Funciones en matemáticas y en programación funcional

Notación en matemáticas	Notación en lenguaje `Haskell`
	`type R = Float`
$f, g : \mathbb{R} \to \mathbb{R}$	`f,g :: R -> R`
$f(x) = x^2$	`f x = x^2`
$g(x) = 2x - 5$	`g x = 2*x-5`
$h = f \circ g$	`h = f.g`
$f(4)$	`> f(4)`
	`16.0`
	o bien, omitiendo los paréntesis:
	`> f 4`
	`16.0`
$g(7.1)$	`> g 7.1`
	`9.2`
$h(7.1)$	`> h 7.1`
	`84.64`
$p : \mathbb{Z} \to \mathbb{B}$	`p :: Int -> Bool`
$p(n) = (3 < n) \land (n < 8)$	`p n = 3<n && n<8`
$c : S \to \mathbb{Z}$	`c :: String -> Int`
$c(s) = \lvert \mathrm{cola}(ss) \rvert$	`c s = length(tail(s++s))`
$q = p \circ c$	`q = p.c`
	`> :t q`
	`q :: [Char] -> Bool`
	`> q "ab"`
	`False`
	`> q "abcd"`
	`True`
	`> q "abcde"`
	`False`

Notación en matemáticas	Notación en lenguaje `Haskell`

$I : A \to A$
$I(x) = x$
	Si no se declara el tipo, se infiere el más general:
	
	`iden x = x`
	
	`> :t iden`
	`iden :: a -> a`
	`> iden 2` `> iden f` `> iden p`
	`2` `f` `p`

Sean A, B y C conjuntos, $\circ : (A \to B) \times (C \to A) \to (C \to B)$	`> :t (.)` `(.) :: (a->b) -> (c->a) -> c->b`
Composición algebraica (de izquierda a derecha):	

$rs = s \circ r$	`(>>>) r s = s.r`
	`> :t (>>>)`
	`(>>>) :: (a->b) -> (b->c) -> a->c`
	
	`q'= c >>> p`
	`> :t q'`
	`q' :: [Char] -> Bool`
	`> q' "abcd"`
	`True`
	
	`h' = g >>> f`
	`h'' = f >>> g`
	
	`> h' 7.1` `> h'' 7.1`
	`84.64` `95.82`

<table>
<tr><th>Notación en matemáticas</th><th>Notación en lenguaje <code>Haskell</code></th></tr>
<tr><td>

$$\mathrm{ff}(x) = \begin{cases} x^2, & \text{si } x \leq 2, \\ 3x, & \text{si } 2 < x < 4, \\ g(x), & \text{si } 4 \leq x. \end{cases}$$

</td><td>

```
ff x | x<=2        = x^2
     | 2<x && x<4 = 3*x
     | 4<=x        = g x

> ff 2        > ff 3       > ff 6
4.0           9.0          7.0
```

</td></tr>
<tr><td>

$\mathrm{fac}, \mathrm{fac}' : \mathbb{N} \rightarrow \mathbb{N}$

$$\mathrm{fac}(n) = \begin{cases} 1, & \text{si } n = 0, \\ n\ \mathrm{fac}(n-1), & \text{si } n > 0. \end{cases}$$

$$\begin{aligned} \mathrm{fac}'(0) &= 1, \\ \mathrm{fac}'(n+1) &= (n+1)\ \mathrm{fac}'(n) \end{aligned}$$

</td><td>

```
fac, fac' :: Int -> Int

fac n  = if n==0 then 1
                 else n*fac(n-1)

fac' 0     = 1
fac' (n+1) = (n+1)*fac'(n)

> fac 5                > fac' 5
120                    120
```

</td></tr>
<tr><td></td><td>

```
divisores n =
  [d | d<-[1..n], n `mod` d == 0]

primo n = divisores n == [1,n]

> divisores 847
[1, 7, 11, 77, 121, 847]
> primo 847
False
> primo 853
True
> [p | p<-[830..880], primo p]
[839, 853, 857, 859, 863, 877]
```

