Paradigma Funcional

Módulo 1:
Introducción al paradigma.
Función. Variable.
Primeros ejemplos.
Aplicación. Guardas.

por Fernando Dodino
Carlos Lombardi
Nicolás Passerini
Daniel Solmirano

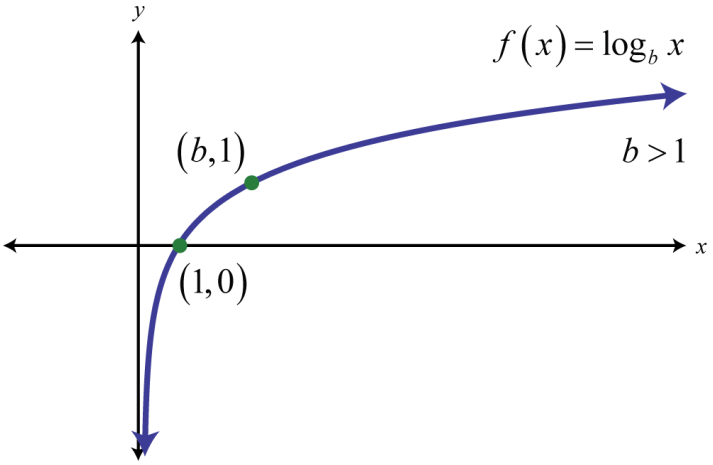
Versión 2.1
Marzo 2018

Contenido

1 Introducción al paradigma

1.1 Función

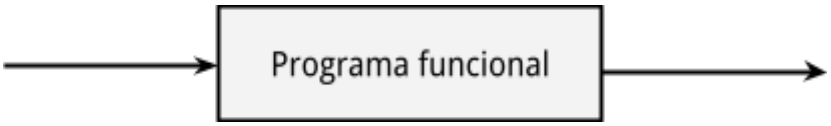
En el sentido matemático, una función implica que un hecho depende de otro. Por ejemplo…



…implica que el valor de ***y*** depende del logaritmo ***x*** (está “en función de”).

2 El Paradigma Funcional

Entonces, 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.
La metáfora asociada al paradigma funcional es la **calculadora:**

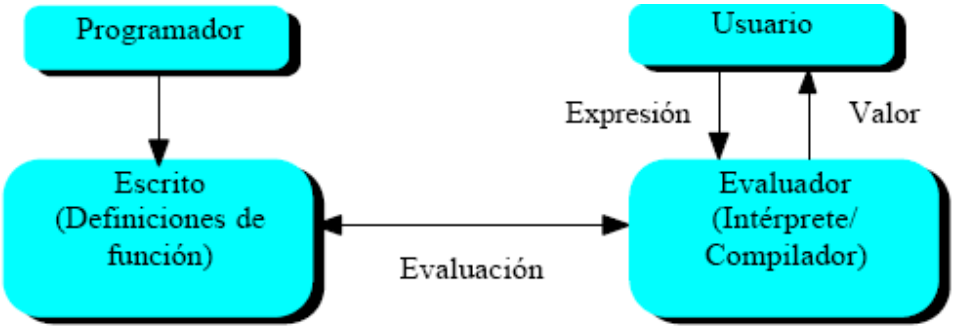
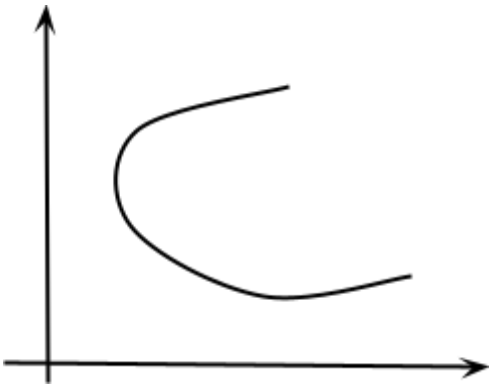


