Apunte Final PLP

Mateo Ziffer

5 de julio de 2025

Índice

1. Introducción

En esta sección escribiré algunas definiciones generales y el objetivo de la materia.
Def: la *programación* es el proceso de escribir instrucciones que una computadora puede ejecutar para resolver algún problema.
Def: un *programa* es una serie de instrucciones/definiciones que una computadora sigue para realizar una tarea específica.
Def: un *lenguaje de programación* es un formalismo artificial en el que se pueden describir compataciones.
Def: un *paradigma* es una marco filosófico y teórico de una escuela científica o disciplina en la que se formulan teorías, leyes y generalizaciones y se llevan a cabo experimentos que les dan sustento.
Def: un *paradigma de programación* es una marco filosófico y teórico en el que se formulan soluciones a problemas de naturaliza algorítmica.
Lo entendemos como un estilo de programación en el que se escriben soluciones a problemas en términos de algoritmos.

Estudiamos la gramática, semántica, pragmática e implementación de los lenguajes de programación.
Def: la *gramática* responde a ¿Qué frases son correctas?, establece el alfabeto, las palabras (o tokens), es decir, la secuencia válida de símbolos y la sintaxis, es decir, las secuencias de palabras que son frases legales.
Def: la *semántica* responde a ¿Qué significa una frase correcta?, estableciendole un significado a cada frase correcta.
Def: la *pragmática* responde a ¿Cómo usamos una frase significativa?. Las frases con el mismo significado pueden usarse de diferentes maneras, diferentes contextos pueden requerir frases más elegantes, eficientes, dialectales, etc.
Def: la *implementación* responde a ¿Cómo ejecutar una frase correcta, de manera que respetemos la semántica?. Es fundamentas para los diseñadores e implementadores del lenguaje, no necesariamente para el usuario (programador).

Hay tres apectos importantes de los lenguajes de programación:
Motivación de la *programación*: los lenguajes de programación tienen distintas características que permiten abordar un mismo problema de distintas ma los lenguajes de programación tienen distintas características que permiten abordar un mismo problema de distintas maneras.
Motivación de la *semántica*: probar teoremas sobre el comportamiento de los programas, para darles significado matemático y poder confiar en que hace lo que queremos, en AED vimos como hacerlo con triplas de Hoare, pero en PLP veremos otras maneras de dar semántica.
Motivación: la *implementación*: una computadora física ejecuta programas escritos en un lenguaje, el código máquina, pero necesita poder ejecutar programas escritos en otros lenguajes, a través de la interpretación, el chequeo e inferencia de tipos y la compilación.

2. Haskell

2.1. Programación funcional

La *programación funcional* consiste en definir funciones y aplicarlas para procesar información.

Las *funciones* son verdaderamente funciones (parciales):

- Aplicarlas no tiene efectos secundarios.
- A una misma entrada corresponde siempre la misma salida (determinismo).
- Las estructuras de datos son inmutables.

Las funciones son datos como cualquer otro, se pueden pasar como parámetros, devolver como resultado y formar parte de estructuras de datos.

Un *programa funcional* está dado por un conjunto de ecuaciones.

2.1.1. Gramática

Las *expresiones* son secuencias de símbolos que sirven para representar datod, funciones y funciones aplicadas a los datos. Una expresión puede ser:

1. Un contructor: True False [] (:) 0 1 2 ...
2. Una variable: longitud ordenar x xs (+) (*) ...
3. La aplicación: de una expresión a otra: ordenar lista, not True, ((+) 1) (alCuadrado 5)
4. ...

La aplicación es asociativa a izquierda

$$\begin{aligned} \text{f x y} &\equiv \text{(f x) y} \not\equiv \text{f (x y)} \\ \text{f a b c d} &\equiv \text{(((f a) b) c) d} \end{aligned} \tag{1}$$

2.1.2. Tipos

Hay secuencias de símbolos que no son expresiones bien formadas como 1„2 ó)f x(, y hay expresiones que están bien pormadas pero no tienen sentido, como True + 1, 0 1 y [[], (+)].

