Introducción a la programación

Práctica 4: Recursión sobre números enteros

Repaso de la teórica

Vamos a implementar la función factorial, entre todos como repaso de lo visto de recursión en la teórica.
Recordemos:

$$n! = \prod_{k=1}^{n} k \qquad\qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n > 0 \end{cases}$$

Posible solución para factorial

```haskell
factorial :: Int -> Int
factorial n
        | n == 0 = 1
        | n > 0 = n * factorial (n-1)
```

Comentando el código

Para poder escribir texto que no sea ejecutado por Haskell, pero que podamos ver y leer en el código, podemos utilizar los comentarios.
Se pueden agregar comentarios de una sóla línea:

```
factorial :: Int -> Int
--Este es un comentario, no interrumpe la ejecución
factorial n
    | n == 0 = 1
    | n > 0 = n * factorial (n-1)
```

Comentando el código

Si queremos dejar comentarios de varias líneas, por ejemplo cuando estamos *debugeando* el código y queremos saber dónde esta el error, podemos comentar toda una función para que no se ejecute:

```
{--
factorial :: Int -> Int
factorial n
    | n == 0 = 1
    | n > 0 = n * factorial (n-1)
--}
```

Ej 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 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 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 1 Posible solución en Haskell

Lo planteamos en Haskell:

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

Otra forma de resolverlo ...

Ej 1 Haskell

Lo planteamos en Haskell usando guardas:

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

Ej 1 Haskell

Lo planteamos en Haskell usando guardas:

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

Esta no es la única forma de implementar la función en Haskell. Veamos otras

Ej 1 Haskell

La podemos definir también usando pattern matching:

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

Ej 1 Haskell

La podemos definir también usando pattern matching:

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

- ¿Qué pasa si introducimos n=-1 en nuestra función?
- ¿Debemos preocuparnos por este caso?

Ej 2

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

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

Ej 2

Probemos con algunos ejemplos:

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

Ej 2

Probemos con algunos ejemplos:

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

Ej 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 && x < 1 = 0
              | ...
```

Ej 2 Haskell

Y luego agregarle el paso recursivo:

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

Ej 2 Haskell

Y luego agregarle el paso recursivo:

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

Cuidado! Revisemos la especificación.
¿Cubrimos todos los casos?

Ej 2 Haskell

Completamos con los casos negativos:

```haskell
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 2 Haskell

Podemos omitir el caso base de $-1 < x < 0 = -1$, pues cuando sumamos 1 pasamos al caso base de los positivos que devuelve cero.
Nota: Sugerimos realizar una ejecución manual empezando con $-1{,}5$ para comparar las dos soluciones propuestas.

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

Ej 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:` $\{ res = true \leftrightarrow$ todos los dígitos de n son iguales $\}$
}

Ej 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:` $\{\ res = true \leftrightarrow$ todos los dígitos de n son iguales $\}$
}

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 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 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 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.

Ponemos nombre a las funciones

Recordemos que en la guía 3 definimos la función *digitoUnidades* y podemos ponerle nombre a la segunda función como *sacarUnidades*.
La modularización de código y la legibilidad es un atributo evaluable en nuestra materia.

Ej 7 Haskell

La escribimos en Haskell. Consideremos primero el caso base:

