Taller de Álgebra I

Clase 3 - Recursión

Primer cuatrimestre 2020

Ejercicio Motivador

Supongamos una bacteria cuyo ciclo de vida se organiza en etapas tales que:

- Las bacterias nunca mueren
- Cada bacteria es estéril o reproductora
- Las bacterias estériles pasan a ser reproductoras en la siguiente etapa
- Las bacterias reproductoras nunca vuelven a ser estériles
- Cada bacteria reproductora crea una bacteria estéril en cada etapa

Objetivo:

- Empezando con una bacteria estéril en la etapa 0, determinar cuántas bacterias hay en la etapa *i* para cualquier *i* dado.

Ejemplo:

	0	1	2	3	4	5	6	7	8	9	10	13	14
Reproductoras	0	1	1	2	3	5	8	13	21	34	55	89	144
Bacterias	1	1	2	3	5	8	13	21	34	55	89	144	233

Ejercicio Motivador (cont)

Definición del problema:

- Dado un número *n* que representa el número de etapa...
- ...computar cuántas bacterias existen en la etapa *n*.

Función Haskell: signatura

```
-- |bact(n) indica la cantidad de bacterias en la etapa n
--siguiendo las reglas establecidas en el enunciado
bact :: Int -> Int
```

Idea conceptual:

```
bact 0 = 1
bact 1 = 1
bact 2 = 2
bact 3 = 3
bact 4 = 5
bact 6 = 8
...
```

¡Necesitamos nuevas herramientas para que la computadora trabaje!

Ejercicio Motivador (cont)

Para una etapa dada n, sean:

- B_n la cantidad de bacterias totales en la etapa n
- R_n la cantidad de bacterias reproductoras en la etapa n

	1	2	3	4	5	7	8	9	10	11	12	13	14
R	0	1	1	2	3	5	8	13	21	34	55	89	144
B	1	1	2	3	5	8	13	21	34	55	89	144	233

Relación entre los valores

- Ciertamente, $R_n = B_{n-1}$ para todo $n > 0$, mientras que
- $B_n = B_{n-1} + R_{n-1}$ para todo $n > 0$
- Por lo tanto: $B_n = B_{n-1} + R_{n-1} = B_{n-1} + B_{n-2}$ para todo $n > 1$

Análisis: sabiendo que $B_{20} = 10946$ y $B_{21} = 17711$

- ¿Cuánto vale B_{22}? ¿Y B_{23}? ¿Y B_{19}?
- ¡Tenemos una herramienta de cómputo! (hay que llevarla a Haskell)

Definición recursiva de una función f

- Cuando f se usa para definir algunos de sus valores
- A veces se dice que f es una función recursiva

$$B_n = \begin{cases} 1 & n \leq 1 \\ B_{n-1} + B_{n-2} & n > 1. \end{cases}$$

Siendo una función partida, podemos escribirla en Haskell

```
bact :: Int -> Int
bact n | n <= 1 = 1
       | otherwise = bact (n-1) + bact (n-2)
```

Nota: B es la función de fibonacci porque $B_0, \ldots, B_n, \ldots$ es la sucesión de Fibonacci

Cómputo de funciones recursivas

¿Cómo funciona la recursión en Haskell?

- Ecuación dirigida: `<expresión a reducir> = <expresión reducida>`
- Algoritmo: reemplazar una expresión reducible a su reducción...
- ...hasta que la expresión resultante sea irreducible

```haskell
bact :: Int -> Int
bact n | n <= 1 = 1
       | otherwise = bact (n-1) + bact (n-2)
```

Un posible proceso de reducción

```
bact 4 ⇝ (bact 3)+(bact 2) ⇝ ((bact 2)+(bact 1))+(bact 2) ⇝
(((bact 1)+(bact 0))+(bact 1))+(bact 2) ⇝
((1+(bact 0))+(bact 1))+(bact 2) ⇝ ((1+1)+(bact 1))+(bact 2) ⇝
(2+(bact 1))+(bact 2) ⇝ (2+1)+(bact 2) ⇝ 3+(bact 2) ⇝
3+((bact 1)+(bact 0)) ⇝ 3+(1+(bact 0)) ⇝ 3+(1+1) ⇝ 3+2 ⇝ 5
```

