Tema 3. Patrones y Definiciones de Funciones

3.1 Comparación de Patrones

- ✓ Permiten modelar cómo se define una función para distintas formas del argumento.

- ✓ Es posible definir una función mediante varias ecuaciones.
  - ◇ Cada ecuación define la función para distintas formas del argumento (patrón).
  - ◇ Al utilizar la función, la comparación de patrones determina la ecuación adecuada.

- ✓ El orden de las ecuaciones es importante:
  - ◇ Se prueban las distintas ecuaciones en el orden dado por el programa.
  - ◇ Dentro de una misma ecuación, se intentan unificar los patrones correspondientes a los argumentos de izquierda a derecha.
  - ◇ En cuanto un patrón falla para un argumento, se pasa a la siguiente ecuación.
  - ◇ Se selecciona solamente la primera ecuación que unifique (No hay backtracking).
  - ◇ Si ninguna ecuación unifica se produce un error en tiempo de ejecución.

Patrones constantes

✓ Puede ser un número, un carácter o un constructor de dato

✓ Con un patrón constante solo *unifica* un argumento que coincida con dicha constante.

$$
\begin{array}{lcl}
f & :: & Integer \to Bool \\
f\ 1 & = & True \\
f\ 2 & = & False
\end{array}
$$

? $f\ 1$
$True\ ::\ Bool$

? $f\ 2$
$False\ ::\ Bool$

? $f\ 3$
$Program\ error:\ \{f\ 3\}$

$$
\begin{array}{lcl}
fact & :: & Integer \to Integer \\
fact\ 0 & = & 1 \\
fact\ n & = & n * fact\ (n-1)
\end{array}
$$

Ejemplo: reducción de $fact\ 2$:

$\underline{fact\ 2}$
$\Longrightarrow$ {segunda ecuación de $fact\ \langle\ n \leftarrowtail 2\ \rangle$}
$2 * fact\ \underline{(2-1)}$
$\Longrightarrow$ {definición de $(-)$}
$2 * \underline{fact\ 1}$
$\Longrightarrow$ {segunda ecuación de $fact\ \langle\ n \leftarrowtail 1\ \rangle$}
$2 * (1 * fact\ \underline{(1-1)})$
$\Longrightarrow$ {definición de $(-)$}
$2 * (1 * \underline{fact\ 0})$
$\Longrightarrow$ {primera ecuación de $fact$}
$2 * \underline{(1 * 1)}$
$\Longrightarrow$ {definición de $(*)$}
$\underline{2 * 1}$
$\Longrightarrow$ {definición de $(*)$}
2

Patrones para listas

✓ Toman las siguientes formas:

- ◇ [ ] unifica con una lista vacía.
- ◇ $[x]$, $[x, y]$, etc. solo unifican con listas de uno, dos, etc. argumentos.
- ◇ $(x : xs)$ unifica con listas no vacías. x queda ligada a la *cabeza* y xs queda ligada a la *cola*. También se puede usar $(x : y : zs)$, $(x : y : v : zs)$, etc. para listas de al menos dos, tres, etc. elementos.

Ejemplo: suma lista de enteros:

$$\begin{aligned} &suma &&:: [Integer] \rightarrow Integer \\ &suma\ [\,] &&= 0 && \text{-- caso base} \\ &suma\ (x : xs) &&= x + suma\ xs && \text{-- caso recursivo} \end{aligned}$$

$$\begin{aligned} & suma\ [1, 2, 3] \\ \Longrightarrow\ & \{\text{sintaxis de listas}\} \\ & \underline{suma\ (1 : (2 : (3 : [\,])))} \\ \Longrightarrow\ & \{\text{segunda ecuación de } suma\ \langle\ x \longleftarrow 1,\ xs \longleftarrow 2 : (3 : [\,])\ \rangle\} \\ & 1 + \underline{suma\ (2 : (3 : [\,]))} \\ \Longrightarrow\ & \{\text{segunda ecuación de } suma\ \langle\ x \longleftarrow 2,\ xs \longleftarrow 3 : [\,]\ \rangle\} \\ & 1 + (2 + \underline{suma\ (3 : [\,])}) \\ \Longrightarrow\ & \{\text{segunda ecuación de } suma\ \langle\ x \longleftarrow 3,\ xs \longleftarrow [\,]\ \rangle\} \\ & 1 + (2 + (3 + \underline{suma\ [\,]})) \\ \Longrightarrow\ & \{\text{primera ecuación de } suma\} \\ & 1 + (2 + (3 + 0)) \\ \Longrightarrow\ & \{\text{definición de } (+) \text{ tres veces}\} \\ & 6 \end{aligned}$$

