Paradigma Funcional

Módulo 2:
Composición.
Aplicación Parcial.

por Fernando Dodino
Carlos Lombardi
Nicolás Passerini
Daniel Solmirano

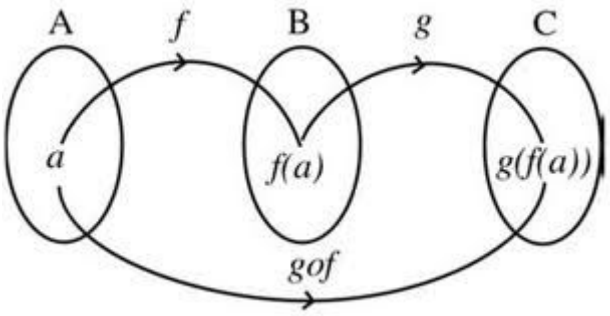
Versión 2.0
Febrero 2017

Contenido

1 Composición

1.1 Definición

Matemáticamente, podemos aplicar dos funciones,



$$g(f(x)) = (g \circ f)\ x$$

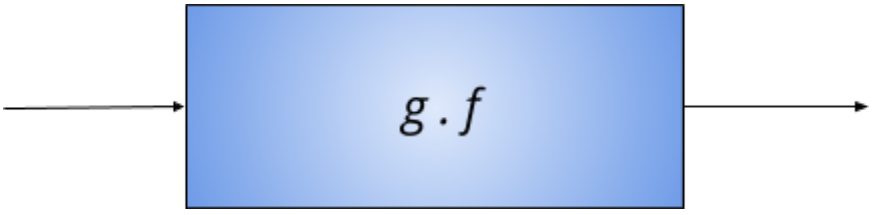
esto termina generando una **nueva función**, como se ve en la flecha que va de a a g(f(a)). Como restricción, la imagen de f tiene que coincidir con el dominio de g.

1.2 Implementación

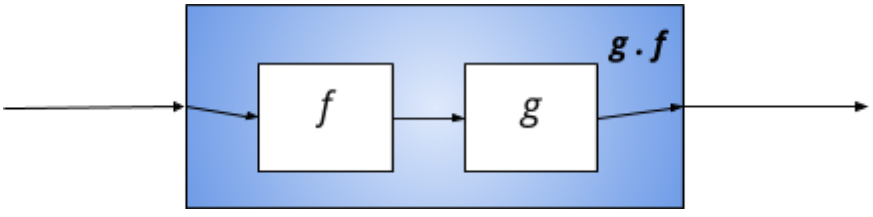
El mismo concepto se da en Haskell, donde la notación es

```
g (f x) = (g . f) x
```

Y que podemos representar como



que es una nueva función que resulta de aplicar f primero, y luego g, en ese orden:



Nótese que primero aplicamos f y luego g, pero a la hora de escribirlo en Haskell, lo hacemos al revés:

$g \cdot f$

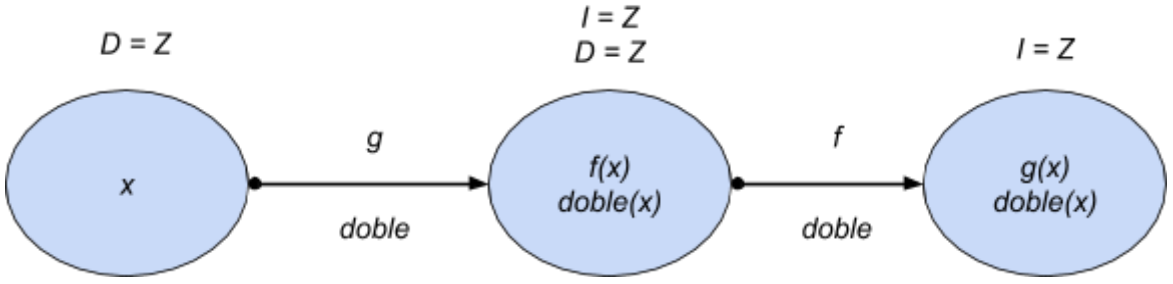
A partir de ahora solo utilizaremos el operador (.) porque queremos ser explícitos en que estamos generando nuevas funciones, no aplicando sucesivamente las funciones existentes.

