PRÁCTICA PF2

Tipos de datos definidos por el programador

1. Definir la función **hanoi**, que dada una cierta cantidad de discos (mayor a cero), devuelva la lista de movimientos (de poste a poste) que deberían realizarse para resolver el juego de las torres de Hanoi. Especificar cómo son los tipos **Poste** y **Movimiento**.

2. Definir el tipo **Extension**, que "agrega" un elemento indefinido a otro tipo dado. Usar este tipo en funciones que devuelven casos indefinidos, como por ejemplo cabeza de una lista, resta de naturales, etc. En Haskell este tipo se llama Maybe.

3. Definir el tipo **TipoDeSangre**, cuyas instancias contienen las características de cada tipo de sangre (grupo y factor). Definir además la función puedeDonarA, que dice si un tipo de sangre puede donarse a otro tipo.

4. Definir el tipo **Nat**, que representa a los números naturales. Definir funciones de suma, resta y producto y la relación menor o igual. Definir también funciones de conversión del/al tipo Int.

5. Definir el tipo **LambdaTerm**, de λ-términos. Definir también las funciones varLibs (variables libres), subExp (subexpresiones), sust (sustituciones), y betaRed (beta reducción), ésta última cuando sea posible realizarse.

6. Definir el tipo **ListOrd** (lista ordenada). La diferencia con las listas comunes es que la inserción de un dato a la lista lo hace en forma ordenada. Crear funciones similares a la del tipo lista y una función de conversión a lista común.

7. ¿Qué diferencia puede haber al definir un tipo de dato sin participación de ningún tipo existente por un lado, y a partir de otro tipo existente por el otro (con constructores diferentes)? ¿En función de qué puede medirse esta diferencia?

8. Definir el tipo **Pila** con funciones crear, esVacia, cabeza, agregar, sacar y cantidad. Idem con el tipo **Cola**.

9. i) Definir el tipo **ArbBin** (árbol binario rotulado) con sus funciones constructoras y selectoras.
   ii) Definir las funciones nroNodos, nroHojas y altura.
   iii) Definir funciones que devuelvan la lista de rótulos del árbol en los tres modos posibles: preorden, inorden y postorden.
   iv) Definir la función igEstrucArb, que decida si dos árboles tienen la misma estructura (los rótulos pueden diferir).

10. Definir el tipo **ArbBinRotHoj**, con rótulos sólo en las hojas.

11. i) Definir el tipo **ArbGen** (árbol genérico), donde cada nodo puede tener una cantidad no fija (arbitraria) de hijos.
    ii) Definir todas las funciones equivalentes del tipo ArbBin para el tipo ArbGen.

12. Definir el tipo **Conjunto** con sus operaciones de unión, intersección, esVacio (dice si un conjunto es vacío), agElem (agrega un elemento a un conjunto), sacElem (quita un elemento de un conjunto) y pertenece (si un elemento pertenece a un conjunto). ¿Qué problema ocurre con la estructura de este tipo?

13. Definir el tipo **Matriz** de números con operaciones de suma, trasposición y producto.

14. Definir el tipo **Secuencia** de números con las siguientes funciones asociadas:
    - crearSecVac : Crea una secuencia vacía de elementos.
    - regAct : Dado un número de registro de una secuencia (según su orden de aparición) lo pone activo. Si ese número de registro no existe, debe devolver error.
    - leerReg : Lee el registro activo (si existe) y activa el siguiente registro en el orden. Si antes ya se había leído el último registro, se devolverá una señal de fin de secuencia y desactiva el registro activo si existía. Si no existía registro activo, se devolverá una señal de error.
    - borrReg : Borra el registro activo y no deja activo ningún registro.
    - agReg : Agrega un registro al final de la secuencia. Si había un registro activo, éste se desactiva.
    - numAct : Devuelve el número de registro activo. Si no existe, deberá devolver una señal de error.

