PARADIGMAS DE LENGUAJES DE PROGRAMACIÓN

Programación Funcional

Pablo E. "Fidel" Martínez López
(fidel@unq.edu.ar)

“Cuando sepas reconocer la cuatrifolia en todas sus sazones, raíz, hoja y flor, por la vista y el olfato, y la semilla, podrás aprender el verdadero nombre de la planta, ya que entonces conocerás su escencia, que es más que su utilidad.”

Un mago de Terramar

Úrsula K. Le Guin

Programación

- ¿Cuáles son los dos aspectos fundamentales?
  - transformación de información
  - interacción con el medio
- Ejemplos:
  - calcular el promedio de notas de examen
  - cargar datos de un paciente en su historia clínica
- Este curso se concentra en el primer aspecto.

Preguntas

- ¿Cómo saber cuándo dos programas son iguales?
- Ejemplo:
  - ¿Son equivalentes '**f(3)+f(3)**' y '**2*f(3)**'?
  - ¿Siempre?
  - ¿Sería deseable que siempre lo fueran? ¿Por qué?

Ejemplo

- ¿Qué imprime este programa en Javascript?

```
// Test.js   (gentileza de Martín Goffan, 2018)
let x = 0;
function f(y) {  x = x + 1;
                 return x + y; }
console.log(f(3) + f(3));
```

- ¿Y con '**2*f(3)**' en lugar de '**f(3)+f(3)**'?

Valores y Expresiones

- Valores
  - entidades (matemáticas) abstractas con ciertas propiedades
    - **Ejs:** el número dos, el valor de verdad falso.
- Expresiones
  - cadenas de símbolos utilizadas para denotar (escribir, nombrar, referenciar) valores
    - **Ejs:** 2, (1+1), False, (True && False)

Transparencia Referencial

- "El valor de una expresión depende sólo de los elementos que la constituyen."
- Implica:
  - consideración sólo del comportamiento externo de un programa (abstracción de detalles de ejecución).
  - posibilidad de demostrar propiedades usando las propiedades de las subexpresiones y métodos de deducción lógica.

Expresiones

- Expresiones atómicas
  - son las expresiones más simples
  - llamadas también formas normales
  - por abuso de lenguaje, les decimos *valores*
    - **Ejs:** 2, False, (3,True)
- Expresiones compuestas
  - se 'arman' combinando subexpresiones
  - por abuso de lenguaje, les decimos *expresiones*
    - **Ejs:** (1+1), (2==1), (4 - 1, True || False)

Expresiones

- Puede haber expresiones incorrectas ("mal formadas")
  - por errores sintácticos

    `*12` `(True` `('a',)`
  - por errores de tipo

    `(2+False)` `(2||'a')` `4 'b'`
- ¿Cómo saber si una expresión está "bien formada"?
  - Reglas sintácticas
  - Reglas de asignación de tipo

Funciones

- Valores especiales, que representan "transformación de datos"
- Dos formas de entender las funciones
  - VISIÓN DENOTACIONAL
    - una función es un valor matemático que relaciona cada elemento de un conjunto (de partida) con un único elemento de otro conjunto (de llegada).
  - VISIÓN OPERACIONAL
    - una función es un mecanismo (método, procedimiento, algoritmo, programa) que dado un elemento del conjunto de partida, calcula (devuelve, retorna) el elemento correspondiente del conjunto de llegada.

Funciones

- Ejemplo: doble x = x+x
- Visión denotacional
  - a cada número x, doble le hace corresponder otro número, cuyo valor es la suma de x más x

    { (0,0), (1,2), (2,4), (3,6), ... }
- Visión operacional
  - dado un número x, doble retorna ese número sumado consigo mismo

    doble 0 $\rightarrow$ 0 doble 1 $\rightarrow$ 2

    doble 2 $\rightarrow$ 4 doble 3 $\rightarrow$ 6 ...

Funciones

- ¿Cuál es la operación básica de una función?
  - la APLICACIÓN a un elemento de su partida
- Regla sintáctica:
  - la aplicación se escribe por yuxtaposición
    - (f x) denota al elemento que se corresponde con x por medio de la función f.
    - **Ej:** (doble 2) denota al número 4

Funciones

- ¿Qué expresiones denotan funciones?
  - Nombres (variables) definidos como funciones
    - **Ej**: doble
  - Funciones anónimas (lambda abstracciones)
    - **Ej:** (\x -> x+x)
  - Resultado de usar otras funciones
    - **Ej:** doble . doble

Ecuaciones Orientadas

- Dada una expresión bien formada, ¿cómo determinamos el valor que denota?
  - Mediante ECUACIONES que establezcan su valor
- ¿Y cómo calculamos el valor de la misma?
  - Reemplazando subexpresiones, de acuerdo con las reglas dadas por las ecuaciones (REDUCCIÓN)
    - Por ello usamos ECUACIONES ORIENTADAS

Ecuaciones Orientadas

- Expresión-a-definir = expresión-definida

  e1 = e2

- Visión denotacional
  - se define que el valor denotado por e1 (su significado) es el mismo que el valor denotado por la expresión e2
- Visión operacional
  - para calcular el valor de una expresión que contiene a e1, se puede reemplazar e1 por e2

Programas Funcionales

- Definición de programa funcional (*script*):
  - Conjunto de ecuaciones que definen una o más funciones (valores).
- Uso de un programa funcional
  - Reducción de la aplicación de una función a sus datos (reducción de una expresión).

Funciones como valores

- Las funciones son valores, al igual que los números, las tuplas, etc.
  - pueden ser argumento de otras funciones
  - pueden ser resultado de otras funciones
  - pueden almacenarse en estructuras de datos
  - pueden ser estructuras de datos
- Funciones que manipulan funciones
  - Las llamamos “de alto orden”, abusando de esa nomenclatura

Funciones como valores

- Ejemplos

```
doble x = x + x
cuadruple x = 4 * x
sqr x = x*x
twice f = g   where g x = f (f x)
fs = [ sqr, doble, cuadruple, twice doble ]
```

(twice doble) 2 → ?

- ¿Será cierto que cuadruple = twice doble?

Lenguaje Funcional Puro

- Definición de lenguaje funcional puro:

"lenguaje de expresiones con transparencia referencial y funciones como valores, cuyo modelo de cómputo es la reducción realizada mediante el reemplazo de iguales por iguales"

Tipos

- Toda expresión válida denota un valor
- Todo valor pertenece a un conjunto
- Los tipos denotan conjuntos
- Entonces...

  TODA EXPRESIÓN DEBERÍA TENER UN TIPO PARA SER VÁLIDA

  (si una expresión no tiene tipo, es inválida)

Asignación de Tipos

- Notación: e :: A
  - se lee "la expresión e tiene tipo A"
  - significa que el valor denotado por e pertenece al conjunto de valores denotado por A
- Ejemplos:

2 :: Int	False :: Bool
'a' :: Char	doble :: Int -> Int
[sqr, doble] :: [Int -> Int]	

Asignación de tipos

- Se puede deducir el tipo de una expresión a partir de su constitución
- Algunas reglas
  - si e1 :: A y e2 :: B, entonces (e1,e2) :: (A,B)
  - si m, n :: Int, entonces (m+n) :: Int
  - si f :: A->B y e :: A, entonces f e :: B
  - si d = e y e :: A, entonces d :: A

Asignación de tipos

- Se puede deducir el tipo de una expresión a partir de su constitución (cont.)
- La regla más importante es

$$\frac{\begin{array}{l}\boxed{f} :: \boxed{A} \text{->} \boxed{B} \\ \boxed{e} :: \boxed{A}\end{array}}{\boxed{f}\,\_\,\boxed{e} :: \boxed{B}}$$

  - Dice que si tenemos una aplicación, la parte izquierda debe ser una función, y el tipo del parámetro debe coincidir con el tipo del argumento

Asignación de tipos

- Ejemplo: doble x = x+x
  - Por la primera parte

$$\begin{array}{l} \boxed{\texttt{doble}} :: \boxed{} \texttt{->} \boxed{} \\ \boxed{\texttt{x}} :: \boxed{} \\ \hline \boxed{\texttt{doble}}\ \boxed{\texttt{x}} :: \boxed{} \end{array}$$

  - y por la segunda parte x+x :: Int,
    y eso solamente si x :: Int
  - Además, doble x y x+x tienen el mismo tipo.

Asignación de tipos

- Ejemplo: doble x = x+x
  - Entonces, continuando y volcando lo inferido

$$\frac{\boxed{\text{doble}} :: \boxed{\text{Int}} \to \boxed{\text{Int}} \qquad \boxed{\text{x}} :: \boxed{\text{Int}}}{\boxed{\text{doble}}\ \boxed{\text{x}} :: \boxed{\text{Int}}}$$

  - De esto puedo deducir que

doble :: Int -> Int

Asignación de tipos

- Ejemplo: doble x = x+x
  twice' (f,y) = f (f y)
  - x+x :: Int, y entonces sólo puede ser que x :: Int
  - doble x :: Int y x :: Int, entonces sólo puede ser que doble :: Int -> Int
  - si y :: A y f :: A -> A, entonces f y :: A, f (f y) :: A
  - como twice' (f,y) :: A, y (f,y) :: (A->A, A), sólo puede ser que twice' :: (A->A, A) -> A

Asignación de tipos

- Propiedades deseables
  - que sea automática (que haya un programa)
  - que le dé tipo al mayor número posible de expresiones con sentido
  - que no le dé tipo al mayor número posible de expresiones sin sentido
  - que se conserve por reducción
  - que los tipos sean descriptivos y razonablemente sencillos de leer

Asignación de tipos

- Inferencia de tipos
  - dada una expresión e, determinar si tiene tipo o no según las reglas, y cuál es ese tipo
- Chequeo de tipos
  - dada una expresión e y un tipo A, determinar si e :: A según las reglas, o no
- Sistema de tipado fuerte (strong typing)
  - sistema que acepta una expresión si, y sólo si ésta tiene tipo según las reglas y tal que las expresiones aceptadas nunca fallan por problemas de tipos

Sistema de tipos

- ¿Para qué sirven los tipos?
  - detección de errores comunes
  - documentación
  - especificación rudimentaria
  - oportunidades de optimización en compilación
- Es una buena práctica en programación empezar dando el tipo del programa que se quiere escribir.

Sistema Hindley-Milner

- Tipos básicos
  - enteros Int
  - caracteres Char
  - booleanos Bool
- Tipos compuestos
  - tuplas (A,B)
  - listas [A]
  - funciones (A->B)
- Variables (polimorfismo) a,b,c,...

Polimorfismo

- ¿Qué tipo tendrá la siguiente función?

  twice :: ??

  twice f = g where g x = f (f x)

    twice doble :: ??

    twice not :: ??

