Práctica 1

Ramírez Rojas José David
Vázquez Cordero Néstor Semer

16 de febrero de 2020

1. Descripción

1.1. Punto Medio

Para resolver este problema se usó la fórmula del punto medio, dada por:

$$(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2})$$

Tomamos los valores de los dos puntos dados y escribimos la fórmula con la correcta sintaxis de Haskell.

1.2. Raices

Tomamos las tres entradas de la función, primero definimos el tipo de dato Complejo el cual lo definimos como una tupla de dos tuplas, luego de se realiza la operacion dentro de la raíz para poder saber si el resultado es positivo o negativo, en caso de ser positivo, la parte imaginaria del resultado sera 0.0, y procedemos a calcular la parte real. En caso de ser negativo el resultado, dividimos la formula general en dos partes, la parte real y la parte imaginaria y calculamos individualmente.

1.3. Segmento

Para este problema utilizamos dos funciones auxiliares definidas así:

1. drop :: Int → [a] → [a]
2. take :: Int → [a] → [a]

La funcion *drop* toma un entero n y una lista, devuelve el sufijo de la lista despues del elemento n. La funcion *take* toma un entero *n* y una lista, devuelve el prefijo de la lista de longitud n. La firma de la funcion obtiene dos numeros *(m, n)*, una lista y devuelve una lista con los elementos comprendidos entre el elemento *m y n.* por lo tanto se usa la funcion *take* para obtener los elementos hasta el indice *n*, con esto obtenemos el prefijo de la lista, luego aplicamos la funcion *take* a la lista resultante del proceso anterior y con esto obtenemos el sufijo de la lista hasta el elemento *m-1*.

1.4. Extremos

Dado el número n y la lista l, verificamos que *2n* no sea mayor o igual al tamaño de la lista l; si este es el caso devolvemos la lista l ya que sus primeros n elementos se sobreponerían a los últimos n elementos de otra forma.
Luego, cuando *2n* es menor al tamaño de la lista usamos las funciones ya definidas para listas de Haskell *take* y *reverse* para obtener dos listas que contengan:

1. A los primeros n elementos
2. La lista reversada de los primeros n elementos de la lista l reversada

Finalmente concatenamos ambas listas.

1.5. Intervalos

En este problema tambien usamos las funciones que se muestran en el problema 1.3 *(drop, take)*.
Dados los enteros m y n tomamos el prefijo de la lista hasta ele elemento *m-1* y lo concatenamos con en prefijo de la lista desde el elemento y hasta el final de la lista,el resultado es la lista sin los elementos comprendidos entre m y n de la lista que se paso por la entrada.

1.6. Números abundantes

Como sabemos con certeza que el menor número abundante es el 12, creamos una lista con enteros que satisfagan 2 condiciones:

1. Estar en el intervalo [12,13,14,...,n-1,n]
2. La suma de sus divisores fuera mayor al entero mismo.

Para esto último escribimos una función auxiliar para obtener una lista con los divisores para cada número del intervalo.

1.7. Elimina Duplicados

Se utilizará una función llamada elem la cual dado un elemento y una lista verifica si elemento se encuentra en la lista, la siguiente es la firma de la función:
elem ::Eq a $\Rightarrow$ a $\rightarrow$ [a] $\rightarrow$ Bool
Dada una lista *x:xs* tomamos la cabeza x y con la funcion *elem* verificamos si se encuantra en la cola *xs*, si el elemento se encuentra en la cola, se aplica recursión sobre la cola*xs*, si no se encuentra entonces se coloca como cabeza de la nueva lista.
Observacion:El resultado tiene un orden distinto al dado como ejemplo para realizar la practica pero contiene los mismos elementos, esto pasa por la forma en que se agregan los elementos en la recursión.

1.8. Primitivo

Revisamos que el número n sea de un solo dígito por la definición de primitivo.
Si no lo es necesitamos separar los dígitos de n y ponerlos en una lista con una función auxiliar llamada *separaDigitos*, y con la ayuda de la función para listas *product*, realizamos *primitivo (product (separaDigitos n))* para obtener el primitivo de n.

1.9. SipLis

Dado un natural y dos listas, el natural n es el índice de las listas donde comenzarán a concatenarse los *i-ésimos elementos* definimos recursivamente sipLis como:

1. Para *Cero*, una lista *Nula*y otra lista *ys* como uan lista *Nula*
2. Para *Cero*, una lista *xs* y una lista *Nula* como una lista *Nula*
3. Para *Cero*, una lista *(Cons x xs)*, otra lista *(Cons y ys)* como la lista concatenando *x y* y aplicando recursion con las colas de cada una de las listas *sipLis xs ys*
4. Para *Suc Cero*, una lista *(Cons x xs)*, otra lista *(Cons y ys)* como la lista concatenando *x y* y aplicando recursión con las colas de cada una de las listas *sipLis xs ys*
5. Para *Suc m*, una lista *(Cons x xs)*, otra lista *(Cons y ys)* solamante se aplica recursión a sus colas y dando como natural a m *sipLis m xs ys*

1.10. Aplana Árbol

Primero definimos tres funciones auxiliares para acceder a campos de un árbol:

1. obtenerValorNodo: Que devuelve el *Int* que guarda el nodo de un árbol.
2. obtenerArbolIzq: Que devuelve el subárbol izquierdo de todo árbol.
3. obtenerArbolDer: Que devuelve el subárbol derecho de todo árbol

Y así, definimos recursivamente a aplanaArbolPre como:

1. Para el arbol *Vacio* se devuelva la lista [ ]
2. Para un árbol t se devuelva la lista :
   [*obtenerValorNodo (t)*] ++ *aplanaArbolPre (obtenerArbolIzq (t))* ++ *aplanaArbolPre (obtenerArbolDer (t))*

La definimos de esta forma debido a la definición del recorrido pre-orden.

2. Forma de ejecución

Esta práctica se puede ejecutar usando el ambiente del compilador GHC: GHCi, escribiendo en la terminal lo siguiente:
:∼$ ghci NestorVazquez/Practica1/src/Practica1.hs

3. Conclusiones

Esta practica nos sirvió para recordar la forma de trabajar asi como la sintaxis de haskell.