Patrones para tuplas

✓ Siguen la misma forma que las tuplas

Ejemplos: funciones que permiten seleccionar el primer elemento de tuplas de dos y tres componentes enteras:

```
primero2 :: (Integer, Integer) → Integer
primero2 (x, y) = x

primero3 :: (Integer, Integer, Integer) → Integer
primero3 (x, y, z) = x
```

```
? primero2 (5,8)
5 :: Integer

? primero3 (5,8,7)
5 :: Integer

? primero2 (5,8,7)
ERROR                : Type error in application
*** Expression       : primero2 (5,8,7)
*** Term             : (5,8,7)
*** Type             : (a,b,c)
*** Does not match   : (Int,Int)
```

Patrones aritméticos

- ✓ Para argumentos enteros
- ✓ Tienen la forma $(n + k)$, donde k es una constante natural
- ✓ Solo unifica con un número entero mayor o igual a k
- ✓ La variable n toma el valor del argumento unificado menos k.

Ejemplo: la función factorial:

$$\begin{array}{lcl} factorial & :: & Integer \to Integer \\ factorial\ 0 & = & 1 \\ factorial\ (n+1) & = & (n+1) * factorial\ n \end{array}$$

- ✓ No unifica con argumentos negativos

La reducción de $factorial\ 2$ es:

$$\begin{array}{l} \underline{factorial\ 2} \\ \Longrightarrow \{\text{segunda ecuación de } factorial\ \langle\ n \leftarrowtail 1\ \rangle\} \\ \underline{(1+1)} * factorial\ 1 \\ \Longrightarrow \{\text{definición de } (+)\} \\ 2 * \underline{factorial\ 1} \\ \Longrightarrow \{\text{segunda ecuación de } factorial\ \langle\ n \leftarrowtail 0\rangle\} \\ 2 * (\underline{(0+1)} * factorial\ 0) \\ \Longrightarrow \{\text{definición de } (+)\} \\ 2 * (1 * \underline{factorial\ 0}) \\ \Longrightarrow \{\text{primera ecuación de } factorial\} \\ 2 * (1 * 1) \\ \Longrightarrow \{\text{definición de } (*) \text{ dos veces}\} \\ 2 \end{array}$$

Patrones nombrados o seudónimos

✓ Permiten nombrar un patrón compuesto y utilizar el nombre en vez del patrón

✓ Toman la forma $nombre@patrón$

$$\begin{array}{lcl} factorial' & :: & Integer \to Integer \\ factorial'\ 0 & = & 1 \\ factorial'\ m@(n+1) & = & m * factorial'\ n \end{array}$$

- El cualificador @ en la segunda ecuación asigna el nombre m al patrón $(n+1)$.
- m queda asociada con el valor del argumento
- n queda asociada con el valor del argumento menos uno.

Por ejemplo, la reducción de $factorial''\ 2$ es:

$\underline{factorial'\ 2}$
$\Longrightarrow$ {segunda ecuación de $factorial'\ \langle n \leftarrowtail 1,\ m \leftarrowtail 2 \rangle$}
$2 * \underline{factorial'\ 1}$
$\Longrightarrow$ {segunda ecuación de $factorial'\ \langle n \leftarrowtail 0,\ m \leftarrowtail 1 \rangle$}
$2 * (1 * \underline{factorial'\ 0})$
$\Longrightarrow$ {primera ecuación de $factorial'$}
$2 * (1 * 1)$
$\Longrightarrow$ {definición de $(*)$ dos veces}
2

✓ Los patrones nombrados pueden mejorar ligeramente la eficiencia de una función.

El patrón subrayado

- ✓ Toman la forma _
- ✓ Unifican con cualquier argumento
- ✓ No producen ninguna ligadura

Pueden utilizarse cuando el argumento no es usado en el cuerpo de la función.