- ¿Es una expresión con sentido?
- ¿Debería tener un tipo?
- En realidad:

  twice :: (A->A)->(A->A), cualquiera sea A

Polimorfismo paramétrico

- Solución: ¡variables de tipo!

  twice :: (a -> a) -> (a -> a)

  se lee: "twice es una función que dado una función de *algún tipo* a->a, retorna otra función de ese mismo tipo"

- Esta es una función *polimórfica*
  - el tipo de su argumento puede ser *instanciado* de diferentes maneras en diferentes usos

  twice doble :: Int->Int y aquí twice :: (Int->Int) -> (Int->Int)

  twice not :: Bool->Bool y aquí twice :: (Bool->Bool) -> (Bool->Bool)

Polimorfismo paramétrico

- Polimorfismo
  - Característica del sistema de tipos
  - Dada una expresión que puede ser tipada de infinitas maneras, el sistema puede asignarle un tipo que sea más general que todos ellos, y tal que en cada uso pueda transformarse en uno particular.
  - Ej: twice :: (a->a) -> (a->a) ← más general
    twice :: (Int->Int) -> (Int->Int)
    twice :: (Bool->Bool) -> (Bool->Bool) ... ← particulares
    - Reemplazando a por Int, por ejemplo, se obtiene un tipo particular
  - Se llama "paramétrico" pues a es el *parámetro*.

Polimorfismo paramétrico

- ¿Tienen tipo las siguientes expresiones? ¿Cuáles?
  (Recordar: twice f = g where g x = f (f x))

  twice :: ??

  (twice doble) 3 :: ??

  (twice twice) doble :: ??

  twice twice :: ??

  (twice twice) twice :: ??

  ((twice twice) twice) doble :: ??


¿VOS SABÍAS QUE MI MAMÁ ES TRADUCTORA DE FRANCÉS, MANOLITO? YO TAMBIÉN SÉ FRANCÉS; SÉ DECIR "PAPÁ" EN FRANCÉS
¿SÍÍ? ¿CÓMO SE DICE, A VER?
PAPÁ
¡AH, ES FÁCIL! ¡SE DICE IGUAL!
¿FÁCIL? ¿IGUAL? ¡NADA DE ESO, EL ASUNTO ES PENSAR EN FRANCÉS! ¡TRATÁ DE DECIR "PAPÁ" PENSÁNDOLO EN FRANCÉS! ¡DALE! ¿A VER? ¡DALE!
¡ES INÚTIL! ¡JAMÁS PODRÉ HABLAR ESE MALDITO IDIOMA!

Aplicación del alto orden

- Considere las siguientes definiciones

```
suma' :: ??
suma' (x,y) = x+y

suma :: ??
suma x = f   where  f y = x+y
```

- ¿Qué tipo tienen las funciones?
- ¿Qué similitudes observa entre suma y suma'?
- ¿Qué diferencias observa entre ellas?

Aplicación del alto orden

- Similitudes
  - ambas retornan la suma de dos enteros:
    suma' (x,y) = (suma x) y, para x e y cualesquiera
- Diferencias
  - una toma un par y retorna un número;
    la otra toma un número y retorna una *función*
  - con suma se puede definir la función sucesor sin usar variables extra:
    succ = suma 1

Curificación

- Correspondencia entre cada función de múltiples parámetros y una de alto orden que retorna una función intermedia que completa el trabajo.
  - Por cada f’ definida como

```
f’ :: (a,b) -> c
f’ (x,y) = e
```

  siempre se puede escribir

```
f :: a -> (b -> c)
(f x) y = e
```

Currificación

- Correspondencia entre los tipos

```
(a,b) -> c      y      a -> (b -> c)

curry :: ((a,b) -> c) -> (a -> (b -> c))
curry …

uncurry :: (a -> (b -> c)) -> ((a,b) -> c)
uncurry …
```

  - Se puede demostrar que

```
curry (uncurry f) = f
uncurry (curry f') = f'
```

Currificación - Sintaxis

- ¿Cómo escribimos una función currificada y su aplicación?
- Considerar las siguientes definiciones

```
twice :: (Int->Int) -> (Int -> Int)
twice1 f = g where g x = f (f x)
twice2 f = \x -> f (f x)
(twice3 f) x = f (f x)
```

- ¿Son equivalentes? ¿Cuál es preferible? ¿Por qué?

Currificación

- ¿Qué pasa con un ejemplo más grande?
  - Consideremos una función para sumar 5 números

```
sum5' :: (Int, Int, Int, Int, Int) -> Int
sum5' (x,y,z,v,w) = x+y+z+v+w
```

vs.

```
sum5 :: ??
sum5 = ??
```

Currificación

- ¿Qué pasa con un ejemplo más grande? (cont.)
  - Con nombres intermedios...

```
sum5 :: Int -> (Int -> (Int -> (Int -> (Int -> Int))))
sum5 x = sum4
        where sum4 y = sum3
          where sum3 z = sum2
            where sum2 v = sum1
              where sum1 w = x+y+z+v+w
```

Currificación

- ¿Qué pasa con un ejemplo más grande? (cont.)
  - Con aplicación reiterada...

```
sum5 :: Int -> (Int -> (Int -> (Int -> (Int -> Int))))
((((sum5 x) y) z) v) w = x+y+z+v+w
```

vs.

```
sum5' :: (Int, Int, Int, Int, Int) -> Int
sum5' (x,y,z,v,w) = x+y+z+v+w
```

Currificación

- ¿Cómo podemos evitar usar paréntesis? Convenciones de notación
  - La aplicación de funciones asocia a izquierda
  - El tipo de las funciones asocia a derecha

suma :: Int ->Int ->Int	suma :: Int -> (Int ->Int)
suma x y = x+y	(suma x) y = x+y

Currificación

- Por abuso de lenguaje

  suma :: Int ->Int ->Int

  suma x y = x+y

  suma es una función que toma dos enteros y retorna otro entero.

  en lugar de

  suma :: Int -> (Int -> Int)

  (suma x) y = x+y

  suma es una función que toma un entero y devuelve una función, la cual toma un entero y devuelve otro entero.

Currificación

- Ventajas.
  - Mayor expresividad

    ```
    derive :: (Int -> Int) -> (Int -> Int)
    derive f x = (f (x+h) - f x) / h   where h = 0.0001
    ```

  - Aplicación parcial

    ```
    derive f            ( = \x -> (f (x+h) - f x) / h )
    ```

  - Modularidad para tratamiento de código
    - Al inferir tipos
    - Al transformar programas

Aplicación Parcial

- Definir un función que calcule la derivada n-ésima de una función

```
deriveN :: Int -> (Int -> Int) -> (Int -> Int)
deriveN 0 f = f
deriveN n f = deriveN (n-1) (derive f)
```

Aplicación parcial de derive.

- ¿Cómo lo haría con derive'?

Expresividad

- Definir un función que calcule la aplicación n veces de una función

```
many :: Int -> (a -> a) -> (a -> a)
many 0 f x = x
many n f x = many (n-1) f (f x)
```

- Se pueden probar (o definir) muchas ideas ya vistas...

```
twice = many 2
deriveN n = many n derive
```

Currificación

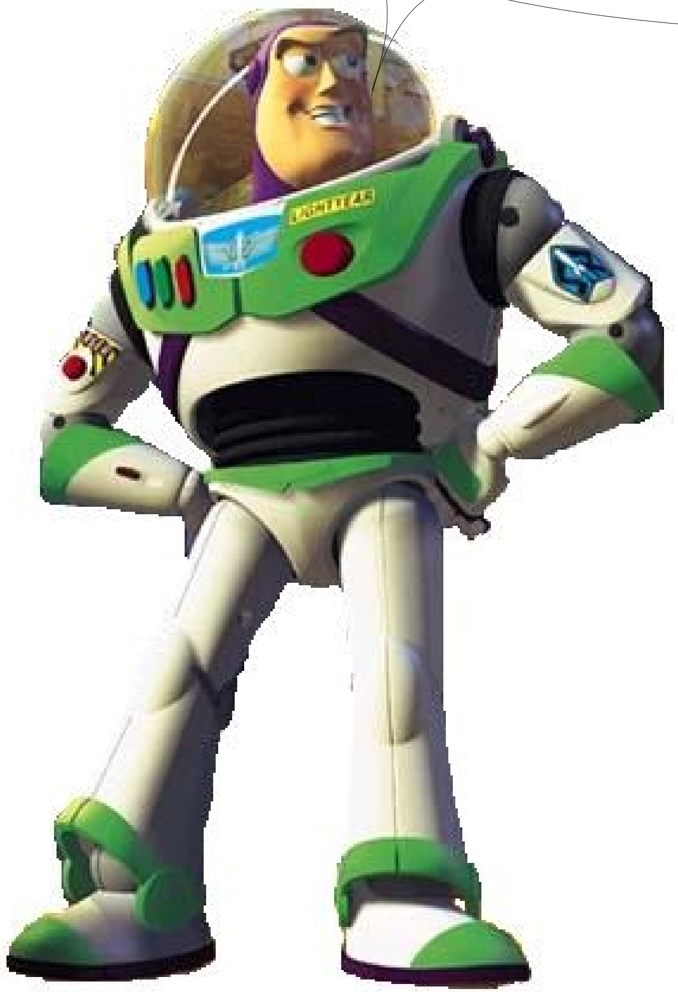
- Decir que algo está currificado es una CUESTIÓN DE INTERPRETACIÓN

```
movePoint :: (Int, Int) -> (Int, Int)
movePoint (x,y) = (x+1,y+1)

distance :: (Int, Int) -> Int
distance (x,y) = sqrt (sqr x + sqr y)
```

- ¿Están currificadas? ¿Por qué?

To infinity and beyond!



Otros conjuntos

- Los tipos definidos hasta ahora, ¿son suficientes para la tarea de programar?
  - por ejemplo, si estuvieramos programando un intérprete para un lenguaje de programación, ¿cómo representaríamos un programa?
- ¿Cómo definimos conjuntos con infinitos elementos?
- ¿Cómo definimos funciones recursivas que no se cuelguen?
- ¿Cómo probamos propiedades de estos conjuntos?

Inducción/Recursión

- Para solucionar los tres problemas, usaremos INDUCCIÓN
- La inducción es un mecanismo que nos permite:
  - Definir conjuntos infinitos
  - Probar propiedades sobre sus elementos
  - Definir funciones recursivas sobre ellos, con garantía de terminación

Inducción estructural

- Una definición inductiva de un conjunto $\Re$ consiste en dar condiciones de dos tipos:
  - reglas base ($z \in \Re$)
    - que afirman que algún elemento simple x pertenece a $\Re$
  - reglas inductivas ($y_1 \in \Re, \ldots, y_n \in \Re \Rightarrow y \in \Re$)
    - que afirman que un elemento compuesto y pertenece a $\Re$ siempre que sus partes $y_1, \ldots, y_n$ pertenezcan a $\Re$ (e y no satisface otra regla de las dadas)

  y pedir que $\Re$ sea el menor conjunto (en sentido de la inclusión) que satisfaga todas las reglas dadas.

Funciones recursivas

- Sea S un conjunto inductivo, y T uno cualquiera. Una definición recursiva *estructural* de una función f :: S -> T es una definición de la siguiente forma:
  - por cada elemento base z, el valor de (f z) se da directamente usando valores previamente definidos
  - por cada elemento inductivo y, con partes inductivas y1, ..., yn, el valor de (f y) se da usando valores previamente definidos ***y*** los valores (f y1), ..., (f yn).

Principio de inducción

- Sea S un conjunto inductivo, y sea P una propiedad sobre los elementos de S. ***Si se cumple que***:
  - para cada elemento $z \in$S tal que z cumple con una regla base, P(z) es verdadero, y
  - para cada elemento $y \in$S construído en una regla inductiva utilizando los elementos y_1, ..., y_n, si P(y_1), ..., P(y_n) son verdaderos entonces P(y) lo es,

entonces P(x) se cumple para todos los $x \in$S.