</td></tr>
</table>

Notación en matemáticas	Notación en lenguaje `Haskell`
Producto vectorial:	
	`type R3 = (R,R,R)`
$\times : (\mathbb{R}^3)^2 \to \mathbb{R}^3$	`(>*<) :: R3 -> R3 -> R3`
$(x_1, y_1, z_1) \times (x_2, y_2, z_2) =$	`(>*<) (x1,y1,z1) (x2,y2,z2) =`
$(y_1 z_2 - y_2 z_1,$	`(y1*z2-y2*z1,`
$z_1 x_2 - z_2 x_1,$	`z1*x2-z2*x1,`
$x_1 y_2 - x_2 y_1)$	`x1*y2-x2*y1)`
Producto de un escalar por un elmento de $\mathbb{R}^3$:	
$a \in \mathbb{R}, (x, y, z) \in \mathbb{R}^3,$	`(<.>) :: R -> R3 -> R3`
$a(x, y, z) = (ax, ay, az)$	`(<.>) a (x,y,z) = (a*x,a*y,a*z)`
Producto interior de dos elementos de $\mathbb{R}^3$:	
$\bullet : (\mathbb{R}^3)^2 \to \mathbb{R}$	`(<*>) :: R3 -> R3 -> R`
$(x_1, y_1, z_1) \bullet (x_2, y_2, z_2) =$	`(<*>) (x1,y1,z1) (x2,y2,z2) =`
$x_1 x_2 + y_1 y_2 + z_1 z_2$	`x1*x2 + y1*y2 + z1*z2`
Norma o tamaño de un elementos de $\mathbb{R}^3$:	
$\vert\vert : \mathbb{R}^3 \to \mathbb{R}$	`norma :: R3 -> R`
$\vert(x, y, z)\vert = \sqrt{x^2 + y^2 + z^2}$	`norma (x,y,z) = sqrt (x^2+y^2+z^2)`
Triple producto escalar:	
	`tpe :: R3 -> R3 -> R3  -> R`
$A \bullet (B \times C)$	`tpe a b c = a <*> (b >*< c)`
$(3, 4, 5) \times (4, 5, 6)$	`> (3,4,5) >*< (4,5,6)`
	`(-1.0,2.0,-1.0)`
$2.1(3, 4, 5)$	`> 2.1 <.> (3,4,5)`
	`(6.3,8.4,10.5)`
$(3, 4, 5) \bullet (4, 5, 6)$	`> (3,4,5)<*>(4,5,6)`
	`62.0`
$\vert(3, 4, 5)\vert$	`> norma (3,4,5)`
	`7.07107`
$(3, 4, 5) \bullet (4, 5, 6) \times (2, -1, 3)$	`> tpe (3,4,5) (4,5,6) (2,-1,3)`
	`-7.0`

<table>
<tr><th>Notación en mats.</th><th>Notación en lenguaje <code>Haskell</code></th></tr>
<tr><td colspan="2">Suma de Riemann (aproximación a la integral de Riemann):</td></tr>
<tr><td>

$S_{[a,b]}(f,n) =$
$\Sigma_{i=0}^{n-1} f(x_i)\Delta x$

donde $\Delta x = \dfrac{b-a}{n}$

</td><td>

```
suma_de_Riemann f (a,b) n =
  sum [f(a+i*deltax)*deltax | i<-[0..n-1]]
  where deltax = (b-a)/n
> :t suma_de_Riemann
suma_de_Riemann :: (Fractional a, Enum a) =>
                         (a->a)->(a,a)->a -> a
> suma_de_Riemann f (1,2) 100
2.31835
> suma_de_Riemann f (1,2) 1000
2.33183
> suma_de_Riemann g (1,2) 1000
-2.001
> suma_de_Riemann h (1,2) 1000
4.33733
> suma_de_Riemann ff (1,6) 2000
30.3121
> suma_de_Riemann (\x->x^3) (0,2) 1000
3.992

suma_de_RiemannB f (a,b) n =
  sum [f(xi)*deltax | xi<-xis]
  where deltax = (b-a)/fromInt(n)
        xis = take (n-1) (iterate (+deltax) a)
> :t suma_de_RiemannB
suma_de_RiemannB ::
  Fractional a => (a->a)->(a,a)->Int -> a
> :t fromInt
fromInt :: Num a => Int -> a
> :t iterate
iterate :: (a -> a) -> a -> [a]
> suma_de_RiemannB h (1,2) 1000
4.33617
```