Un *tipo* es una especificación del invariante de un dato o de una función. El tipo de una función expresa un *contrato*.

```
99 :: Int
not :: Bool -> Bool
not True :: Bool
((+) 1) 2 :: Int
```

→es asociativo a derecha.

$$\begin{aligned} \text{a} \to\text{b} \to\text{c} &\equiv \text{a} \to(\text{b} \to\text{c}) \not\equiv (\text{a} \to\text{b}) \to\text{c} \\ \text{a} \to\text{b} \to\text{c} \to\text{d} &\equiv \text{a} \to(\text{b} \to(\text{c} \to\text{d})) \end{aligned} \tag{2}$$

2.1.3. Condiciones de tipado

Para que un programa esté *bien tipado*:

1. todas las expresiones deben tener tipo.
2. cada variable se debe usar siempre con un mismo tipo.
3. los dos lados de una ecuación deben tener el mismo tipo.
4. el argumento de una función debe tener el tipo del dominio.
5. el resultado de una función debe tener el tipo del codominio.

$$\frac{\text{f} :: \text{a} \rightarrow \text{b} \quad \text{x} :: \text{a}}{\text{f x} :: \text{b}} \tag{3}$$

Sólo tienen sentido los programas bien tipados. No es necesario escribir explícitamente los tipos (Inferencia).

2.1.4. Polimorfismo

Hay expresiones que tienen más de un tipo, usamos variables de tipo a, b, c para denotar tipos desconocidos:

```
id :: a -> a
[] :: [a]
(:) :: a -> [a] -> [a]
fst :: (a,b) -> a
flip :: (a -> b -> c) -> b -> a -> c
```

2.1.5. Modelo de cómputo

Dada una expresión, se computa su valor usando las ecuaciones.
Hay expresiones bien tipadas que no tienen valor como 1 / 0. Decimos que se indefinen o tienen el valor $\perp$.
Un programa funcional está dado por un conjunto de *ecuaciones orientadas*. Ena ecuación e1 = e2 se interpreta desde dos puntos de vista:
Def: Punto de vista denotacional: declara que e1 y e2 tienen el mismo significado.
Def: Punto de vista operacional: computar el valor de e1 se reduce a computar el valor de e2.
El lado izquiedo de una ecuación no es una expresión arbitraria, debe ser una función aplicada a *patrones*. Un patrón puede ser una variable, un comodín ó un consstructor aplicado a patrones. Este no debe contener variables repetidas.

Evaluar una expresión consiste en:

1. buscar la subexpresión más externa que coincida con el lado izquerdo de una ecuación.
2. reemplazar la subexpresión que coincide con el lado izquierdo de la ecuación por la expresión correspondiente al lado derecho.
3. continuar evaluendo la expresión resultante.

La evaluación se detiene cuando se da uno de los siguientes casos:

1. la expresión en un constructor o un constructor aplicado. True, (:) 1, [1,2,3].

2. la expresión es una función parcialmente aplicada. (+), (+) 5.
3. se alcanza un estado de error, es decir, una expresión que no coincide con las ecuaciones que definen a la función aplicada.

Ejemplos de evaluaciones y resultados

1. constructor: tail (tail [1,2,3]) ⇝tail [2,3] ⇝[3]
2. función parcialmente aplicada: const (const 1) 2 ⇝const 1
3. error: head (head [[], [1], [1,1]]) ⇝head [] ⇝⊥
4. no terminación: loop n = loop (n + 1), loop 0 ⇝loop (1 + 0) ⇝loop(1 + (1 + 0)) ⇝loop (1 + (1 + (1 + 0))) ⇝...
5. evalucación no estricta: indefinido = indefinido, head (tail [indefinido, 1, indefinido]) ⇝head [1, indefinido] ⇝1
6. listas infinitas: desde n = n : desde (n + 1), desde 0 ⇝0 : desde 1 ⇝0 : (1 : desde 2) ⇝0 : (1 : (2 : desde 3)) ⇝...
7. listas infinitas 2: head (tail (desde 0)) ⇝head (tail (0 : desde 1)) ⇝head (desde 1) ⇝head (1 : desde 2) ⇝1

Obs: en Haskell, el orden de las ecuaciones es relevante, si hay varias ecuaciones que coinciden siempre se usa la primera.