$\forall x.$P(x) se demostró por *inducción estructural* en x

“...de más está decir que rehusarse a explotar este poder de las matemáticas concretas equivale a suicidio intelectual y tecnológico. La moraleja de la historia es: traten a todos los elementos de un conjunto ignorándolos y trabajando con la definición del conjunto.”

On the cruelty of really teaching computing science
(EWD 1036)
Edsger W. Dijkstra

Ejemplo: LISTAS

- Dado un tipo cualquiera a, definimos inductivamente al conjunto [a] con las siguientes reglas:
  - [] :: [a]
  - si x :: a y xs :: [a] entonces x:xs :: [a]
- ¿Qué elementos tiene [Bool]? ¿Y [Int]?
- Notación:

  $[\ x_1, x_2, x_3\ ] = (x_1 : (x_2 : (x_3 : [\ ])))$

Ejemplo: LISTAS

- Definir por recursión una función len que cuente los elementos de una lista.

```
len :: [a] -> Int
  -- Por inducción en la estructura de la lista xs
len [ ]      = ...
len (x:xs)  = ...  len xs ...
```

Aplicación inductiva de len

Caso base

Caso inductivo

Ejemplo: LISTAS

- Definir por recursión una función len que cuente los elementos de una lista.

```
len :: [a] -> Int
  -- Por inducción en la estructura de la lista xs
len [ ]       = 0
len (x:xs)  = 1+ len xs
```

Aplicación inductiva de len

Caso base

Caso inductivo

Funciones sobre listas

- Siguiendo el patrón de recursión

```
len :: [a] -> Int
len [ ]      = 0
len (x:xs) = 1 + len xs

append :: [a] -> [a] -> [a]
append [ ]      ys = ys
append (x:xs) ys = x : (append xs ys)

(++) = append
```

Funciones sobre listas

- Sin seguir el patrón de recursión

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs

null :: [a] -> Bool
null [ ]       = True
null (x:xs)  = False
```

Funciones sobre listas

- Más funciones siguiendo el patrón de recursión

```
sum :: [Int] -> Int
sum [ ]      = ...
sum (n:ns) = ... n ... sum ns ...

prod :: [ Int ] -> Int
prod [ ]      = ...
prod (n:ns) = ... n ... prod ns ...
```

- ¿Cómo definir (sum [ ]) y (prod [ ])?

Funciones sobre listas

- Más funciones siguiendo el patrón de recursión

```
sum :: [Int] -> Int
sum [ ]       = 0
sum (n:ns) = n + sum ns

prod :: [ Int ] -> Int
prod [ ]       = 1
prod (n:ns) = n * prod ns
```

- ¿Por qué se puede definir (sum [ ]) y (prod [ ]) de esta manera?

Funciones sobre listas

- Más funciones siguiendo el patrón de recursión

```
upperl :: [Char] -> [Char]
upperl [ ]      = ...
upperl (c:cs)   = ... c ... (upperl cs)

novacias :: [ [ a ] ] -> [ [ a ] ]
novacias [ ]         = ...
novacias (xs:xss)    = ... xs ... novacias xss
```

Funciones sobre listas

- Más funciones siguiendo el patrón de recursión

```
upperl :: [Char] -> [Char]
upperl [ ]      = [ ]
upperl (c:cs)   = (upper c) : (upperl cs)

novacias :: [ [ a ] ] -> [ [ a ] ]
novacias [ ]         = [ ]
novacias (xs:xss)    = if null xs  then novacias xss
                                   else xs : novacias xss
```

Funciones sobre listas

- Siguiendo otro patrón de recursión

```
maximum :: [ a ] -> a
maximum [ x ]     = x
maximum (x:xs) = x `max` maximum xs

last :: [ a ] -> a
last [ x ]     = x
last (x:xs) = last xs
```

  - ¿puede establecer cuál es el patrón?
  - ¿por qué (maximum [ ]) no está definida?

Funciones sobre listas

- Otras funciones

```
reverse :: [ a ] -> [ a ]
reverse [ ]       = ...
reverse (x:xs) = ... reverse xs ... x ...

insert :: a -> [ a ] -> [ a ]
insert x [ ]       = ...
insert x (y:ys) = ... y ... x ... ys ... (insert x ys) ...
```

Funciones sobre listas

- Otras funciones

```
reverse :: [ a ] -> [ a ]
reverse [ ]       = [ ]
reverse (x:xs) = reverse xs ++ [ x ]

insert :: a -> [ a ] -> [ a ]
insert x [ ]       = [ x ]
insert x (y:ys) = if x <= y  then x : (y : ys)
                             else y : (insert x ys)
```

“'Detrás de cada acontecimiento se esconde un truco de espejos. Nada es, todo parece. Escóndase, si quiere. Espíe por las ranuras. Alguien estará preparando otra ilusión. Las diferencias entre las personas son las diferencias entre las ilusiones que perciben.'

(Consejero, 121:6:33)”

El Fondo del Pozo
Eduardo Abel Giménez

Parametrización

- ¿Qué es un parámetro? Consideremos

  f x = x + $\boxed{1}$ (ó f = \x -> x + $\boxed{1}$)

  g x = x + $\boxed{17}$ (ó g = \x -> x + $\boxed{17}$)

  h x = x + $\boxed{42}$ (ó h = \x -> x + $\boxed{42}$)

  - Sólo las partes recuadradas son distintas ¿podremos aprovechar ese hecho?

Parametrización

- ¿Qué es un parámetro?

$f\ x = x + \boxed{1}$

$g\ x = x + \boxed{17}$

$h\ x = x + \boxed{42}$

Abstracción

$(\mathrm{genF}\ y)\ x = x + \boxed{y}$

Instanciación

$f = \mathrm{genF}\ \boxed{1}$

$g = \mathrm{genF}\ \boxed{17}$

$h = \mathrm{genF}\ \boxed{42}$

  - Sólo las partes recuadradas son distintas
    ¿podremos aprovechar ese hecho?
  - Técnica de los “recuadros”
  - **Parámetro**: valor que cambia en cada uso

Esquemas de funciones

- Probemos con funciones sobre listas
  - Escribir las siguientes funciones:

```
succl :: [ Int ] -> [ Int ]
-- suma uno a cada elemento de la lista
upperl :: [ Char ] -> [ Char ]
-- pasa a mayúsculas cada carácter de la lista
test :: [ Int ] -> [ Bool ]
-- cambia cada número por un booleano que
-- dice si el mismo es cero o no
```

  - ¿Observa algo en común entre ellas?

Esquemas de funciones

- Solución:

```
succl [ ]        = ...
succl (n:ns)     = ... n ... succl ns ...

upperl [ ]       = ...
upperl (c:cs)    = ... c ... upperl cs ...

test [ ]         = ...
test (x:xs)      = ... x ...  test xs ...
```

  - Usamos el esquema de recursión estructural sobre listas

Esquemas de funciones

- Solución:

```
succl [ ]        = [ ]
succl (n:ns)     = ( n +1)  :  succl ns

upperl [ ]       = [ ]
upperl (c:cs)    =  upper  c   :  upperl cs

test [ ]         = [ ]
test (x:xs)      = ( x ==0)  :  test xs
```

  - Sólo las partes recuadradas son distintas... pero los círculos rojos “molestan”

Esquemas de funciones

- Solución:

```
succl [ ]        = [ ]
succl (n:ns)     = (n+1) :  succl ns

upperl [ ]       = [ ]
upperl (c:cs)    = upper c :  upperl cs

test [ ]         = [ ]
test (x:xs)      = (x==0) :  test xs
```

  - Sólo las partes recuadradas son distintas... pero los círculos rojos “molestan”

Esquemas de funciones

- Técnica de los "recuadros" (extendida)

```
succl [ ]        = [ ]
succl (n:ns)     = (\n' -> n'+1) n : succl ns
upperl [ ]       = [ ]
upperl (c:cs)    = (\c' -> upper c') c : upperl cs
test [ ]         = [ ]
test (x:xs)      = (\n -> n ==0) x : test xs
```

  - Reescribimos los recuadros (azules) para que no dependan del contexto (círculos rojos)

Esquema de map

- Procedemos con la abstracción:

```
map :: ??
map   [ ]      =  [ ]
map   (x:xs) = [ ] x : map   xs
```

Esquema de map

- Completamos la definición

```
map :: ??
map f [ ]      =  [ ]
map f (x:xs) = f x : map f xs
```

Esquema de map

- Completamos la definición

```
map :: ??
map f [ ]      =  [ ]
map f (x:xs) = f x : map f xs
```

- Y entonces

```
succl'  = map (\n' -> n'+1)
upperl' = map upper
test'   = map (==0)
```

Esquema de map

- Agregamos el tipo

```
map :: (a -> b) -> [a] -> [b]
map f [ ]      =  [ ]
map f (x:xs) = f x : map f xs
```

- Y entonces

```
succl'  = map (\n' -> n'+1)
upperl' = map upper
test'   = map (==0)
```

Esquema de map

- Agregamos el tipo

```
map :: (a -> b) -> ([a] -> [b])
map f [ ]      =  [ ]
map f (x:xs) = f x : map f xs
```

- Y entonces

```
succl'   = map (\n' -> n'+1)
upperl' = map upper
test'     = map (==0)
```

- ¿Podría probar que succl' = succl? ¿Cómo?

Esquema de map

- **Demostración**
  - Por principio de extensionalidad, probamos que para toda lista finita xs, succl' xs = succl xs por inducción en la estructura de la lista.
  - Caso base: xs = [ ]
    - Usar succl', map.1, y succl.1
  - Caso inductivo: xs = x:xs'
    - Usar succl', map.2, succl', HI, y succl.2
- ¡Observar que no estamos contemplando el caso ⊥ ni el de listas no finitas, o con elementos ⊥!

Esquemas de funciones

- Una vez más, con otras funciones
  - Escribir las siguientes funciones:

masQueCero :: [Int] -> [Int]
-- retorna la lista que sólo contiene los números
-- mayores que cero, en el mismo orden

digitos :: [Char] -> [Char]
-- retorna los caracteres que son dígitos

noVacias :: [[a]] -> [[a]]
-- retorna sólo las listas no vacías

  - ¿Observa algo en común entre ellas?

Esquemas de funciones

- Solución:

```
digitos [ ]       = ...
digitos (c:cs)  =
               ... c ... digitos cs ...

noVacias [ ]          = ...
noVacias (xs:xss) =
               ... xs ... noVacias xss ...
```

  - Siempre recursión estructural

Esquemas de funciones

- Solución:

```
digitos [ ]        = [ ]
digitos (c:cs)  =
   if  (isDigit  c )  then c : digitos cs
                          else digitos cs

noVacias [ ]          = [ ]
noVacias (xs:xss) =
   if  (null  xs )  then noVacias xss
                       else xs : noVacias xss
```

  - Otra vez, técnica de los “recuadros” extendida

Esquemas de funciones

- Solución:

```
digitos [ ]       = [ ]
digitos (c:cs)  =
   if (isDigit c) then c : digitos cs
                  else digitos cs

noVacias [ ]          = [ ]
noVacias (xs:xss) =
   if (null xs) then noVacias xss
                else xs : noVacias xss
```

  - Otra vez, técnica de los “recuadros” extendida

Esquemas de funciones

- Solución:

