# Paradigma Funcional

Un programa según el Paradigma Funcional será una función, una transformación de un input en un output. El output dependerá del input que se ingrese.

## Transparencia referencial

Una solución tiene transparencia referencial si podemos reemplazar en un programa todas las expresiones E por el valor V sin alterar el resultado del programa, independientemente del lugar donde aparezca E (no depende del contexto de ejecución, ni del orden de evaluación de dicha expresión).

¿Qué ventajas tiene?

* Es mucho más sencillo demostrar la corrección de un programa (simplifica el testing).
* Podemos calcular el resultado de la expresión la primera vez y luego almacenarlo para mejorar la performance.
* No nos importa mucho en qué momento resolvemos la expresión, podría ser al comienzo del programa o después.

## Variable

Mientras que una variable en el paradigma imperativo representa una posición de memoria donde almaceno valores, y me sirve para recordar estados intermedios, una variable en el paradigma funcional respeta la definición matemática de variable, como una incógnita, un valor desconocido o que todavía no fue calculado.

## Definiciones con guardas

En matemática podemos definir una función por partes (o definición “por trozos”). Para ciertos valores del dominio mi conjunto imagen era uno y para otros valores mi conjunto imagen era otro. Un ejemplo es la función max:

max x y | x > y = x

| otherwise = y

Algunos comentarios:

* como otherwise siempre se cumple, se debe escribir como última condición
* es importante dejar al menos un espacio en blanco para la segunda línea, ya que si se escribe así

# Composición

Se puede representar como que es una nueva función que resulta de aplicar f primero, y luego g, en ese orden. Primero se aplica f y luego g, pero a la hora de escribirlo en Haskell, se hace al revés: g . f

# Aplicación parcial

Todas las funciones devuelven otra función, cuando son evaluadas con una menor cantidad de argumentos de los que tiene definidos. Los tipos de datos en el prototipo de una función están implícitamente asociados del final hacia el principio.

La utilidad principal de este proceso de evaluación parcial de funciones es permite crear nuevas funciones a partir de las funciones existentes.

Existe un caso especial importante. Una expresión de la forma: (-e) es interpretada como negate e, no como la aplicación parcial de “-“que resta el valor de e de su argumento. Su uso está relacionado con el concepto de currificación, que plantea la utilización de argumentos individuales en vez de estructuras de tipo tuplas en el prototipo de la función. De esta manera, es condición necesaria para que se pueda dar la aplicación parcial. Solo puede evaluarse parcialmente una función de varios parámetros si está currificada.

# Listas

Una lista es una serie de elementos del mismo tipo. En esa serie puede no haber elementos, en ese caso la lista es vacía. Son útiles para modelar las materias aprobadas por un alumno, las mascotas que tiene una persona, los empleados de una compañía, etc.

## Formas de generar una lista

La forma más sencilla de generar una lista es enumerar los elementos encerrados entre corchetes y separar cada elemento mediante una coma [1, 2, 3] ó ["Hola", "mundo"].

En una lista de Haskell todos los elementos deben ser homogéneos, de lo contrario se generará un mensaje de error:

En el caso de las listas numéricas, se puede generar una lista de enteros especificando cota inferior y superior:

λ [1...10] = [1,2,3,4,5,6,7,8,9,10]

## Pattern matching sobre listas

|  |  |
| --- | --- |
| Pattern | Denota |
| [] | Lista vacía, no se puede separar en cabeza y cola |
| (x:xs) | El operador: separa cabeza y cola de una lista. Lista que tiene al menos un elemento, donde la cabeza es un elemento x, y la cola es una lista (son muchos x, por eso la convención es xs). |
| (x:y:ys) | Lista de al menos 2 elementos. |
| [x] | Lista de exactamente un elemento. |
| [x, y] | Lista de exactamente dos elementos. |

## Funciones que aplican sobre listas

* head: cabeza de una lista. La función head/1 devuelve la cabeza de una lista.
* tail: cola de una lista. De la misma manera tenemos la función que devuelve la cola de una lista.
* length: Devuelve la longitud de una lista.
* sum: Suma los elementos de una lista de números.
* (++): Concatena dos listas.
* Take: Toma los primeros n elementos de la lista.
* Drop: Devuelve la lista sin los primeros n elementos.
* (!!): Devuelve el elemento que está en la posición n (donde 0 es el primer elemento).
* Reverse: Devuelve una lista con los elementos en orden inverso.