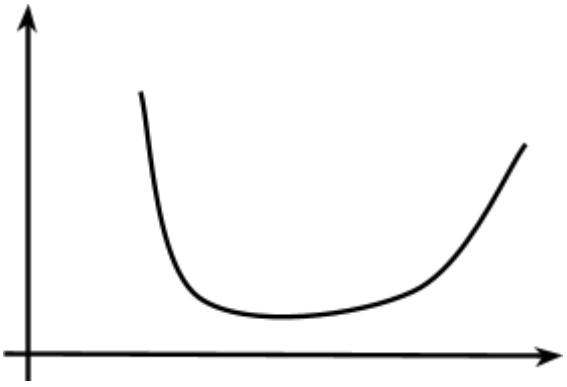
Fig.1 - Aplicación en el Paradigma Funcional
Fuente: *Introducción al lenguaje Haske*ll, José A.Labra, Univ.Oviedo, 1998

2.1 Repaso de funciones matemáticas

Esto, ¿es una función?



No, porque para cada ***x*** sólo puede haber un ***y*** posible. Matemáticamente: para cada elemento del dominio debe haber una sola imagen (concepto que se conoce como **unicidad**). El siguiente gráfico sí respeta esta restricción y representa una función:



2.2 Unicidad de funciones en programación

Veamos un ejemplo en pseudocódigo[1]:

```
program Prueba
var
   flag: BOOLEAN

function f (n: INTEGER): INTEGER
begin
   flag ← NOT flag
   if flag then
      ↑ n
   else
      ↑ 2 * n
   end if
end
```

Cuando evaluamos el programa principal nos llevamos algunas sorpresas...

```
--- Programa Principal
begin
   flag ← true;
   ...
   write( f(1) );     ← escribe 2
   write( f(1) );     ← escribe 1
   ...
   write( f(1) + f(2) );  ← escribe 4
   write( f(2) + f(1) );  ← escribe 5
   ...
end
```

f no es matemáticamente una función, para un mismo input tengo dos resultados posibles. Además, no cumple la propiedad reflexiva (`f(1) ≠ f(1)`) ni la conmutativa (`f(1) + f(2) ≠ f(2) + f(1)`).

2.3 Transparencia referencial

f(1) es una expresión *E* cuyo resultado da un valor *V*.

[1] ↑ significa return y ←asignación

Decimos que 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).

O sea, si en cada lugar donde apareciera f(1) pudiéramos encontrar un único valor V que corresponde al resultado de evaluar f(1), entonces la solución tendría transparencia referencial (que justamente no es el caso, V puede ser 1 ó 2).

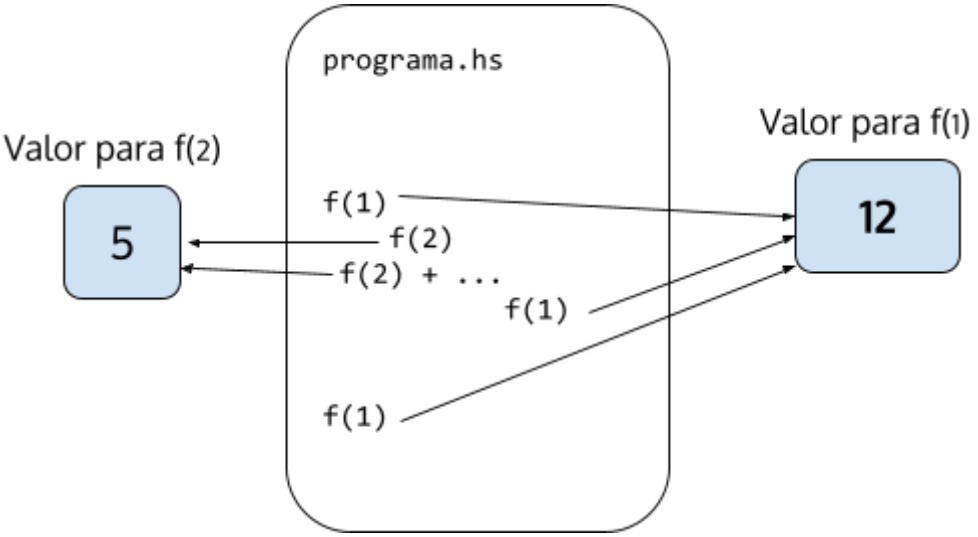


Fig.2 - Ejemplo de un programa funcional con transparencia referencial

¿Qué ventajas tiene?

