Práctica de ejercicios # 1 - Tipos algebraicos

Estructuras de Datos, Universidad Nacional de Quilmes

5 de abril de 2022

Aclaraciones:

- ***Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados.*** *No se saltee ejercicios sin consultar antes a un docente.*
- ***Recuerde que puede aprovechar en todo momento las funciones que ha definido****, tanto las de esta misma práctica como las de prácticas anteriores.*
- ***Pruebe todas sus implementaciones****, al menos en una consola interactiva.*
- *Es sumamente aconsejable* ***resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en videos publicados o clases presenciales****, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

1. Consejos

- Recordar probar las funciones definidas con ejemplos.
- Recordar determinar las precondiciones de las funciones que se definen.
- Recordar analizar la calidad de las soluciones, controlando:
  - que la cantidad de casos sea razonable;
  - que la división en subtareas sea adecuada;
  - que no haya vicios como "miedo al booleano";
  - que los nombres de las subtareas sean descriptivos y significativos.

2. Números enteros

1. Defina las siguientes funciones:
   a) `sucesor :: Int -> Int`
   Dado un número devuelve su sucesor
   b) `sumar :: Int -> Int -> Int`
   Dados dos números devuelve su suma utilizando la operación `+`.
   c) `divisionYResto :: Int -> Int -> (Int, Int)`
   Dado dos números, devuelve un par donde la primera componente es la división del primero por el segundo, y la segunda componente es el resto de dicha división. Nota: para obtener el resto de la división utilizar la función `mod :: Int -> Int -> Int`, provista por Haskell.
   d) `maxDelPar :: (Int,Int) -> Int`
   Dado un par de números devuelve el mayor de estos.
2. De 4 ejemplos de expresiones diferentes que denoten el número 10, utilizando en cada expresión a todas las funciones del punto anterior.
   Ejemplo: `maxDePar (divisionYResto (suma 5 5) (sucesor 0))`

3. Tipos enumerativos

1. Definir el tipo de dato Dir, con las alternativas Norte, Sur, Este y Oeste. Luego implementar las siguientes funciones:

   a) `opuesto :: Dir -> Dir`
   Dada una dirección devuelve su opuesta.

   b) `iguales :: Dir -> Dir -> Bool`
   Dadas dos direcciones, indica si son la misma. Nota: utilizar pattern matching y no `==`.

   c) `siguiente :: Dir -> Dir`
   Dada una dirección devuelve su siguiente, en sentido horario, y suponiendo que no existe la siguiente dirección a `Oeste`. ¿Posee una precondición esta función? ¿Es una función total o parcial? ¿Por qué?

2. Definir el tipo de dato DiaDeSemana, con las alternativas Lunes, Martes, Miércoles, Jueves, Viernes, Sabado y Domingo. Supongamos que el primer día de la semana es lunes, y el último es domingo. Luego implementar las siguientes funciones:

   a) `primeroYUltimoDia :: (DiaDeSemana, DiaDeSemana)`
   Devuelve un par donde la primera componente es el primer día de la semana, y la segunda componente es el último día de la semana. Considerar definir subtareas útiles que puedan servir después.

   b) `empiezaConM :: DiaDeSemana -> Bool`
   Dado un día de la semana indica si comienza con la letra M.

   c) `vieneDespues :: DiaDeSemana -> DiaDeSemana -> Bool`
   Dado dos días de semana, indica si el primero viene después que el segundo. Analizar la calidad de la solución respecto de la cantidad de casos analizados (entre los casos analizados en esta y cualquier subtarea, deberían ser no más de 9 casos).

   d) `estaEnElMedio :: DiaDeSemana -> Bool`
   Dado un día de la semana indica si no es ni el primer ni el ultimo dia.