Ejemplo: número de elementos de una lista de enteros

$$
\begin{array}{lcl}
longitud & :: & [Integer] \to Integer \\
longitud\ [\,] & = & 0 \\
longitud\ (x : xs) & = & 1 + longitud\ xs
\end{array}
$$

Puede ser escrita usando un patrón subrayado como

$$
\begin{array}{lcl}
longitud & :: & [Integer] \to Integer \\
longitud\ [\,] & = & 0 \\
longitud\ (\_ : xs) & = & 1 + longitud\ xs
\end{array}
$$

Anidando patrones

✓ Se pueden anidar patrones

Ejemplo: función que suma todos los elementos de una lista de pares:

$$\begin{aligned} &sumaPares &&:: [(Integer,\ Integer)] \to Integer \\ &sumaPares\ [\,] &&= 0 \\ &sumaPares\ ((x,y):xs) &&= x + y + sumaPares\ xs \end{aligned}$$

Por ejemplo

$sumaPares\ [(1,2),(3,4)]$

$\Longrightarrow$ {sintaxis de listas}

$\underline{sumaPares\ ((1,2):((3,4):[\,]))}$

$\Longrightarrow$ {segunda ecuación de $sumaPares$ $\langle\ x \leftarrow 1,\ y \leftarrow 2,\ xs \leftarrow (3,4):[\,]\ \rangle$}

$1 + 2 + \underline{sumaPares\ ((3,4):[\,])}$

$\Longrightarrow$ {segunda ecuación de $sumaPares$ $\langle\ x \leftarrow 3,\ y \leftarrow 4,\ xs \leftarrow [\,]\ \rangle$}

$1 + 2 + (3 + 4 + \underline{sumaPares\ [\,]})$

$\Longrightarrow$ {primera ecuación de $sumaPares$}

$1 + 2 + (3 + 4 + 0)$

$\Longrightarrow$ {definición de $(+)$ cuatro veces}

10

Errores comunes

✓ Un nombre de variable no puede aparecer repetido en la parte izquierda (a la izquierda del signo igual) de una misma ecuación:

```
sonIguales      :: Integer → Integer → Bool
sonIguales x x  =  True      -- INCORRECTO: x REPETIDA !!!
sonIguales x y  =  False
```

En su lugar, hay que escribir:

```
sonIguales      :: Integer → Integer → Bool
sonIguales x y  =  (x == y)
```

✓ No es un error repetir el patrón subrayado:

```
siempreVerdad _ _  :: Integer → Integer → Bool
siempreVerdad _ _  =  True
```

✓ El tipo de todas las ecuaciones correspondientes a la definición de una función debe ser el mismo:

```
f 0     =  0   -- INCORRECTO: Bool e Integer son tipos distintos !!!
f True  =  2
```

3.2 Expresiones condicionales

✓ Expresiones cuyo resultado depende de una condición.

$$\textbf{if}\ \boxed{\text{exprBool}}\ \textbf{then}\ \boxed{\text{exprSi}}\ \textbf{else}\ \boxed{\text{exprNo}}$$

✓ El tipo de *exprBool* debe ser *Bool*.

✓ Los tipos de *exprSi* y *exprNo* deben ser iguales.

✓ La parte **else** es obligatoria

Comportamiento:

1. Se evalúa el valor de *exprBool*.
2. Si el valor es *True*, el valor de la expresión es el de *exprSi*.
3. En otro caso, el valor de la expresión es el de *exprNo*.

Ejemplo. Máximo de dos enteros:

```
máximo     :: Integer → Integer → Integer
máximo x y = if x >= y then x else y
```

✓ La evaluación es perezosa:

```
? if 5 > 2 then 10.0 else (10.0/0.0)
10.0 :: Double

? if 5 < 2 then 10.0 else (10.0/0.0)
Program error : {primDivFloat 10.0 0.0}

? 2 * if 'a' < 'z' then 10 else 4
20 :: Integer
```

3.3 Funciones por casos

✓ Generalización de las expresiones condicionales

```
absoluto      :: Integer → Integer
absoluto x
  | x >= 0 = x
  | x < 0  = -x
```

✓ Las expresiones entre los símbolos | y = se denominan *guardas* (tipo *Bool*)

✓ Se devuelve el resultado correspondiente a la **primera** guarda cierta

Ejemplo: devuelve -1, 0 ó 1 dependiendo del signo del argumento:

```
signo         :: Integer → Integer
signo x
  | x > 0  = 1
  | x == 0 = 0
  | x < 0  = -1
```

✓ *otherwise* es equivalente al valor *True*. Suele aparecer como última guarda:

```
signo         :: Integer → Integer
signo x
  | x > 0     = 1
  | x == 0    = 0
  | otherwise = -1
```

✓ Cuidado con el sangrado

3.4 Expresiones case

✓ Permiten calcular un resultado que depende de la forma de una expresión

Sintaxis:

$$\begin{array}{l}\textbf{case}\ \boxed{\text{expr}}\ \textbf{of}\\ \quad \boxed{Patrón_1} \rightarrow \boxed{expr_1}\\ \quad \boxed{Patrón_2} \rightarrow \boxed{expr_2}\\ \quad \ldots\\ \quad \boxed{Patrón_n} \rightarrow \boxed{expr_n}\end{array}$$

Comportamiento:

1. Se evalúa $expr$.
2. Se devuelve la primera $expr_i$ tal que $Patrón_i$ unifica con el resultado de evaluar $expr$
3. Si ningún $Patrón_i$ unifica se produce un error

✓ $expr$ y todos los $Patrón_i$ han de tener el mismo tipo.