</td></tr>
</table>

Cálculo lambda: un modelo de computación mediante funciones.

Definición de *expresión–λ* (*exp–λ*). Sea $\mathcal{V}$ un conjunto de *identificadores* o *variables* (como x, y, x_1, x_2, etc.) y $\mathcal{C}$ un conjunto de *constantes* (como 2, 5, +, *, etc.).

e1) Las variables (y las constantes) son exp–λ.

e2) Si **x** es una variable y **e** es una exp–λ, entonces (λ**x**.**e**) es exp–λ. (Se le llama *abstracción* o *definición de una función*, en la que **x** es la variable *ligada* y **e** el *cuerpo* de la abstracción.)

e3) Si $\mathbf{e}_1$ y $\mathbf{e}_2$ son exp–λ, entonces ($\mathbf{e}_1\,\mathbf{e}_2$) es exp–λ. (Se le llama *aplicación* o *combinación*. Sirve para modelar la aplicación del *operador* $\mathbf{e}_1$ al *operando* $\mathbf{e}_2$.)

Definición de variable *libre* en una exp–λ.

l1) **x** es libre en **x**.

l2) **x** es libre en (λ**y**.**e**) ⇔ **x**≠**y** y **x** es libre en **e**.

l3) **x** es libre en ($\mathbf{e}_1\,\mathbf{e}_2$) ⇔ **x** es libre en $\mathbf{e}_1$ o en $\mathbf{e}_2$.

Definición de variable *ligada* (o *enlazada* o *acotada*) en una exp–λ.

a1) **x** no es ligada en **y**. (En particular, **x** no es ligada en **x**.)

a2) **x** es ligada en (λ**y**.**e**) ⇔ **x**=**y** o **x** es ligada en **e**.

a3) **x** es ligada en ($\mathbf{e}_1\,\mathbf{e}_2$) ⇔ **x** es ligada en $\mathbf{e}_1$ o en $\mathbf{e}_2$.

Definición. El *alcance* de **x** en (λ**x**.**e**) es toda esta exp–λ, salvo las subexpresiones de la forma (λ**x**.**e'**) que ocurran en **e**.

Ejemplos.

En cálculo–λ	En `Haskell`
$(\lambda x.x^2)$	`(\x->x^2)`
$(\lambda x.2x-5)$	`(\x->2*x-5)`
$(\lambda x.x^2)\,4$	`> (\x->x^2) 4`
	`16`
$(\lambda x.2x-5)\,7.1$	`(\x->2*x-5) 7.1`
	`9.2`
$(\lambda x.x^2)((\lambda x.2x-5)(7.1))$	`> (\x->x^2) ((\x->2*x-5) 7.1)`
	`84.64`
	`> (\x -> (\y -> x*(y+5))) 2 3`
	`16`
	`> (\x -> \y -> x*(y+5)) 2 3`
	`16`
	`(\x y -> x*(y+5)) 2 3`
	`16`
	`> (\(x,y) ->x*(y+5)) (2,3)`
	`16`
	`> :t (\x -> \y -> x*(y+5))`
	`\x -> \y -> x*(y+5) :: Num a => a->a->a`
	`> :t (\x -> \y -> x*(y+5)) 2`
	`(\x -> \y -> x*(y+5)) 2 :: Num a => a->a`