3. Los booleanos también son un tipo de enumerativo. Un booleano es `True` o `False`. Defina las siguientes funciones utilizando pattern matching (*no usar* las funciones sobre booleanos ya definidas en Haskell):

   a) `negar :: Bool -> Bool`
   Dado un booleano, si es `True` devuelve `False`, y si es `False` devuelve `True`.
   En Haskell ya está definida como `not`.

   b) `implica :: Bool -> Bool -> Bool`
   Dados dos booleanos, si el primero es `True` y el segundo es `False`, devuelve `False`, sino devuelve `True`. Esta función debe ser tal que `implica False (error "Mal")` devuelva `True`.
   Nota: no viene implementada en Haskell.

   c) `yTambien :: Bool -> Bool -> Bool`
   Dados dos booleanos si ambos son `True` devuelve `True`, sino devuelve `False`. Esta función debe ser tal que `yTambien False (error "Mal")` devuelva `False`.
   En Haskell ya está definida como `\&\&`.

   d) `oBien :: Bool -> Bool -> Bool`
   Dados dos booleanos si alguno de ellos es `True` devuelve `True`, sino devuelve `False`. Esta función debe ser tal que `oBien True (error "Mal")` devuelva `True`.
   En Haskell ya está definida como `||`.

4. Registros

1. Definir el tipo de dato Persona, como un nombre y la edad de la persona. Realizar las siguientes funciones:

   - `nombre :: Persona -> String`
     Devuelve el nombre de una persona
   - `edad :: Persona -> Int`
     Devuelve la edad de una persona
   - `crecer :: Persona -> Persona`
     Aumenta en uno la edad de la persona.
   - `cambioDeNombre :: String -> Persona -> Persona`
     Dados un nombre y una persona, devuelve una persona con la edad de la persona y el nuevo nombre.
   - `esMayorQueLaOtra :: Persona -> Persona -> Bool`
     Dadas dos personas indica si la primera es mayor que la segunda.
   - `laQueEsMayor :: Persona -> Persona -> Persona`
     Dadas dos personas devuelve a la persona que sea mayor.

2. Definir los tipos de datos `Pokemon`, como un `TipoDePokemon` (agua, fuego o planta) y un porcentaje de energía; y `Entrenador`, como un nombre y dos Pokémon. Luego definir las siguientes funciones:

   - `superaA :: Pokemon -> Pokemon -> Bool`
     Dados dos Pokémon indica si el primero, en base al tipo, es superior al segundo. Agua supera a fuego, fuego a planta y planta a agua. Y cualquier otro caso es falso.
   - `cantidadDePokemonDe :: TipoDePokemon -> Entrenador -> Int`
     Devuelve la cantidad de Pokémon de determinado tipo que posee el entrenador.
   - `juntarPokemon :: (Entrenador, Entrenador) -> [Pokemon]`
     Dado un par de entrenadores, devuelve a sus Pokémon en una lista.

5. Funciones polimórficas

1. Defina las siguientes funciones polimórficas:

   a) `loMismo :: a -> a`
   Dado un elemento de algún tipo devuelve ese mismo elemento.

   b) `siempreSiete :: a -> Int`
   Dado un elemento de algún tipo devuelve el número 7.

   c) `swap :: (a,b) -> (b, a)`
   Dadas una tupla, invierte sus componentes.
   ¿Por qué existen dos variables de tipo diferentes?

2. Responda la siguiente pregunta: ¿Por qué estas funciones son polimórficas?

6. Pattern matching sobre listas

1. Defina las siguientes funciones polimórficas utilizando pattern matching sobre listas (no utilizar las funciones que ya vienen con Haskell):

2. `estaVacia :: [a] -> Bool`
   Dada una lista de elementos, si es vacía devuelve `True`, sino devuelve `False`.
   Definida en Haskell como `null`.

3. `elPrimero :: [a] -> a`
   Dada una lista devuelve su primer elemento.
   Definida en Haskell como `head`.
   Nota: tener en cuenta que el constructor de listas es `:`

4. `sinElPrimero :: [a] -> [a]`
   Dada una lista devuelve esa lista menos el primer elemento.
   Definida en Haskell como `tail`.
   Nota: tener en cuenta que el constructor de listas es `:`

5. `splitHead :: [a] -> (a, [a])`
   Dada una lista devuelve un par, donde la primera componente es el primer elemento de la lista, y la segunda componente es esa lista pero sin el primero.
   Nota: tener en cuenta que el constructor de listas es `:`