2.1.6. Listas

El tipo [a] denota listas de elementos de tipo a. Por ejemplo, [1,2,3] abrevia a 1:2:3:[], (:) :: a ->[a] ->[a] es el operador cons, que es el constructor de listas, asocia a derecha y no tiene una definición asociadia ya que es un constructor, luego la expresión no se puede simplificar más.
Las listas toman una de las siguientes formas. Una lista indefinida, undefined :: [a], una lista vacía [] :: [a] y una lista de la forma x:xs donde x :: a y xs :: [a].
Luego hay tres tipos de listas. Las listas finitas como 1:2:3:[], las listas parciales como 1:2:3:undefined ó filter (<4) [1..] ⇝1:2:3:undefined y las listas infinitas como [1..].

```haskell
iterate :: (a -> a) -> a -> [a]
iterate f x = x:iterate f (f x)
-- ghci> iterate (+1) 1

-- ghci> divisors n = filter (\m -> n `mod` m == 0) [1..(n-1)]
-- ghci> divisors 6
-- [1,2,3]
-- ghci> head (filter (\n -> n == sum (divisors n)) [1..])
-- 6

until p f = head . filter p . iterate f
-- ghci> until (\n -> n == sum (divisors n)) (+1) 1
-- 6
```

Las listas pueden ser enumeradas por la clase Enum, por ejemplo:

```
[m..n] ⇝ [m,m+1,..,n]
[m..] ⇝ [m,m+1,..]
[m,n..p] ⇝ [m,m+(n-m),m+2(n-m),..,p]
[m,n..] ⇝ [m,m+(n-m),m+2(n-m),..]
['a'..'z'] ⇝ "abcdefghijklmnopqrstuvwxyz"
```

Las listas también pueden ser definidas por comprensión:

```
[x*x | x <- [1..5]] ⇝ [1,4,9,16,25]
[(i,j) | i <- [1..5], even i, j <- [i..5]]
   ⇝ [(2,2),(2,3),(2,4),(2,5),(4,4),(4,5)]
[x | xs <- [[(3,4)],[(5,4),(3,2)]], (3,x) <- xs] ⇝ [4,2]

divisors x = [d | d <- [2..x-1], x `mod` d == 0]

disjoint [] _ = True
disjoint (x:xs) ys = not (x `elem` ys) && disjoint xs ys

coprime x y = disjojint (divisors x) (divisors y)

triads n = [(x,y,z) | x <- [1..m], y <- [x+1..n],
                      coprime x y,
                      z <- [y+1..n], x*x+y*y==z*z]
           where m = floor (fromIntegral n / sqrt 2) -- optimizacion aprovechando
                                                     -- que 2x^2 < x^2 + y^2 = z^2 ≤ n^2, luego x < ⌊n/√2⌋

map f xs = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
concat xss = [x | xs <- xss, x <- xs]
```

En realidad Haskell hace lo contrario, evalua las listas por comprensión en términos de map y concat, sus reglas de traducción son:

```
[e | True]        = [e]
[q | q]           = [e | q, True]
[e | b, Q]        = if b then [e | Q] else []
[e | p <- xs, Q] = let ok p = [e | Q]
                       ok _ = []
                   in concat (map ok xs)
[e | Q1, Q2] = concat [[e | Q2] | Q1]
```

La definición de ok usa un patrón don't care, o un wild card. Dice que la lista vacia se retorna para cualquier elemento que no une con el patrón p.

Podemos definir funciones sobre listas con patterrn matching y sabemos que [] y x:xs son disjuntos y exhaustivos. También podemos usar el don't care pattern.

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

null2 :: [a] -> Bool
```

```haskell
null2 [] = True
null2 _ = False

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

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

-- en last el orden importa!
last :: [a] -> a
last [x] = x -- [x] ≡ (x:_)
last (_:xs) = last xs
```

2.1.7. Funciones de orden superior

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
(g . f) x = g (f x)
-- o bien (g . f) = \x -> g (f x), con notación lambda

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

dobleL = map (*2)
longitudL = map length

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

curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

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

Prop: Sean `suma x y = x + y`, `suma' (x, y) = x + y`,

- `suma = curry suma'`
- `suma' = uncurry suma`

Obs: Por notación lambda, una expresión de la forma `\ x -> e` representa una función que recibe un parámetro x y devuelve e.

$$(\backslash\ x1\ x2\ \ldots\ xn\ \texttt{->}\ e) \equiv (\backslash\ x1\ \texttt{->}\ (\backslash\ x2\ \texttt{->}\ \ldots\ (\backslash\ xn\ \texttt{->}\ e)))$$

2.1.8. Más funciones y propiedades

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

Ejemplo de secuencia de evaluación

```
[1,2] ++ [3,4,5]
= (1:(2:[])) ++ (3:(4:(5:[])))
= 1:((2:[]) ++ (3:(4:(5:[]))))
= 1:(2:([] ++ (3:(4:(5:[])))))
= 1:(2:(3:(4:(5:[]))))
= [1,2,3,4,5]
```

La concatenación es asociativa, ¿Pero esta implementación la hace conmutativa?

