Programación Funcional

Ejercicios de Práctica Nro.8

Inducción/Recursión II

Aclaraciones:
- Los ejercicios siguen un orden de complejidad creciente, y cada uno puede servir a los siguientes. No se recomienda saltear ejercicios sin consultar antes a un docente.
- Recordar que se pueden aprovechar en todo momento las funciones ya definidas, tanto las de esta misma práctica como las de prácticas anteriores.
- Probar todas las implementaciones, al menos en una consola interactiva.
- Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evalúan principalmente este aspecto. Para utilizando formas alternativas al resolver los ejercicios consultar a los docentes.

Sección I

Ejercicio 1) Definir las siguientes funciones sobre listas utilizando recursión estructural:

a. **`length :: [a] -> Int`**, que describe la cantidad de elementos de la lista.
b. **`sum :: [Int] -> Int`**, que describe la suma de todos los elementos de la lista.
c. **`product :: [Int] -> Int`**, que describe el producto entre todos los elementos de la lista.
d. **`concat :: [[a]] -> [a]`**, que describe la lista resultante de concatenar todas las listas que son elementos de la dada.
e. **`elem :: Eq a => a -> [a] -> Bool`**, que indica si el elemento dado pertenece a la lista.
f. **`all :: (a -> Bool) -> [a] -> Bool`**, que indica si todos los elementos de la lista cumplen el predicado dado.
g. **`any :: (a -> Bool) -> [a] -> Bool`**, que indica si algún elemento de la lista cumple el predicado dado.
h. **`count :: (a -> Bool) -> [a] -> Int`**, que describe la cantidad de elementos de la lista que cumplen el predicado dado.
i. **`subset :: Eq a => [a] -> [a] -> Bool`**, que indica si todos los elementos de la primera lista se encuentran en la segunda.
j. **`(++) :: [a] -> [a] -> [a]`**, que describe el resultado de agregar los elementos de la primera lista adelante de los elementos de la segunda.
k. **`reverse :: [a] -> [a]`**, que describe la lista que tiene los elementos en el orden inverso a la lista dada.
l. **`zip :: [a] -> [b] -> [(a,b)]`**, que describe la lista resultante de juntar de a pares los elementos de ambas listas, según la posición que comparten en cada una.

m. **`unzip :: [(a,b)] -> ([a],[b])`**, que describe el par de listas que resulta de desarmar la lista dada; la primera componente del resultado se corresponde con las primeras componentes de los pares dados, y la segunda componente con las segundas componentes de dichos pares.

Ejercicio 2) Demostrar por inducción estructural las siguientes propiedades:

a. **`length (xs ++ ys) = length xs + length ys`**
b. **`count (const True) = length`**
c. **`elem = any . (==)`**
d. **`any (elem x) = elem x . concat`**
e. **`subset xs ys = all (flip elem ys) xs`**
f. **`all null = null . concat`**
g. **`length = length . reverse`**
h. **`reverse (xs ++ ys) = reverse ys ++ reverse xs`**
i. **`all p (xs ++ ys) = all p (reverse xs) && all p (reverse ys)`**
j. **`unzip (zip xs ys) = (xs, ys)`**
   (en este caso, mostrar que no vale)

Sección II

Ejercicio 1) Dada la siguiente definición

```
data N = Z | S N
```

cuya intención es describir representaciones unarias de números naturales,

a. implementar las siguientes funciones:

   i. **`evalN :: N -> Int`**, que describe el número representado por el elemento dado.
   ii. **`addN :: N -> N -> N`**, que describe la representación unaria de la suma de los números representados por los argumentos. La resolución debe ser exclusivamente *simbólica*, o sea, SIN calcular cuáles son esos números.
   iii. **`prodN :: N -> N -> N`**, que describe la representación unaria del producto de los números representados por los argumentos. La resolución debe ser exclusivamente *simbólica*.
   iv. **`int2N :: Int -> N`**, que describe la representación unaria del número dado usando el tipo **`N`**.

b. demostrar las siguientes propiedades:

   i. **`evalN (addN n1 n2) = evalN n1 + evalN n2`**
   ii. **`evalN (prodN n1 n2) = evalN n1 * evalN n2`**
   iii. **`int2N . evalN = id`**
   iv. **`evalN . int2N = id`**

Ejercicio 2) Dada la siguiente definición

```
type NU = [()]
```

cuya intención es describir representaciones unarias de números como listas de símbolos. El tipo `()` se lee Unit, y su único elemento es `()`; es equivalente a la siguiente definición:

```
data Unit = Unit
```

a. implementar las siguientes funciones por recursión estructural:

   i. `evalNU :: NU -> Int`, que describe el número representado por el elemento dado.
   ii. `succNU :: NU -> NU`, que describe la representación unaria del resultado de sumarle uno al número representado por el argumento. La resolución debe ser exclusivamente *simbólica*.
   iii. `addNU :: NU -> NU -> NU`, que describe la representación unaria de la suma de los números representados por los argumentos. La resolución debe ser exclusivamente *simbólica*.
   iv. `nu2N :: NU -> N`, que describe la representación unaria dada por el tipo `N` correspondiente al número representado por el argumento.
   v. `n2nu :: N -> NU`, que describe la representación unaria dada por el tipo `NU` correspondiente al número representado por el argumento.

b. demostrar las siguientes propiedades:

   i. `succNU = (+1) . evalNU`
   ii. `evalNU (addNU n1 n2) = evalNU n1 + evalNU n2`
   iii. `nu2n . n2nu = id`
   iv. `n2nu . nu2n = id`