✓ Todas las $expr_i$ han de tener el mismo tipo.

Ejemplo:

$$\begin{array}{lcl} suma & :: & [Integer] \rightarrow Integer\\ suma\ ls & = & \textbf{case}\ ls\ \textbf{of}\\ & & \quad [\,] \rightarrow 0\\ & & \quad (x : xs) \rightarrow x + suma\ xs\end{array}$$

3.5 La función error

✓ Abortan la evaluación de una expresión y muestran un mensaje por pantalla.

Ejemplo:

$$
\begin{array}{lcl}
cabeza & :: & [Integer] \to Integer \\
cabeza\ [\,] & = & error\ \ \text{"cabeza de lista vacía no definida"} \\
cabeza\ (x : \_\,) & = & x
\end{array}
$$

✓ El intérprete mostrará el mensaje cuando se aplique $cabeza$ a una lista vacía:

? $cabeza\ [1,2,3]$
$1\ \ ::\ Integer$

? $cabeza\ [\,]$
$Program\ error :\ cabeza\ de\ lista\ vacía\ no\ definida$

✓ $error$ permite controlar casos para los que la definición de la función no tiene sentido y emitir un mensaje por pantalla.

3.6 Definiciones locales

✓ Definiciones con visivilidad limitada

✓ Suelen usarse para nombrar una expresión que aparece varias veces en una función

Ejemplo:

```
raíces          :: Float → Float → Float → (Float, Float)
raíces a b c
  | disc >= 0   = ((−b + raízDisc)/denom, (−b − raízDisc)/denom)
  | otherwise   = error "raíces no reales"
  where
    disc        = b^2 − 4 ∗ a ∗ c
    raízDisc    = sqrt disc
    denom       = 2 ∗ a
```

$disc$, $raízDisc$ y $denom$ son *definiciones locales* de la función $raíces$.

✓ Se gana eficiencia: definiciones locales constantes son calculadas como máximo una vez.

✓ Las definiciones locales **where** solo pueden aparecer al final de una declaración de función.

✓ Es posible introducir definiciones locales en cualquier parte de una expresión usando las palabras **let** e **in**:

```
? let f n = n^2 + 2 in f 100
10002 :: Integer
```

3.7 Operadores

- ✓ Funciones de dos argumentos con nombre simbólico
- ✓ Suelen usarse de modo infijo

```
? 1 + 2
3 :: Integer
```

- ✓ Pueden usarse de modo prefijo (entre paréntesis)

```
? (+) 1 2
3 :: Integer
```

- ✓ Una función de dos argumentos se puede usar infija (entre acentos franceses):

```
? 10 `div` 3
3 :: Integer
```

- ✓ Para definir operadores se pueden utilizar uno o más de:

: ! # $ % & * + . / < = > ? @ \ ^ | - ~

- ✓ ~ solo se puede usar una vez y al principio
- ✓ Si comienzan por dos puntos (:) son *constructores de datos infijos*.

Algunos ejemplos:

```
+   ++   &&   ||   <=   ==   /=   .   //   $
::  ->   =>   :    ..   =    @    \   |    <-   ~
%   @@   -*-       /    <+>  ?
```

- ✓ Los de la primera línea están predefinidos
- ✓ Los de la segunda están reservados

Operadores (2)

- ✓ Prioridad:
  - ◇ Entre 0 y 9
  - ◇ Valor mayor significa mayor prioridad
- ✓ Asociatividad
  - ◇ Izquierda(**infixl**), derecha (**infixr**) o ninguna (**infix**)
- ✓ Tabla Prelude

```
infixr  9  .
infixl  9  !!
infixr  8  ^, ^^, **
infixl  7  *, /, `quot`, `rem`, `div`, `mod`
infixl  6  +, -
infixr  5  :
infixr  5  ++
infix   4  ==, /=, <, <=, >=, >, `elem`, `notElem`
infixr  3  &&
infixr  2  ||
infixl  1  >>, >>=
infixr  1  =<<
infixr  0  $, $!, `seq`
```

- ✓ Ejemplo

```
-- O exclusivo
infixr 2 |||

(|||)          :: Bool → Bool → Bool
True  ||| True  = False
False ||| False = False
_     ||| _     = True
```

3.8 Bibliotecas

✓ Haskell proporciona un conjunto de definiciones globales que pueden ser usadas por el programador sin necesidad de definirlas.