# Tuplas

Las tuplas permiten representar un tipo de dato compuesto, pero con elementos que pueden ser de distinto tipo. El número de elementos es fijo (siempre el mismo).

## Comparación entre tuplas y listas

* Las listas requieren que todos los elementos sean homogéneos: no podemos mezclar en una misma lista números y strings. Las tuplas pueden ser heterogéneas.
* El número de elementos de una lista es variable, puede ser infinito. En una tupla el número de elementos es fijo.
* La lista es un tipo de dato recursivo, la tupla no, aunque ambos son compuestos.

## Funciones que aplican sobre tuplas

Hay funciones estándar para separar una tupla de dos elementos: fst devuelve el primer elemento de la tupla, snd devuelve el segundo elemento.

# Tipos propios

Se pueden definir nuevos tipos de datos que se agregan a los ya existentes en el lenguaje, mediante la expresión data". Si queremos modelar varias personas que tienen nombre y edad, lo hacemos de la siguiente manera:

data Persona = Persona String Int

Esto se lee como: “el tipo de dato Persona utiliza un constructor –de nombre Persona- que recibe un String y un Int”.

## Diferencias entre data y tupla

La tupla ofrece una solución general que sirve para modelar personas, empleados y muchas otras cosas. Justamente por eso no es intuitiva, la estructura tiene que ser conocida por quien la usa: tengo una tupla con dos elementos String e Int, no es fácil darse cuenta de que se trata de una estructura que modela una persona, donde el primer elemento se asocia al nombre y el segundo elemento a la edad.

## Record syntax

Para sumar expresividad y evitar definiciones que podríamos considerar “redundantes”, tenemos la notación record syntax que consiste en definir Persona de la siguiente manera:

data Persona = Persona {

nombre :: String,

edad :: Int,

domicilio :: String,

telefono :: String,

fechaNacimiento :: (Int, Int, Int),

buenaPersona :: Bool,

plata :: Float

}

## Múltiples constructores

Es posible definir un tipo de dato propio que admita más de un constructor. Por ejemplo, podemos definir:

data Sexo = Femenino | Masculino

# Recursividad

Una función puede llamarse a sí misma, y ese mecanismo recursivo es el que implementa una estructura de repetición en el paradigma funcional.

El concepto de recursividad viene atado al de inducción: verifico P(0), y defino P(N + 1) en base a P(N). Volviendo al ejemplo del factorial, primero definimos el caso base, que es el que corta la recursividad: para 0, el factorial es 1:

factorial 0 = 1

factorial n = n \* factorial (n - 1)

Entonces recordemos: todo algoritmo recursivo debe tener

* un caso base para cortar la recursividad.
* un caso recursivo para que verdaderamente exista recursividad.

## Recursividad con listas

Repasemos la definición de una lista, que tiene una estructura recursiva:

* el caso base es la lista vacía.
* el caso recursivo es una lista de 1 ó más elementos, cuya cabeza es el primer elemento y cuya cola es una lista con los restantes.

### Funciones con recursividad con listas

|  |  |
| --- | --- |
| length [] = 0  length (x:xs) = 1 + length xs | drop 0 xs = xs  drop n (x:xs) = drop (n - 1) xs |
| sum [] = 0  sum (x:xs) = x + sum xs | (!!) (x:\_) 0 = x  (!!) (\_:xs) n = xs !! (n - 1) |
| last [x] = x  last (x:xs) = last xs | take n \_ | n <= 0 = []  take \_ [] = []  take n (x:xs) = x : take (n - 1) xs |
| elem \_ [] = False  elem e (x:xs) = e == x || elem e xs |
| concat [] x2 = x2  concat (x:xs) x2 = x:concat xs x2 | maximum [x] = x  maximum (x:y:ys)  | x > y = maximum (x:ys)  | otherwise = maximum (y:ys) |
| reverse [] = []  reverse (x:xs) = (reverse xs) ++ [x] |

# Evaluación diferida

Como estamos acostumbrados a trabajar en lenguajes con evaluación ansiosa, nos resulta extraño pensar que esta expresión λ (sum . take 10 . muchosDe) 5 pueda converger a un valor sin entrar en un loop infinito. Sin embargo, Haskell y muchos otros lenguajes del paradigma funcional trabajan con el concepto de evaluación diferida o perezosa (lazy evaluation), de manera de evaluar los argumentos a medida que los va necesitando.