Es interesante observar que el operador de composición de funciones se puede construir en términos de las exp–λ $\mathbf{K}=(\lambda xy.x)$ y $\mathbf{S}=(\lambda fgx.fx(gx))$ así:

$$\circ = \mathbf{S}\,(\mathbf{K}\,\mathbf{S})\,\mathbf{K}$$

es decir: $\circ = \Big(\lambda fgx.fx(gx)\Big)\,\Big((\lambda xy.x)\,(\lambda fgx.fx(gx))\Big)\,\Big(\lambda xy.x\Big)$

Experimentemos con Haskell:

```
(|.|) = (\f g x -> f x (g x))
          ( (\x y -> x) (\f g x -> f x (g x)) ) (\x y -> x)
> :t (|.|)
(|.|) :: (a -> b) -> (c -> a) -> c -> b
> (p |.| c) "abcd"
True
```

Reglas de sustitución.

s1) $[\mathbf{e}/\mathbf{x}]\,\mathbf{x} = \mathbf{e}$
$[\mathbf{e}/\mathbf{x}]\,\mathbf{y} = \mathbf{y} \quad \text{si } \mathbf{x} \neq \mathbf{y}$

s2) $[\mathbf{e}_1/\mathbf{x}]\,(\lambda\mathbf{x}.\mathbf{e}_2) = (\lambda\mathbf{x}.\mathbf{e}_2)$
$[\mathbf{e}_1/\mathbf{x}]\,(\lambda\mathbf{y}.\mathbf{e}_2) = \lambda\mathbf{y}.[\mathbf{e}_1/\mathbf{x}]\,\mathbf{e}_2$ si $\mathbf{x} \neq \mathbf{y}$ y $\mathbf{y}$ no es libre en $\mathbf{e}_1$
$[\mathbf{e}_1/\mathbf{x}]\,(\lambda\mathbf{y}.\mathbf{e}_2) = \lambda\mathbf{z}.[\mathbf{e}_1/\mathbf{x}]\,([\mathbf{z}/\mathbf{y}]\,\mathbf{e}_2)$ si $\mathbf{x} \neq \mathbf{z} \neq \mathbf{y}$ y $\mathbf{z}$ no es libre en $\mathbf{e}_1$ ni en $\mathbf{e}_2$

s3) $[\mathbf{e}_1/\mathbf{x}]\,(\mathbf{e}_2\,\mathbf{e}_3) = ([\mathbf{e}_1/\mathbf{x}]\,\mathbf{e}_2)\,([\mathbf{e}_1/\mathbf{x}]\,\mathbf{e}_3)$

Reglas de reescritura (y reducción).

r1) Conversión–α: $(\lambda\mathbf{x}.\mathbf{e}) \leftrightarrow (\lambda\mathbf{y}.[\mathbf{y}/\mathbf{x}]\mathbf{e})$ si $\mathbf{y}$ no es libre en $\mathbf{e}$.

r2) Conversión–β: $(\lambda\mathbf{x}.\mathbf{e}_1)\,\mathbf{e}_2 \leftrightarrow [\mathbf{e}_2/\mathbf{x}]\,\mathbf{e}_1$.

r3) Conversión–η: $\lambda\mathbf{x}.(\mathbf{e}\,\mathbf{x}) \leftrightarrow \mathbf{e}$ si $\mathbf{x}$ no es libre en $\mathbf{e}$. (La función de $\mathbf{x}$ que al aplicarse a $\mathbf{x}$ da $(\mathbf{e}\,\mathbf{x})$ surte el mismo efecto que $\mathbf{e}$.)

En estas reglas se dice que la exp–λ del lado izquierdo se *reduce* a la del lado derecho. Una *redex* es una exp–λ de la forma $((\lambda\mathbf{x}.\mathbf{e}_1)\,\mathbf{e}_2)$. Se dice que una exp–$\lambda$ está en *forma normal* si no contiene subexpresiones redex. No toda exp–λ es reducible a forma normal, por ejemplo: $(\lambda x.xx)(\lambda x.xx)$. Está demostrado que si una exp–λ tiene forma normal esta es única (salvo conversión–α).

