Práctica 1

Brenda Jiménez Ruiz, Yessica Vianney Montes de Oca Aguilar, Irany Solano Marcial

February 12, 2025

1. buscar: En esta función observamos que el caso base es la lista vacia, posteriormente comparamos con la cabeza si es igual al elemento que estamos buscando, sino es así usamor otherwise para buscarlo en el resto de la lista.

2. suma_ordinaria: El Caso base (sumar_ordinario [] = 0): Si la lista está vacía ([]), la suma de los elementos es 0. Este es el caso base de la recursión.
   El Caso recursivo (sumar_ordinario (x:xs) = x + sumar_ordinario xs): Si la lista no está vacía, la lista se divide en la cabeza x y la cola xs. La función suma el valor de x al resultado de la recursión sobre el resto de la lista (sumar_ordinario xs).

3. sumar_lista: Aqui llamamos a una función auxiliar sumar_lista_aux y usa la lista que el usuario le da y un acumulador inicial en 0.
   Si la lista está vacía ([]), significa que hemos llegado al final de la lista, por lo que la función simplemente devuelve el valor del acumulador. Si la lista no está vacía y tiene al menos un elemento ((x:xs)), se separa en el primer elemento x y el resto de la lista xs.
   Aquí la función recursiva se llama a sí misma con la lista restante xs y un nuevo acumulador que es el acumulador actual más el valor del primer elemento de la lista (x). Por tanto usamos la recursión en cola para hacer mas eficiente el método.

4. filterB: El caso base es si la lista está vacía ([]), la función retorna una lista vacía. Esto es porque no hay elementos en la lista para filtrar.
   Ahora tenemos que dividir la lista en el primer elemento x y el resto de la lista xs.
   Se aplica el predicado p al primer elemento x: si p x es True, el elemento x se incluye en el resultado. Esto porque con x : filterB p xs, donde : es el operador de construcción de lista que agrega x al principio del resultado de aplicar filterB al resto de la lista xs.
   Si p x es False, entonces se omite el elemento x y se continúa con el filtrado de la lista xs mediante filterB p xs.

5. mapear: En el caso base si la lista está vacía ([]), la función retorna una lista vacía. Esto es porque no hay elementos que mapear.
   Ahora en el caso recursivo (mapear f (x:xs)): la lista se divide en el primer elemento x y el resto de la lista xs. La función f se aplica al primer elemento x con f x. El resultado de f x se agrega a la lista resultante, y luego se mapea recursivamente sobre el resto de la lista xs, con mapear f xs.
   El operador : se usa para construir una nueva lista, donde f x es el primer elemento, seguido de la lista resultante de aplicar mapear al resto de la lista.

6. preorder: Aquí el caso base es cuando el árbol está vacío (el nodo Empty), la función retorna una lista vacía []. No hay elementos que agregar al recorrido.
   Ahora el caso caso recursivo es cuando el árbol tiene un nodo (Node root left right), en este caso la funcion hace lo siguiente:

   - Visita el nodo actual: El valor del nodo actual es root, así que lo coloca en la lista resultante. Recorre el subárbol izquierdo: Llama recursivamente a preorder left, lo que va hacer que devuelva la lista de valores del subárbol izquierdo en preorden.
   - Recorre el subárbol derecho: Llama recursivamente a preorder right, lo que va hacer que devuelva la lista de valores del subárbol derecho en preorden.
   - Luego, concatena el valor del nodo ([root]) con las listas resultantes de los subárboles.

7. buscar_tree: El caso base (buscar_tree Empty _ = False): Sucede cuando el árbol está vacío (Empty), no hay ningún elemento en él para comparar con e, por lo que la función devuelve False.
   En el caso recursivo (buscar_tree (Node root left right) e): Sucede cuando el árbol tiene un nodo (Node root left right), la función hace lo siguiente:
   - Compara el valor del nodo: Si el valor del nodo raíz (root) es igual al valor que estamos buscando (e), la función devuelve True.
   - Si no es igual, la búsqueda continúa recursivamente en el subárbol izquierdo (left) y en el subárbol derecho (right):
   - Se evalúa la expresión buscar_tree left e —— buscar_tree right e. Esta expresión devuelve True si el valor e se encuentra en cualquiera de los subárboles (izquierdo o derecho), y False si no se encuentra en ninguno de los dos.

8. hojas: El caso base (hojas Empty = 0) sucede cuando el árbol está vacío (Empty), no hay nodos, por lo que no hay hojas. Por lo tanto, la función devuelve 0.
   Caso hoja (hojas (Node _ Empty Empty) = 1): Aquí sucede que el árbol tiene un nodo (Node _ left right) y tanto el subárbol izquierdo como el derecho son vacíos (Empty), significa que el nodo es una hoja, es decir no tiene hijos. En este caso, la función devuelve 1, porque hemos encontrado una hoja.
   Finalmente el caso recursivo (hojas (Node _ left right) = hojas left + hojas right): Si el árbol tiene un nodo con hijos, se cuenta la cantidad de hojas en el subárbol izquierdo (hojas left) y en el subárbol derecho (hojas right). El resultado total es la suma de las hojas de ambos subárboles.

