# Programación Funcional

Curso 2019-20

## EXPRESIONES SINTÁCTICAS DE HASKELL

# Expresiones condicionales

`if b then e else e'`

*b* expresión booleana | *e* , *e'* expresiones del mismo tipo

```
> if 2+2==4 then [1,2] else []
[1,2]
```

`if _ then _ else _` función de aridad 3,
no es una instrucción de control

```
> [1,if False then 2 else 3,3]
[1,3,3]
```

`if _ then _ else _` puede aparecer en cualquier sitio donde pueda aparecer una expresión

```
> if True then 1 else 3 `div` 0
1
```

evaluación perezosa

```
> if True then 1 else [1,2]
```

Mal tipada

Las partes `then` y `else` deben ser del mismo tipo

El tipo de una expresión se preserva durante su evaluación

# Definiciones locales (expresiones o ligaduras *let*)

```
> length [1..10^8]
100000000
(1.70 secs, 4000551148 bytes)
> length [1..10^8] + length [1..10^8]
200000000
(3.39 secs, 8000301264 bytes)
> let x=length [1..10^8] in x+x
```

Definición local de la variable `x`

```
200000000
(1.68 secs, 3999167476 bytes)
```

`x` se computa una sola vez

Hay compartición (*sharing*)

# Definiciones locales (II)

`let` $x$`=`$e$ `in` $e'$

$x$ variable ligada | $e$ ligadura de $x$ | $e'$ expresión principal

- Una expresión *let* $x$=$e$ *in* $e'$ es una expresión más.
- El valor de *let* $x$=$e$ *in* $e'$ es el valor que tenga $e'$.
- En la evaluación de $e'$ el valor de $x$ es el valor de $e$.
- Valor de *let* $x$=$e$ *in* $e'$ = valor de $e'[x/e]$
  donde $e'[x/e] \equiv$ resultado de sustituir $x$ por $e$ en $e'$.
  Pero en *let* $x$=$e$ *in* $e'$ se comparte el cómputo de $e$ a través de la ligadura de $x$ mientras que en $e'[x/e]$ se repite (o puede repetirse) el cómputo de $e$.
- El valor de $x$ se computa según lo requiera $e'$ (evaluación perezosa).
- La ligadura de $x$ se circunscribe a $e'$.
- $x$ es una variable *muda*. Se puede renombrar (consistentemente) sin riesgo.
- Más adelante: otras variantes de definiciones locales.

# Definiciones locales (III)

```
> 5 + let x=2*3 in x+x
17
```

`let _ in _` puede aparecer en cualquier sitio donde pueda aparecer una expresion

```
> 5 + let y=2*3 in y+y
17
```

renombramiento de variables

```
> let x=div 1 0 in fst (1,x)
1
```

evaluación perezosa

```
> let x=length [1..]  in fst (1,x)
1
```

evaluación perezosa

```
> let x=length [1..10^8] in x+x
200000000
(1.68 secs)
> let x=length [1..10^8] in 3
3
(0.00 secs)
```

evaluación perezosa

```
> (let x=5 in x+x) + (let x=1 in 2*x)
12
```

ámbito de las ligaduras

```
> (let x=5 in x+x) + x
Exception:  x not in scope
```

ámbito de las ligaduras

# Definiciones locales (IV)

```
> let x=2 in let y=x+x in y*y
```
anidamiento de `let`'s

```
16
> let y= (let x=2 in x+x) in y*y
16
> let x=2 in let y=x+x in y*y*x
32
> let y= (let x=2 in x+x) in y*y*x
Exception:  x not in scope
> let y=x+x in let x=2 in y*y*x
Exception:  x not in scope
> let x=1 in let x=2 in x
2
```
ámbito de las ligaduras

```
> let {x=2 ; y=x+x} in y*y*x
```
bloque de `let`'s

```
32
> let {y=x+x ; x=2} in y*y*x
```
bloque de `let`'s

```
32
> (let x=x in x+x)
```
Ligadura recursiva o circular

No termina *(pero no es un error sintáctico)*

```
> let x=1:2:x in take 3 x
```
x se liga a [1,2,1,2,...

```
[1,2,1]
```
Ligadura recursiva + evaluación perezosa

# Expresiones *case*

```
case e of
       t1      -> e1
       ...     -> ...
       tn      -> en
```

- $t_1, \ldots, t_n$: patrones lineales
- $e, e_1, \ldots, e_n$: expresiones
- Se evalúa $e$ y se intenta ajustar con $t_1, \ldots$
- Si $t_i$ es el primer patrón con el que ajusta, el resultado es la evaluación de $e_i$ (si no ajusta ninguno, error)
- Regla de indentación para $t_i$->$e_i$