```
digitos [ ]      = [ ]
digitos (c:cs)  =
   if (\c' -> isDigit c') c  then c : digitos cs
                             else digitos cs

noVacias [ ]        = [ ]
noVacias (xs:xss) =
   if (\xs' -> not (null xs')) xs  then xs : noVacias xss
                                   else noVacias xss
```

- Observar el cambio en el if de noVacias para que ambas funciones se parezcan

Esquema de filter

- Procedemos con la abstracción

```
filter :: ??
filter    [ ]       = [ ]
filter    (x:xs)  = if  (□ x)  then x : filter   xs
                                else filter   xs
```

Esquema de filter

- Completamos la definición

```
filter :: ??
filter p [ ]        = [ ]
filter p (x:xs) = if  (p x)  then x : filter p xs
                              else filter p xs
```

- Y entonces

```
masQueCero' = filter (>0)
digitos'        = filter isDigit
noVacias'      = filter (not . null)
```

Esquema de filter

- Agregamos el tipo

```
filter :: (a->Bool) -> [a] -> [a]
filter p [ ]      = [ ]
filter p (x:xs) = if  (p x)  then x : filter p xs
                             else filter p xs
```

- Y entonces

```
masQueCero' = filter (>0)
digitos'    = filter isDigit
noVacias'   = filter (not . null)
```

Esquema de filter

- Agregamos el tipo

```
filter :: (a->Bool) -> ([a] -> [a])
filter p [ ]       = [ ]
filter p (x:xs)  = if  (p x)  then x : filter p xs
                               else filter p xs
```

- Y entonces

```
masQueCero' = filter (>0)
digitos'        = filter isDigit
noVacias'      = filter (not . null)
```

- ¿Podría probar que noVacias' = noVacias?

Esquemas de funciones

- Una vez más, con más complejidad
  - Escribir las siguientes funciones:

```
sonCincos :: [ Int ] -> Bool
-- dice si todos los elementos son 5

cantTotal :: [ [a] ] -> Int
-- dice cuántos elementos de tipo a hay en total

concat :: [ [a] ] -> [ a ]
-- hace el append de todas las listas en una
```

  - ¿Observa algo en común entre ellas?
    ¿Qué es?

Esquemas de funciones

- Solución:

```
sonCincos [ ]      = ...
sonCincos (n:ns) =
            … n ... sonCincos ns ...

concat [ ]         = ...
concat (xs:xss)  =
            ... xs ...  concat xss ...
```

  - Recursión estructural

Esquemas de funciones

- Aplicandola técnica de las cajas

```
sonCincos [ ]       = True
sonCincos (n:ns) =
            n ==5 && sonCincos ns

concat [ ]          = [ ]
concat (xs:xss)  =
            xs ++  concat xss
```

  - Los “recuadros” son más complicadas, pero la técnica es la misma

Esquemas de funciones

- Aplicandola técnica de las cajas

```
sonCincos [ ]       = True
sonCincos (n:ns) =
            n ==5 && sonCincos ns

concat [ ]          = [ ]
concat (xs:xss)  =
            xs ++ concat xss
```

  - Los "recuadros" son más complicadas, pero la técnica es la misma

Esquemas de funciones

- Aplicando la técnica de las cajas

```
sonCincos [ ]      = True
sonCincos (n:ns) =
  (\x b -> x==5 && b) n (sonCincos ns)

concat [ ]         = [ ]
concat (xs:xss)  =
  (\ys zs -> ys ++ zs) xs ( concat xss )
```

  - Los “recuadros” son más complicadas, pero la técnica es la misma

Esquema de recursión (fold)

- Procedemos con la abstracción

```
foldr :: ??
foldr     [ ]     = □
foldr     (x:xs) = □ x (foldr     xs)
```

Esquema de recursión (fold)

- Completamos la definición

```
foldr :: ??
foldr f z [ ]       = z
foldr f z (x:xs)  = f  x  (foldr f z xs)
```

Esquema de recursión (fold)

- Completamos la definición

```
foldr :: ??
foldr f z [ ]       = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

- Y entonces

```
sonCincos' = foldr check True
                  where check x b = (x==5) && b
cantTotal' = foldr ((+) . len) 0
concat' = foldr (++) [ ]
```

Esquema de recursión (fold)

- Agregamos el tipo

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr f z [ ]      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

- Y entonces

```
sonCincos' = foldr check True
        where check x b = (x==5) && b
cantTotal' = foldr ((+) . len) 0
concat' = foldr (++) [ ]
```

Esquema de recursión (fold)

- Agregamos el tipo

```
foldr :: (a->b->b) -> b -> ([a] -> b)
foldr f z [ ]      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

- Y entonces

```
sonCincos' = foldr check True
             where check x b = (x==5) && b
cantTotal' = foldr ((+) . len) 0
concat' = foldr (++) [ ]
```

- ¿Podría probar que concat' = concat?

Esquemas de funciones

- ¿Qué ventajas tiene trabajar con esquemas?

  Permite
  - definiciones más concisas y modulares
  - reutilizar código
  - demostrar propiedades generales
- ¿Qué requiere trabajar con esquemas?
  - Familiaridad con funciones de alto orden
  - Detección de características comunes (¡ABSTRACCIÓN!)

Propiedades de esquemas

- Propiedades de los esquemas
- Analicemos el ejemplo de cantTotal

```
cantTotal :: [ [a] ] -> Int
-- dice cuántos elementos de tipo a hay en total

-- canTotal [ ] = 0
-- cantTotal (xs:xss) = length xs + cantTotal xss

cantTotal = foldr (\zs n -> length zs + n) 0
-- cantTotal = foldr ((+) . length) 0
```

- ¿Hay otra forma de pensarlo?

Propiedades de esquemas

- Alternativa para cantTotal

```
cantTotal' :: [ [a] ] -> Int
cantTotal' xss = sum (map length xss)
sum = foldr (+) 0
```

- ¿Será cierto que cantTotal es igual a cantTotal'?
- Sugiere una propiedad, que para todo xs

```
foldr f z (map g xs) = foldr (f . g) z xs
```

  - O sea, procesar primero cada elemento y luego unir los resultados da lo mismo que unir los resultados procesando cada elemento al unirlo
  - ¡Demostración por inducción estructural!

Esquemas y alto orden

- ¿Cómo definir append con foldr?

```
append :: [a] -> ([a] -> [a])
append []     = \ys -> ys
append (x:xs) = \ys -> x : append xs ys
```

- Expresado así, es rutina:

```
\ys -> x : append xs ys =
              (\x' h -> \ys -> x' : h ys) x (append xs)
```

y entonces

```
append = foldr (\x h ys -> x : h ys) id
       = foldr (\x h -> (x:) . h) id = foldr ((.) . (:)) id
```

Esquemas y alto orden

- ¿Cómo definir take con foldr?

```
take :: Int -> [a] -> [a]
take _ []       = []
take 0 (x:xs) = []
take n (x:xs) = x : take (n-1) xs
```

¡El n cambia en cada paso!

- Primero debo cambiar el orden de los argumentos

```
take' :: [a] -> (Int -> [a])
take' []       = \_ -> []
take' (x:xs) = \n -> case n of 0 -> []
                                _ -> x : take' xs (n-1)
```

Esquemas y alto orden

- ¿Cómo definir take con foldr? (Cont.)

```
take' :: [a] -> (Int -> [a])
take' = foldr g (const [])
  where g _ _ 0 = []
        g x h n = x : h (n-1)
```

y entonces

```
take :: Int -> [a] -> [a]
take = flip take'

flip f x y = f y x
```

Esquemas y alto orden

- Un ejemplo más: la función de Ackerman

  (¡con notación unaria!)

```
data One = One

ack :: Int -> Int -> Int
ack n m = u2i (ack' (i2u n) (i2u m))
              where i2u n = repeat n One
                    u2i = length

ack' :: [ One ] -> [ One ] -> [ One ]
ack' []       ys      = One : ys
ack' (x:xs) []        = ack' xs [ One ]
ack' (x:xs) (y:ys) = ack' xs (ack' (x:xs) ys)
```

Esquemas y alto orden

- La función de Ackerman (cont.)

```
ack' :: [ One ] -> [ One ] -> [ One ]
ack' []       = \ys -> One : ys
ack' (x:xs) = g
   where g []      = ack' xs [ One ]
            g (y:ys) = ack' xs (g ys)
```

- Reescribimos ack' (x:xs) = g como un foldr

```
ack' (x:xs) = foldr (\_ -> ack' xs) (ack' xs [ One ])
```

Esquemas y alto orden

- Y finalmente podemos definir ack' con foldr

```
ack' :: [ One ] -> [ One ] -> [ One ]
ack' = foldr (const g) (One :)
        where g h = foldr (const h) (h [ One ])
```

- Con esto podemos ver que la función de Ackerman termina para todo par de números naturales.

Esquemas en otros tipos

- Los esqumas de recursión también se pueden definir para otros tipos.
- Los naturales son un tipo inductivo.

```
foldNat :: (b -> b) -> b -> Nat -> b
foldNat s z 0 = z
foldNat s z n = s (foldNat s z (n-1))
```

- Los casos de la inducción son cero y el sucesor de un número, y por eso los argumentos del foldNat.

Recursión Primitiva (Listas)

- No toda función sobre listas es definible con foldr.
- Ejemplos:

```
tail :: [a] -> [a]
tail (x:xs) = xs

insert :: a -> [a] -> [a]
insert x []       = [x]
insert x (y:ys) = if x<y then (x:y:ys) else (y:insert x ys)
```

  - (**Nota:** en listas es complejo de observar. La recursión primitiva se observa mejor en árboles.)

Recursión Primitiva (Listas)

- El problema es que, además de la recursión sobre la cola, ¡utilizan la misma cola de la lista!
- Solución

```
recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z f []     = z
recr z f (x:xs) = f x xs (recr z f xs)
```

¡Observar el parámetro adicional de f!

Recursión Primitiva (Listas)

- El problema es que, además de la recursión sobre la cola, ¡utilizan la misma cola de la lista!
- Solución

```
recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z f []       = z
recr z f (x:xs) = f x xs (recr z f xs)
```

- Entonces

```
tail = recr (error “Lista vacía”) (\_ xs _ -> xs)
insert x = recr [x] (\y ys zs -> if x<y then (x:y:ys)
                                          else (y:zs))
```

Recursión Primitiva (Listas)

- Otras funciones que se pueden definir con recr.

```
init :: [a] -> [a]
init = recr …



maximum :: [a] -> a
maximum = recr ...
```

Recursión Primitiva (Listas)

- Otras funciones que se pueden definir con recr.