```
undefined ++ [1,2] = undefined
[1,2] ++ undefined = 1:2:undefined
```

Listaré propiedades que pueden ser probadas con razonamineto ecuacional:

```
filter p = concat . map (\x -> if p x then [x] else [])

-- estas dos se llamas functor laws of map, nombre prestado de Teoría de Categorías
map id = id
map (f . g) = map f . map g

-- Haskell provee una clase de tipos Function cuya definición es
class Functor f where
  fmap :: (a -> b) -> f a -> f b

-- ahora que lo generalizé puedo aplicarlo a otras estructuras
data Tree a = Tip a | Fork (Tree a) (Tree a)
instance Functor Tree whee
  fmap f (Tip x) = Tip (f x)
  fmap f (Fork u v) = Fork (fmap f u) (fmap f v)

-- En realidad map es un sinónimo para la instancia fmap para listas
map(+1) [2,3,4] ≡ fmap (+1) [2,3,4]

-- Para operaciones que no dependen de la naturaleza de los elementos de la lista.
-- Son funciones que mezclan, descartan o extraen elementos de listas.
-- funciones con tipos polimorficos satisfacen alguna.
-- ley que se pueden cambiar valores antes de aplicar la función.
-- En matemáticas se llaman transformaciones naturales y
-- las leyes asociadas leyes de naturalidad.
f . head = head . map f -- sólo si f es estricta (mirar caso [])
map f . tail = tail . map f
map f . concat = concat . map (map f)

-- otro ejemplo
```

```haskell
map f . reverse = reverse . map f

-- mas propiedades
concat . map concat = concat . concat
filter p . map f = map f . filter (p . f)
```

Las siguientes son las definiciones de zip y zipWith

```haskell
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y): zip xs ys
zip _ _ = []

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _ _ = []

-- zip se puede definir con zipWith y el constructor de pares
zip ≡ zipWith (,)

nondec :: (Ord a) => [a] -> Bool
nondec xs = and (zipWith (<=) xs (tail xs))

position :: (Eq a) => a -> [a] -> Int
position x xs = head ([j | (j,y) <- zip [0..] xs, y==x] ++ [-1])

sort :: (Ord a) => [a] -> [a]
sort [] = []
sort [x] = [x]
sort xs = merge (sort ys) (sort zs)
  where (ys,zs) = halve xs

halve xs = (take n xs, drop n xs)
  where n = length xs `div` 2

merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y = x:merge xs (y:ys)
  | otherwise = y:merge (x:xs) ys

-- esta ultima linea se podria escribir asi
merge xs'(x:xs) ys'(y:ys)
  | x <= y = x:merge xs ys'
  | otherwise = y:merge xs' ys

enumerarPares :: (Num a) -> [(a,a)]
enumerarPares = [(x,y) | n <- [0..], x <- [0..n], y <- [n-x]]
```

Obs: Haskell define la comparación <= en pares como

```
(x1,y1) <= (x2,y2) = (x1,x2) || (x1 == x2 && y1 <= y2)
```

2.1.9. Tipos de datos algebraicos

Conocemos algunos tipos de datos "primitivos", como Char, Int, Float, (a ->b), (a, b), [a], String (sinónimo de [Char]).

Se pueden definir nuevos tipos de datos con la cláusula data `data Tipo =` <declaración de los constructores>

Por ejemplo, `data Dia = Dom | Lun | ... | Sab` declara que existen constructores `Dom :: Dia`, `Lun :: Dia`, ..., `Sab :: Dia` y que son los únicos constructores del tipo Dia.

```
esFinDeSemana :: Dia -> Bool
esFinDeSemana Sab = True
esFinDeSemana Dom = True
esFinDeSemana _ = False
```

Un constructor puede tener muchos parámetros, por ejemplo, `data Persona = LaPersona String Stri` declara que el único constructor de persona es `LaPersona :: String -> String -> Int -> Persona`.