No terminación

Considerar las siguientes funciones y su traducción Haskell

$$\mathrm{ida}(n) = \begin{cases} 0 & n = 0 \\ \mathrm{ida}(n-1)+1 & \end{cases} \qquad \mathrm{idb}(n) = \begin{cases} 0 & n = 0 \\ \mathrm{idb}(n+1)-1 & \end{cases}$$

```
ida :: Int -> Int
ida 0 = 0
ida n = ida (n-1) + 1
```

```
idb :: Int -> Int
idb 0 = 0
idb n = idb (n+1) - 1
```

Matematica vs. Computación (Inteligencia vs. Mecánica)

- Ambas funciones son matemáticamente iguales; ¿por qué?
- Los seres humanos entendemos la ambigüedad; Haskell usa un proceso mecánico

```
ida 1 ⇝ (ida 0) + 1 ⇝ 0 + 1 ⇝ 1
idb 1 ⇝ (idb 2) - 1 ⇝...⇝ (idb n) - (n-1) ⇝...

ida (-1) ⇝ (ida (-2)) + 1 ⇝...⇝ (idb (-n)) + (n-1) ⇝...
idb (-1) ⇝ (idb 0) - 1 ⇝ 0 - 1 ⇝ -1
```

No terminación

Considerar las siguientes funciones y su traducción Haskell

$$\mathrm{ida}(n) = \begin{cases} 0 & n = 0 \\ \mathrm{ida}(n-1)+1 & n \neq 0 \end{cases} \qquad \mathrm{idb}(n) = \begin{cases} 0 & n = 0 \\ \mathrm{idb}(n+1)-1 & n \neq 0 \end{cases}$$

```
ida :: Int -> Int
ida 0 = 0
ida n = ida (n-1) + 1
```

```
idb :: Int -> Int
idb 0 = 0
idb n = idb (n+1) - 1
```

Matematica vs. Computación (Inteligencia vs. Mecánica)

- Ambas funciones son matemáticamente iguales; ¿por qué?
- Los seres humanos entendemos la ambigüedad; Haskell usa un proceso mecánico

```
ida 1 ⇝ (ida 0) + 1 ⇝ 0 + 1 ⇝ 1
idb 1 ⇝ (idb 2) - 1 ⇝...⇝ (idb n) - (n-1) ⇝...

ida (-1) ⇝ (ida (-2)) + 1 ⇝...⇝ (idb (-n)) + (n-1) ⇝...
idb (-1) ⇝ (idb 0) - 1 ⇝ 0 - 1 ⇝ -1
```

No terminación (cont)

No terminación

- Haskell utiliza un proceso mecánico basado en ecuaciones dirigidas
- Los procesos recursivos pueden no finalizar
- Si f no termina para el argumento x, entonces x esta indefinido para f
- Corolario: `ida` e `idb` son funciones distintas y ninguna calcula la identidad

Funciones iguales, aunque con comportamientos diferentes

```
f 0 = 0                    g 0 = 0
f _ = undefined            g x = g x
```

Ejemplos conocidos

- Indefinición con error: pantalla azul de la muerte
- Indefinición por no terminación: “parece que el programa no termina”
- Problema de la parada (de colectivo)

No terminación → casos base

Casos base vs. pasos recursivos

- Caso base de f: ecuación donde f no aparece en el lado derecho
- Paso recursivo de f: ecuación donde f sí aparece en el lado derecho

```
fibo n | n <= 1 = 1                  -- casos base: n = 0 y n = 1
       | otherwise = fibo (n-1) + fibo (n-2)  --pasos recursivos
```

Cadena de reducción

- Las ecuaciones recursivas definen una relación $\leadsto$ de reducción entre los argumentos

  $5 \leadsto 4$, $5 \leadsto 3$, $4 \leadsto 3$, $4 \leadsto 2$, ...
- Los casos base son aquellos que no estan a la izquierda de $\leadsto$
- Una cadena de reducción es una secuencia $x_1, \ldots, x_k$ tal que $x_i \leadsto x_{i+1}$

  $5 \leadsto 3 \leadsto 2 \leadsto 1$, $7 \leadsto 5 \leadsto 4$, $1000 \leadsto 998 \leadsto 996 \leadsto \ldots \leadsto 2 \leadsto 0$

Condición suficiente para terminación (acercarse a los casos base)

Si no existen cadenas de reducción infinitas empezando desde x, entonces la recursión termina para el argumento x