La *evaluación* de una exp–λ consiste en obtener su forma normal (el *valor* de la exp–λ). Si el proceso de evaluación no termina —sea porque la exp–λ no tiene forma normal o debido al orden en que se aplican las reducciones— el valor se considera *indefinido* ($\perp$). Consideremos la evaluación de una aplicación $(\mathbf{F}\,\mathbf{X})$; el orden *aplicativo* consiste en evaluar el "argumento" $\mathbf{X}$ antes de ("llamar" a) la función $\mathbf{F}$; en el orden *normal* $\mathbf{F}$ se reduce ("llama") antes de evaluar $\mathbf{X}$. Veamos algunos ejemplos de evaluaciones:

$$(\lambda x.x*x)\,(3+2) \stackrel{\beta}{\Rightarrow} (\lambda x.x*x)\,5 \stackrel{\beta}{\Rightarrow} 5*5 \stackrel{\beta}{\Rightarrow} 25$$
$$(\lambda x.x*x)\,(3+2) \stackrel{\beta}{\Rightarrow} (3+2)*(3+2) \stackrel{\beta}{\Rightarrow} 5*(3+2) \stackrel{\beta}{\Rightarrow} 5*5 \stackrel{\beta}{\Rightarrow} 25$$
$$(\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \stackrel{\beta}{\Rightarrow} 5$$
$$(\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \stackrel{\beta}{\Rightarrow} (\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \stackrel{\beta}{\Rightarrow}$$
$$(\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \stackrel{\beta}{\Rightarrow} (\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \stackrel{\beta}{\Rightarrow} \cdots$$

Lenguajes representativos de programación funcional. LISP

```
==> (quote a)
a
==> 'a
a
==> (car '(a b c))
a
==> (cdr '(a b c))
(b c)
==> (cdr '(a))
nil
==> (cons 'a '(b c))
(a b c)
==> (cons '(a b c) '(d e f))
((a b c) d e f)
==> (atom 'a)
t
```

```
==> (atom '(a))
nil
==> (equal (car '(a b)) 'a)
t
==> (equal 'A 'a)
nil
==> (cond ( (atom 'a) 'b)
        ( t         'c) )
b
==> ((lambda (a b c) (+ a b c))
         1 (* 2 3) (- 5 4) )
8
==> (defun f (a b c) (+ a b c))
f
==> (f 1 (* 2 3) (- 5 4))
8
```

```
==> (defun subst (E1 x E2)
    (cond ((atom E2)(cond ((equal E2 x) E1)
                          ( t           E2)))
          ( t (cons (subst E1 x (car E2))
                    (subst E1 x (cdr E2)))) ))
subst
==> ( subst  '((P Q) R)  'b  '(a b (c b) ((a ((b)) c))) )
( a  ((P Q) R)  (c ((P Q) R))  ((a ((((P Q) R))) c)) )
==> (subst '(+ x y) 'v '(* x v))
(* x (+ x y))
==> (list 'lambda '(x y) (subst '(+ x y) 'v '(* x v)))
(lambda (x y) (* x (+ x y)))
==> (apply (list 'lambda '(x y) (subst '(+ x y) 'v '(* x v)))
         '(2 3))
10
==> ( mapcar 'atom '(a b (c b) c (a)) )
(t t nil t nil)
```