```
nombre :: Persona -> String
nombre (LaPersona n _ _) = n
```

Un tipo puede tener muchos constructores,

```
data Forma = Rectangulo Float Float
           | Circulo Float
```

declara que el tipo Forma tiene dos constructores y sólo esos.

Los constructores pueden ser recursivos,

```
data Nat = Zero
         | Succ Nat

-- valores de la forma
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...

doble :: Nat -> Nat
doble Zero = Zero
doble (Succ n) = Succ (Succ (doble n))

-- Haskell permite trabajar con estructurAas infinitas
-- porque las definiciones recursivas se interpretan de manera
-- coinductiva en lugar de inductiva
infinito :: Nat
infinito = Succ infinito
```

Forma general de tipos algebráicos:

```
data T = CBase_1 <parámetros>
         ...
       | CBase_n <parámetros>
       | CRecursivo_1 <parámetros>
         ...
       | CRecursivo_n <parámetros>
```

Los constructores base no reciben parámetros de tipo T y los recursivos reciben al menos uno. Los valores de tipo T son los que se pueden construir aplicando constructores base y recursivos un número finito de veces y sólo esos, es decir, entendemos la definición de T de forma inductiva.

Ejemplo `data Lista a = Vacia | Cons a (Lista a)`

Ejemplo `data AB a = Nil | Bin (AB a) a (AB a)`. Definir preorder, postorder e inorder.

```
data AB a = Nil | Bin (AB a) a (AB a)

preorder :: AB a -> [a]
preorder Nil = []
preorder (Bin i v d) = v : (preorder i ++ preorder d)

inorder :: AB a -> [a]
inorder Nil = []
inorder (Bin i v d) = inorder i ++ (v : inorder d)

postorder :: AB a -> [a]
postorder Nil = []
postorder (Bin i v d) = postorder i ++ postorder d ++ [v]
```

2.2. Esquemas de Recursion

2.2.1. Sobre Listas

2.2.1.1. Recursión estructural

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g []     = <caso base>
g (x:xs) = < caso recursivo>
```

g está dada por recursión estructural si:

- El caso base devuelve un valor fijo z.
- El caso recursivo se escribe usando (cero, una o muchas veces) x y (g xs), pero sin usar el valor de xs ni de otros llamados recursivos.

Ejemplo suma, (++), isort están dadas por recursión estructural, pero ssort no.

```
-- dada por recursion estructural
isort :: Ord a => [a] -> [a]
```

```
isort [] = []
isort (x:xs) = insertar x (isort xs)

-- no dada por recursion estructural
ssort :: Ord a => [a] -> [a]
ssort [] = []
ssort (x:xs) = minimo (x:xs)
             : ssort (sacarMinimo (x:xs))
```

foldr abstrae el esquema de recursión estructural

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

toda recursión estructural es una instancia de foldr.

Ejemplo

```
suma = foldr (+) 0

suma [1, 2]
  ⇝ foldr (+) 0 [1,2]
  ⇝ (+) 1 (foldr (+) 0 [2])
  ⇝ (+) 1 ((+) 2 (foldr (+) 0 []))
  ⇝ (+) 1 ((+) 2 0)
  ⇝* 3
```

Ejemplo

```
reverse = foldr (\x rec -> rec ++ [x]) []
  ≡ foldr (\x rec -> (++) rec [x]) []
  ≡ foldr (\x rec -> flip (++) [x] rec) []
  ≡ foldr (\x -> flip (++) [x]) []
  ≡ foldr (\x -> flip ((++) (:[]) x)) []
  ≡ foldr (flip (++) . (:[])) []
```

Prop:

- `foldr (:) [] = id`
- `foldr ((:) . f) [] = map f`
- `foldr (const (+ 1)) = [] = length`

Ejemplo Casos degenerados

```
length [] = 0
length (_:xs) = 1 + length xs -- no usa la cabeza

head [] = error "No tiene cabeza."
head (x:_) = x -- no usa el llamado recursivo sobre la cola
```

2.2.1.2. Propiedad universal de fold

Al igual que el operador fold encapsula un patrón simple de recursión para procesar listas, la propiedad universal de fold encapsula un patrón simple de prueba inductiva para listas.