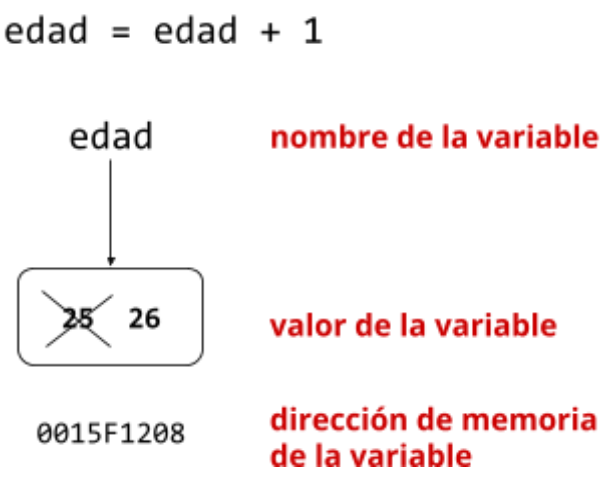
1. Es mucho más sencillo demostrar la corrección de un programa (simplifica el testing).
2. Podemos calcular el resultado de la expresión la primera vez y luego almacenarlo para mejorar la performance.
3. No nos importa mucho en qué momento resolvemos la expresión, podría ser al comienzo del programa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

La variable *flag* causa que la solución no tenga transparencia referencial: el valor de la función depende de un contexto que es “recordado”.

2.4 Variable

Mientras que una variable en el paradigma imperativo representa una posición de memoria donde almaceno valores, y me sirve para recordar estados intermedios[2]...

[2] Esta idea fue diseñada por el matemático húngaro Lajos Neuman (John Von Neumann)



...una variable en el paradigma funcional respeta la [definición matemática de variable], como una incógnita, un valor desconocido o que todavía no fue calculado.

$f(x) = y = \log x$

tanto ***y*** como ***x*** son variables matemáticas:

- y es variable dependiente
- x es variable independiente

Por eso se suele definir una función acompañando la variable independiente: Velocidad = V(t).

No tiene sentido la idea x = x + 1. Si tengo x = 2 , x = 3 corresponden a dos puntos diferentes del dominio que tendrán sus correspondientes valores para la función. No son posiciones de memoria, son incógnitas temporales que se resuelven cuando quiero conocer el valor de la función en un punto determinado.

Cuando se quiere saber el valor de la variable dependiente que se corresponde con cierto valor de la variable independiente, se dice que se evalúa o aplica la función. En el ejemplo anterior de la función logaritmo, al evaluar f(1), se asume que x es 1 y se evalua log 1, obteniendo 0 como resultado. En otras palabras, se puede decir que f(1) es equivalente a 0.

3 Conociendo Haskell

3.1 1, 2, 3... probando

Cuando ejecutamos por primera vez la consola GHCi de Haskell, desde allí podemos escribir expresiones, que serán evaluadas y reducidas:

```
> 2 + 3
5     -- ok, la suma funciona
> not True
False
> length "Hola"
4
```

3.2 Valores y tipos, primera aproximación

En la primera expresión a evaluar: `2 + 3`

- tenemos dos valores: 2 y 3, dos números
- y un operador, la suma que se denota + y donde se utiliza la notación *infija*[3], para mayor comodidad (el operador se ubica entre medio de los valores)
- el resultado es 5,
    - un número (Num)