1.3 Primer ejemplo: Cuádruple

Para obtener el cuádruple de un número, podemos valernos de la propiedad asociativa:

4 * x = 2 * (2 * x)

Es decir, estamos aplicando dos veces la función doble:



g ∘ f donde f y g corresponden a la función doble

En Haskell

```
cuadruple numero = (doble . doble) numero
```

Como el tipo de doble es

```
doble :: Numero -> Numero
```

se puede componer consigo mismo, dado que su imagen coincide con su dominio.

Incluso, matemáticamente, podemos simplificar esta igualdad[1]:

```
cuadruple numero = (doble . doble) numero
cuadruple = (doble . doble)
cuadruple = doble . doble
```

[1] La simplificación de términos recibe el nombre de "conversión eta" (η-conversion). Para más información pueden verse los artículos http://www.haskell.org/haskellwiki/Eta_conversion y http://www.haskell.org/haskellwiki/Pointfree. La idea de esta reducción de términos es mejorar la legibilidad de los programas eliminando paréntesis y variables redundantes.

Esta simplificación se puede dar porque la operación doble . doble está encerrada entre paréntesis.

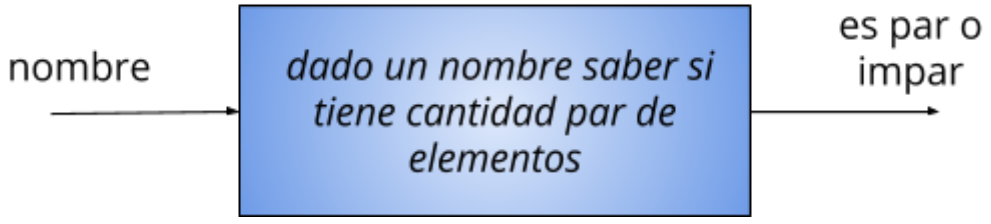
También podríamos haber pedido en la consola que calcule el cuádruple de un número:

```
> (doble . doble) 12
```

Y aquí vemos la primera ventaja de la composición: construyo funciones sin tener que escribirlas en un archivo de definiciones *.hs*.

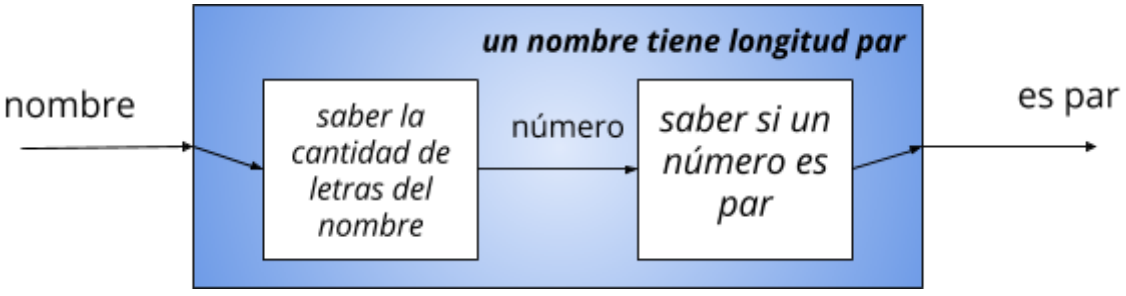
2 Segundo ejemplo: longitud de un nombre par

El requerimiento dice: "Saber si la longitud del nombre de una persona es una cantidad par". ¿De qué tipo es esta función?



- El nombre lo modelamos como un String.
- Si es par o impar, con un boolean.
- Y el requerimiento se modela con una función.