Prop: universalidad de fold

```
g [] = v                    ⇔ g = fold f v
g (x:xs) = f x (g xs)
```

Ejemplo universalidad como principio de prueba
Queremos ver que `(+1) . sum = fold (+) 1`. Primero notamos que satisface el lado derecho de la propiedad universal de fold, con `g = (+1) . sum`, `f = (X)` y `v = 1`. Luego, por la propiedad universal, concluimos que la ecuación a ser probada es equivalente a las dos siguientes ecuaciones:

```
((+1) . sum) [] = 1
((+1) . sum) (x:xs) = (+) x (((+1) . sum) xs)
```

Luego simplificando

```
sum [] + 1 = 1
sum (x:xs) + 1 = x + (sum xs + 1)
```

Ahora, se puede calcularlo sólo con definiciones

```
sum [] + 1
  = 0 + 1      {Definicion de sum}
  = 1          {Aritmética}
sum (x:xs) + 1
  = (x + sum xs) + 1 {Definicion de sum}
  = x + (sum xs + 1) {Asociatividad de la suma}
```

Y queda probado sin uso explícito de inducción. □

Con la propiedad de universalidad de fold se pueden probar propiedades más específicas, como la propiedad de fusión de fold. Esta puede ser un poco más simple en los casos que se puede aplicar, pero siempre que se puede hacer una prueba for fusión se puede hacer por universalidad.

Ejemplo universalidad como principio de definición.
Sea la función sum sum definida por recursión estructural de la siguiente forma:

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Queremos definir sum con fold. Luego queremos resolver la ecuación `sum = fold f v` para una función f y un valor v. Observamos que la ecuación une con el lado derecho de la propiedad universal, así que concluimos que la ecuación es equivalente a las siguientes ecuaciones:

```
sum [] = v
sum (x:xs) = f x (sum xs)
```

De la primera ecuación y la def de sum, v = 0. De la segunda ecuación calculamos f

```
sum (x:xs) = f x (sum xs)
⇔ x + sum xs = f x (sum xs) {Definición de sum}
⇐ x + y = f x y             {Generalizando (sum xs) a y}
⇔ f = (+)                   {Funciones}
```

Luego, usando la propiedad universal obtuvimos que `sum = fold (+) 0`. □

2.2.1.3. Recursión primitiva

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g []     = <caso base>
g (x:xs) = <caso recursivo>
```

g está dada por recursión primitiva si:

- El caso base devuelve un valor fijo z.
- El caso recursivo se escribe usando (cero, una o muchas veces) x, (g xs) y también xs, pero sin hacer otros llamados recursivos.

Es decir, es similar a la recursión estructural, pero permite referirse a xs.

Obs: Todas las definiciones dadas por recursión estructural también están dadas por recursión primitiva.
Obs: Hay definiciones dadas por recursión primitiva que no están dadas por recursión estructural.
Es decir, estructural ⇒ primitiva, pero primitiva ⇏ estructural.

Ejemplo trim se puede definir con recursión primitiva, pero no con recursión estructural.

recr abstrae el esquema de recursión primitiva

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

toda recursión primitiva es una instancia de recr.

Ejemplo

```
trim = recr (\ x xs rec -> if x == ' ' then rec else x:xs) []
```

2.2.1.4. Recursión iterativa

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g ac []     = <caso base>
g ac (x:xs) = < caso recursivo>
```

g está dada por recursión iterativa si:

- El caso base devuelve el acumulador ac.
- El caso recursivo invoca inmediatamente a (g ac' xs) donde ac' es el acumulador acutualizado en función de su valor anterior y el valor de x.

Es decir, es similar a la recursión estructural, pero permite referirse a xs.

Ejemplo

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

foldl abstrae el esquema de recursión iterativa

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

toda recursión iterativa es una instancia de foldl.

En general foldr y foldl tienen comportamiento diferentes:

```
foldr (⊕) z [a,b,c] = a ⊕ (b ⊕ (c ⊕ z))
foldl (⊕) z [a,b,c] = ((z ⊕ a) ⊕ b) ⊕ c
```

si $\oplus$ es un operador asociativo y conmutativo, foldr y foldl definen la misma función.

Ejemplo

```
suma = foldr (+) 0 = foldl (+) 0
producto = foldr (*) 1 = foldl (*) 1
and = foldr (&&) True = foldl (&&) True
or = foldr (||) False = foldl (||) False
```

Ejemplo

```
tern2dec :: [Int] -> Int
tern2dec = foldl (\ ac b -> b + 3 * ac) 0

tern2dec [2,1,1]
  ⇝ foldl (\ ac b -> b + 3 * ac) 0 [2,1,1]
  ⇝ foldl (\ ac b -> b + 3 * ac) (2 + 3 * 0) [2,1]
  ⇝ foldl (\ ac b -> b + 3 * ac) (1 + 3 * (2 + 3 * 0)) [1]
  ⇝ foldl (\ ac b -> b + 3 * ac) (1 + 3 * (1 + 3 * (2 + 3 * 0))) []
  ⇝ 1 + 3 * (1 + 3 * (2 + 3 * 0))
  ⇝* 22
```

La función foldl es un operador de iteración. Pseudocódigo imperativo:

```
función foldl f ac xs
  mientras xs no es vacía
    ac := f ac (head xs)
    xs := tail xs
  devolver ac
```

Se puede demostrar que `foldl (flip (:)) [] = reverse`.