No terminación: ejemplos

¿Qué problemas tienen las siguientes funciones? ¿Cómo se arreglan?

```haskell
-- |Determina si un numero n >= 0 es par
esPar :: Int -> Bool
esPar 0 = True
esPar n = esPar (n-2)

-- |Determina si un numero n cualquiera es par
esPar' :: Int -> Bool
esPar' 0 = True
esPar' 1 = False
esPar' n = esPar (n+2) && esPar (n-2)

unoAdelanteTresAtras :: Int -> Int
unoAdelanteTresAtras 0 = 0
unoAdelanteTresAtras n | esPar n = 1 + unoAdelanteTresAtras (n+1)
                       | otherwise = 1 + unoAdelanteTresAtras (n-3)
```

Terminación: un problema imposible

Tomemos una pausa

- No existe ningún algoritmo que, dada (la escritura de) una función recursiva $f : \mathbb{N} \to \mathbb{N}$ y un valor n, determine si $f(n)$ termina
- Peor aún, tal algoritmo no puede existir (Church, Turing, 1936)
- Caso importante: si $n \rightsquigarrow m$ solo para $n > m$, entonces f termina para todo n
- Generalizado: si existe $v : \mathbb{N} \to \mathbb{N}$ tal que $v(n) > v(m)$ cuando $n \rightsquigarrow m$, entonces f termina para todo n

El problema $3n + 1$ **de Collatz (1937):** ¿la siguiente recursión termina para todo n?

```
collatz 1 = 1
collatz n | n `mod` 2 == 0 = collatz (n `div` 2)
          | otherwise = collatz (3*n+1)
```

Cortocircuitos

Evaluación con cortocircuitos

- Cuando evaluamos `&&` y `||`, Haskell puede ignorar el segundo argumento

  Ejemplo: `False && x == False`, `True || x == True`
- Esto, aunque `x` sea un valor indefinido

Ejecutar

```
False && undefined     ⇝ False
undefined && False     ⇝ undefined
True || undefined      ⇝ True
cuelga = cuelga
False && cuelga        ⇝ False
cuelga && False        ⇝ no termina
```

Explotando cortocircuitos

```
esPar :: Int -> Bool
esPar n = parAux (abs n)
    where parAux n = n == 0 || not (parAux (n-1))
```

Diseño de una función recursiva

¿Cómo diseño una definición recursiva para una función f?

- Objetivo: derivar una ecuación recursiva como en el ejemplo bacterias $\rightarrow$ fibonacci

  Es decir, queremos encontrar una función que se use a si misma en valores mas "chicos"
- Diseño clásico de una solución recursiva:
  1. Defino un orden $x_0, \ldots, x_n, \ldots$ para calcular cada $f(x_i)$
  2. Fijo x_n, $n \gg 0$, y pienso en cómo resolver $f(x_n)$ suponiendo que ya conozco $f(x_0), \ldots, f(x_{n-1})$. Estos valores me fueron comunicados por un oráculo
  3. Defino una ecuación para x_n que utilice sólo $f(x_0), \ldots, f(x_{n-1})$
  4. Defino ecuaciones para los casos base necesarios en forma *ad-hoc*
- En casos difíciles se empieza por 3 y luego se averigua 1.

Ejemplo

- Juego por turnos, en cada turno se pueden sacar 7 u 11 puntos.
- Dado $n \geq 0$, ¿es posible sacar n puntos?
- Reformulando: ¿existen $i, j \geq 0$ tales que $n = 7i + 11j$?

Diseño de una función recursiva

Sea $f(n)$ la función tal que $f(n)$ es True sii existen $i, j \geq 0$ tales que $n = 7i + 11j$

1. Defino un orden $x_0, \ldots, x_n, \ldots$ para calcular cada $f(x_i)$

   Considero el orden $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $\ldots$, $x_i = i$, $\ldots$ ¿por qué?

2. Supongamos que conocemos la respuesta para todo $0 \leq m < n$. Quiero $f(n)$.

   - Como $n \geq 1$, $f(n)$ vale sii $n = 7i + 11j$ para $i + j > 0$.
   - Luego, $f(n)$ vale sii
     a. $i > 0$, $n \geq 7$ y $(n-7) = 7(i-1) + 11j$ o
     b. $j > 0$, $n \geq 11$ y $(n-11) = 7i + 11(j-1)$.
   - Es decir, $f(n) = (n \geq 7$ y $f(n-7))$ o $(n \geq 11$ y $f(n-11))$.