Pero vemos que la función puede dividirse en dos partes:



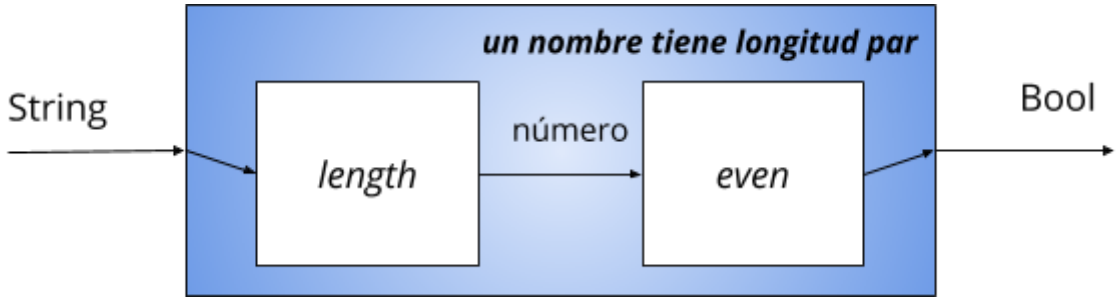
Las alternativas son

- construir nosotros las funciones que necesitamos
- o ver si no vienen ya con Haskell

Tenemos suerte:

- para saber la cantidad de letras de un nombre tenemos la función length, que en este caso va de String a Int
- y para saber si un número es par, existe la función even, que va de Int a Bool.

de esa composición, tendremos una función que dado un String devuelve un Bool, lo que estamos necesitando:

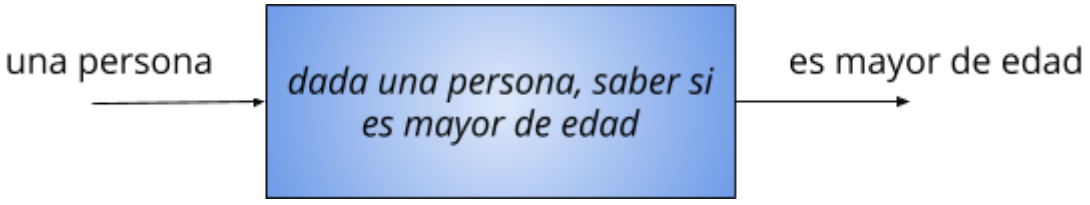


Pero recordemos que en Haskell, al igual que en la notación matemática, debemos invertir el orden en el que escribimos las funciones compuestas:

```
nombrePar :: String -> Bool
nombrePar nombre = (even . length) nombre
nombrePar ~~nombre~~ = (even . length) ~~nombre~~
nombrePar = even . length
```

3 Cómo saber si una persona es mayor de edad

Dado el siguiente requerimiento: “saber si una persona es mayor de edad”, vemos que es algo que se puede modelar con una función:



¿Cómo modelo a la persona? Hasta el momento conocemos los tipos String, caracter, número, booleano y función. Una restricción adicional parece complicar las cosas: “Conocemos el nombre y la edad de una persona”.

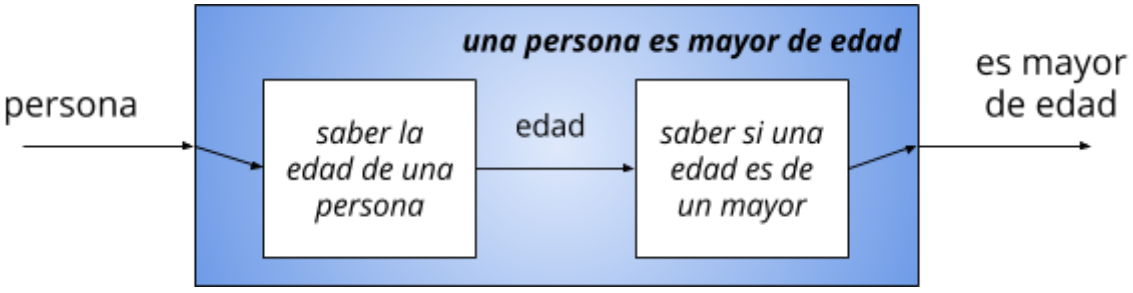
- el nombre es un String,
- y la edad un número entero

para que puedan trabajar juntos vamos a adelantar un concepto del próximo módulo y presentaremos la tupla, un tipo de dato compuesto que agrupa

información relacionada y que nos viene muy bien para poder modelar una persona.