```
init :: [a] -> [a]
init = recr (error “No definida”)
            (\x xs rs -> if null xs then [] else x:rs)


maximum :: [a] -> a
maximum = recr (error “No definida”)
               (\x xs m -> if null xs then x
                                      else max x m)
```

Recursión Primitiva (Nats)

- Recursión primitiva sobre naturales

```
recNat :: b -> (Nat -> b -> b) -> Nat -> b
recNat z f 0 = z
recNat z f n = f (n-1) (recNat z f (n-1))
```

- Ejemplos (no definibles como foldNat)

```
fact = recNat 1 (\n p -> (n+1)*p)
```

-- fact n = $\prod_{i=1}^{n} i$

```
sumatoria f = recNat 0 (\x y -> f (x+1) + y)
```

-- sumatoria f n = $\sum_{i=1}^{n} f\ i$

Otros esquemas de listas

- En el caso de maximum o minimum, podemos identificar otro esquema:
  - el de fold de listas no vacías (foldr1)

```
maximum, minimum :: [a] -> a
maximum = foldr1 (\x m -> max x m)
minimum = foldr1 min

foldr1 :: (a->a->a) -> [a] -> a
foldr1 f (x:xs) = foldr f x xs
```

“We claim that advanced data structures and algorithms can be better taught at the functional paradigm than at the imperative one.”

"A Second Year Course on Data Structures Based on Functional Programming"

M. Núñez, P. Palao y R. Peña

Functional Programming Languages in Education, LNCS 1022

Definición de Tipos 1

- Para definir un tipo de datos podemos:
  - establecer qué *forma* tendrá cada *elemento*, y
  - dar un *mecanismo único* para *inspeccionar* cada elemento
  - entonces: TIPO ALGEBRAICO

ó

  - determinar cuáles serán las *operaciones* que manipularán los elementos, SIN decir cuál será la forma exacta de éstos o aquéllas
  - entonces: TIPO ABSTRACTO

Tipos Algebraicos

- ¿Cómo damos en Haskell la forma de un elemento de un tipo algebraico?
  - Mediante **constantes** llamadas *constructores*
    - nombres con mayúsculas
    - no tienen asociada una regla de reducción
    - pueden tener argumentos
- Ejemplos:

False :: Bool   True :: Bool

Tipos Algebraicos

- La cláusula data
  - introduce un nuevo tipo algebraico
  - introduce los nombres de los constructores
  - define los tipos de los argumentos de los constructores
- Ejemplos:

```
data Sensacion = Frio | Calor
data Shape = Circle Float | Rect Float Float
```

Tipos Algebraicos

- data Shape = Circle Float | Rect Float Float

  Ejemplos de elementos:

  c1 = Circle 1.0

  c2 = Circle (4.0-3.0)

  r1 = Rect 2.5 3.0

  Ejemplos de funciones que arman Shapes:

  circuloPositivo x = Circle (abs x)

  cuadrado x = Rect x x

Tipos Algebraicos

- data Shape = Circle Float | Rect Float Float

Ejemplo de alto orden:

```
construyeShNormal :: (Float -> Shape) -> Shape
construyeShNormal c = c 1.0
```

Uso de funciones de alto orden:

```
c3 = construyeShNormal circuloPositivo
c4 = construyeShNormal cuadrado
c5 = construyeShNormal (Rect 2.0)
```

- ¿Cuál es el tipo de Circle? ¿Y el de Rect?

Pattern Matching

- ¿Cuál es el mecanismo único de acceso?
  - *Pattern matching* (correspondencia de patrones (?))
- Pattern: expresión especial
  - sólo con constructores y variables sin repetir
  - argumento en el lado izquierdo de una ecuación
- Matching: operación asociada a un pattern
  - inspecciona el valor de una expresión
  - puede fallar o tener éxito
  - si tiene éxito, liga las variables del pattern

Pattern Matching

- Ejemplos:

```
area :: Shape -> Float
area (Circle radio) = pi * radio^2
area (Rect base altura) = base * altura

isCircle :: Shape -> Bool
--isCircle1 (Circle radio) = True
--isCircle1 (Rectangle base altura) = False
isCircle (Circle _) = True
isCircle _ = False
```

Pattern Matching

- Uso de pattern matching:
  - Al evaluar (area (circuloPositivo (-3.0)))
    - se reduce (circuloPositivo (-3.0)) a (Circle 3.0)
    - luego se verifica cada ecuación, para hacer el matching
    - si lo hace, la variable toma el valor correspondiente
  - radio se liga a 3.0, y la expresión retorna 28.2743
- ¿Cuánto valdrá (area (cuadrado 2.5))?
- ¿Y (area c2)?

Tuplas

- Son tipos algebraicos con sintaxis especial

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y

distance :: (Float, Float) -> Float
distance (x,y) = sqrt (x^2 + y^2)
```

- ¿Cómo definir distance sin usar pattern matching?

```
distance p = sqrt ((fst p)^2 + (snd p)^2)
```

Tipos Algebraicos

- Pueden tener argumentos de tipo
- Ejemplo:

  data Maybe a = Nothing | Just a

- ¿Qué elementos tiene (Maybe Bool)? ¿Y (Maybe Int)?
- En general:
  - tiene los mismos elementos que el tipo a (pero con Just adelante) más uno adicional (Nothing)

Tipos Algebraicos

- ¿Para qué se usa el tipo Maybe?
- Ejemplo:

```
buscar :: clave -> [(clave,valor)] -> valor
buscar k [] = error "La clave no se encontró"
          -- Única elección posible con polimorfismo!
buscar k ((k',v):kvs) = if k==k'
                        then v
                        else buscar k kvs
```

- ¿La función buscar es total o parcial?

Tipos Algebraicos

- ¿Para qué se usa el tipo Maybe?
- Ejemplo:

```
lookup :: clave -> [(clave,valor)] -> Maybe valor
lookup k []               = Nothing
lookup k ((k',v):kvs) = if k==k'
                              then Just v
                              else lookup k kvs
```

- ¿La función lookup es total o parcial?

Tipos Algebraicos

- El tipo Maybe
  - permite expresar la posibilidad de que el resultado sea erróneo, sin necesidad de usar 'casos especiales'
  - evita el uso de ⊥ hasta que el programador decida, permitiendo controlar los errores

```
sueldo :: Nombre -> [Empleado] -> Int
sueldo nombre empleados
      = analizar (lookup nombre empleados)
analizar Nothing = error "No es de la empresa!"
analizar (Just s) = s
```

Tipos Algebraicos

- El tipo Maybe (cont.)
  - evita el uso de ⊥ hasta que el programador decida, permitiendo controlar los errores

```
sueldoGUI :: Nombre -> [Empleado] -> GUI Int
sueldoGUI nombre empleados =
    case (lookup nombre empleados) of
       Nothing -> ventanaError "No es de la empresa!"
       Just s    -> mostrarInt “El sueldo es ” s
```

Tipos Algebraicos

- Otro ejemplo:

  data Either a b = Left a | Right b

- ¿Qué elementos tiene (Either Int Bool)?
- En general:
  - representa la unión disjunta de dos conjuntos (los elementos de uno se identifican con Left y los del otro con Right)

Tipos Algebraicos

- ¿Para qué sirve Either?
  - Para mantener el tipado fuerte y poder devolver elementos de distintos tipos
    - Ejemplo: [Left 1, Right True] :: [Either Int Bool]
  - Para representar el origen de un valor
    - Ejemplo: lectora de temperaturas

```
data Temperatura = Celsius Int | Fahrenheit Int

convertir :: Either Int Int -> Temperatura
convertir (Left t)   = Celsius t
convertir (Right t) = Fahrenheit t
```

Tipos Algebraicos

- ¿Por qué se llaman tipos algebraicos?
- Por sus características:
  - toda combinación válida de constructores y valores es elemento de un tipo algebraico
(y sólo ellas lo son)
  - dos elementos de un tipo algebraico son iguales si y sólo si están construídos utilizando los mismos constructores aplicados a los mismos valores

Tipos Algebraicos

- Expresividad: números complejos
  - Toda combinación de dos flotantes es un complejo
  - Dos complejos son iguales si tienen las mismas partes real e imaginaria

```
data Complex = C Float Float

realPart, imagePart :: Complex -> Float
realPart (C r i) = r
imagePart (C r i) = i

mkPolar :: Float -> Float -> Complex
mkPolar r theta = C (r * cos theta) (r * sin theta)
```

Tipos Algebraicos

- Expresividad: números racionales
  - No todo par de enteros es un número racional (R 1 0)
  - Hay racionales iguales con distinto numerador y denominador (R 4 2 = R 2 1)

```
data NoRacional = R Int Int
numerador, denominador :: NoRacional -> Int
numerador (R n d) = n
denominador (R n d) = d
```

- ¡No se puede representar a los racionales como tipo algebraico!

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar tipos ad-hoc, combinando las ideas

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto (Maybe Baño)
            | Capelina Gusto Gusto [Agregado]
            | Pote Gusto Gusto Gusto
data Gusto = Chocolate | …
data Agregado = Almendras | Rocklets | …
data Baño = Blanco | Negro
```

- Así se pueden expresar elementos de dominios específicos

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar funciones por pattern matching

```
precio :: Helado -> Float
precio h = costo h * 0.3 + 5

costo :: Helado -> Float
costo (Vasito g) = 1 + costoGusto g
costo (Cucurucho g1 g2 mb) = 2 + costoGusto g1
                               + costoGusto g2
                               + costoBaño mb
...
```

Tipos Algebraicos

- Expresividad: ejemplos
  - Se pueden armar funciones por pattern matching (cont.)

```
costoGusto :: Gusto -> Float
costoGusto Chocolate = 2
…

costoBaño :: Maybe Baño - > Float
costoBaño Nothing         = 0
costoBaño (Just Negro)   = 2
costoBaño (Just Blanco)  = 1
```

Tipos Algebraicos

- Podemos clasificarlos en:
  - Enumerativos (Sensacion, Bool)
    - Sólo constructores sin argumentos
  - Productos (Complex, Tuplas)
    - Un único constructor con varios argumentos
  - Sumas (Shape, Maybe, Either)
    - Varios constructores con argumentos
  - Recursivos (Listas)
    - Utilizan el tipo definido como argumento

Tipos de Datos Recursivos

- Un tipo algebraico recursivo
  - tiene al menos uno de los constructores con el tipo que se define como argumento
  - es la concreción en Haskell de un conjunto definido inductivamente
- Ejemplos:

```
data N = Z | S N
data BE = TT | FF | AA BE BE | NN BE
```

- ¿Qué elementos tienen estos tipos?

Tipos de Datos Recursivos

- Cada constructor define un caso de una definición inductiva de un conjunto.
  - Si tiene al tipo definido como argumento, es un *caso inductivo*, si no, es un *caso base*.
- El pattern matching
  - provee análisis de casos
  - permite acceder a los elementos inductivos que forman a un elemento dado
- Por ello, se pueden definir funciones recursivas

Tipos de Datos Recursivos

- Ejemplo: `data N = Z | S N`
  - Asignación de significado

```
evalN :: N -> Int
evalN Z      = ...
evalN (S x) = ... evalN x ...
```

  - ¡Usamos recursión estructural!

Tipos de Datos Recursivos

- Ejemplo: `data N = Z | S N` (cont.)
  - Asignación de significado

```
evalN :: N -> Int
evalN Z     = 0
evalN (S x) = 1 + evalN x
```

  - El tipo `N` es notación unaria para expresar números enteros (`Int`)

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Manipulación simbólica

```
addN :: N -> N -> N
addN Z     m = ...
addN (S n) m = ... (addN n m) ...
```

  - Otra vez usamos recursión estructural

Tipos de Datos Recursivos

- Ejemplo: `data N = Z | S N` (cont.)
  - Manipulación simbólica