2.2.1.5. Ejercicios

Ejercicio Definir la siguiente función usando foldr.

```
zip :: [a] -> [b] -> [(a,b)]
zip [] [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Primero mostraré que zip está definida sii los largos de las dos listas son iguales.

Demo: Sean a, b tipos, $A :: [a], B :: [b]$ listas, $n, m \in \mathbb{Z}$ las longitudes de las listas A y B respectivamente. Quiero ver que `zip A B` está definida si y sólo si $n = m$ y no hay elementos indefinidos en las listas.

Si hay algun elemento indefinido en alguna de las listas claramente en algún momento de la recursión ese elemento será x ó y si $n \neq m$, luego la expresión se indefinirá ó ocurrirá que $n = m$. Luego me reduzco a ver que zip está definida sii $n = m$.

Para la demostración usaré como invariante que a través de las llamadas recursivas de zip, la igualdad/desigualdad de n y m se preserva. En el caso base se ve que no hay llamadas recursivas. En el caso recursivo sólo se llama a `zip xs ys`, es decir, los nuevos valores de n y m son $n - 1$ e $y - 1$, luego como $n = m \Leftrightarrow n - 1 = m - 1$ se mantiene el invariante.

Si $n = m$. Por inducción en n,

Caso base. Si $n = m = 0$ luego A y B son listas con cero elementos y por lo tanto `A = []` y `B = []`. Luego `zip A B = zip [] [] = []` por la primera regla y entonces está definida.

Caso inductivo. Si $n = m > 0$, luego A y B son listas con $n > 0$ elementos, luego se usa la segunda ecuación de zip, que está definida si y sólo si `(x, y) : zip xs ys` está definida. Claramente `(x, y)` está definido, pues x e y están definidos. Además `zip xs ys` está definido por hipótesis inductiva. Luego la expresión entera está definida.

Luego por inducción en n, $\forall k$, si ambas listas tienen tamaño k la función está definida.

Si $n \neq m$, por invariante se mantiene la desigualdad de las llamadas recursivas. Además el tamaño de ambas listas se reduce en uno en cada llamado recursivo. Luego eventualmente alguna de las listas se intentará llamar con tamaño 0 y la otra con tamaño mayor a 0, por lo cual no habrá definición de la función.

Queda probado que `zip A B` está definida si y sólo si $n = m$ y los elementos de ambas listas están definidos.

□

Quisiera escribir algo de la forma `zip xs ys = foldr f z xs ys`, con f siendo la función que hay que definir y z el caso base, para poder reescribirlo como `zip = foldr f z`. Si hago esto, estoy evaluando el foldr con xs, y esto debería devolver una función que al evaluarla con ys me de el resultado de zip. Es decir, `foldr f z xs :: [b] -> [(a, b)]`. Entonces `z :: [b] -> [(a,b)]`, así que defino z como `(\ _ -> [])` definiendo el caso base de la función que retornaré. Ahora quiero en cada paso de la recursión agregar un caso a la función. Luego defino a f como `(\ x rec (y:ys) -> (x, y) : rec ys)`.

Luego tengo
```
zip xs ys = foldr (\ x rec -> (y:ys) -> (x, y) : rec ys) (_ -> []) xs ys
```

Que es equivalente a
```
zip = foldr (\ x rec (y:ys) -> (x, y) : rec ys) (_ -> [])
```

Ejercicio Definir foldr en términos de recr.

Recordemos que todas las definiciones dadas por recursión estructural también están dadas por recursión primitiva. La única diferencia entre recr y foldr es que en la función que se pasa se recibe el parámetro xs, solo debemos ignorarlo.

Ahora, `foldr f z xs = recr (\ x xs rec -> f x rec) z xs`.
Esto es equivalente a `foldr f = recr (\ x _ -> f x)`.

Ejercicio Definir recr en términos de recr.

Recordemos que hay definiciones dadas por recursión primitiva que no están dadas por recursión estructural. Esto no significa que el ejercicio sea imposible, sino que tenemos que modificarla definición para que siempre sea posible. Esto lo haremos devolviendo una tupla con una copia de la lista original en el caso base de la recursión, así toda la lista es accesible en todas las ejecuciones de la función a través de la tupla rec que renombramos (xs,rec) ya que contiene la lista original ademas el llamado recursivo que antes devolvíamos.