Encontré entonces una abstracción para representar esa **entidad**. ¿Me interesa todo sobre la persona? No, me quedo con lo que es esencial para mí y dejo un par ordenado o *tupla* con el nombre y la edad: (String, Int).

Ahora sí, podemos partir el requerimiento en dos:



Y lo que nos falta es ver si tenemos funciones que puedan servirnos para cada caso:

- en el primer caso, hay una función *snd*, que dada una tupla de dos elementos devuelve el segundo. Pero *snd* no es representativa del dominio que estamos modelando, entonces vamos a construir una nueva función que mejore la expresividad de nuestra solución:

```
edad = snd
```

¿Qué acabamos de hacer? Simplemente escribiendo un sinónimo para *snd*, sin volver a definir la función. Si el operador = representa la igualdad matemática, solamente estamos diciendo a Haskell que cuando necesite reducir

`edad ("laura", 41)`, esta expresión será equivalente a hacer
`snd ("laura", 41)`

¿De qué tipo es edad? Del mismo que snd (más adelante lo veremos). Pero en particular

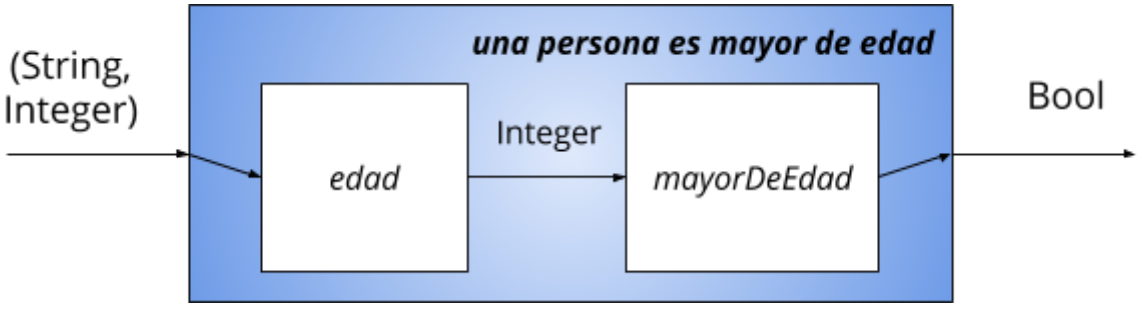
```
> edad ("laura", 41)
41
> :t edad ("laura", 41)
edad ("laura", 41) :: Num b => b
```

Más allá de algún detalle técnico que todavía nos falta explicar, lo que devuelve `edad ("laura", 41)` es un número.

- en el segundo caso, no hay ninguna función que dado un número me diga si es considerable como mayoría de edad. La construimos:

```
mayorDeEdad :: Integer -> Bool
mayorDeEdad edad = edad > 18
```

Ahora sí, podemos componer edad y mayorDeEdad:




```
esMayorEdad :: (String, Integer) -> Bool
esMayorEdad = mayorDeEdad . edad
```

Podemos definir que una Persona es un (String, Int), esto facilita la lectura del tipo de la función que acabamos de construir:

```
type Persona = (String, Integer)
esMayorEdad :: Persona -> Bool
```

Si definimos una expresión para Laura...
`laura = ("Laura", 41)`
...podemos usarla con esMayorEdad:

```
> esMayorEdad laura
True
```