Otros operadores: == (comparación por igualdad), - (resta), / (división), * (multiplicación), < (menor que), > (mayor que), ^ (elevado a), !! (desplazamiento), etc.[4]

En el segundo caso: `not True`

- *not* es una función, por lo tanto se utiliza la notación *prefija* por defecto: primero se escribe el nombre de la función y luego los parámetros
- *True* representa al valor booleano verdadero, con mayúscula
- el resultado de evaluar esta expresión es *False*,
    - otro booleano (Bool)

Por último, en `length "Hola"`

- *length* es una función, que puede calcular la longitud...
- en este caso de un String, o lista de caracteres

[3] Los operadores pueden utilizarse con notación prefija si se los encierra entre paréntesis:

```
> (+) 2 3
5
```

[4] Podés ver aquí una lista bastante completa de operadores

- el resultado de evaluar esta expresión es 4,
    - un número (Num)

3.3 Modelando valores

Si nos piden que modelemos

- la edad de una persona
- el nombre de una materia
- si un alumno aprobó un parcial

Podemos ver qué tipo es adecuado para cada caso:

- edad de una persona ⟹ es un número, sin decimales
- el nombre de una materia ⟹ lo podemos representar con un String
- si un alumno aprobó un parcial ⟹ en este caso particular tenemos opciones, podríamos decir que es un valor booleano, pero...

... ¿no podría ser una función? Algo como

3.4 Primera función: saber si un alumno aprobó

La función recibe una nota, que lo representamos como un número de 1 a 10 sin decimales, por lo tanto:

- recibimos un número entero sin decimales
- esperamos como retorno un valor booleano (aprobó / no aprobó).

Entonces definimos el tipo de la función:

```
aproboAlumno :: Int -> Bool
```

Y luego escribimos el cuerpo de la función, que como criterio pide que el alumno saque 6 o más para poder aprobar:

```
aproboAlumno nota = nota >= 6
```

nota es una variable, porque en el momento de definir la función no conozco su valor: es una incógnita.

Estas dos líneas se escriben en un editor de texto[5]:

```haskell
aproboAlumno :: Int -> Bool
aproboAlumno nota = nota >= 6
```
alumnos.hs

y se guarda el archivo con extensión *.hs*.

3.5 Aplicación

Para probar la función que acabamos de construir, levantamos el GHCi y desde la consola cargamos el archivo:

```
$ ghci (o stack ghci)
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :l alumnos.hs
[1 of 1] Compiling Main             ( alumnos.hs, interpreted )
Ok, modules loaded: Main.
```

Entonces podemos probar nuestra función, **aplicando la función aproboAlumno con diferentes parámetros**:

```
*Main> aproboAlumno 8
True
*Main> aproboAlumno 3
False
```

Al pasarle parámetros a una función, se evalúa la expresión hasta obtener un valor que es irreductible para Haskell.

3.6 Consulta de tipos

Podemos preguntar cuál es el tipo de cada una de las siguientes expresiones:

```
*Main> :t aproboAlumno
aproboAlumno :: Int -> Bool
*Main> :t 8
8 :: Num a => a
```

[5] En algunos cursos que utilizan el pdepreludat hay instrucciones ligeramente diferentes para crear un proyecto desde cero. Consultá con tu docente cuál de las dos variantes aplica a tu cursada. En ese caso el tipo es

```haskell
aproboAlumno :: Number -> Bool
```

```
*Main> :t True
True :: Bool
```
(es algo que viene bien saber en caso de duda)

En este caso, parece no tener mucho sentido, pero también podemos saber qué tipo toma una función cuando la aplico con algún parámetro concreto:

```
*Main> :t aproboAlumno 8
aproboAlumno 8 :: Bool
```

Más adelante volveremos sobre el tema.

