Mecanismo de Reducción

Taller de Álgebra I

Primer cuatrimestre 2018

Ejercicios de números enteros

Ejercicios

Dar el tipo y luego implementar las siguientes funciones:

1. `unidades`: dado un entero, devuelve el dígito de las unidades del número (el dígito menos significativo).
2. `sumaUnidades3`: dados 3 enteros, devuelve la suma de los dígitos de las unidades de los 3 números.
3. `todosImpares`: dados 3 números enteros determina si son todos impares.
4. `alMenosUnImpar`: dados 3 números enteros determina si al menos uno de ellos es impar.
5. `alMenosDosImpares`: dados 3 números enteros determina si al menos dos de ellos son impares.
6. `alMenosDosPares`: dados 3 números enteros determina si al menos dos de ellos son pares.

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.
    ```
    suc :: Integer -> Integer
    suc x = x + 1
    ```
  - Funciones parciales: hay argumentos para los cuales se indefinen.
    ```
    inv :: Float -> Float
    inv x | x /= 0 = 1/x
    ```

Ejercicios de relaciones

Ejercicios

7. Dados dos enteros a, b implementar funciones:
   ($r1$, $r2$ y $r3$) `::` `Integer -> Integer -> Bool` que determinen si $a \sim b$ donde:
   1. $a \sim b$ si tienen la misma paridad
   2. $a \sim b$ si $2a + 3b$ es divisible por 5
   3. $a \sim b$ si los dígitos de las unidades de a, b y ab son todos distintos
8. Se define en $\mathbb{R}$ la relación de equivalencia asociada a la partición
   $$\mathbb{R} = (-\infty, 3) \cup [3, +\infty)$$
   Determinar el tipo e implementar una función que dados dos números $x, y \in \mathbb{R}$ determine si $x \sim y$.
9. Repetir el ejercicio anterior para la partición
   $$\mathbb{R} = (-\infty, 3) \cup [3, 7) \cup [7, +\infty).$$
10. Dados (a, b) y (p, q) en $\mathbb{Z} \times \mathbb{Z} - \{(0, 0)\}$, determinar el tipo e implementar funciones que determinen si $(a, b) \sim (p, q)$ para las siguientes relaciones:
    1. $(a, b) \sim (p, q)$ si existe $k \in \mathbb{Z}$ tal que $(a, b) = k(p, q)$
    2. $(a, b) \sim (p, q)$ si existe $k \in \mathbb{R}$ tal que $(a, b) = k(p, q)$

Reducción

¿Cómo ejecuta Haskell?

¿Qué sucede en Haskell si escribo una expresión? ¿Cómo se transforma esa expresión en un resultado?

- Dado el siguiente programa:

```
resta :: Integer -> Integer -> Integer
resta x y = x - y

suma :: Integer -> Integer -> Integer
suma x y = x + y

negar :: Integer -> Integer
negar x = -x
```

- ¿Qué sucede al evaluar la expresión `suma (resta 2 (negar 42)) 4`

Reducción

`suma (resta 2 (negar 42)) 4`

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*re*ducible *ex*pression).
     - Buscamos un redex: $\texttt{suma}\ (\underbrace{\texttt{resta 2 (negar 42)}}_{\text{redex}})\ \texttt{4}$
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - `x` ← `2`
     - `y` ← `(negar 42)`
  4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.
     - `suma (resta 2 (negar 42)) 4` ⇝ `suma (2 - (negar 42)) 4`
  5. Si la expresión resultante aún puede reducirse, volvemos al paso 1.

Órdenes de evaluación en Haskell

Orden normal o lazy (“perezoso”):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝  (3+4) + (suc (2*3))
⇝  7 + (suc (2*3))
⇝  7 + ((2*3) + 1)
⇝  7 + (6 + 1)
⇝  7 + 7
⇝  14
```

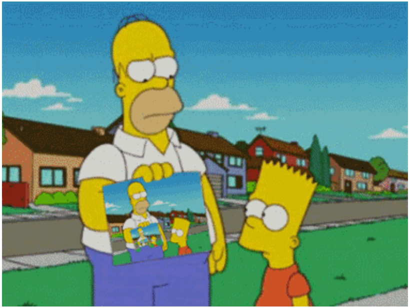
Recursión

- Hasta ahora, especificamos funciones que consistían en "expresiones sencillas".
- ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

$$n! = \prod_{k=1}^{n} k, \qquad n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si no} \end{cases}$$

¡La segunda definición de `factorial` involucra a esta misma función del lado derecho!

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



Sucesiones recursivas

Ejercicios

1. Implementar la función `sc :: Integer -> Integer` definida por

$$sc(n) = \begin{cases} 0 & \text{si } n = 0 \\ sc(n-1) + n^2 & \text{si no} \end{cases}$$

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

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

3. Implementar funciones recursivas para calcular el n-ésimo término de las siguientes sucesiones del Ejercicio 16 y 20 de la Práctica 2.

   1. $a_1 = 2$, $a_{n+1} = 2na_n + 2^{n+1}n!$, para todo $n \in \mathbb{N}$.
   2. $a_1 = 1$, $a_2 = 2$ y $a_{n+2} = na_{n+1} + 2(n+1)a_n$, para todo $n \in \mathbb{N}$.
   3. $a_1 = -3$, $a_2 = 6$ y $a_{n+2} = \begin{cases} -a_{n+1} - 3 & \text{si } n \text{ es impar} \\ a_{n+1} + 2a_n + 9 & \text{si } n \text{ es par} \end{cases}$

Definiciones recursivas

¿Cómo pensar recursivamente?

- Si queremos definir una función recursiva, por ejemplo `factorial`,
  - en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿qué falta para poder obtener el resultado que quiero?
    En este caso, suponemos ya calculado `factorial (n-1)` y lo combinamos multiplicándolo por `n` para lograr obtener `factorial n`.
  - además, identificamos el o los casos base. En el ejemplo de `factorial`, definimos como casos base la función sobre 0: `factorial n | n == 0 = 1`
- Propiedades de una definición recursiva:
  - las llamadas recursivas tienen que *"acercarse"* a un caso base.
  - tiene que tener uno o más casos base que dependerán del tipo de llamado recursivo. Un caso base, es aquella expresión que no tiene paso recursivo.
- En cierto sentido, la recursión es el equivalente computacional de la inducción para las demostraciones.