Lenguajes representativos de programación funcional. APL

<table>
<tr><th>En APL</th><th>En Haskell</th></tr>
<tr><td colspan="2">Generador ι</td></tr>
<tr><td></td><td><code>iota n = [1..n]</code></td></tr>
<tr><td>ι5</td><td><code>> iota 5</code></td></tr>
<tr><td>1 2 3 4 5</td><td><code>[1, 2, 3, 4, 5]</code></td></tr>
<tr><td>ι4;ι5</td><td><code>> iota 4 ++ iota 5</code></td></tr>
<tr><td>1 2 3 4 1 2 3 4 5</td><td><code>[1, 2, 3, 4, 1, 2, 3, 4, 5]</code></td></tr>
<tr><td colspan="2">Operador de compresión /</td></tr>
<tr><td></td><td><code>comp m l = [e | (b,e)<-zip m l, b]</code></td></tr>
<tr><td>1 0 0 1 / ι4</td><td><code>[True,False,False,True] ‘comp‘ (iota 4)</code></td></tr>
<tr><td>1 4</td><td><code>[1, 4]</code></td></tr>
<tr><td colspan="2">Operador de reducción /</td></tr>
<tr><td>A$\leftarrow$2 2 ρ 1 2 3 4</td><td><code>(//) f x = map f x</code></td></tr>
<tr><td>+/A</td><td><code>> sum // [[1,2],[3,4]]</code></td></tr>
<tr><td>3 7</td><td><code>[3, 7]</code></td></tr>
<tr><td colspan="2">Producto interno .</td></tr>
<tr><td></td><td><code>(f >< g) ma mb =</code></td></tr>
<tr><td></td><td><code> [[row ‘gg‘ col | col<-mb’] | row<-ma]</code></td></tr>
<tr><td>A$\leftarrow$2 3 ρ 1 0 -1 3 2 1</td><td><code> where v ‘gg‘ w = foldl1 f (zipWith g v w)</code></td></tr>
<tr><td>B$\leftarrow$3 2 ρ 2 3 -1 2 5 4</td><td><code>   mb’   = transpose mb</code></td></tr>
<tr><td>A +.$\times$ B</td><td><code>> ((+) >< (*)) [[1,0,-1], [3,2,1]]</code></td></tr>
<tr><td></td><td><code>   [[2,3], [-1,2], [5,4]]</code></td></tr>
<tr><td></td><td><code>[[-3, -1], [9, 17]]</code></td></tr>
<tr><td></td><td><code>> ((++)><(:)) [[’A’,’B’,’C’],[’1’,’2’,’3’]]</code></td></tr>
<tr><td></td><td><code>   [["si","uu"],["esa","vvv"],["ar","w"]]</code></td></tr>
<tr><td></td><td><code>[["AsiBesaCar", "AuuBvvvCw"],</code></td></tr>
<tr><td></td><td><code> ["1si2esa3ar", "1uu2vvv3w"]]</code></td></tr>
<tr><td colspan="2">Producto externo $\circ$</td></tr>
<tr><td></td><td><code>o f as bs = [[f a b | b<-bs] | a<-as]</code></td></tr>
<tr><td>A $\circ$.$\times$ B</td><td><code>> o (*) [1,2,3] [5,6,7,8]</code></td></tr>
<tr><td></td><td><code>[[5,6,7,8], [10,12,14,16], [15,18,21,24]]</code></td></tr>
<tr><td colspan="2">Números primos manores o iguales que N</td></tr>
<tr><td>(2=(+/0=ιN$\circ$.|ιN))/ιN</td><td></td></tr>
</table>

Ejemplo de uso de estructuras, signaturas y functores en SML

```
signature MONOID =
  sig
    type set;
    val one: set;
    val ope: set*set -> set
  end;
signature POWMON =
  sig
    type set;
    val one: set;
    val ope: set*set -> set;
    val pow: set*int -> set
  end;
```

```
functor
 Pow(Monoid: MONOID): POWMON =
  struct
    type set = Monoid.set;
    val one = Monoid.one;
    val ope = Monoid.ope;
    fun pow(a,n) =
      if n=0 then one
      else ope(pow(a,n-1),a)
  end;
```

```
structure RealAdd =
  struct
    type set = real;
    val one = 0.0;
    fun ope(r1:real, r2:real)
      = r1+r2
  end;
```

```
structure RealMul =
  struct
    type set = real;
    val one = 1.0;
    fun ope(r1:real, r2:real)
      = r1*r2
  end;
```

```
structure Fun =
  struct
    type set = int -> int;
    val one =
      fn(x) => x;
    fun ope(f,g) =
      fn(x) => f(g(x))
  end;
```

```
structure PowRA = Pow(RealAdd);
structure PowRM = Pow(RealMul);
structure PowF = Pow(Fun);
```

```
- PowRA.pow(17.1,0);
0.0 : PowRA.set
- PowRA.pow(2.0,10);
20.0 : PowRA.set
```

```
- PowRM.pow(17.1,0);
1.0 : PowRM.set
- PowRM.pow(2.0,10);
1024.0 : PowRM.set
```

```
- PowF.pow((fn(n)=>(~n)),7)(10);
~10 : int
- PowF.pow((fn(n)=>n*n),3)(2);
256 : int
```