```
addN :: N -> N -> N
addN Z      m = m
addN (S n) m = S (addN n m)
```

  - No hay significado en sí mismo en esta manipulación

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Coherencia de significación con manipulación
  - ¿Puede probarse la siguiente propiedad?
    Sean n,m::N finitos, cualesquiera; entonces

    evalN (addN n m) = evalN n + evalN m

  - ¿Cómo? ...
  - La propiedad captura el vínculo entre el significado y la manipulación simbólica

Tipos de Datos Recursivos

- Por inducción estructural
  (pues el tipo representa a un conjunto inductivo)
- **Demostración**: por inducción en la estructura de n
  - Caso base: n = Z
    - Usar addN.1, 0 neutro de (+) y evalN.1
  - Caso inductivo: n = S n'
  - HI: size (addN n' m) = size n' + size m
    - Usar addN.2, evalN.2, HI, asociatividad de (+), y evalN.2

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Más manipulación simbólica

```
prodN :: N -> N -> N
prodN Z       m = Z
prodN (S n)  m = addN (prodN n m) m
```

- ¿Puede probar la siguiente propiedad?
  Sean n,m::N finitos, cualesquiera; entonces

  evalN (prodN n m) = evalN n * evalN m

Listas

- Una definición equivalente a la de listas

  data List a = Nil | Cons a (List a)

- La sintaxis de listas es equivalente a la de esta definición:
  - [ ] es equivalente a Nil
  - (x:xs) es equivalente a (Cons x xs)
- Sin embargo, (List a) y [a] son tipos distintos

Listas

- Considerar las definiciones

```
sum :: [Int] -> Int          -- Significado
sum [ ]        = 0
sum (n:ns)     = n + sum ns

(++) :: [a] -> [a] -> [a]    -- Manipulación simbólica
[ ] ++ ys      = ys
(x:xs) ++ ys   = x : (xs ++ ys)
```

- Coherencia entre significado y manipulación:
  Demostrar que para todas xs, ys listas finitas vale que:

  sum (xs ++ ys) = sum xs + sum ys

Árboles

- Un árbol es un tipo algebraico tal que al menos un elemento compuesto tiene dos componentes inductivas
- Se pueden usar TODAS las técnicas vistas para tipos algebraicos y recursivos
- Ejemplo:
  `data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)`
- ¿Qué elementos tiene el tipo (Arbol Int)?

Árboles

- Si representamos elementos de tipo Arbol Int mediante diagramas jerárquicos

```
aej = Nodo 1 (Hoja 0)
             (Nodo 3 (Hoja 2) (Hoja 0))
```

```
        Nodo 1
       /      \
  Hoja 0     Nodo 3
            /      \
        Hoja 2    Hoja 0
```

Árboles

- ¿Cuántas hojas tiene un (Arbol a)?

```
hojas :: Arbol a -> Int
hojas (Hoja x)         = ...
hojas (Nodo x t1 t2) = ... hojas t1 ... hojas t2 ...
```

- ¿Y cuál es la altura de un (Arbol a)?

```
altura :: Arbol a -> Int
altura (Hoja x)         = ...
altura (Nodo x t1 t2) = ... altura t1 ... altura t2 ...
```

- Observar el uso de recursión estructural

Árboles

- ¿Cuántas hojas tiene un (Arbol a)?

```
hojas :: Arbol a -> Int
hojas (Hoja x)       = 1
hojas (Nodo x t1 t2) = hojas t1 + hojas t2
```

- ¿Y cuál es la altura de un (Arbol a)?

```
altura :: Arbol a -> Int
altura (Hoja x)       = 0
altura (Nodo x t1 t2) = 1+ (altura t1 `max` altura t2)
```

- ¿Puede mostrar que para todo árbol finito a,
  hojas a ≤ 2^(altura a)? ¿Cómo?

Árboles

- ¿Cómo reemplazamos una hoja?
- Ej: Cambiar los 2 en las hojas por 3.

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n) = …


cambiar2 (Nodo n t1 t2) =
              … (cambiar2 t1) … (cambiar2 t2) …
```

- ¡Más recursión estructural!

Árboles

- ¿Cómo reemplazamos una hoja?
- Ej: Cambiar los 2 en las hojas por 3.

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n) = if n==2
                      then Hoja 3
                      else Hoja n
cambiar2 (Nodo n t1 t2) =
         Nodo n (cambiar2 t1) (cambiar2 t2)
```

- ¿Cómo trabaja cambiar2? Reducir (cambiar2 aej)

Árboles

- Más funciones sobre árboles

```
duplA :: Arbol Int -> Arbol Int
duplA (Hoja n) = ... n ...
duplA (Nodo n t1 t2) =
          ... n ... (duplA t1) ... (duplA t2) ...
sumA :: Arbol Int -> Int
sumA (Hoja n) = ... n ...
sumA (Nodo n t1 t2) =
          ... n ... sumA t1 ... sumA t2 ...
```

- ¡Recursión estructural!

Árboles

- Más funciones sobre árboles

```
duplA :: Arbol Int -> Arbol Int
duplA (Hoja n) = Hoja (n*2)
duplA (Nodo n t1 t2) =
          Nodo (n*2) (duplA t1) (duplA t2)
sumA :: Arbol Int -> Int
sumA (Hoja n) = n
sumA (Nodo n t1 t2) = n + sumA t1 + sumA t2
```

- ¿Cómo evalúa la expresión (duplA aej)?
- ¿Y (sumA aej)?

Árboles

- Recorridos de árboles
  - Transformación de un árbol en una lista (estructura no lineal a estructura lineal)

```
inOrder, preOrder :: Arbol a -> [ a ]
inOrder (Hoja n) = ... n ...
inOrder (Nodo n t1 t2) =
        ... inOrder t1 ... n ... inOrder t2 ...

preOrder (Hoja n) = ... n ...
preOrder (Nodo n t1 t2) =
        ... n ... preOrder t1 ... preOrder t2 ...
```

  - ¿Cómo sería posOrder?

Árboles

- Recorridos de árboles
  - Transformación de un árbol en una lista (estructura no lineal a estructura lineal)

```
inOrder, preOrder :: Arbol a -> [ a ]
inOrder (Hoja n) = [ n ]
inOrder (Nodo n t1 t2) =
         inOrder t1 ++ [ n ] ++ inOrder t2

preOrder (Hoja n) = [ n ]
preOrder (Nodo n t1 t2) =
         n : (preOrder t1 ++ preOrder t2)
```

  - ¿Cómo sería posOrder?

Expresiones Aritméticas

- Definimos expresiones aritméticas
  - constantes numéricas
  - sumas y productos de otras expresiones

    data ExpA = Cte Int | Suma ExpA ExpA
    | Mult ExpA ExpA
- Ejemplos:
  - 2 se representa (Cte 2)
  - (4*4) se representa (Mult (Cte 4) (Cte 4))
  - ((2*3)+4) se representa
    Suma (Mult (Cte 2) (Cte 3)) (Cte 4)

Expresiones Aritméticas

- Definimos expresiones aritméticas
  - alternativa más simbólica...

    data ExpS = CteS N | SumaS ExpS ExpS
| MultS ExpS ExpS
- Ejemplos:
  - 2 se representa (CteS (S (S Z)))
  - comparar con (Cte 2), donde Int representa números como semántica y no como símbolos (como lo hace N)

Expresiones Aritméticas

- ¿Cómo dar el significado de una ExpA?

```
evalEA :: ExpA -> Int
evalEA (Cte n)         = ...
evalEA (Suma e1 e2)  = evalEA e1 ... evalEA e2
evalEA (Mult e1 e2)   = evalEA e1 ... evalEA e2
```

  - Observar el uso de recursión estructural

Expresiones Aritméticas

- ¿Cómo dar el significado de una ExpA?

```
evalEA :: ExpA -> Int
evalEA (Cte n)          = n
evalEA (Sum e1 e2)      = evalEA e1 + evalEA e2
evalEA (Mult e1 e2)     = evalEA e1 * evalEA e2
```

- Reduzca:

```
evalEA (Suma (Mult (Cte 2) (Cte 3)) (Cte 4))
evalEA (Mult (Cte 2) (Suma (Cte 3) (Cte 4)))
```

Expresiones Aritméticas

- Comparar con la versión más simbólica

```
evalES :: ExpS -> Int
evalES (CteS n)         = evalN n
evalES (SumaS e1 e2)    = evalES e1 + evalES e2
evalES (MultS e1 e2)    = evalES e1 * evalES e2
```

  - Se observa el uso de la función de asignación semántica (evalN) a los números representados como N (símbolos)

Expresiones Aritméticas

- ¿Cómo simplificar una ExpA?
  Manipulación simbólica

```
simplEA :: ExpA -> ExpA
simplEA (Cte n)         = ...
simplEA (Suma e1 e2) = ... (simplEA e1) ...
                           (simplEA e2) ...
simplEA (Mult e1 e2)  = ... (simplEA e1) (simplEA e2)
```

  - Observar otra vez el uso de recursión estructural

Expresiones Aritméticas

- ¿Cómo simplificar una ExpA?
  Manipulación simbólica

```
simplEA :: ExpA -> ExpA
simplEA (Cte n)          = Cte n
simplEA (Suma e1 e2) = armarSuma (simplEA e1)
                                 (simplEA e2)
simplEA (Mult e1 e2)   = Mult (simplEA e1) (simplEA e2)

armarSuma (Cte 0) e  = e
armarSuma e (Cte 0)  = e
armarSuma e1 e2      = Sum e1 e2
```

Expresiones Aritméticas

- ¿La manipulación es correcta?
  - Coherencia entre significado y manipulación simbólica
  - Expresado mediante la siguiente propiedad

    para todo e se cumple que

    evalEA (simplEA e) = evalEA e

Expresiones con variables

- ¿Cómo agregar variables a las expresiones?
  - Nuevo constructor

```
type Variable = String

data NExp = Vble Variable
          | NCte Int
          | Add NExp NExp | Sub NExp NExp
          | Mul NExp NExp
          | Div NExp NExp  | Mod NExp NExp
```

  - Además agregamos nuevas operaciones

Expresiones con variables

- ¿Y para asignarles significado?
  - Necesitamos conocer el valor de las variables

    ```
    evalNExp :: NExp -> (Memoria -> Int)
    ```

  - ¡El significado es una función!
  - Observar el uso del alto orden y la currificación
  - ¿Qué es la memoria?

Expresiones con variables

- Tipo abstracto para representar la memoria

```
data Memoria                -- Tipo abstracto de datos
enBlanco :: Memoria
  -- Una memoria vacía, que no recuerda nada
cuantoVale :: Memoria -> Variable -> Maybe Int
  -- Retorna el valor recordado para la variable dada
recordar :: Memoria -> Variable -> Int -> Memoria
  -- Recuerda un valor paraa una variable
variables :: Memoria -> [ Variable ]
  -- Retorna las variables que recuerda
```

Expresiones con variables

- Semántica de expresiones con variables

```
evalN :: NExp -> (Memoria -> Int)
evalN (Vble x)      mem  =
      ... mem … x ...



evalN (NCte n)      mem  = ... n ...
evalN (Add e1 e2)  mem  = evalN e1 mem … evalN e2 mem
        ...
```

- Observar
  - que las variables complican la semántica
  - el uso de la currificación para pasar la memoria

Expresiones con variables

- Semántica de expresiones con variables

