Recursión

Taller de Álgebra I

Segundo cuatrimestre de 2013

Funciones en Haskell

- Hasta ahora, especificamos funciones que consistían de "expresiones sencillas".
- Sin embargo, el tipo de funciones que hicimos hasta la clase pasada no permite programar todas las funciones computables.

Funciones en Haskell

- Hasta ahora, especificamos funciones que consistían de "expresiones sencillas".
- Sin embargo, el tipo de funciones que hicimos hasta la clase pasada no permite programar todas las funciones computables.
- Por ejemplo, ¿cómo es una función en Haskell para calcular el factorial de un número entero?

Funciones en Haskell

- Hasta ahora, especificamos funciones que consistían de "expresiones sencillas".
- Sin embargo, el tipo de funciones que hicimos hasta la clase pasada no permite programar todas las funciones computables.
- Por ejemplo, ¿cómo es una función en Haskell para calcular el factorial de un número entero?
- `factorial :: Int -> Int`

Funciones en Haskell

- Hasta ahora, especificamos funciones que consistían de "expresiones sencillas".
- Sin embargo, el tipo de funciones que hicimos hasta la clase pasada no permite programar todas las funciones computables.
- Por ejemplo, ¿cómo es una función en Haskell para calcular el factorial de un número entero?
- ```
  factorial ::  Int -> Int
  factorial n | n == 0 =
  ```
  ```

Funciones en Haskell

- Hasta ahora, especificamos funciones que consistían de "expresiones sencillas".
- Sin embargo, el tipo de funciones que hicimos hasta la clase pasada no permite programar todas las funciones computables.
- Por ejemplo, ¿cómo es una función en Haskell para calcular el factorial de un número entero?
- ```
  factorial ::  Int -> Int
  factorial n | n == 0 = 1
  ```
  ```

Funciones en Haskell

- Hasta ahora, especificamos funciones que consistían de "expresiones sencillas".
- Sin embargo, el tipo de funciones que hicimos hasta la clase pasada no permite programar todas las funciones computables.
- Por ejemplo, ¿cómo es una función en Haskell para calcular el factorial de un número entero?
- ```
  factorial ::  Int -> Int
  factorial n | n == 0 = 1
  factorial n | n > 0 =
  ```
  ```

Funciones en Haskell

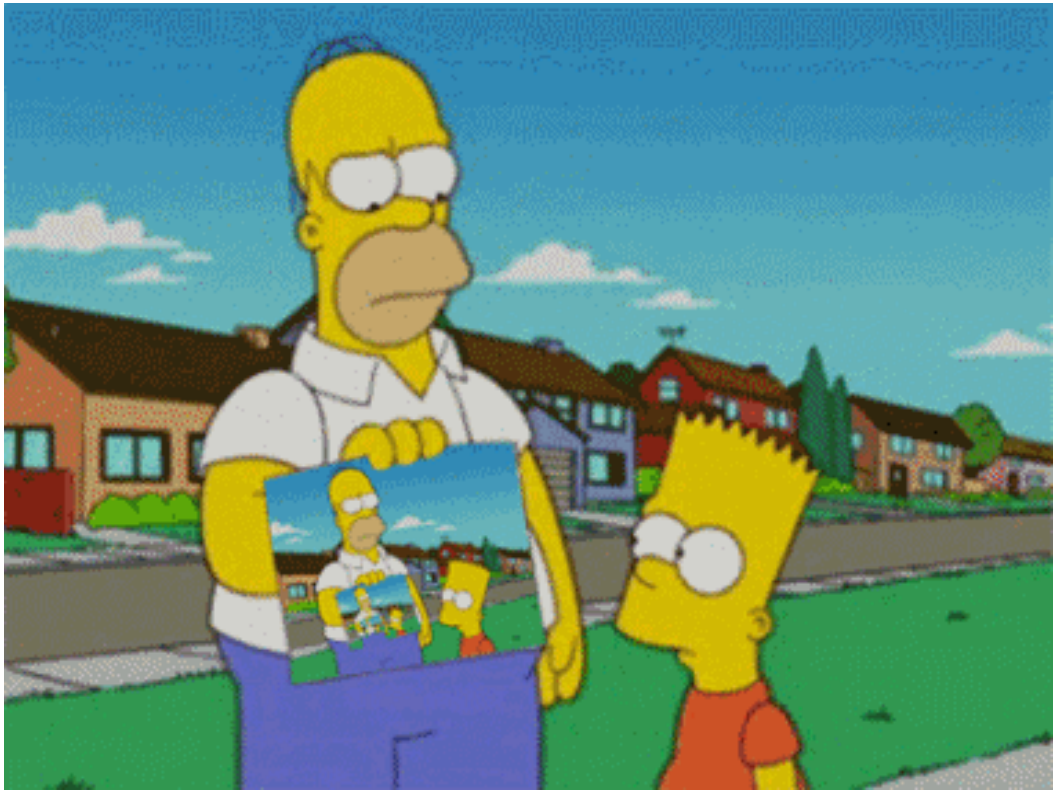
- Hasta ahora, especificamos funciones que consistían de "expresiones sencillas".
- Sin embargo, el tipo de funciones que hicimos hasta la clase pasada no permite programar todas las funciones computables.
- Por ejemplo, ¿cómo es una función en Haskell para calcular el factorial de un número entero?
- ```
  factorial ::  Int -> Int
  factorial n | n == 0 = 1
  factorial n | n > 0 = n * factorial (n-1)
  ```
  ```

