este proceso de recursión se repite hasta que todas las sublistas están ordenadas y finalmente se combinan en una lista ordenada completa.

La función ordenarPorEdad divide la lista de estudiantes en partes más pequeñas basadas en un pivote, y luego las une después de que cada parte ha sido ordenada recursivamente. Este método de ordenamiento, conocido como "quick sort", es eficiente para listas pequeñas o medianas y garantiza que los estudiantes se ordenen por edad de manera ascendente.

Código:

```
ordenarPorEdad [] = []
ordenarPorEdad (x:xs) = ordenarPorEdad menores ++ [x] ++ ordenarPorEdad mayores
  where
    menores = [y | y <- xs, edad y <= edad x]
    mayores = [y | y <- xs, edad y > edad x]
```

Obtener al estudiante menor

Esta función busca al estudiante más joven en una lista de estudiantes. Utiliza una recursión simple para comparar la edad de cada estudiante con el resto de la lista. Si la lista contiene solo un estudiante, este es retornado directamente. Si la lista tiene más de un estudiante, la función llama recursivamente para encontrar el menor de la lista restante y compara el resultado con el estudiante actual. Si el estudiante actual tiene una edad menor o igual al estudiante más joven encontrado en la lista restante, se devuelve el estudiante actual.

Código:

```
menorEstudiante [x] = x
menorEstudiante (x:xs) =
  let menorResto = menorEstudiante xs
```

```
  in if edad x <= edad menorResto then x else menorResto
```

Variables:
- x: El estudiante actual que está siendo evaluado.
- xs: El resto de la lista de estudiantes.
- menorResto: El estudiante más joven encontrado recursivamente en la lista restante.

Obtener al estudiante mayor
De manera similar a `menorEstudiante`, esta función busca al estudiante más viejo. Utiliza una lógica recursiva para comparar la edad de cada estudiante en la lista. Si solo queda un estudiante en la lista, se devuelve directamente. Para listas más grandes, se compara recursivamente el estudiante actual con el resto de la lista y devuelve el que tenga mayor edad.

Código:
```
mayorEstudiante [x] = x
mayorEstudiante (x:xs) =
  let mayorResto = mayorEstudiante xs
  in if edad x >= edad mayorResto then x else mayorResto
```

Variables:
- x: El estudiante actual que está siendo evaluado.
- xs: El resto de la lista de estudiantes.
- mayorResto: El estudiante más viejo encontrado recursivamente en la lista restante.

Obtener el promedio de edades
Esta función calcula el promedio de las edades de una lista de estudiantes sin utilizar las funciones predefinidas de suma o longitud. Primero se calcula la suma total de las edades con una función recursiva llamada `sumaEdades`, que acumula las edades. Luego, se cuenta el número de estudiantes utilizando la función `contarElementos`, que también es recursiva. Finalmente, el promedio se obtiene dividiendo la suma de las edades entre la cantidad de estudiantes.

Código:
```
promedioEdades estudiantes = sumaEdades estudiantes / fromIntegral (contarElementos estudiantes)

sumaEdades [] = 0
sumaEdades (x:xs) = fromIntegral (edad x) + sumaEdades xs
contarElementos [] = 0
contarElementos (_:xs) = 1 + contarElementos xs
```

Variables:
- estudiantes: La lista de estudiantes.
- x: El estudiante actual cuyas edades se suman.
- xs: El resto de la lista de estudiantes.
- sumaEdades: Función que calcula la suma de las edades.
- contarElementos: Función que cuenta el número de estudiantes.

Árboles

Tipo de Árbol
El tipo de datos `Arbol` representa un árbol binario. Un árbol puede estar vacío (Vacio) o contener un nodo con un valor y dos subárboles, el izquierdo y el derecho. Este tipo de dato se utiliza para representar una estructura jerárquica donde se pueden insertar y buscar elementos de forma eficiente.

Código:
```
data Arbol a = Vacio | Nodo a (Arbol a) (Arbol a) deriving (Show)
```

Insertar un elemento en un árbol
Esta función inserta un elemento en el árbol binario. Si el árbol está vacío, se crea un nuevo nodo con el elemento. Si no, el valor del nodo actual se compara con el valor que se está insertando. Si el valor es menor, se inserta en el subárbol izquierdo; si es mayor, se inserta en el subárbol derecho. Este proceso continúa recursivamente hasta encontrar la posición adecuada para el nuevo valor.

Código:
```
insertar x Vacio = Nodo x Vacio Vacio
insertar x (Nodo y izq der)
  | x < y     = Nodo y (insertar x izq) der
  | otherwise = Nodo y izq (insertar x der)
```

Variables:
- x: El valor que se está insertando.
- Vacio: Representa un árbol vacío.
- y: El valor del nodo actual.
- izq: Subárbol izquierdo.
- der: Subárbol derecho.

Insertar desde un arreglo
Esta función toma una lista de elementos y los inserta uno por uno en el árbol binario, comenzando con un árbol vacío. Para cada elemento en la lista, se llama a la función `insertar`, lo que da como resultado un árbol binario que contiene todos los elementos de la lista.

Código:
```
insertarDesdeArreglo :: (Ord a) => [a] -> Arbol a
insertarDesdeArreglo [] = Vacio
insertarDesdeArreglo (x:xs) = insertar x (insertarDesdeArreglo xs)
```

Variables:
- x: El valor que se está insertando.
- xs: El resto de la lista que se insertará.

Buscar un elemento en un árbol
Esta función busca un valor en el árbol binario. Compara el valor que se está buscando con el valor del nodo actual. Si coinciden, devuelve `True`. Si el valor es menor, busca en el subárbol izquierdo, y si es mayor, busca en el subárbol derecho. Si llega a un nodo vacío, devuelve `False`, indicando que el valor no se encontró.

Código:
```
buscar _ Vacio = False
buscar x (Nodo y izq der)
  | x == y    = True
  | x < y     = buscar x izq
  | otherwise = buscar x der
```

Variables:
- x: El valor que se está buscando.
- Vacio: Representa un árbol vacío.
- y: El valor del nodo actual.
- izq: Subárbol izquierdo.
- der: Subárbol derecho.

Recorridos en el árbol

Recorrido Inorden
El recorrido inorden visita primero el subárbol izquierdo, luego el nodo actual, y finalmente el subárbol derecho. El resultado es una lista con los elementos en orden ascendente, siempre que el árbol esté bien balanceado.

Código:
```
inorden :: Arbol a -> [a]
inorden Vacio = []
inorden (Nodo x izq der) = inorden izq ++ [x] ++ inorden der
```

Recorrido Posorden
El recorrido posorden visita primero ambos subárboles, izquierdo y derecho, y luego el nodo actual. Es útil para operaciones en las que se deben procesar primero los subárboles antes de procesar el nodo.

Código:
```
posorden :: Arbol a -> [a]
posorden Vacio = []
posorden (Nodo x izq der) = posorden izq ++ posorden der ++ [x]
```

Recorrido Preorden
El recorrido preorden visita primero el nodo actual, luego el subárbol izquierdo y finalmente el subárbol derecho. Este recorrido es útil cuando es necesario procesar el nodo antes que sus hijos.

Código:

```
preorden Vacio = []
preorden (Nodo x izq der) = [x] ++ preorden izq ++ preorden der
```

Variables:

- x: El valor del nodo actual.
- izq: Subárbol izquierdo.
- der: Subárbol derecho.