```
evalN :: NExp -> (Memoria -> Int)
evalN (Vble x)       mem  =
      case (cuantoVale mem x) of
        Nothing   -> error ("variable "++x++" indefinida")
        Just v      -> v
evalN (NCte n)       mem   = n
evalN (Add e1 e2)  mem   = evalN e1 mem + evalN e2 mem
        ...
```

- Observar
  - Lo mejor es fallar si la variable está indefinida
  - Algunos lenguajes dan otras cosas (0, o “basura”)

Definición de LIS

- Definimos un Lenguaje Imperativo Simple
  - asignación de expresiones numéricas
  - sentencias if y while sobre expresiones booleanas
  - secuencia de sentencias
- Ejemplo:

```
a := n; facn := 1
while (a > 0)
  { facn := a * facn; a := a – 1 }
```

- ¡Solo sintaxis!

Definición de LIS (cont.)

- Definimos tipos algebraicos para representar un programa LIS

```
data Program = P Bloque
type Bloque = [Comando]
data Comando  = Skip
                        | Assign Variable NExp
                        | If BExp Bloque Bloque
                        | While BExp Bloque
```

Definición de LIS (cont.)

- Definimos tipos algebraicos para representar un programa LIS
  - Los constructores carecen de significado
  - Una definición “equivalente” sería

```
data A = B C
type C = [D]
data D  = E
          | F Variable NExp
          | G BExp C C
          | H BExp C
```

Definición de LIS (cont.)

- Usamos las NExp anteriores, y agregamos expresiones booleanas

```
data BExp = BCte Bool      | Not BExp
          | And BExp BExp  | Or BExp BExp
          | RelOp ROp NExp NExp

data ROp = Equal             | NotEqual
         | Greater           | Lower
         | GreaterEqual      | LowerEqual
```

Definición de LIS (cont.)

- Usamos las NExp anteriores, y agregamos expresiones booleanas
  - Continuando con la definición “equivalente”

```
data BExp = I Bool      | J BExp
          | K BExp BExp  | L BExp BExp
          | M ROp NExp NExp

data ROp = N  | O
         | P  | Q
         | R  | S
```

Definición de LIS

- ¿Cómo queda el programa de ejemplo?

```
a := n; facn := 1
while (a > 0)
  {  facn := a * facn; a := a – 1 }
```

se expresa como

```
P [ Assign “a” (Vble “n”)
  , Assign “facn” (NCte 1)
  , While (RelOp Greater (Vble “a”) (NCte 0))
      [  Assign “facn” (Mul (Vble “a”) (Vble “facn”))
      ,  Assign “a” (Sub (Vble “a”) (NCte 1))
      ]
  ]
```

Definición de LIS

- ¿Cómo queda el programa de ejemplo?

```
a := n; facn := 1
while (a > 0)
  {  facn := a * facn; a := a – 1 }
```

se expresa “equivalentemente” como

```
B [ F “a” (Vble “n”)
  , F “facn” (NCte 1)
  , H (M P (Vble “a”) (NCte 0))
      [  F “facn” (Mul (Vble “a”) (Vble “facn”))
      ,  F “a” (Sub (Vble “a”) (NCte 1))
      ]
  ]
```

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas

```
evalB :: BExp -> (Memoria -> Bool)
evalB (BCte b) mem = ...
evalB (RelOp rop e1 e2) mem
   = ... rop ... (evalN e1 mem) ... (evalN e2 mem) ...
evalB (And e1 e2) mem
   = ... evalB e1 mem ... evalB e2 mem ...
...
```

- ¿Por qué hace falta la memoria para dar significado a una expresión booleana?

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas

```
evalB :: BExp -> (Memoria -> Bool)
evalB (BCte b) mem = b
evalB (RelOp rop e1 e2) mem
   = evalROp rop (evalN e1 mem) (evalN e2 mem)
evalB (And e1 e2) mem
   = evalB e1 mem && evalB e2 mem
…
```

- Nuevamente, observar el uso de currificación
- ¿Y la función auxiliar evalROp?

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas (cont.)

```
evalROp :: ROp -> (Int -> Int -> Bool)
evalROp Equal      = (==)
evalROp NotEqual   = (!=)
evalROp Greater    = (>)
…
```

- ¡Observar que el significado se define directamente como una función!

Evaluador de LIS (cont.)

- Semántica de programas LIS

```
evalP :: Program -> (Memoria -> Memoria)
evalP (P bloque) = evalBloque bloque

evalBloque [ ] = \mem -> mem
evalBloque (c:cs) =
          \mem -> let mem' = evalCom c mem
                  in evalP cs mem'
```

- ¡¡El significado es una función!!
- ¡Observar cómo la secuencia de comandos ALTERA la memoria antes de proseguir!

Evaluador de LIS (cont.)

- Semántica de sentencias LIS

```
evalCom :: Comando -> (Memoria -> Memoria)
evalCom Skip  = ...
evalCom (Assign x ne)
          = ... (evalN ne mem) ...
evalCom (If be bl1 bl2)
          = ... (evalB be mem)
                  ... (evalBloque bl1 mem)
                  ... (evalBloque bl2 mem)
evalCom (While be p)
          = ...??
```

Evaluador de LIS (cont.)

- Semántica de sentencias LIS

```
evalCom :: Comando -> (Memoria -> Memoria)
evalCom Skip  = \mem -> mem
evalCom (Assign x ne)
        = \mem -> recordar mem x (evalN ne mem)
evalCom (If be bl1 bl2)
        = \mem -> if (evalB be mem)
                   then (evalBloque bl1 mem)
                   else (evalBloque bl2 mem)
evalCom (While be p)
      = evalCom (If be (p ++ [While be p]) [Skip])
```

¡No es estructural!

Manipulacion simbólica

- ¿Qué pasa con programas como éste?

```
p :: Program
p = P [ Assign “x” (Add (NCte 10) (NCte 7))
      , Assign “y” (Add (Sub (NCte 59) (Var “x”))
                        (Sub (NCte 2) (NCte 2)) ]
```

- ¿Se podría hacer más eficiente ANTES de ejecutarlo?
  - Constant folding, simplification

```
p = P [ Assign “x” (NCte 17))
      , Assign “y” (Sub (NCte 59) (Var “x”)) ]
```

Manipulación simbólica (cont.)

- Expresiones sin variables

  ```
  groundNExp :: NExp -> Bool
  ```

- Simplificación y evaluación

  ```
  simplifyNExp :: NExp -> NExp
  evalGroundNExp :: NExp -> Int
    -- PRECOND: el argumento es ground
  ```

  - ¡simplify debería usar evalGroundNExp!

- Análisis simbólico del programa

  ```
  optimize :: Program -> Program
  ```

Manipulación simbólica (cont.)

- Expresiones sin variables

```
groundNExp :: NExp -> Bool
groundNExp ...
```

Manipulación simbólica (cont.)

- Simplificación y evaluación

```
simplifyNExp :: NExp -> NExp

simplifyNExp …
    -- OBS: usa evalGroundNExp


evalGroundNExp :: NExp -> Int
  -- PRECOND: el argumento es ground
evalGroundNExp …
```

Manipulación simbólica (cont.)

- Análisis simbólico del programa

  optimize :: Program -> Program

  optimize ...

“'Todo es pasajero. La verdad depende del momento. Baje los ojos. Incline la cabeza. Cuente hasta diez. Descubrirá otra verdad.'

(Consejero, 74:96:3)”

El Fondo del Pozo
Eduardo Abel Giménez

Esquemas en árboles

- Esquema de map en árboles:

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)

mapArbol :: (a -> b) -> Arbol a -> Arbol b
mapArbol f (Hoja x)          = Hoja (f x)
mapArbol f (Nodo x t1 t2)  =
    Nodo (f x) (mapArbol f t1) (mapArbol f t2)
```

- ¿Cómo definiría la función que multiplica por 2 cada elemento de un árbol? ¿Y la que los eleva al cuadrado?

Esquemas en árboles

- Solución:

```
dupArbol :: Arbol Int  -> Arbol Int
dupArbol = mapArbol (*2)

cuadArbol :: Arbol Int -> Arbol Int
cuadArbol = mapArbol (^2)
```

- ¿Podría definir, usando mapArbol, una función que aplique dos veces una función dada a cada elemento de un árbol? ¿Cómo?

Esquemas en árboles

- La función `foldr` expresa el patrón de recursión estructural sobre listas como función de alto orden
- Todo tipo algebraico recursivo tiene asociado un patrón de recursión estructural
- ¿Existirá una forma de expresar cada uno de esos patrones como una función de alto orden?
- ¡Sí, pero los argumentos dependen de los casos de la definición!

Esquemas en árboles

- Ejemplo:

```
foldArbol :: (a->b) -> (a->b->b->b) -> Arbol a -> b
foldArbol f g (Hoja x)         = f x
foldArbol f g (Nodo x t1 t2) =
          g x (foldArbol f g t1) (foldArbol f g t2)
```

- ¿Cuál es el tipo de los constructores?

```
Hoja :: a -> Arbol a
Nodo :: a -> Arbol a -> Arbol a -> Arbol a
```

- ¿Qué similitudes observa con el tipo de foldArbol?

Esquemas en árboles

- Defina una función que sume todos los elementos de un árbol

```
sumArbol :: Arbol Int -> Int
sumArbol = foldArbol id (\n n1 n2 -> n1 + n + n2)
```

- ¿Podría identificar las llamadas recursivas?
- ¿Y si expandimos la definición de foldArbol?

```
sumArbol (Hoja x)          = id x
sumArbol (Nodo x t1 t2) =
                 sumArbol t1 + x + sumArbol t2
```

Esquemas en árboles

- Defina, usando foldArbol una función que:
  - cuente el número de elementos de un árbol

```
sizeArbol = foldArbol (\x->1) (\x s1 s2 -> 1+s1+s2)
```

  - cuente el número de hojas de un árbol

```
hojas = foldArbol (const 1) (\x h1 h2 -> h1+h2)
```

  - calcule la altura de un árbol

```
altura = foldArbol (\x->0) (\x a1 a2 -> 1 + max a1 a2)
```

  - ¿Puede identificar los llamados recursivos?
  - ¿Por qué el primer argumento es una función?

Esquemas en ExpA

- Recordando que

  data ExpA = Cte Int | Sum ExpA ExpA | Mult ExpA ExpA

- ¿Cómo sería la función que representa la recursión estructural sobre ExpA?

```
foldExpA :: (Int->b) -> (b->b->b) -> (b->b->b) -> ExpA -> b
foldExpA fc fs fm (Cte n)           = fc n
foldExpA fc fs fm (Sum e1 e2)       =
                   fs (foldExpA fc fs fm e1) (foldExpA fc fs fm e2)
foldExpA fc fs fm (Mult e1 e2)      =
                   fm (foldExpA fc fs fm e1) (foldExpA fc fs fm e2)
```

Esquemas en ExpA

- ¿Por qué tiene 3 parámetros?
- ¿Por qué esos parámetros tienen esos tipos?
  - Comparar con los tipos de los constructores

```
Cte  :: Int -> ExpA
Sum  :: ExpA -> ExpA -> ExpA
Mult :: ExpA -> ExpA -> ExpA
```

- ¡Cada parámetro coincide con un constructor, pero reemplazando ExpA por b!