Ejercicio 3) Dada la siguiente definición

```
type NBin = [DigBin]
```

cuya intención es describir representaciones binarias de números con el dígito menos significativo a la izquierda, y siendo `DigBin` el tipo definido en el ejercicio 2 de la práctica 5. Es recomendable reusar las funciones definidas en el ejercicio mencionado.

a. implementar las siguientes funciones por recursión estructural:

   i. `evalNB :: NBin -> Int`, que describe el número representado por el elemento dado.
   ii. `normalizarNB :: NBin -> NBin`, que describe la representación binaria del número representado por el argumento, pero sin *"ceros a la izquierda"* (dígitos redundantes).

iii. **`succNB :: NBin -> NBin`**, que describe la representación binaria *normalizada* del resultado de sumarle uno al número representado por el argumento. La resolución debe ser exclusivamente *simbólica*, y no debe utilizar **`normalizarNB`**. Se puede suponer como precondición que el argumento está normalizado.

iv. **`addNB :: NBin -> NBin -> NBin`**, que describe la representación binaria normalizada de la suma de los números representados por los argumentos. La resolución debe ser exclusivamente *simbólica*, y no debe utilizar **`normalizarNB`**. Se puede suponer como precondición que los argumentos está normalizados.

v. **`nb2N :: NBin -> N`**, que describe la representación unaria dada por el tipo **`N`** correspondiente al número representado por el argumento.

vi. **`n2nb :: N -> NBin`**, que describe la representación binaria dada por el tipo **`NBin`** correspondiente al número representado por el argumento.

b. demostrar las siguientes propiedades:

i. **`evalNB . normalizarNB = evalNB`**

ii. **`succNB = (+1) . evalNB`**

iii. **`evalNB (addNB n1 n2) = evalNB n1 + evalNB n2`**

iv. **`nb2n . n2nb = id`**

v. **`n2nb . nb2n = id`**

Ejercicio 4) Dada la siguiente definición

```
type NDec = [DigDec]
```

cuya intención es describir representaciones decimales de números con el dígito menos significativo a la izquierda, y siendo **`DigDec`** el tipo definido en el ejercicio 3 de la práctica 5. Es recomendable reusar las funciones definidas en el ejercicio mencionado.

a. implementar las siguientes funciones:

i. **`evalND :: NDec -> Int`**, que describe el número representado por el elemento dado.

ii. **`normalizarND :: NDec -> NDec`**, que describe la representación decimal del número representado por el argumento, pero sin *"ceros a la izquierda"* (dígitos redundantes).

iii. **`succNDec :: NDec -> NDec`**, que describe la representación decimal *normalizada* del resultado de sumarle uno al número representado por el argumento. La resolución debe ser exclusivamente *simbólica*, y no debe utilizar **`normalizarND`**. Se puede suponer como precondición que el argumento está normalizado.

iv. **`addNDec :: NDec -> NDec -> NDec`**, que describe la representación decimal normalizada de la suma de los números representados por los argumentos. La resolución debe ser exclusivamente *simbólica*, y no debe utilizar **`normalizarND`**. Se puede suponer como precondición que los argumentos está normalizados.
v. **`nd2nb :: NDec -> NBin`**, que describe la representación binaria correspondiente al número representado por el argumento.
vi. **`nb2nd :: NBin -> NDec`**, que describe la representación decimal correspondiente al número representado por el argumento.

b. demostrar por inducción estructural las siguientes propiedades:

i. **`succNDec = (+1) . evalNDec`**
ii. **`evalNDec (addNDec n1 n2) = evalNDec n1 + evalNDec n2`**
iii. **`nd2nb . nb2nd = id`**
iv. **`nb2nd . nd2nb = id`**

Ejercicio 5) Dar la representación de los números 17 y 42 como **`N`**, **`NU`**, **`NBin`** y **`NDec`**.

Sección III

Ejercicio 6) Dada la siguiente representación de expresiones aritméticas

```
data ExpA = Cte Int
          | Sum ExpA ExpA
          | Prod ExpA ExpA
```

a. implementar las siguientes funciones:

i. **`evalEA :: ExpA -> Int`**, que describe el número que resulta de evaluar la cuenta representada por la expresión aritmética dada.
ii. **`simplificarEA :: ExpA -> ExpA`**, que describe una expresión aritmética con el mismo significado que la dada, pero que no tiene sumas del número 0, ni multiplicaciones por 1 o por 0. La resolución debe ser exclusivamente *simbólica*.
iii. **`cantidadDeSumaCero :: ExpA -> Int`**, que describe la cantidad de veces que aparece suma cero en la expresión aritmética dada. La resolución debe ser exclusivamente *simbólica*.

b. demostrar por inducción estructural las siguientes propiedades:

i. **`evalEA . simplificarEA = evalEA`**
ii. **`cantidadSumaCero . simplificarEA = const 0`**

Ejercicio 7) Dada la siguiente representación de expresiones aritméticas

```
data ExpS = CteS N
          | SumS ExpS ExpS
          | ProdS ExpS ExpS
```

a. implementar las siguientes funciones:

   i. **`evalES :: ExpS -> Int`**, que describe el número que resulta de evaluar la cuenta representada por la expresión aritmética dada.
   ii. **`es2ea :: ExpS -> ExpA`**, que describe una expresión aritmética representada con el tipo ExpA, que tiene el mismo significado que la dada.
   iii. **`ea2es :: ExpA -> Int`**, que describe la cantidad de veces que aparece suma cero en la expresión aritmética dada. La resolución debe ser exclusivamente *simbólica*.

b. demostrar por inducción estructural las siguientes propiedades:
   i. **`evalEA . es2ea = evalES`**
   ii. **`es2ea . ea2es = id`**
   iii. **`ea2es . es2ea = id`**