3.7 Variables asociadas a valores

Queremos modelar la edad del señor Burns: 120 años definitivamente entra dentro de un Int. Podemos encontrar una abstracción para eso:

```
edadSeniorBurns = 120

*Main> :t edadSeniorBurns
edadSeniorBurns :: Integer
```

edadSeniorBurns es una variable que se asocia al valor 120. **No es una función, ya que al no tener parámetros, no se puede aplicar.**

Definir una variable hace que una expresión sea más fácil de entender:

```
*Main> puedeConducir edadSeniorBurns
```

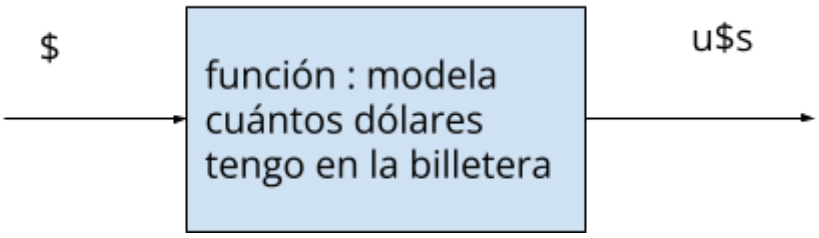
3.8 Más ejemplos de modelado

El segundo nombre de Homero... es un String
```
segundoNombreHomero = "Jay"

*Main> :t segundoNombreHomero
segundoNombreHomero :: [Char]
```

Como vemos, para Haskell un String es una lista de caracteres, donde el tipo lista se asocia con los corchetes. Un caracter se escribe con comillas simples:
```
*Main> :t 'a'
'a' :: Char
```

Queremos saber en pesos cuántos dólares tenemos en la billetera.

Esto... es una función: dada la cantidad de pesos que tengamos en la billetera y el factor de conversión de pesos a dólares, podremos saber cuántos dólares tenemos.



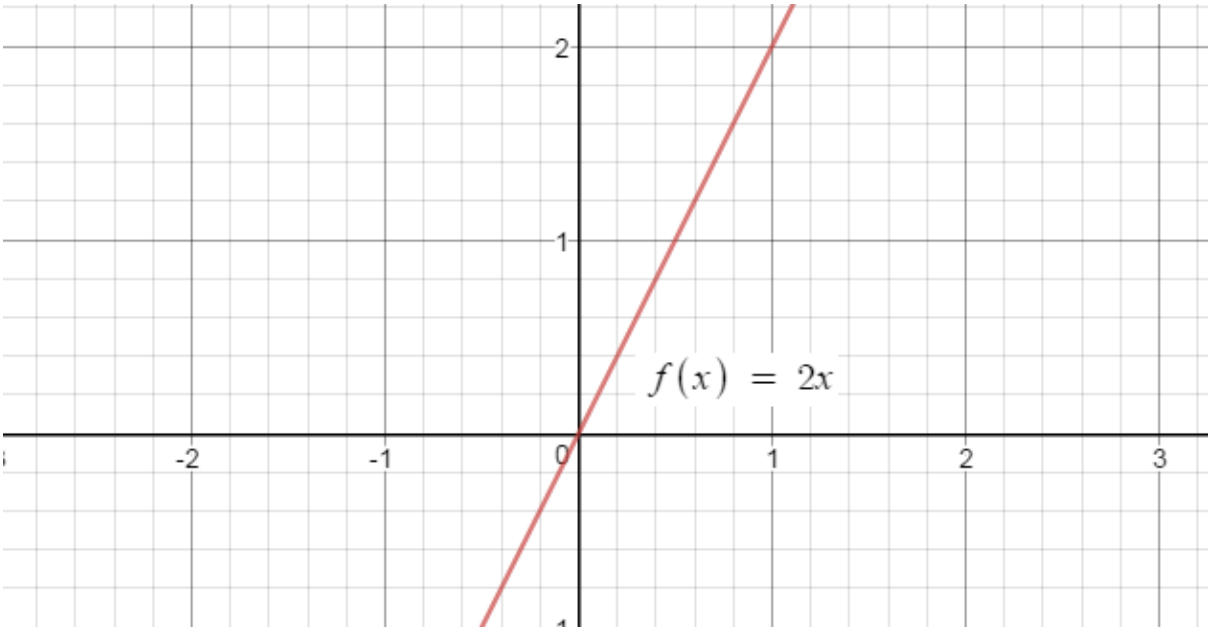

```
pesosADolares :: Float -> Float
pesosADolares pesos = pesos / 206.50

*Main> pesosADolares 500
2.4213075060532688
```

¿Y la conversión de millas a kilómetros? Claramente, es una función.