Definiciones recursivas

- Notar que la definición de `factorial` involucra a esta misma función del lado derecho de la definición!

Definiciones recursivas

- Notar que la definición de `factorial` involucra a esta misma función del lado derecho de la definición!



Recursión

Definiciones recursivas

- Propiedades de una definición recursiva:
  1. Tiene que tener uno o más casos base.
  2. Las llamadas recursivas del lado derecho tienen que *acercarse* al caso base, con relación a los parámetros del lado izquierdo de la ecuación.

Definiciones recursivas

- Propiedades de una definición recursiva:
  1. Tiene que tener uno o más casos base.
  2. Las llamadas recursivas del lado derecho tienen que *acercarse* al caso base, con relación a los parámetros del lado izquierdo de la ecuación.
- En cierto sentido, la recursión es el equivalente computacional de la inducción para las demostraciones.
- Para aquellos que han programado en lenguajes imperativos: La recursión reemplaza a las estructuras de control iterativas (ciclos: `while`).

Más sobre la función factorial

- Dado que las ecuaciones se evalúan en secuencia, también podemos definir:

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

Más sobre la función factorial

- Dado que las ecuaciones se evalúan en secuencia, también podemos definir:
- ```
  factorial ::  Int -> Int
  factorial 0 = 1
  factorial n = n * factorial (n-1)
  ```
- Observar que `factorial 50 = 0`! ¿Por qué sucede esto?
  ```

Más sobre la función factorial

- Dado que las ecuaciones se evalúan en secuencia, también podemos definir:

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

- Observar que `factorial 50 = 0`! ¿Por qué sucede esto?

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

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

- Podemos programar en Haskell una función que dado n calcule F_n:

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

- Podemos programar en Haskell una función que dado n calcule F_n:
- `fib :: Integer -> Integer`

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

- Podemos programar en Haskell una función que dado n calcule F_n:

- ```
  fib ::  Integer -> Integer
  fib n | n == 0 =
  ```
  ```

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

- Podemos programar en Haskell una función que dado n calcule F_n:

- ```
  fib ::  Integer -> Integer
  fib n | n == 0 = 0
  ```
  ```

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

- Podemos programar en Haskell una función que dado n calcule F_n:

- ```
  fib ::  Integer -> Integer
  fib n | n == 0 = 0
  fib n | n == 1 =
  ```
  ```

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

- Podemos programar en Haskell una función que dado n calcule F_n:

- ```
  fib ::  Integer -> Integer
  fib n | n == 0 = 0
  fib n | n == 1 = 1
  ```
  ```

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

- Podemos programar en Haskell una función que dado n calcule F_n:

- ```
  fib ::  Integer -> Integer
  fib n | n == 0 = 0
  fib n | n == 1 = 1
  fib n | n >= 2 =
  ```
  ```

Otro ejemplo clásico

- Consideremos la sucesión de Fibonacci:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad n \geq 2 \end{aligned}$$

- Podemos programar en Haskell una función que dado n calcule F_n:

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

Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:
- ```
  par ::  Int -> Bool
  par 0 = True
  par n = par (n-2)
  ```
- ¿Qué problema tiene esta función?
  ```

Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:
- ```
  par ::  Int -> Bool
  par 0 = True
  par n = par (n-2)
  ```
- ¿Qué problema tiene esta función?
- ¿Cómo se arregla?
  ```

Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:
- ```
  par ::  Int -> Bool
  par 0 = True
  par n = par (n-2)
  ```
- ¿Qué problema tiene esta función?
- ¿Cómo se arregla?
- ```
  par 0 = True
  par 1 = False
  par n = par (n-2)
  ```
- ```
  par 0 = True
  par n = not (par (n-1))
  ```
  ```

Calculando sumatorias

- **Ejercicio**: (¡para hacer ahora!) Escribir una función que dado $n \in \mathbb{N}$ sume los primeros n números naturales.
- Sabemos que esta suma es $n(n+1)/2$, pero para practicar hagamos esta sumatoria en forma recursiva, sin usar la fórmula!

Calculando sumatorias

- **Ejercicio**: (¡para hacer ahora!) Escribir una función que dado $n \in \mathbb{N}$ sume los primeros n números naturales.
- Sabemos que esta suma es $n(n+1)/2$, pero para practicar hagamos esta sumatoria en forma recursiva, sin usar la fórmula!
- ```
  suma ::  Integer -> Integer
  suma n | n == 0 = 0
  suma n | n > 0 = n + suma (n-1)
  ```
  ```

Calculando sumatorias

- **Ejercicio**: (¡para hacer ahora!) Escribir una función que dado $n \in \mathbb{N}$ sume los primeros n números naturales.
- Sabemos que esta suma es $n(n+1)/2$, pero para practicar hagamos esta sumatoria en forma recursiva, sin usar la fórmula!
- ```
  suma ::  Integer -> Integer
  suma n | n == 0 = 0
  suma n | n > 0 = n + suma (n-1)
  ```
- **Ejercicio**: Escribir una función que dado $n \in \mathbb{N}$ sume los números impares entre los primeros n números naturales.
  ```

Calculando sumatorias

- **Ejercicio**: (¡para hacer ahora!) Escribir una función que dado $n \in \mathbb{N}$ sume los primeros n números naturales.
- Sabemos que esta suma es $n(n+1)/2$, pero para practicar hagamos esta sumatoria en forma recursiva, sin usar la fórmula!
- ```
  suma ::  Integer -> Integer
  suma n | n == 0 = 0
  suma n | n > 0 = n + suma (n-1)
  ```
- **Ejercicio**: Escribir una función que dado $n \in \mathbb{N}$ sume los números impares entre los primeros n números naturales.
- ```
  sumaImpares ::  Integer -> Integer
  sumaImpares n | n == 0 = 0
  sumaImpares n | mod n 2 == 0 = sumaImpares (n-1)
  sumaImpares n | mod n 2 == 1 = n + sumaImpares (n-1)
  ```

Calculando sumatorias

- Podemos generalizar esta función haciendo que tome como parámetro un filtro, que determina qué números debemos sumar:

- ```
  suma ::  Integer -> (Integer -> Bool) -> Integer
  suma n f | n == 0 = 0
  suma n f | f n = n + suma (n-1) f
  suma n f | not (f n) = suma (n-1) f
  ```
  ```

Calculando sumatorias

- Podemos generalizar esta función haciendo que tome como parámetro un filtro, que determina qué números debemos sumar:

- ```
  suma ::  Integer -> (Integer -> Bool) -> Integer
  suma n f | n == 0 = 0
  suma n f | f n = n + suma (n-1) f
  suma n f | not (f n) = suma (n-1) f
  ```

- Para usar esta función ...

- ```
  impar ::  Integer -> Bool
  impar n = mod n 2 == 1
  ```

- ```
  > suma 15 impar
  ```
  ```

Ejercicios

1. Escribir una función para determinar recursivamente si un número es múltiplo de 3. No se puede usar las funciones `mod` y `div` para resolver este ejercicio.
2. Usando la función `suma` de la transparencia anterior, sumar todos los números entre 1 y 1000 que terminen en 12.
3. Escribir una función recursiva que no termine si se ejecuta con números negativos.
4. Escribir una función para calcular $n!! = n(n-2)(n-4)\ldots 1$. La función se debe indefinir para los números impares.
5. Escribir una función para calcular $n!!$, tal que si n es impar calcule $(n-1)!!$
6. Generalizar la función `suma` para que, en vez de sólo sumar $(+)$ los números que pasen el filtro, pueda realizar cualquier otra operación entre `Integers` que uno quiera, pasándole la operación apropiada como un nuevo parámetro.