Apunte Final PLP

Mateo Ziffer

3 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
```

2.1.8. Más funciones y propiedades

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

Ejemplo de secuencia de evaluación

```haskell
[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
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

```
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 _ _ _ = []
```

```haskell
-- 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

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

2.2. Esquemas de Recursion

2.3. Razonamiento Ecuacional

2.4. Inducción Estructural