9. leafSum: En el caso base (leafSum Empty = 0): Si el árbol está vacío (Empty), no hay hojas en el árbol, por lo que la suma es 0.
   Caso de hoja (leafSum (Node n Empty Empty) = n): Si el árbol tiene un nodo (Node n left right) y ambos subárboles izquierdo y derecho son vacíos (Empty), significa que el nodo es una hoja. En este caso, se retorna el valor de la hoja, que es n.
   En el caso recursivo (leafSum (Node _ left right) = leafSum left + leafSum right): Sucede que si el nodo no es una hoja (tiene hijos), se recurre recursivamente en el subárbol izquierdo (leafSum left) y en el subárbol derecho (leafSum right). El resultado final es la suma de las hojas de ambos subárboles.

10. vecinos: El caso base (vecinos []_ = []): es cuando el grafo está vacío (representado por la lista vacía []), la función devuelve una lista vacía, ya que no hay vértices en el grafo y, por lo tanto, no hay vecinos para buscar.
    En el caso recursivo (vecinos ((v, ns):xs) x): El patrón ((v, ns):xs) representa una lista de adyacencia del grafo, donde: v es un vértice, ns es la lista de vecinos de ese vértice (v), xs es el resto de la lista de adyacencias.
    La función compara si el vértice v es igual al vértice x que estamos buscando (v==x). Si son iguales (v == x), entonces hemos encontrado el vértice x, por lo que la función retorna la lista de vecinos ns de ese vértice. Si no son iguales (otherwise), la búsqueda continúa recursivamente en el resto del grafo xs.

11. dfs: El Caso base (v elem visited): Si el vértice v ya está en la lista visited (es decir, ya ha sido visitado), la función simplemente retorna la lista de vértices visitados tal como está, sin hacer más cambios.
    En el Caso recursivo (otherwise): Si el vértice v no ha sido visitado, la función agrega v al principio de la lista de vértices visitados (v : visited). Luego, se llama recursivamente a la función dfs sobre todos los vecinos de v. El parámetro vecinos graph v obtiene la lista de vecinos de v (mediante la función vecinos que ya hemos discutido previamente).
    foldl: La función foldl (fold left) se usa para aplicar la función recursiva dfs a cada vecino de v de manera acumulativa. Empieza con la lista (v : visited) y, para cada vecino n, realiza la recursión de forma acumulada, es decir, actualiza la lista de vértices visitados.

12. isConnected: El Caso base (isConnected [] = True): Sucede si el grafo está vacío, la función devuelve True, asumiendo que una gráfica vacía es conexa por definición.
    El caso recursivo (isConnected graph):

- startVertex = fst (head graph): Seleccionamos un vértice de inicio para comenzar el recorrido DFS. En este caso, seleccionamos el primer vértice del grafo (el primer vértice de la primera tupla de la lista de adyacencias).
- visited = dfs graph startVertex []: Ejecutamos un recorrido DFS a partir del vértice startVertex. El resultado de dfs es la lista de vértices visitados. Inicialmente, la lista de vértices visitados está vacía ([]), y la función dfs irá llenando esta lista a medida que visita los vértices.
- allVertices = map fst graph: Extraemos todos los vértices del grafo. La función map fst toma la primera parte de cada tupla en la lista de adyacencias (el vértice de cada tupla) y crea una lista de todos los vértices del grafo.
- all (elem visited) allVertices: Finalmente, la función verifica si todos los vértices del grafo han sido visitados. La función all toma una función y una lista, y verifica si esa función devuelve True para todos los elementos de la lista. En este caso, la función (elem visited) verifica si un vértice dado está en la lista de vértices visitados. Si todos los vértices están en la lista visited, significa que el grafo es conexo, y la función devuelve True. Si algún vértice no ha sido visitado, la función devuelve False.

13. isTree: El caso base (hojas Empty = 0): Sucede si el árbol está vacío (Empty), no hay nodos, por lo que no hay hojas. Por lo tanto, la función devuelve 0.
    Caso de hoja (hojas (Node _ Empty Empty) = 1): Si el árbol tiene un nodo (Node _ left right) y tanto el subárbol izquierdo como el derecho son vacíos (Empty), significa que el nodo es una hoja (no tiene hijos). En este caso, la función devuelve 1, porque hemos encontrado una hoja.
    Finalmente el Caso recursivo (hojas (Node _ left right) = hojas left + hojas right): Si el árbol tiene un nodo con hijos, se cuenta la cantidad de hojas en el subárbol izquierdo (hojas left) y en el subárbol derecho (hojas right). El resultado total es la suma de las hojas de ambos subárboles.

Tarea Moral: ¿Por qué es más flexible ++ que :, en haskell?
Recordemos en Haskell, ++ como : son operadores, pero se usan para diferentes tipos de estructuras de datos y tienen propósitos distintos.

- ++: Este operador ++ se utiliza para concatenar dos listas. Podemos decir que es más flexibled porque puede operar con cualquier tipo de lista, independientemente de los elementos quecontenga. Por ejemplo podemos agregar un elemento siempre y cuando lo volvamos lista.

- Operador : Este operador : se usa para construir listas, añadiendo un elemento al principio de una lista, y esto sucede siempre, agrega un solo elemento al principio de una lista.

En conclusión encontramos que ++ puede trabajar con cualquier tipo de lista, ademas ++ se usa también para la concatenación de listas lo hace más flexible cuando estás trabajando con listas de tipos arbitrarios. Ahora : sólo puede agregar un único elemento de tipo "a" a una lista de tipo "[a]". Por lo que podriamos decir que no es tan flexible, ademas siempre estás agregando un solo elemento a la cabeza de la lista.

Blibliografía: Autor2017, Haskell (:) and (++) differences [en linea]; Consultado el 11 de febrero de 2025; de: https://stackoverflow.com/questions/1817865/|haskell-and-differences