15. Definir el tipo **ConjInf** (conjuntos posiblemente infinitos) a partir de funciones características (dado un dato dice si pertenece o no al conjunto). Definir funciones similares a las del tipo conjunto (sin la función esVacio).

16. i) Definir el tipo de dato **GNO** (grafo no orientado) con sus constructores.
    ii) Definir la función **esArbol**, que dice si un grafo cumple con la condición de ser árbol. Asumir definida la función **existeCamino**, que dados dos nodos y un GNO, devuelve si el segundo nodo es accesible desde el primero a través de un camino de arcos del GNO dado de longitud mayor o igual que uno.

17. Definir el tipo **DiagClases**, que represente un gráfico asociado a un diagrama de clases válido perteneciente al lenguaje de modelado UML. Debe contener los siguientes componentes:
    - Clases
    - Dominios
    - Atributos (con su dominio)
    - Asociaciones (con multiplicidades y roles)
    - Signaturas de métodos
    - Clases de asociación

18. i) Definir el tipo de dato **Elemento** de la tabla periódica de elementos, que se componen de un nombre y de una o más valencias. Las valencias están asociadas a números enteros. Asumir que si existe más de una valencia asociada, entonces ninguna puede ser cero.

ii) Definir la función **esMolecula**, que dada una lista de elementos (que pueden estar repetidos) devuelva si entre ellos pueden formar una molécula. El resultado será verdadero si ningún elemento es gas noble (tiene valencia cero) y además existe una combinación de valencias de cada uno de los elementos tal que su suma sea cero.

19. i) Describir el tipo **GR** (gramática regular con producciones lambda) con terminales de cualquier tipo.
    ii) Programar la función currificada **esInfinito**, que decida si el lenguaje generado por una gramática regular es infinito o no. Asumir que existe una función esAlcanzable, que dado un no terminal y una gramática regular, diga si genera algún elemento a partir del no terminal dado de la gramática regular dada.

20. i) Describir el tipo **AFD** (autómata finito determinístico) con caracteres como rótulos de transiciones.
    ii) Programar la función **esAceptada**, que dado un AFD y una lista de caracteres, diga si esa lista es aceptada por el AFD.
    iii) Programar la función **esAlcanzable**, que dado un AFD y un estado del AFD, diga si existe un camino desde ese estado hasta un estado final.

21. i) Programar el tipo de dato **Expresion** que permite representar expresiones aritméticas enteras con operadores de suma, resta, multiplicación, división, negación unaria y elevación al cuadrado.
    ii) Programar una función **listaAExp** que dada una lista de cadenas de caracteres que representa una expresión correctamente escrita en notación prefija, devuelva la expresión asociada de tipo expresion.
    Ej: lista_a_exp ["+", "-u", "10", "*", "5", "3"] → (-10)+5*3 (de tipo expresion)
    iii) Programar además la función inversa de la anterior (**expALista**), que dada una expresión del tipo expresión, devuelve una lista con el formato antes visto.

Nota: Asumir que existe una función strAInt, que dado un número en formato de lista de char devuelve un Int, y su inversa intAStr. No programar estas funciones.

22. i) Definir el tipo de datos **LogicalExp**, de expresiones lógicas (booleanas). En este tipo se deben poder representar sus constantes (verdadero y falso), así como también conjunciones, disyunciones y negaciones.
    ii) Definir un evaluador **evalLogExp** que reciba una expresión de tipo LogicalExp y devuelva su valor de verdad. Debe resolverlo por cortocircuito (short circuit), es decir que si uno de los componentes de la disyunción es verdadero, el otro no se evaluará; si uno de los componentes de la conjunción es falso, el otro no se evalúa. Asumir que las funciones primitivas de conjunción y disyunción del lenguaje son estrictas.

23. i) Definir el tipo de datos **FirstOrderExp**, de expresiones lógicas que representen fórmulas de la lógica de primer orden
    ii) Definir un evaluador **evalFirstOrderExp** que reciba una expresión de tipo FirstOrderExp y devuelva su valor de verdad. Pasar como parámetro todo aquello que crea necesario.