   ```
   f(n) = (n >= 7 && f(n-7)) || (n >= 11 && f(n-11))
   ```

3. Defino los casos base en forma *ad-hoc*. En este caso, $f(0)$.

Función resultante

```haskell
puntajeValido :: Int -> Bool
--opcion 1
puntajeValido 0 = True
puntajeValido n = (n >= 7 && puntajeValido(n-7)) || (n >= 11 &&
    puntajeValido(n-11))
--opcion 2: cortocircuito
puntajevalido n = (n==0) || (n>=7 && puntajeValido(n-7)) || (n>=11
    && puntajeValido(n-11))
```

Ejercitación conjunta

Implementar las siguientes **funciones recursivas**

```
-- |cantDigitos(n) cuenta la cantidad de digitos de n
cantDigitos :: Int -> Int

-- |esBinario(n) = True cuando todos los digitos de n son 0 o 1
esBinario :: Int -> Bool
```

Recursión indirecta

- En rigor, una definición de f es recursiva si f aparece en alguna cadena de reducción...
- ... incluso aunque f no aparezca en el lado derecho de ninguna ecuación

```
esPar :: Int -> Bool                esImpar :: Int -> Bool
esPar 0 = True                      esImpar 0 = False
esPar n = esImpar (n-1)             esImpar n = esPar (n-1)
-- esPar 3 ⇝ esImpar 2 ⇝ esPar 1 ⇝ esImpar 0 ⇝ False
```

Ejemplo 2: recordemos que la cantidad de bacterias en la etapa $n > 0$ es

- $B_n = B_{n-1} + R_{n-1}$, donde
- $R_m = B_{m-1}$ es la cantidad de bacterias reproductoras para $m > 0$

```
bact :: Int -> Int                          repr :: Int -> Int
bact 0 = 1                                  repr 0 = 0
bact n = bact (n-1) + repr (n-1)            repr n = bact (n-1)
```

Recursión con más de un parámetro

Caso general: se extiende lo anterior en forma natural

```
-- Ejemplo de recursion con dos parametros
dosParams :: Int -> Int -> Int
dosParams 0 _ = 0
dosParams _ 0 = 0
dosParams m n | m `mod` 2 == 0 = n + dosParams (m-2) (n+1)
              | otherwise = n + dosParams (m+2) (n-1)
```

- Casos base: `(0, n)` y `(m, 0)` para cualquier `n` y `m`
- Si `m` es par, `(m,n)` $\rightsquigarrow$ `(m-2,n+1)` $\rightsquigarrow \ldots \rightsquigarrow$ `(0,n+m/2)`
- Si no, `(m,n)` $\rightsquigarrow$ `(m+2,n-1)` $\rightsquigarrow \ldots \rightsquigarrow$ `(m+2n,0)`

Caso típico: un parámetro x controla la recursión $\rightarrow$ recursión sobre x

```
-- Ejemplo de recursion sobre el parametro m
parametroMcontrola :: Int -> Int -> Int
parametroMcontrola 0 _ = 0
parametroMcontrola m n = n + (parametroMcontrola (m `div` 2) (2*n))
```

Ejercitación conjunta

Crear una biblioteca de funciones para números naturales que incluya funciones para

- sumar,
- multiplicar,
- restar (en forma saturada),
- comparar por igualdad,
- comparar por mayor y menor,
- conocer el maximo y mínimo de dos números,

utilizando únicamente los siguientes tipos y funciones de `Prelude`

- `Bool(True, False)`, `not`, `&&`, `||`, `otherwise`
- `Int`, `succ`, `pred`

Ejercitación

Ejercicios

En todos los ejercicios los números son naturales (incluyendo 0)

1. Implementar la función `tribonacci :: Int -> Int` que computa

$$T(n) = \begin{cases} n & n \leq 2 \\ T(n-1) + T(n-2) + T(n-3) & n > 2 \end{cases}$$

2. Implementar una función para determinar si un número es múltiplo de 3. No está permitido utilizar `mod` ni `div`
3. Implementar la función `diabolico :: Int -> Bool` que determine si todos los dígitos de un número son 6
4. Extender la función anterior para determinar si todos los dígitos de un número son iguales
5. Implementar las funciones restantes para completar el módulo de los números naturales (alcanza con usar recursión solo para `resta`)
   - a. `resta :: Int -> Int -> Int` es la resta natural $\overset{\circ}{-}$, i.e., $n \overset{\circ}{-} m = \max\{0, n-m\}$
   - b. `menor :: Int -> Int -> Bool` determina si el primer argumento es menor
   - c. `mayor :: Int -> Int -> Bool` determina si el primer argumento es mayor
   - d. `iguales :: Int -> Int -> Bool` determina si dos valores son iguales
6. Implementar una función para determinar si un número n es potencia de otro m. Está permitido utilizar `mod` y `div` (mérito extra por implementarlo con una única ecuación)