3.1 Acoplamiento de funciones

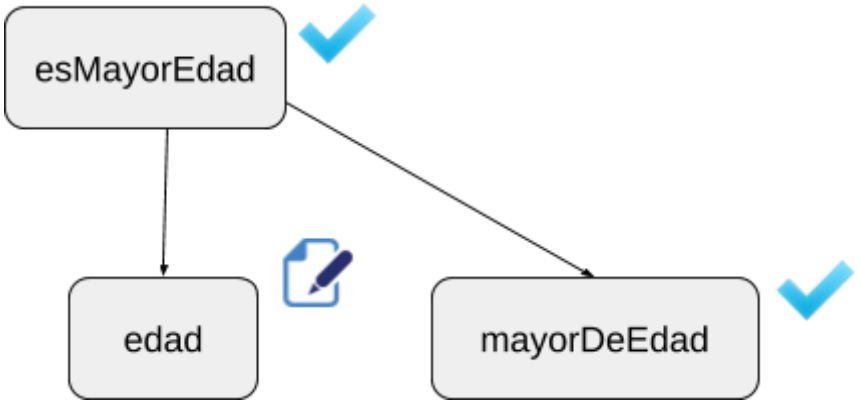
Aparece un cambio en la forma de modelar una persona, nos piden que además del nombre y la edad, también deberíamos conocer el domicilio. Sabemos que se modela con un String, ¿qué debemos cambiar en nuestra solución?

```
type Persona = (String, Integer, String)

laura :: Persona
laura = ("Laura", 41, "Medrano 951 CABA")

edad :: Persona -> Integer
edad (_, e, _) = e
```

- La definición de Persona debe aceptar un String adicional para el domicilio
- También la definición de la expresión laura, que agrega el domicilio
- y la función edad utiliza **pattern matching**[2], para devolver la edad en la tupla de 3 elementos que conforma la persona

Por otra parte, veamos que estas definiciones **no cambiaron**:

```
mayorDeEdad :: Integer -> Bool
mayorDeEdad edad = edad > 18

esMayorEdad :: Persona -> Bool
esMayorEdad = mayorDeEdad . edad
```



Y vemos que pese a que la forma de representar una persona cambió, la definición para la función esMayorEdad no se ve impactada.

Acabamos de ver un ejemplo concreto de **acoplamiento**, el grado en que dos componentes se conocen.

- ¿Existe acoplamiento entre esMayorEdad y edad? Sí, ya que de lo contrario no podríamos resolver el requerimiento: “Saber si una persona es mayor de edad”

[2] En el próximo capítulo explicaremos en detalle este tema

- Pero ese acoplamiento es bajo: un cambio interno en la codificación de la función edad no afecta a la función esMayorEdad.
- De hecho, la función esMayorEdad solo necesita saber que puede resolver el requerimiento **componiendo dos funciones cuya implementación no necesita conocer**
- En general, mientras edad siga devolviendo un valor entero, podremos seguir componiendo edad con mayorDeEdad sin inconvenientes. Si cambiamos la interfaz de la función (lo que recibe o lo que devuelve) será más difícil para nosotros no hacer cambios en los demás componentes. Si la edad cambia de valor entero a decimales, ese será un cambio con mayor impacto que incorporar más información a una persona.
- Un detalle, la prueba por consola sigue siendo exactamente igual:

```
> esMayorEdad laura
True
```

