Introducción a la programación

Práctica 4: Recursión sobre números enteros

Ej 4.1

Implementar la función `fibonacci: Integer -> Integer` que devuelve el i-ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

`problema` `fibonacci` (n: $\mathbb{Z}$) : $\mathbb{Z}$ {
  `requiere:` $\{ n \geq 0 \}$
  `asegura:` $\{ resultado = fib(n) \}$
}

Ej 4.1

Implementar la función `fibonacci: Integer -> Integer` que devuelve el i-ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

Podemos comenzar pensando cual es el caso base (o mejor dicho, los casos base):

- $n = 0 => (resultado = 0)$
- $n = 1 => (resultado = 1)$

Ej 4.1

Implementar la función `fibonacci: Integer -> Integer` que devuelve el i-ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

Y luego consideramos el paso recursivo:

- $n = 0 => (resultado = 0)$
- $n = 1 => (resultado = 1)$
- $n >= 2 => (resultado = fib(n-1) + fib(n-2))$

Ej 4.1 Haskell

Lo planteamos en Haskell:

```haskell
fibonacci :: Integer -> Integer
fibonacci n | n == 0 = ...
            | n == 1 = ...
            | n >=2 = ...
```

Ej 4.1 Haskell

Lo planteamos en Haskell usando guardas:

```
fibonacci :: Integer -> Integer
fibonacci n | n == 0 = 0
            | n == 1 = 1
            | n >= 2 = (fibonacci (n-1)) +
                       (fibonacci (n-2))
```

Ej 4.1 Haskell

Lo planteamos en Haskell usando guardas:

```
fibonacci :: Integer -> Integer
fibonacci n | n == 0 = 0
            | n == 1 = 1
            | n >= 2 = (fibonacci (n-1)) +
                       (fibonacci (n-2))
```

Esta no es la unica forma de implementar la funcion en Haskell.
Veamos otras

Ej 4.1 Haskell

Podriamos evitar los casos en que la recursion se hace infinita. Ej: si n = -1

```
fibonacci :: Integer -> Integer
fibonacci n | n == 0 = 0
            | n == 1 = 1
            | n >= 2 = (fibonacci (n-1)) +
                       (fibonacci (n-2))
            | otherwise = undefined
```

Ej 4.1 Haskell

La podemos definir tambien usando pattern matching:

```
fibonacci :: Integer -> Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = (fibonacci (n-1)) +
              (fibonacci (n-2))
```

Ej 4.2

Implementar una función `parteEntera :: Float -> Integer` que calcule la parte entera de un número real positivo.

`problema` `parteEntera` (x: $\mathbb{R}$) : $\mathbb{Z}$ $\{$
  `requiere:` $\{\ True\ \}$
  `asegura:` $\{\ resultado \leq x < resultado + 1\ \}$
$\}$

Ej 4.2

Probemos con algunos ejemplos:

```
parteEntera 8.124 = ?
parteEntera 1.999999 = ?
parteEntera 0.12 = ?
```

Ej 4.2

Probemos con algunos ejemplos:

```
parteEntera 8.124 = 8
parteEntera 1.999999 = 1
parteEntera 0.12 = 0
```

Ej 4.2

Probemos con algunos ejemplos:

```
parteEntera 8.124 = 8
parteEntera 1.999999 = 1
parteEntera 0.12 = 0
```

Podemos pensar en un caso base:

```
parteEntera :: Float -> Integer
parteEntera x | 0 <= x < 1 = 0
              | ...
```

Ej 4.2 Haskell

Y luego agregarle el paso recursivo:

```
parteEntera :: Float -> Integer
parteEntera x | 0 <= x < 1 = 0
              | otherwise = 1 + parteEntera (x-1)
```

Ej 4.2 Haskell

Y luego agregarle el paso recursivo:

```
parteEntera :: Float -> Integer
parteEntera x | 0 <= x < 1 = 0
              | otherwise = 1 + parteEntera (x-1)
```

Cuidado! Revisemos la especificacion. Cubrimos todos los casos?

Ej 4.2 Haskell

Completamos con los casos negativos:

```
parteEntera :: Float -> Integer
parteEntera x | x < 1 && x >= 0 = 0
              | x > -1 && x <= 0 = -1
              | x >= 1 = 1 + parteEntera (x - 1)
              | otherwise = (-1) + parteEntera (x + 1)
```

Ej 4.7

Implementar la función `todosDigitosIguales :: Integer -> Bool` que determina si todos los dígitos de un número natural son iguales.

`problema` `todosDigitosIguales` (n: $\mathbb{Z}$) : $\mathbb{B}$ $\{$

  `requiere:` $\{\ n > 0\ \}$

  `asegura:` $\{$

$$res = true \leftrightarrow (\exists d, k : \mathbb{Z})(0 \leq d \leq 9 \wedge n = \sum_{i=0}^{k} d * 10^i)\ \}$$

$\}$

Ej 4.7

Implementar la función `todosDigitosIguales :: Integer -> Bool` que determina si todos los dígitos de un número natural son iguales.