Esto es equivalente a pedirle a un ferretero 10 cm. de cable, y que el vendedor desenrolle todo el cable de 100 metros para luego cortar los 10 cms que pedimos. El ferretero trabaja con evaluación ansiosa, nosotros preferimos hacerlo con evaluación diferida.

## Ventajas de la evaluación diferida

En los lenguajes imperativos estamos acostumbrados a la que las cosas funcionen en forma ansiosa, pero en funcional se nos abre una nueva puerta.

* Con la evaluación diferida solo se evalúa aquello que realmente se necesita. Como corolario, puedo trabajar con estructuras potencialmente infinitas (como las listas), mientras asegure que el algoritmo converge
* Si una expresión puede andar, con evaluación diferida seguro que anda. Si se rompe, es porque necesité algo que se rompía.
* ¿Por qué no tengo esto en C? Por el efecto colateral. Yo solo puedo cambiar el orden esperado en la evaluación de mis expresiones si sé que este cambio no va a afectar al resto del mundo. Y funcional es un paradigma atemporal: se relaja la idea de secuencia en el paradigma ya que no hay un antes y un después en una función matemática.

# Orden Superior

Esto se relaciona con el concepto de cohesión de los componentes de un sistema, en el caso del paradigma funcional los componentes se implementan mediante funciones:

* una función es más cohesiva que otra si se enfoca en menos objetivos a la vez.
* al disminuir la cohesión, no solo tengo más responsabilidades para cubrir, sino que es más probable cometer errores: puedo equivocarme en el criterio para filtrar, o puedo equivocarme en el algoritmo que recorre la lista.

## Filter

filter f [] = []

filter f (x:xs) | f x = x : filter f xs

| otherwise = filter f xs

filter :: (a -> Bool) -> [a] -> [a]

## Composición y orden superior

(palindromo . nombre) vs. filter even [1..10]

En el primer caso se construye una función adhoc a partir de la composición de dos funciones existentes: esa función recibe un cliente y permite determinar si su nombre es un palíndromo. Ambas son funciones de primer orden, si coinciden el dominio e imagen de ambas funciones puedo componerlas en uno u otro sentido.

En el segundo caso tenemos una función como valor de primer orden. Pasamos even a filter, y luego filter usa esa función que le paso como parámetro. Filter es una función de orden superior (mientras que even es de orden simple). Lo potente es que filter recibe una función que ni siquiera conoce. Simplemente la usa (delega la responsabilidad a la otra función).

# Map: transformación

Transforma una lista en otra aplicando una función a todos sus elementos.

map f [] = []

map f (x:xs) = f x : map f xs

* Recibe una función que transforma un elemento en otro que puede ser del mismo tipo o no. Por eso utilizamos una letra nueva, b puede coincidir con a o ser un tipo distinto.
* la lista original de as
* la lista resultante es de bs

map :: (a -> b) -> [a] -> [b]

# Fold: reducción

Fold es una familia de funciones de orden superior, que tienen todas un objetivo similar: combinar los elementos de ciertas estructuras (como por ejemplo, las listas), usando una operación binaria (una función de dos parámetros).

Una primera aproximación gráfica e informal a los folds es la noción de “intercalar” sintácticamente una operación binaria entre los elementos de la lista.

### foldl vs. foldr

foldr f i [] = i

foldr f i (x:xs) = f x (foldr f i xs)

foldl f i [] = i

foldl f i (x:xs) = foldl f (f i x) xs

Una característica interesante de la definición que hace foldl es que es recursiva a la cola: si podemos resolver el valor de la expresión (f i x) y no hay otras operaciones pendientes con posterioridad a la llamada recursiva, Haskell no necesita mantener el estado de la pila.

La ventaja típica de las funciones recursivas a la cola es que son fáciles de traducir a iterativas. En el caso particular de Haskell, sin embargo, esto no constituye una ventaja significativa, dado que el motor es lo suficientemente inteligente como para evaluar funciones no recursivas a la cola de forma eficiente. En otras tecnologías, tener un algoritmo con recursividad a la cola implica poder procesar una lista de valores grande sin que el tamaño de la pila se desborde (el famoso “Stack Overflow”).

La definición de foldr, entre tanto, no es recursiva a la cola, dado que la aplicación más exterior es la aplicación de f.

Necesitamos mantener el estado del juego de variables para que al calcular (foldr f i xs) podamos aplicarla con f x. Pero contamos con la ayuda de Haskell y su mecanismo de evaluación diferida, entonces solo va a reducir las expresiones que necesite.