3.2 Composición de más de dos funciones

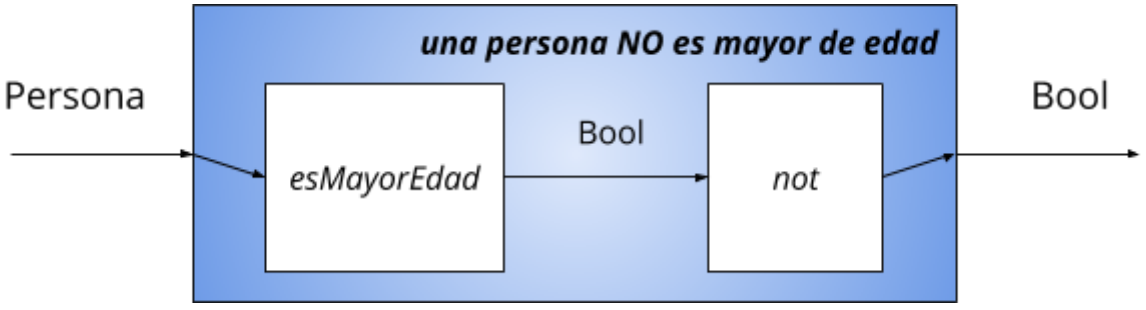
Para resolver este requerimiento: “Saber si una persona no es mayor de edad”, partiendo de lo que ya sabemos, podemos hacer una consulta directamente por consola:

```
> (not . esMayorEdad) laura
False
```

not es una función que niega el valor de verdad de un booleano.

```
> :t not
not :: Bool -> Bool
```

Como recibe un Bool, se puede componer con cualquier función que devuelva un Bool:

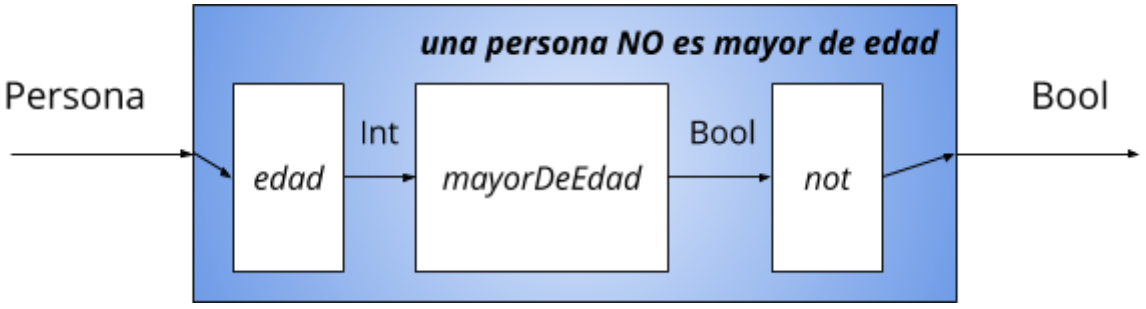


También podríamos haber compuesto sucesivamente varias veces. Por ejemplo, cuando una persona no es mayor de edad, entonces decimos que es menor de edad. Por lo tanto, podríamos definir:

```
esMenorEdad persona = (not . mayorDeEdad . edad) persona[3]
esMenorEdad ~~persona~~ = ~~(~~not . mayorDeEdad . edad~~) persona~~
esMenorEdad = not . mayorDeEdad . edad
```

```
> esMenorEdad laura
False
```

La única restricción es que las imágenes de cada función coincidan con el dominio de la función siguiente:



4 Aplicación parcial

4.1 Introducción

La función mod permite conocer el resto de la división de dos números

```
> mod 6 4
2
> mod 10 5
0
```

¿Qué pasa si “nos olvidamos” de pasarle un parámetro?

```
> mod 6
<interactive>:27:1:
    No instance for (Show (a0 -> a0))
      (maybe you haven't applied enough arguments to a function?)
      arising from a use of ‘print’
```

[3] Obviamente, podríamos habernos basado en esMayorEdad, pero para lo que queremos mostrar preferimos hacerlo de esta forma.

```
    In the first argument of ‘print’, namely ‘it’
    In a stmt of an interactive GHCi command: print it
```

Ah, Haskell tira error[4]. Pero si editamos un archivo de definiciones hs, y escribimos en la primera línea:

```
import Text.Show.Functions
```

Cerramos el editor y repetimos la pregunta:

```
> :e
[1 of 1] Compiling Main             ( mayorEdad.hs, interpreted )
Ok, modules loaded: Main.
> mod 6
<function>
```

Ah, ahora no aparece el error, sino que explícitamente dice que mod 6 es una función. Exacto, y esa función (delimitada por mod 6)

- recibe un número
- y devuelve el resto de dividir 6 por ese número

¿De qué tipo es esa nueva función?

```
> :t mod 6
mod 6 :: Numero -> Numero
```

Recibe un número, devuelve un número[5].