`problema todosDigitosIguales` (n: $\mathbb{Z}$) : $\mathbb{B}$ {
  `requiere:` $\{ n > 0 \}$
  `asegura:` {

$$res = true \leftrightarrow (\exists d, k : \mathbb{Z})(0 \leq d \leq 9 \wedge n = \sum_{i=0}^{k} d * 10^i) \; \}$$

}

Veamos algunos ejemplos:

- $44 = 4 * 10 + 4 = 4 * 10^1 + 4 * 10^0$
- $8888 = 8 * 10^3 + 8 * 10^2 + 8 * 10^1 + 8 * 10^0$

Ej 4.7

Algunas operaciones útiles para manipular enteros:

- mod:
  - mod 8123 10 = ?
  - mod 2142 10 = ?
  - mod 4 10 = ?
- div:
  - div 8123 10 = ?
  - div 2142 10 = ?
  - div 4 10 = ?

Ej 4.7

Algunas operaciones útiles para manipular enteros:

- mod:
  - mod 8123 10 = 3
  - mod 2142 10 = 2
  - mod 4 10 = 4
- div:
  - div 8123 10 = 812
  - div 2142 10 = 214
  - div 4 10 = 0

Ej 4.7

Algunas operaciones útiles para manipular enteros:

- mod:
  - mod 8123 10 $=$ 3
  - mod 2142 10 $=$ 2
  - mod 4 10 $=$ 4
- div:
  - div 8123 10 $=$ 812
  - div 2142 10 $=$ 214
  - div 4 10 $=$ 0

mod n 10 $\rightarrow$ me da el ultimo digito de n.
div n 10 $\rightarrow$ le *saca* el ultimo digito a n.

Ej 4.7 Haskell

La escribimos en Haskell. Consideremos primero el caso base:

```
tDI :: Integer -> Bool
tDI n | n < 10 = True
      | ...
```

Ej 4.7 Haskell

Y luego consideramos el paso recursivo usando **mod 10** y **div 10**:

```
tDI :: Integer -> Bool
tDI n | n < 10 = True
      | otherwise = (ultimoDigito n == ultimoDigito (sacarUltimo n)) &&
                    tDI (sacarUltimo n)

ultimoDigito :: Integer -> Integer
ultimoDigito n = mod n 10

sacarUltimo :: Integer -> Integer
sacarUltimo n = div n 10
```

Ej 4.8

Implementar la función `iesimoDigito :: Integer -> Integer -> Integer` que dado un $n \in \mathbb{N}_{\geq 0}$ y un $i \in \mathbb{N}$ menor o igual a la cantidad de dígitos de n, devuelve el i-ésimo dígito de n.

problema `iesimoDigito` (n: $\mathbb{Z}$, i: $\mathbb{N}$) : $\mathbb{Z}$ {
  `requiere:` $\{ n \geq 0 \land 1 \leq i \leq cantDigitos(n) \}$
  `asegura:` $\{ resultado = (n \text{ div } 10^{cantDigitos(n)-i}) \text{ mod } 10 \}$
}

problema `cantDigitos` (n: $\mathbb{Z}$) : $\mathbb{N}$ {
  `requiere:` $\{ n \geq 0 \}$
  `asegura:` $\{ n = 0 \rightarrow res = 1 \}$
  `asegura:` $\{ n \neq 0 \rightarrow (n \text{ div } 10^{res-1} > 0 \land n \text{ div } 10^{res} = 0) \}$
}

Ej 4.8 Haskell

Implementemos primero la funcion auxiliar cantDigitos:

```
cantDigitos :: Integer -> Integer
cantDigitos n | n < 10 = 1
              | otherwise = 1 + cantDigitos (sacarUltimo n)
              where sacarUltimo n = div n 10
```

Ahora podemos implementar la funcion que nos pedian:

```
iesimoDigito :: Integer -> Integer -> Integer
iesimoDigito n i | i == cantDigitos n = ultimoDigito n
                 | otherwise = iesimoDigito (sacarUltimo n) i
                 where sacarUltimo n = div n 10
                       ultimoDigito n = mod n 10
```

Ej 4.14 Haskell

Implementar una función `sumaPotencias :: Integer - > Integer -> Integer -> Integer` que dados tres naturales q, n, m sume todas las potencias de la forma q^{a+b} con $1 \leq a \leq n$ y $1 \leq b \leq m$.

```
sumatoriaInterna2 :: Integer -> Integer -> Integer -> Integer
sumatoriaInterna2 _ _ 0 = 0
sumatoriaInterna2 n j k = n^(j+k) + sumatoriaInterna2 n j (k-1)

sumaPotencias :: Integer ->Integer ->Integer ->Integer
sumaPotencias _ 0 _ = 0
sumaPotencias q n m = sumaPotencias q (n-1) m + sumatoriaInterna2 q n m
```

Ej 4.18 Haskell

Implementar una función `mayorDigitoPar :: Integer -> Integer` que devuelve el mayor de los dígitos pares de un número natural positivo. Si el número no tiene ningún digíto par, entonces la función devuelve -1. TT (Aula 8) Prestar atención que tiene cambios con respecto a lo que se vió en clase.

```haskell
esPar :: Integer -> Bool
esPar x = mod x 2 == 0

elMayor :: Integer -> Integer -> Integer
elMayor a b | a>b = a
            | otherwise = b

mayorDigitoPar :: Integer -> Integer
mayorDigitoPar x | ultimo == resto && not (esPar(x)) = -1
                 | ultimo == resto && esPar(ultimo)  = ultimo
                 | esPar ultimo = elMayor ultimo (mayorDigitoPar resto)
                 | otherwise    = mayorDigitoPar resto
    where
        ultimo = mod x 10
        resto  = div x 10
```