Por otra parte, foldl no se lleva bien con listas demasiado grandes, potencialmente infinitas. Cuando empleemos una operación que podría terminar o arrojar resultados parciales antes de evaluar toda la lista, foldr es una mejor opción. Dicho de otra forma, si la operación es lazy a derecha, foldr puede terminar donde foldl no. Para ilustrar esto que hemos dicho veamos un ejemplo concreto: la función repeat permite construir una lista infinita de un elemento.

# Expresiones Lambda

Haskell está basado en cálculo lambda, que es un sistema de reglas de transformación o reductor de expresiones que diseñó Alonzo Church.

(\x -> x \* x)

La contrabarra (\) es el símbolo que remite a la letra griega lambda λ que como habrán notado es el ícono de la programación funcional. Luego de los parámetros que se separan por espacios, la flecha -> termina de definir el cuerpo de la función.

Las expresiones lambdas permiten definir funciones anónimas, que no tienen nombre y se usan en un contexto limitado (el de la misma función que estoy definiendo). ¿Qué desventaja tiene? El objetivo está dentro de la función y es menos expresiva que tener definida una función cuadrado o suma. La ventaja es que no tengo que definir funciones auxiliares cuando sólo la necesito usar una vez.

## Consecuencias de las lambdas

Algo importante a tener en cuenta es que si no le damos un nombre a nuestras funciones podríamos perder abstracciones útiles que podrían luego ser utilizadas en otros puntos de nuestro programa, por lo tanto es importante ser criteriosos respecto nombrar o no una función.

Por lo general, si tengo una forma sencilla de nombrar una determinada lógica que forma parte de una función más grande, lo más probable es que no quiera definir ese pedacito de lógica usando una lambda, sino con una función que se llame como la idea que tenemos en la cabeza. Si no hay un nombre claro asociado a ese pedacito de lógica, lo más probable es que no sea un concepto del dominio que merezca la pena modelar como algo aparte.

# Sistema de Tipos

Hasta el momento hemos visto los tipos de una función sabiendo que existen

* tipos simples: Int, Bool, Float, Char, las funciones cuyos parámetros se delimitan con flechas.
* tipos compuestos: String, listas, tuplas, los tipos de dato definidos por el usuario (data)

También contamos con los sinónimos de tipo type Nombre = String que permiten establecer equivalencias con nombres más representativos del dominio.

## La función id

Quizás ya hayan conocido la función identidad en Haskell, llamada id: id x = x

¿De qué tipo es la función id?

* es válido para booleanos
* pero también para strings
* incluso funciones

## Polimorfismo paramétrico

Las funciones que aceptan valores de cualquier tipo, sin ningún tipo de restricciones, tienen polimorfismo paramétrico. En particular lo que nos interesa comentar es que solo necesitamos definir una vez la función para aplicarla con cualquier tipo.

## Num como typeclass

Un typeclass

* agrupa tipos
* define una interfaz, que son un conjunto de operaciones que todos los tipos pertenecientes deben implementar
* permite definir un comportamiento por defecto para ciertas operaciones

Los tipos Int, Float y Fractional son instancias del typeclass Num que los agrupa. Por ser Num estos tipos deben implementar las siguientes operaciones:

* (+), (-), (\*) :: (Num a) => a -> a -> a
* negate, abs, signum :: (Num a) => a -> a

## El typeclass Ord

Ord es un typeclass que define las siguientes operaciones

* (<), (>), (<=), (>=) :: (Ord a) => a -> a -> Bool
* max, min :: (Ord a) => a -> a -> a

Y aplica a los tipos

* Bool
* Char
* Double
* Int
* Float

## El typeclass Eq

Eq es un typeclass que define las siguientes operaciones

* (==) :: (Eq a) => a - > a -> Bool
* (/=) :: (Eq a) => a - > a -> Bool

Y aplica a los tipos

* Bool
* Char
* Double
* Int
* Float

## Polimorfismo ad-hoc

Todas estas funciones

* (==) :: (Eq a) => a -> a -> Bool
* (>) :: (Ord a) => a -> a -> Bool
* (+) :: (Num a) => a -> a -> a

admiten una gran variedad de tipos, pero no todos: tienen una restricción demarcada por una typeclass. Por este motivo decimos que tienen polimorfismo ad-hoc.