Lo interesante es que no escribimos esa nueva función, sino que es la consecuencia de no pasar todos los parámetros a mod.

En general

```
mod :: Numero -> Numero -> Numero
      // es una función que devuelve el resto de la
      // división de dos números cualquiera => no hay aplicación

mod 6 :: Numero -> Numero
      // es una función que dado un número devuelve el resto de
      // la división de 6 por ese número => hay aplicación parcial

max 6 4 :: Numero
      // es un número, que resulta de aplicar la función
      // => la función se aplica
```

[4] En los cursos donde utilicen *stack* y la biblioteca especial para PDP no es necesario hacer esto, devuelve **<una función>** cuando evaluamos por consola cualquier función.
[5] La consola muestra información diferente que contaremos más adelante

4.2 Segundo ejemplo: siguiente

Necesitamos resolver una función que dado un número nos devuelva el siguiente. Podríamos escribirla así:

```
siguiente :: Integer -> Integer
siguiente n = n + 1
```

Pero también podemos aprovechar la suma, que está definida como:
`(+) :: Numero -> Numero -> Numero`

Entonces la expresión (1 +) define una nueva función:
`(1 +) :: Numero -> Numero`

Y lo que hace esa función es

- recibir un número
- y devolver el número consecutivo inmediato

Podemos definir siguiente como:
`siguiente = (1 +)`
de la misma manera que podríamos definir doble como:
`doble = (2 *)`

Y las siguientes funciones:
```
cuadrado = (^ 2)
cubo = (^ 3)
```

Nota: Solo para el caso de los operadores como pueden ser (+), (*), (^), también vale pasar el segundo argumento como se muestra en estos ejemplos. Tener en cuenta que en los ejemplos anteriores ((+) y (*)) los parámetros de los operadores son conmutables (a + b = b + a) y por lo tanto (2+) y (+2) es lo mismo. La potenciación (^) no lo es, y por lo tanto es diferente (2^) que (^2).

5 Composición + Aplicación parcial

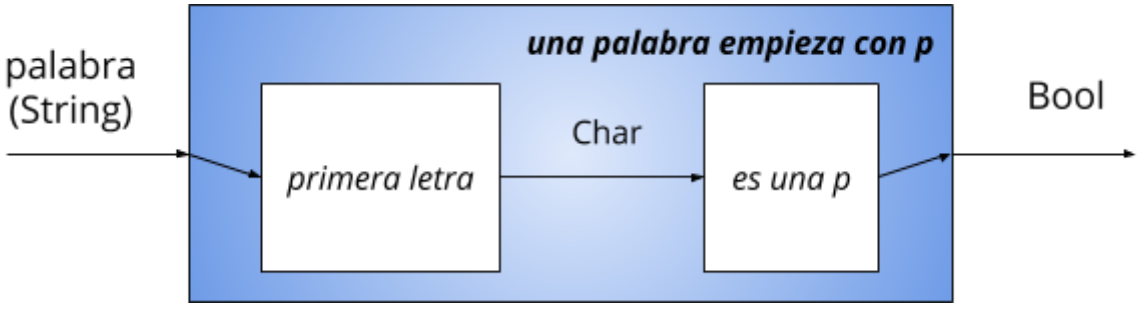
Para entender la utilidad del concepto aplicación parcial, vamos a utilizarlo en conjunto con la composición, para ver que es útil para evitar definiciones específicas y reutilizar funciones existentes:

5.1 Comienza con p

“Queremos saber si una palabra comienza con p”

La función recibe

- una palabra (un String)
- y devuelve un Bool



- Para obtener la primera letra, tenemos una función **head** que toma el primer elemento de una lista
- Ahora bien, para saber si es una 'p' nos convendría tener una función a la que le pasemos un Char y nos diga si ese Char es p. Por suerte existe esa función, si aprovechamos el operador (==) y lo aplicamos parcialmente con el carácter 'p':

```
esP :: Char -> Bool
esP = ('p' ==)
```

Esa función nos dice, dado un carácter, si ese carácter es una p.