✓ Estas definiciones aparecen agrupadas en *módulos de biblioteca*

(véase `http://haskell.org/onlinelibrary`).

Ejemplo: biblioteca *Ratio* (define el tipo *Rational*)

```
import Ratio
sumaCubos       :: Rational→ Rational → Rational
sumaCubos ra rb  =  ra ^ 3 + rb ^ 3
```

Uso de la función anterior

```
? sumaCubos (1 %3) (2 %7)
559 % 9261  :: Rational
```

✓ El módulo de biblioteca especial *Prelude* es automáticamente importado por cualquier programa (elementos *predefinidos* del lenguaje).

✓ El programador también puede definir sus propias bibliotecas.

```
module MiBiblioteca where

doble    :: Integer → Integer
doble x  =  x + x

cuadruple    :: Integer → Integer
cuadruple x  =  doble (doble x)
```

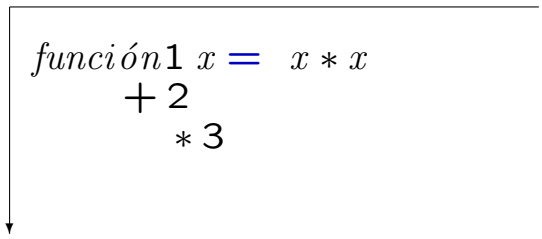
3.9 Sangrado

¿Dónde acaba una definición y comienza otra?.

✓ Regla del *layout* (o *offside rule*):

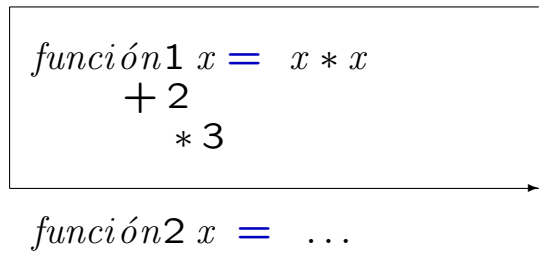
> Una definición acaba con el primer trozo de código que aparece a la misma altura horizontal o a la izquierda del comienzo de la definición actual.

- El primer carácter de una definición abre una caja que alberga la definición:



- La caja permanece abierta hasta que se encuentra algo a la misma altura horizontal o a la izquierda del comienzo de la definición actual.

- En este punto, la caja se cierra:



Sangrado (2)

✓ La regla del *layout* se aplica tras las palabras **let**, **where** , **do** y **of**, además de a las definiciones globales.

✓ Consecuencia de esta regla:

- ◇ todas las definiciones globales deben tener el mismo sangrado (se recomienda que comiencen en la primera columna).
- ◇ todas las definiciones locales introducidas por **where** o **let** deben tener el mismo sangrado.

Ejemplo de sangrado **INCORRECTO**:

```
f1  :: Integer → Integer
f1 x = z + y
  where
    z = 3
      y = 4
```

✓ Una sintaxis alternativa es utilizar llaves y el separador punto y coma:

```
g     :: Integer → Integer → Integer
g x y = doble x + triple y where {doble n = 2 * n; triple n = 3 * n}

h     :: Integer → Integer → Integer
h x y = let{doble n = 2 * n; triple n = 3 * n} in doble x + triple y
```

Objetivos del tema

El alumno debe:

- ✓ Conocer los distintos tipos de patrones
- ✓ Conocer como unifican dichos patrones
- ✓ Saber definir funciones utilizando patrones
- ✓ Saber reducir expresiones en las que sea necesario unificar patrones
- ✓ Conocer las expresiones condicionales, las funciones por casos, las expresiones case y la función error
- ✓ Saber definir funciones utilizando las construcciones anteriores
- ✓ Saber utilizar definiciones locales siempre que sea adecuado:
  - ◇ La definición solo sea usada desde una función
  - ◇ Una expresión constante aparezca varias veces en una función
- ✓ Conocer que existen bibliotecas y cómo importarlas
- ✓ Saber sangrar adecuadamente el código que se escriba