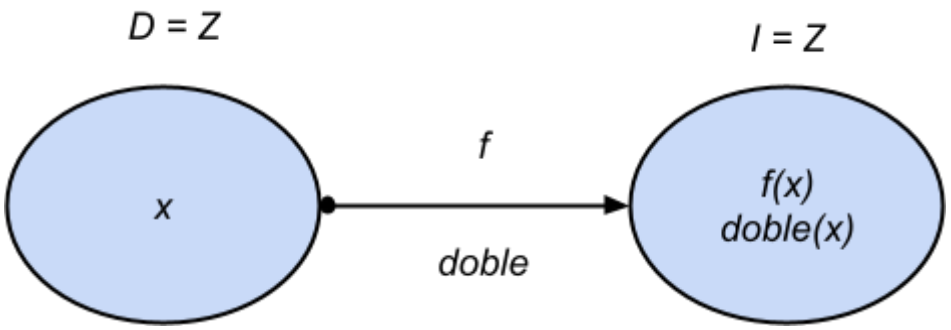
```
millasAKilometros :: Float -> Float
millasAKilometros millas = millas * 1.609344
```

4 Calentando motores

4.1 Función doble



Como hemos visto una función representa una relación (que respeta unicidad y existencia) entre un conjunto dominio y un conjunto imagen.
Ej.: `doble: ℤ -> ℤ / doble(x) = 2 * x`



En este caso la función 'doble', cuyo dominio son los enteros y cuyo conjunto imagen también son los enteros, está definida como el doble de cada elemento del dominio.

Implementamos la función doble:

```
doble :: Int -> Int
doble x = 2 * x
                                                                    pdp.hs
```

El signo = nos define una igualdad en el sentido más matemático posible, nos sirve a la hora de reducir expresiones.

Evaluamos la expresión doble con el parámetro 4:

```
Main> doble 4
8 (doble 4 => 2 * 4 => 8)
```

4.2 Chequeo de tipos de Haskell

Cada expresión se reduce a un valor. Decimos que aplicamos el parámetro 4 a la función doble, que por tener un solo parámetro en su definición produjo su reducción. Ahora bien, si alguien se fijó atentamente entre el mapeo entre la definición matemática y la definición Haskell hay una particularidad que son los conjuntos de partida/llegada

en matemática era $\mathbb{Z}$ -> $\mathbb{Z}$
en Haskell Int->Int

esto quiere decir que desde Haskell vemos a los conjuntos que definen la función como a un tipo de dato. Por eso decimos que la función tiene un tipo y es parte característica de ella. En particular el tipo de doble es:

```
Main>:t doble
```

```
doble:: Int -> Int
```

Aclaración importante: no es necesario definir el tipo de mi función en Haskell dado que el motor sabe inferir éstos a partir de la implementación que nosotros hagamos. Por ende también es válido definir la función doble como:

```
doble x = 2 * x
```

2 * x <= el operador (*) tiene que poder aplicarse sobre 2 y sobre x. Como 2 es un número, está todo bien solo si x es también un número. Si x es un String no lo puedo multiplicar por dos, el operador (*) no es válido si el tipo de x no es el correcto. ¿Quién hace este chequeo? El compilador.

Tres tipos de chequeo que el compilador hace:

1) Si defino
`doble x = "pepe" * x`
el valor `"pepe"` no es del tipo adecuado para el operador de la multiplicación (*).