Como dijimos antes, no necesitamos escribir la función esP, la construimos directamente:

```
> (('p' ==) . head) "palabras"
True
```

La nueva función toma una lista de caracteres (o un String) y devuelve un Bool:

```
> :t (('p' ==) . head)
(('p' ==) . head) :: [Char] -> Bool
```

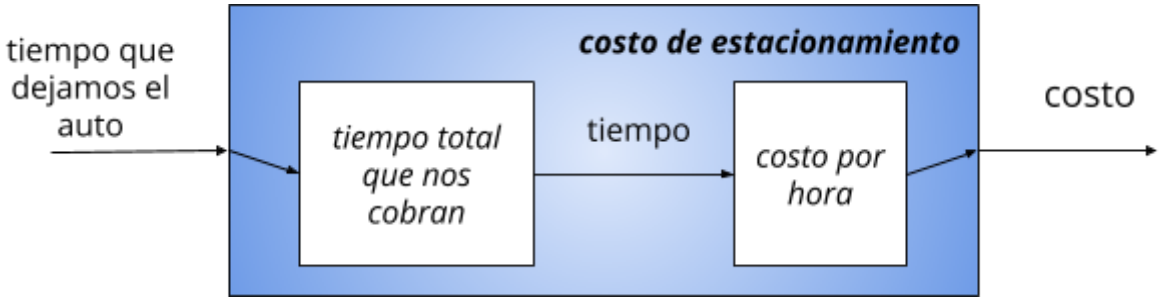
5.1 Costo del estacionamiento

"El costo de estacionamiento es de 50 pesos la hora, con un mínimo de 2 horas", esto significa que

- si estamos 1 hora, nos cobrarán por 2 horas
- si estamos 3 horas, nos cobrarán por 3 horas

Independientemente de eso siempre nos cobran 50 pesos la hora.

Podemos pensar nuestra solución en términos del negocio:



Ahora lo traducimos a funciones de Haskell, ¿qué funciones necesitamos?

- la primera que calcula el tiempo total que nos cobran: deberíamos considerar el máximo entre 2 y el tiempo que dejamos el auto. Ya tenemos nuestra primera función: max 2 (es decir, max aplicada parcialmente)
- por otra parte, si el costo es siempre 50 pesos la hora, podemos aplicar parcialmente el operador de multiplicación y tenemos nuestra segunda función: (* 50).

```haskell
costoEstacionamiento :: Integer -> Integer
costoEstacionamiento horas = (* 50) . max 2 horas
costoEstacionamiento = (* 50) . max 2
```

Lo podemos probar:

```
> costoEstacionamiento 1
100
> costoEstacionamiento 5
250
```

Y efectivamente los paréntesis marcan el límite entre el valor de entrada y la sucesiva aplicación de funciones: esta definición

```haskell
costoEstacionamiento horas = ((* 50) . max 2 horas)
```

es incorrecta, no se puede componer `max 2 horas` con `(* 50)` ya que `max 2 horas` se reduce a un valor entero, que **no es una función**.

6 Resumen

Anteriormente hemos visto que el paradigma funcional cuenta con las funciones como abstracción fundamental, representa un valor tan natural como un entero, un booleano o un string. En este capítulo sumamos la composición y la aplicación parcial como herramientas para poder reutilizar una función en diferentes contextos para resolver requerimientos.

En la composición, construimos funciones nuevas a partir de la aplicación sucesiva de funciones existentes, mientras que en la aplicación parcial generamos una nueva función con la técnica de no pasar todos los parámetros que la definición propone.