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

Ej 7 Haskell

Y luego consideramos el paso recursivo usando las funciones que ya pensamos:

```haskell
todosDigitosIguales :: Integer -> Bool
todosDigitosIguales n
  | n < 10 = True
  | otherwise = (
      digitoUnidades n == digitoUnidades (sacarUnidades n)
    ) && todosDigitosIguales (sacarUnidades n)

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

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

Ej 8

Implementar la función `iesimoDigito :: Integer -> Integer -> Integer` que dado un $n \in \mathbb{Z}$ mayor o igual a 0 y un $i \in \mathbb{Z}$ mayor o igual a 1 menor o igual a la cantidad de dígitos de n, devuelve el i-ésimo dígito de n.

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

Demos algunos ejemplos para asegurarnos que comprendimos la especificación

- cantDigitos 0 = ?
- cantDigitos 12 = ?
- cantDigitos 123 = ?

Demos algunos ejemplos para asegurarnos que comprendimos la especificación

- cantDigitos $0 = 1$
- cantDigitos $12 = (12 \text{ div } 10^{res-1} > 0 \land 12 \text{ div } 10^{res} = 0) = 2$
- cantDigitos $123 =$
  $(123 \text{ div } 10^{res-1} > 0 \land 123 \text{ div } 10^{res} = 0) = 3$

Y ejemplos con iesimoDigito?

- iesimoDigito 468 0 $=$?
- iesimoDigito 468 1 $=$?
- iesimoDigito 468 2 $=$?
- iesimoDigito 468 3 $=$?

Y ejemplos con iesimoDigito?

- iesimoDigito 468 0 $= (468 \text{ div } 10^{cantDigitos(468)-0}) \text{ mod } 10 = (468 \text{ div } 10^{3}) \text{ mod } 10 = (468 \text{ div } 1000) \text{ mod } 10 =$ 0
- iesimoDigito 468 1 $= (468 \text{ div } 10^{cantDigitos(468)-1}) \text{ mod } 10 = (468 \text{ div } 10^{3-1}) \text{ mod } 10 = (468 \text{ div } 10^{2}) \text{ mod } 10 =$ 4
- iesimoDigito 468 2 $= (468 \text{ div } 10^{cantDigitos(468)-2}) \text{ mod } 10 = (468 \text{ div } 10^{3-2}) \text{ mod } 10 = (468 \text{ div } 10^{1}) \text{ mod } 10 =$ 6
- iesimoDigito 468 3 $= (468 \text{ div } 10^{cantDigitos(468)-3}) \text{ mod } 10 = (468 \text{ div } 10^{3-3}) \text{ mod } 10 = (468 \text{ div } 10^{0}) \text{ mod } 10 =$ 8

Y ejemplos con iesimoDigito?

- iesimoDigito 468 0 $= (468 \text{ div } 10^{cantDigitos(468)-0})$ mod $10 =$ $(468 \text{ div } 10^3)$ mod $10 = (468 \text{ div } 1000)$ mod $10 =$ 0
- iesimoDigito 468 1 $= (468 \text{ div } 10^{cantDigitos(468)-1})$ mod $10 =$ $(468 \text{ div } 10^{3-1})$ mod $10 = (468 \text{ div } 10^2)$ mod $10 =$ 4
- iesimoDigito 468 2 $= (468 \text{ div } 10^{cantDigitos(468)-2})$ mod $10 =$ $(468 \text{ div } 10^{3-2})$ mod $10 =(468 \text{ div } 10^1)$ mod $10 =$ 6
- iesimoDigito 468 3 $= (468 \text{ div } 10^{cantDigitos(468)-3})$ mod $10 =$ $(468 \text{ div } 10^{3-3})$ mod $10 =(468 \text{ div } 10^0)$ mod $10 =$ 8

Notemos que el requiere indica que: $n \geq 0 \wedge 1 \leq i \leq cantDigitos(n)$, por lo tanto el primer caso con i $=$ 0 no es válido. Y tampoco sería válido usar i $=$ 4 porque 468 tiene 3 dígitos.

Ej 8 Haskell

Implementemos primero la función auxiliar cantDigitos:

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

Nota: el `where` nos permite renombrar una parte del código de la función actual; esto funcionará de forma local, únicamente dentro de la función cantDigitos.

Ej 8 Haskell

Ahora podemos implementar la función que nos pedían:

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

El caso base será si la cantidad de dígitos es igual al segundo parámetro, en ese caso nos están pidiendo las unidades del número. En caso contrario debemos pensar la recursión, como sabemos que no es el caso actual, entonces sacamos el último dígito del número y disminuimos el segundo parámetro para acercarnos al caso base.