2) Si defino la función doble como:
`doble :: String -> Int`
no va a funcionar en la definición del cuerpo, cuando quiera operar (*) con un argumento de tipo String:
`doble x = 2 * x`

3) Si pido evaluar la expresión
`Main> doble "2"`
el valor “2” (String) no coincide con el tipo Int.

Por eso hablamos de que Haskell es un lenguaje con chequeo estático de tipos.

Tener chequeo de tipos estático o dinámico es una característica del lenguaje y no del paradigma. Hay otros lenguajes funcionales como Joy o Factor donde el compilador no chequea estrictamente los tipos de los argumentos y las expresiones, por eso son lenguajes con tipado dinámico.

5 Definiciones con guardas

5.1 Función max

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:

$$\max : \mathbb{Z}^2 \to \mathbb{Z} \quad \begin{cases} \max(x,y) = x & x > y \\ \max(x,y) = y & y \geq x \end{cases}$$

$$\max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

en Haskell esto se traduce a:

```
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

```
max x y | x > y     = x
| otherwise = y
```

Haskell no lo sabe interpretar correctamente y falla la compilación.

- ¿de qué tipo es max? Repasemos una vez más la definición

```
max x y | x > y     = x
        | otherwise = y
```

Podemos recibir cualquier x y cualquier y, mientras sepan cuál es "mayor" que otro. Esto quiere decir que puedo hacer consultas por número, pero también por caracteres, o Strings:

```
λ max 3 7
7
:: (Num a, Ord a) => a
λ max "francia" "tailandia"
```

```
"tailandia"
:: [Char]
λ max 'd' 'a'
'd'
:: Char
λ max False True
True  -- la verdad es más grande que la mentira
:: Bool
```

¿Podemos comparar Strings con números?

```
λ max "hola" 5
<interactive>:8:12: error:
      Estás operando una lista con un número
      In the second argument of ‘max’, namely ‘5’
      In the expression: max "hola" 5
```

No, aquí vemos que Haskell no puede hacer pasar a 5 como un String. Más adelante profundizaremos este tema.

5.2 Segunda función o el uso innecesario de guardas

Queremos modelar la función que nos dice si podemos avanzar nuestro auto en base a la indicación del semáforo:

- rojo o amarillo implica que no avanzaremos
- verde nos permite avanzar

sin entrar en mayores detalles.

Nuestro input es “rojo”, “amarillo” o “verde”, más adelante veremos formas más interesantes de modelarlo que un String, pero por ahora nos alcanza. El resultado de esta función es un Bool: avanzo o no avanzo.

```
puedoAvanzar :: String -> Bool
puedoAvanzar color | color == "verde" = True
                   | otherwise        = False
```

Esta función parece correctamente definida, y podemos evaluarla.

Pero tiene un uso inadecuado de guardas, si entendemos que `color == "verde"` es una expresión booleana que podemos utilizar perfectamente como valor de retorno de nuestra función. Si eliminamos redundancias, simplificamos nuestra definición a

```
puedoAvanzar color = color == "verde"
```

que es la forma correcta de utilizar expresiones booleanas en el lenguaje.

6 Resumen

En nuestra primera aproximación al paradigma funcional, hemos visto que respeta definiciones matemáticas

- de función como un hecho que depende de otro
- y de variable como incógnita, valor sin resolver, en lugar de una posición de memoria que puede sobreescribirse.

Haskell trae algunas funciones predefinidas y nos permite hacer nuestras propias definiciones de función,

- recibiendo valores de entrada como parámetro
- y devolviendo valores

Cada valor pertenece a un tipo de dato, que nos sirve para representar diferente información de la realidad: los números, los caracteres, los Strings, booleanos e incluso las funciones. Las funciones tienen un tipo, que Haskell es capaz de inferir.

Por último, para trabajar estructuras de selección (if) Haskell utiliza las definiciones con guardas, que debemos utilizar solo cuando no sea posible trabajar con expresiones booleanas como valor de retorno.