Esquemas en ExpA

- Recordando que

  ```
  data ExpA = Cte Int | Sum ExpA ExpA | Mult ExpA ExpA
  ```

- ¿Cómo hacer la evaluación usando foldExpA?

  ```
  evalExpA :: ExpA -> Int
  evalExpA = foldExpA id (+) (*)
  ```

- ¿Y la simplificación?

  ```
  simplificar :: ExpA -> ExpA
  simplificar = foldExpA Cte armarSuma Mult
  ```

Esquemas en otros tipos

```
data A = B | C A Char A | D A A A | E A Char | F Int A
```

- Función de recursión estructural sobre A

```
foldA :: b -> (b->Char->b->b) -> (b->b->b->b)
            -> (b->Char->b) -> (Int->b->b) -> A -> b
foldA b fc fd fe ff B              = b
foldA b fc fd fe ff (C a1 c a2)    = fc (foldA b fc fd fe ff a1) c
                                        (foldA b fc fd fe ff a2)
foldA b fc fd fe ff (D a1 a2 a3)   = fd (foldA b fc fd fe ff a1)
                                        (foldA b fc fd fe ff a2)
                                        (foldA b fc fd fe ff a3)
foldA b fc fd fe ff (E a c)        = fe (foldA b fc fd fe ff a) c
foldA b fc fd fe ff (F n a)        = ff n (foldA b fc fd fe ff a)
```

Esquemas en otros tipos

data A = B | C A Char A | D A A A | E A Char | F Int A

- Contar los caracteres que aparecen en un A

```
contarChar :: A -> Int
contarChar = foldA 0  (\n1 c n2 -> n1+1+n2)
                      (\n1 n2 n3 -> n1+n2+n3)
                      (\n c -> n+1) (\_ n -> n)
```

- Observar que solamente el 2do y el 4to argumentos suman 1 (porque esos constructores tienen un Char)

Esquemas en otros tipos

- Resumiendo:
  - La función fold expresa la recursión sobre un tipo recursivo T (regular)
    - Su resultado es una función T -> b
  - Tiene tantos parámetros como constructores tenga el tipo
  - El tipo de cada parámetro depende del tipo del constructor correspondiente, reemplazando T por b

Árboles alfa-beta

- Considere la siguiente definición

```
data AB a b = Leaf b | Branch a (AB a b) (AB a b)
```

- Defina una función que cuente el número de bifurcaciones de un árbol

```
bifs :: AB a b -> Int
bifs (Leaf x)           = ...
bifs (Branch y t1 t2)   = ... bifs t1 ... bifs t2 ...
```

- Completamos con el significado...

Árboles alfa-beta

- Considere la siguiente definición

```
data AB a b = Leaf b | Branch a (AB a b) (AB a b)
```

- Defina una función que cuente el número de bifurcaciones de un árbol

```
bifs :: AB a b -> Int
bifs (Leaf x)          = 0
bifs (Branch y t1 t2)  = 1 + bifs t1 + bifs t2
```

- ¿Cómo sería el esquema de recursión asociado a un árbol AB?

Árboles alfa-beta

- ¡Utilizamos el esquema de recursión!

```
foldAB :: ??
foldAB f g (Leaf x)             = f x
foldAB f g (Branch y t1 t2) =
          g y (foldAB f g t1) (foldAB f g t2)
```

- ¿Cómo representaría la función bifs?

```
bifs' = foldAB (const 0) (\x n1 n2 -> 1+n1+n2)
```

- ¿Puede probar que bifs' = bifs?

Árboles alfa-beta

- Ejemplo de uso

```
type AExp = AB BOp Int
data BOp = Suma | Producto
```

- ¿Cómo definimos una expresión aritmética usando AExp?

```
exEj = Branch Suma
                   (Branch Producto
                             (Leaf 3)
                             (Leaf 4))
                   (Leaf 5)
-- Representa a (3 * 4) + 5
```

Árboles alfa-beta

- Recordando que

```
data ExpA = Cte Int  | Sum ExpA ExpA
                     | Mult ExpA ExpA
```

- Comparar el ejemplo anterior con la representación equivalente en ExpA

```
exEjEA = Sum  (Mult  (Cte 3)
                     (Cte 4))
              (Cte 5)
-- Representa a (3 * 4) + 5
```

- ¿En qué se diferencian? ¿Cuál elegir?

Árboles alfa-beta

- Ejemplo de uso

```
type AExp = AB BOp Int
data BOp = Suma | Producto
```

- ¿Cómo definimos la semántica de AExp usando foldAB?

```
evalAE :: AExp -> Int
evalAE = foldAB id binOp

binOp :: BOp -> Int -> Int -> Int
binOp Suma      = (+)
binOp Producto = (*)
```

Árboles alfa-beta

- Ejemplo de uso

```
type Decision s a = AB (s->Bool) a
```

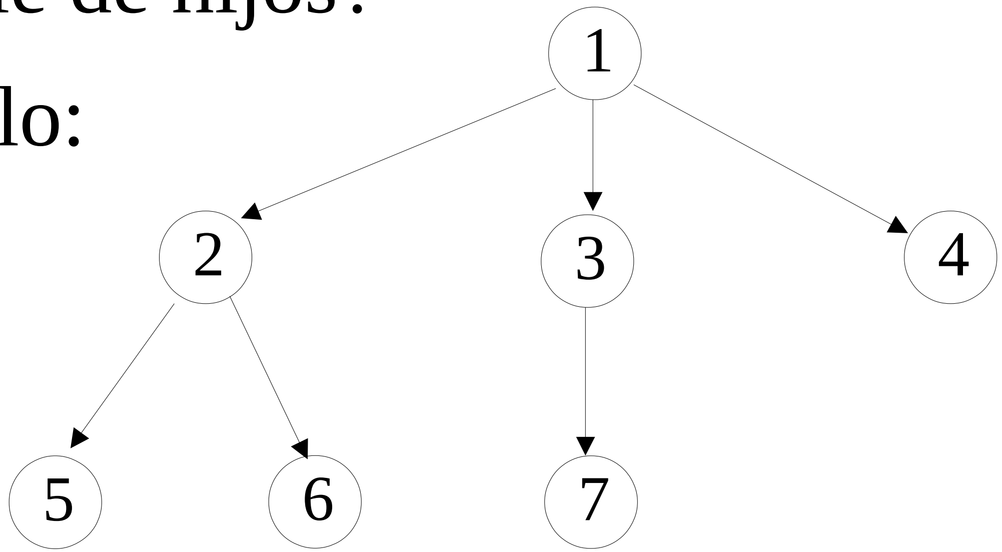
- Definamos una función que dada una situación, decida qué acción tomar basada en el árbol

```
decide :: situation -> Decision situation action -> action
decide s = foldAB id  (\f a1 a2 -> if (f s) then a1 else a2)

ej = Branch f1 (Leaf "Huya")
        (Branch f2 (Leaf "Trabaje") (Leaf "Quédese manso"))
       where f1 s = (s==Fuego) || (s==AtaqueExtraterrestre)
             f2 s = (s==VieneElJefe)
```

Árboles Generales

- ¿Cómo representar un árbol con un número variable de hijos?
- Ejemplo:



- Idea: ¡usar una lista de hijos!

Árboles Generales

- Ello nos lleva a la siguiente definición:

  data AG a = GNode a [AG a]

- Pero, ¿tiene caso base? ¿cuál?
  - Un árbol sin hijos...
- ¡Se basa en el esquema de recursión de listas!
  - O sea, el caso base es (GNode x [ ]); por ejemplo:

```
GNode 1 [ GNode 2 [ GNode 5 [ ], GNode 6 [ ] ]
        , GNode 3 [ GNode 7 [ ] ]
        , GNode 4 [ ]
        ]
```

Árboles Generales

- Definir una función que sume los elementos

  ```
  sumAG :: AG Int -> Int
  ```

- ¿Cómo la definimos?
  - ¡Usando funciones sobre listas!

  ```
  sumAG (GNode x ts) = x + sum (map sumAG ts)
  ```

- Y esto, ¿es estructural?
  - Sí, pues se basa en la estructura de las listas
- Se ve la utilidad de funciones de alto orden...

Árboles Generales

- ¿Cómo sería el esquema de recursión?
  Hay varias posibilidades
  - Según la receta de una función por constructor

```
foldAG0 :: (a->[b]->b) -> AG a -> b
foldAG0 h (GNode x ts) = h x (map (foldAG0 h) ts)
```

  y entonces, la función sumAG queda

```
sumAG0 = foldAG0 (\x ns -> x + sum ns)
```

  - ¡El problema es que no es recursión estructural!

Árboles Generales

- ¿Cómo sería el esquema de recursión? (2)
  - Completamente estructural

```
foldAG1 :: (a->c->b) -> (b->c->c) -> c -> AG a -> b
foldAG1 g f z (GNode x ts) =
        g x (foldr f z (map (foldAG1 g f z) ts))
```

y entonces, la función sumAG queda

```
sumAG1 = foldAG1 (+) (+) 0  -- sum = foldr (+) 0
```

  - Siempre termina, porque es estructural
  - ¡El problema es que es difícil de pensar!

Árboles Generales

- ¿Cómo sería el esquema de recursión? (3)
  - Opción intermedia entre ambas

```
foldAG :: (a->c->b) -> ([b]->c) -> AG a -> b
foldAG g k (GNode x ts) =
        g x (k (map (foldAG g k) ts))
```

y entonces, la función sumAG queda

```
sumAG = foldAG (+) sum
```

  - No es estructural, pero es bastante clara

Árboles Generales

- ¿Cuál es mejor? Depende del uso y el gusto

```
sumAG0 = foldAG0 (\x ns -> x + sum ns)
sumAG1 = foldAG1 (+) (+) 0  -- sum = foldr (+) 0
sumAG' = foldAG (+) sum
```

- Otras funciones sobre árboles generales:

```
depthAG = foldAG (\x d -> 1+d) (maxWith 0)
    where maxWith x []   = x
          maxWith x xs = maximum xs
mirrorAG = foldAG GNode reverse
```

“La tarea de un pensador no consistía para Shevek en negar una realidad a expensas de otra, sino en integrar y relacionar. No era una tarea fácil.”

Los Desposeídos
Úrsula K. Le Guin

“Enseñen a los niños a ser preguntones para que pidiendo el por qué de lo que se les manda, se acostumbren a obedecer a la razón, no a la autoridad como los limitados, ni a la costumbre como los estúpidos.”

Simón Rodríguez,

maestro del Libertador, Simón Bolívar.

“La persona que toma lo banal y lo ordinario y lo ilumina de una nueva forma, puede aterrorizar. No deseamos que nuestras ideas sean cambiadas. Nos sentimos amenazados por tales demandas. «¡Ya conocemos las cosas importantes!», decimos. Luego aparece el Cambiador y echa a un lado todas nuestras ideas.

-El Maestro Zensunni”

Casa Capitular: Dune
Frank Herbert

“Un mago sólo puede dominar lo que está cerca, lo que puede nombrar con la palabra exacta.”

Un mago de Terramar

Úrsula K. Le Guin

“ - Maestro - dijo Ged -, no soy tan vigoroso como para arrancarte el nombre por la fuerza, ni tan sabio como para sacártelo por la astucia. Me contento pues, con quedarme aquí y aprender o servir, lo que tú prefieras; a menos que consintieras, por ventura, a responder a una pregunta mía.

- Hazla.

- ¿Qué nombre tienes?

El Portero sonrió, y le dijo el nombre.”

Un mago de Terramar
Úrsula K. Le Guin