```
recr f z xs = foldr (\ x (xs,rec) -> ??) (z,xs) xs
```

Ahora nos falta devolver el valor recursivo, es decir, `f x xs rec`, pero no hay que olvidarse de preservar la tupla (xs,rec).

```
recr f z xs = foldr (\ x (xs,rec) -> (xs, f x xs rec)) (z,xs) xs
```

El problema que tenemos ahora es que la función recr devolvería la tupla (xs,resultado), y solo queremos el resultado, así que usamos la función snd para obtener el segundo valor de la tupla.

```
recr f z xs = snd (foldr (\ x (xs,rec) -> (xs, f x xs rec)) (z,xs) xs)
```

El último problema que tenemos es que en xs tenemos siempre toda la lista, pero en realidad queremos que tenga los elementos siguientes, no todos. Entonces lo que hacemos es en vez de pasar toda la lista en el caso base, pasamos una lista vacía, y en cada paso de la recursión agregamos el elemento actual, así en el backtracking iremos teniendo una cada vez un elemento más.

```
recr f z xs = snd (foldr (\x((x:xs),rec) -> (xs, f x xs rec)) ([], z) xs)
```

Ejercicio definir foldl en términos de foldr.

Lo único que cambia entre foldl y foldr es el orden de los términos.

```
foldl f z xs = foldr (flip f) z (reverse xs)
```

Y reverse sabemos que se puede escribir en términos de foldr.

Ejercicio definir foldr en términos de foldl.

```
foldr f z xs = foldl (flip f) z (reverse xs)
```

Y reverse sabemos que se puede escribir en términos de foldl.

2.2.2. Sobre Otras Estructuras

La recursión estructural se generaliza a tipos algebraicos en general. Supongamos que T es un tipo algebráico.
Dada una función `g :: T -> Y` definida por ecuaciones:

```
g (CBase_1 <parámetros>) = <caso base_1>
...
g (CBase_n <parámetros>) = <caso base_n>
g (CRecursivo_1 <parámetros>) = <caso recursivo_1>
```

```
...
g (CRecursivo_n <parámetros>) = <caso recursivo_n>
```

Decimos que g está dada por recursión estructural si:

- Cada caso base se escribe combinando los parámetros.
- Cada caso recursivo se escribe combinando los parámetros del constructor que no son de tipo T y el llamado recursivo sobre cada parámetro de tipo T, pero sin unar los parámetros del constructor que son de tipo T y sin hacer otros llamados recursivos.

Ejemplo `data AB a = Nil | Bin (AB a) a (AB a)`

```haskell
foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
foldAB cNil cBin Nil = cNil
foldAB cNil cBin (Bin i r d) =
  cBin (foldAB cNil cBin i) r (foldAB cNil cBin d)

idAB :: AB a -> AB a
idAB = foldAB Nil Bin

mapAB :: (a -> b) -> AB a -> AB b
mapAB f = foldAB Nil (\i r d -> Bin i (f r) d)

maximoAB :: AB a -> Maybe a
maximoAB = foldAB Nothing (\i r d -> maxMaybe i (maxMaybe r d))
  where maxMaybe Nothing Nothing = Nothing
        maxMaybe (Just x) Nothing = x
        maxMaybe Nothing (Just y) = y
        maxMaybe (Just x) (Just y) = max x y

altura :: AB a -> Int
altura = foldAB (const 0) (\i _ d -> 1 + max i d)

-- Se puede definir ordenado :: AB a -> Bool retornado
-- en cada paso recursivo un booleano ordenado, el valor
-- minimo y el valor maximo. Luego en ordenado retorno la
-- primera componente de la tripla.

-- Se puede definir directamente caminoMasLargo :: AB a -> [a]
-- con una idea similar retornando tanto la rama mas larga
-- como el camino mas largo, y eligiendo la rama como la mejor
-- de las dos y el camino como el mejor de los dos caminos y unir las ramas.
```

2.3. Razonamiento Ecuacional

2.4. Inducción Estructural