

Tema 3. Patrones y Definiciones de Funciones

3.1 Comparación de Patrones

Patrones constantes

Patrones para listas

Patrones para tuplas

Patrones aritméticos

Patrones nombrados o seudónimos

El patrón subrayado

Anidando patrones

Errores comunes

3.2 Expresiones condicionales

3.3 Funciones por casos

3.4 Expresiones case

3.5 La función error

3.6 Definiciones locales

3.7 Operadores

3.8 Sangrado

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.
 - ◇ Si ninguna ecuación unifica se produce un error durante la reducción.

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.

```
f  :: Integer → Bool
f 1 = True
f 2 = False
```

```
? f 1
True :: Bool
```

```
? f 2
False :: Bool
```

```
? f 3
Program error : {f 3}
```

Recursividad

- ✓ La función definida puede invocarse a sí misma para un argumento *más simple*

$$\begin{aligned} \text{fact} &:: \text{Integer} \rightarrow \text{Integer} \\ \text{fact } 0 &= 1 \\ \text{fact } n &= n * \text{fact } (n - 1) \end{aligned}$$

Ejemplo: reducción de *fact* 2:

$$\begin{aligned} &\underline{\text{fact } 2} \\ \implies &\{\text{segunda ecuación de } \text{fact} \langle n \leftarrow 2 \rangle\} \\ &2 * \underline{\text{fact } (2 - 1)} \\ \implies &\{\text{definición de } (-)\} \\ &2 * \underline{\text{fact } 1} \\ \implies &\{\text{segunda ecuación de } \text{fact} \langle n \leftarrow 1 \rangle\} \\ &2 * (1 * \underline{\text{fact } (1 - 1)}) \\ \implies &\{\text{definición de } (-)\} \\ &2 * (1 * \underline{\text{fact } 0}) \\ \implies &\{\text{primera ecuación de } \text{fact}\} \\ &2 * (\underline{1 * 1}) \\ \implies &\{\text{definición de } (*)\} \\ &\underline{2 * 1} \\ \implies &\{\text{definición de } (*)\} \\ &2 \end{aligned}$$

- ✓ Recursividad termina si
 - ◇ Existe caso base (no recursivo)
 - ◇ Llamada recursiva se acerca al caso base

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:

```
suma      :: [Integer] → Integer
suma []   = 0                -- caso base
suma (x : xs) = x + suma xs  -- caso recursivo
```

`suma [1, 2, 3]`

≈> {sintaxis de listas}

`suma (1 : (2 : (3 : [])))`

⇒ {segunda ecuación de *suma* $\langle x \leftarrow 1, xs \leftarrow 2 : (3 : []) \rangle$ }

`1 + suma (2 : (3 : []))`

⇒ {segunda ecuación de *suma* $\langle x \leftarrow 2, xs \leftarrow 3 : [] \rangle$ }

`1 + (2 + suma (3 : []))`

⇒ {segunda ecuación de *suma* $\langle x \leftarrow 3, xs \leftarrow [] \rangle$ }

`1 + (2 + (3 + suma []))`

⇒ {primera ecuación de *suma*}

`1 + (2 + (3 + 0))`

⇒ {definición de (+) tres veces}

6

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)
```


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

$$\begin{aligned} \textit{longitud} & \quad \quad \quad \text{:: } [\textit{Integer}] \rightarrow \textit{Integer} \\ \textit{longitud} [] & \quad \quad \quad = 0 \\ \textit{longitud} (x : xs) & = 1 + \textit{longitud} xs \end{aligned}$$

Puede ser escrita usando un patrón subrayado como

$$\begin{aligned} \textit{longitud} & \quad \quad \quad \text{:: } [\textit{Integer}] \rightarrow \textit{Integer} \\ \textit{longitud} [] & \quad \quad \quad = 0 \\ \textit{longitud} (_ : xs) & = 1 + \textit{longitud} xs \end{aligned}$$

Errores comunes

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

$sonIguales \quad :: \quad Integer \rightarrow Integer \rightarrow Bool$
 $sonIguales \ x \ x \ = \ True \quad \text{-- INCORRECTO: x REPETIDA !!!}$
 $sonIguales \ x \ y \ = \ False$

En su lugar, hay que escribir:

$sonIguales \quad :: \quad Integer \rightarrow Integer \rightarrow Bool$
 $sonIguales \ x \ y \ = \ (x == y)$

- ✓ No es un error repetir el patrón subrayado:

$siempreVerdad \quad \quad \quad :: \quad Integer \rightarrow Integer \rightarrow Bool$
 $siempreVerdad \ _ \ _ \ = \ True$

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

$f \ 0 \quad = \ 0 \quad \text{-- INCORRECTO: Bool e Integer son tipos distintos !!!}$
 $f \ True \ = \ 2$

3.2 Expresiones condicionales

- ✓ Expresiones cuyo resultado depende de una condición.

if `exprBool` then `exprSi` else `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 : {*primDiv Float* 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
| $x \geq 0$ = 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:

```
case expr of
  Patrón1 → expr1
  Patrón2 → expr2
  ...
  Patrónn → exprn
```

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:

```
suma    :: [Integer] → Integer
suma ls = case ls of
  []      → 0
  (x : xs) → x + suma xs
```

3.5 La función error

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

Ejemplo:

```
cabeza           :: [Integer] → Integer  
cabeza []        = error "cabeza de lista vacía no definida"  
cabeza (x : _)  = x
```

- ✓ 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 visibilidad 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*.

- ✓ 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:

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

✓ 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 Sangrado

- ✓ La regla del *sangrado* se aplica tras las palabras *let*, *where*, *do* y *of*, además de a las definiciones globales y guardas.
- ✓ 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 el concepto de recursividad
- ✓ 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
- ✓ Saber sangrar adecuadamente el código que escriba