Ejemplo de uso de clases y módulos en Haskell

```
module Monoides where
class Monoide a where
  e  :: a
  op :: a -> a -> a
instance Monoide [a] where
  e  = []
  op = (++)
instance Monoide Int where
 e  = 0
 op = (+)
```

```
class Monoide a =>
 MonoidePot a where
  pot :: a -> Int -> a
  pot x n =
    if n==0 then e
      else op (pot x (n-1)) x
instance MonoidePot [a]
instance MonoidePot Int
```

```
module Modulos where
import Monoides
instance Monoide Float where
  e  = 1
  op = (*)
instance MonoidePot Float
```

```
Modulos> pot [1,2] 3
[1, 2, 1, 2, 1, 2]
Modulos> pot [1,2] 0
[]
Modulos> pot [1,2] 5
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
Modulos> pot "ja-" 3
"ja-ja-ja-"
```

```
Modulos> pot 3 5
ERROR: Unresolved overloading
**type: (MonoidePot a,Num a)=>a
**expression: pot 3 5
Modulos> pot (3::Int) 5
15
Modulos> pot (3::Int) 0
0
Modulos> pot (3::Int) 1
3
```

```
Modulos> pot (2.2::Float) 3
10.648
Modulos> pot (pi::Float) 3
31.0063
Modulos> pot (pi::Float) 0
1.0
```

Ejemplo de uso módulos en Haskell

Parser para la gramática:

$$S \to zMNz$$
$$M \to aMa$$
$$M \to z$$
$$N \to bNb$$
$$N \to z$$

```
module Gram_2_5 where
import Arboles(RoseTree(Node),
               dibujaArbol)
import Parsers
gramatica =
  (noterms, terms, prods, Nont "S")
noterms =
  simb 'S' <|> simb 'M' <|> simb 'N'
terms =
  simb 'z' <|> simb 'a' <|> simb 'b'
prods = "S ::= z M N z   ." ++
        "M ::= z | a M a ." ++
        "N ::= z | b N b ."
simb c = greedy1 (satisfy (==c))
parsG = twopass (sp terms <@ Term)
                (parsgen gramatica)
```

```
> dibujaArbol (some parsG "z z z z")
--Nont "S"--Term "z"
         |
         |--Nont "M"--Term "z"
         |
         |--Nont "N"--Term "z"
         |
         `--Term "z"
```

```
arbPars s = dibujaArbol
              (simp (some parsG s))
simp (Node (Term s) []) = Node s []
simp (Node (Nont s) []) = Node s []
simp (Node (Term s) hs) =
            Node s (map simp hs)
simp (Node (Nont s) hs) =
            Node s (map simp hs)
```

```
> arbPars "z a z a b z b z"
--"S"--"z"
    |
    |--"M"--"a"
    |    |
    |    |--"M"--"z"
    |    |
    |    `--"a"
    |
    |--"N"--"b"
    |    |
    |    |--"N"--"z"
    |    |
    |    `--"b"
    |
    `--"z"
```

```
> arbPars "z a a z a a b z b z"
--"S"--"z"
    |
    |--"M"--"a"
    |    |
    |    |--"M"--"a"
    |    |    |
    |    |    |--"M"--"z"
    |    |    |
    |    |    `--"a"
    |    |
    |    `--"a"
    |
    |--"N"--"b"
    |    |
    |    |--"N"--"z"
    |    |
    |    `--"b"
    |
    `--"z"
```