Programación Declarativa

Curso 2004-2005

Departamento de Electrónica, Sistemas Informáticos y Automática
Universidad de Huelva

Tema 5: Introducción a la programación funcional.

Qué es la programación Funcional (I)

- La programación funcional apareció como un paradigma independiente a principio de los sesenta.
- Su creación es debida a las necesidades de los investigadores en el campo de la inteligencia artificial y en sus campos secundarios del cálculo simbólico, pruebas de teoremas, sistemas basados en reglas y procesamiento del lenguaje natural.
- Estas necesidades no estaban cubiertas por los lenguajes imperativos de la época.

Qué es la programación Funcional (II)

- La característica principal de la programación funcional es que los cálculos se ven como una función matemática que hacen corresponder entradas y salidas.
- No hay noción de posición de memoria y por tanto, necesidad de una instrucción de asignación.
- Los bucles se modelan a través de la recursividad ya que no hay manera de incrementar o disminuir el valor de una variable.
- Como aspecto práctico casi todos los lenguajes funcionales soportan el concepto de variable, asignación y bucle.
- Estos elementos no forman parte del modelo funcional "puro".

Qué es la programación Funcional (III)

Cualquiera que haya programado una hoja de cálculo conoce la experiencia de la programación funcional.

- En una hoja de cálculo, se especifica cada celda en terminos de los valores de otras celdas. El objetivo es que debe ser calculado y no en como debe calcularse.

Por ejemplo:

- No especificamos el orden en el que las celdas serán calculadas, en cambio obtenemos el orden que garantiza que la hoja de cálculo puede calcular las celdas respetando las dependencias.
- No indicamos a la hoja de cálculo como manejar la memoria, en cambio esperamos que nos presente un plano de celdas, aparentemente infinito, pero que solo utiliza la memoria de las celdas que están actualmente en uso.
- Lo más importante, especificamos el valor de una celda por una expresión (cuyas partes pueden ser evaluadas en cualquier orden), en vez de una secuencia de comandos que calculan los valores.

¿Por qué Haskell?

- Haskell es un lenguaje funcional puro
- Haskell no es un lenguaje muy rápido, pero se prioriza el tiempo del programador sobre el tiempo de computación.

Introducción a Haskell

- Haskell es un lenguaje de programación con tipos polimórficos, con evaluación perezoso (lazy), y puramente funcional, muy diferente a otros lenguajes imperativos.
- El lenguaje toma el nombre de ***Haskell Brooks Curry*** (1900-1982 Millis, Massachusetts, USA), cuyos trabajos en lógica matemática sirvieron como base a los lenguajes funcionales.
- Haskell está basado en el cálculo *lambda*, de aquí el símbolo λ (lambda) que aparece en el logo.

  http://www.lfcia.org/openprojects/camllets/doc/html/node11.html
  http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Curry.html

Haskell Brooks Curry

Experto en lógica simbólica, trabajó en el primer ordenador electrónico llamado **ENIAC** (Electronic Numerical Integrator and Computer) durante Segunda Guerra Mundial. Trabajó en los 50 en los fundamentos de la lógica combinatoria y los aplicó en Mitre Corporation Curry Chip en 1986 un innovador elemento hardware basado en los conceptos de combinatoria de Curry.

Función matemática

En Matemáticas, una función f entre dos conjuntos A y B, llamados conjuntos inicial y final, es una correspondencia que a cada elemento de un subconjunto de A, llamado "Dominio de f", le hace corresponder uno y sólo uno de un subconjunto B llamado "Imagen de f".

f: A → B

f(x) → ....

Ejemplo:

sucesor: Z → Z

sucesor(x) → x + 1

pi: R

pi → 3.141592

Las constantes son funciones que no tienen parámetros y devuelven siempre lo mismo

Sesiones y declaraciones

Desde el punto de vista de la programación funcional, el ordenador actúa como una calculadora, o evaluador, que calcula el resultado de las expresiones que introducimos

• El programador dispone de un conjunto de valores, funciones y operadores predefinidos, cuyo significado lo conoce ya el evaluador

```
Prelude> 1 + 3

4 :: Integer
```

Escribiremos ":set +t" para que Haskell muestre el tipo de dato

Sesiones y declaraciones (II)

Las líneas anteriores representan “una sesión” o un diálogo con el evaluador.

Le pedimos a Haskell que calculase el valor de la expresión “1 + 3” y éste respondió “4 :: Integer”

También podemos definir funciones con una sintaxis muy cercana a la matemática, de ahí la facilidad a la hora de escribir programas, ya que serán meras implementaciones de las funciones matemáticas.

Tipos de datos

En Haskell se puede trabajar con diferentes tipos de datos. Cada valor tiene un tipo y hay tipos que están predefinidos, como los enteros, los reales, etc...

```
Prelude> cos(-2*pi)
1.0 :: Double
```

Existen otros tipos de datos más complejos

```
Prelude> [1..5]
[1,2,3,4,5] :: [Integer]
Prelude> sum [1..5]
15 :: Integer
```

Currificación

Las funciones con más de un argumento se pueden interpretar como funciones que toman un único argumento y devuelven como resultado otra función con un argumento menos.

Este mecanismo que permite representar funciones de varios argumentos mediante funciones de un argumento se denomina "CURRIFICACIÓN" (debido a Haskell B. Curry)

La mayoría de los lenguajes funcionales están basados en el

λ-cálculo

λ- Cálculo

Fue concebido originalmente por el lógico-matemático Alonzo Church en los años 30 como parte de una teoría general para modelar funciones y lógica de orden superior.

Haskell es un lenguaje descrito como un λ- Cálculo extendido

Definición: Dado un conjunto(infinito numerable) de variables V y un conjunto C de constantes, el conjunto Λ de los términos del λC queda definido por la sintaxis

$$\Lambda ::= V \mid C \mid (\lambda V.\Lambda) \mid (\Lambda\ \Lambda)$$

Donde λx.M representa la función x→M

Para la función

f: A → B

x → 2x + 1

Podemos escribirla como una λ-expresión de la forma

λx.2x+1

Ejemplo:

f(x,y) = (x + y)*2

λx → λy → (x + y) * 2

Se expresaría en λCalculo como

λx. λy. * (+ x y) 2

Particularidades:

- El ámbito de x en la expresión λx.E se extiende a la derecha tanto como sea posible, hasta el primer paréntesis no cerrado o hasta el final de la expresión.
- $\lambda x1x2...xn.M \equiv \lambda x1.(\lambda x2.(... (\lambda xn.M)...))$

  Esto es muy importante, ya que nos permite descomponer cualquier función multi-variable como composición de funciones de una única variable
- La aplicación es asociativa por la izquierda

  $M\ N1\ N2....Nn \equiv (...((M\ N1)\ N2)...\ Nn))$

Declaración de una función ejemplo

Vamos a definir nuestra primera función en Haskell, que calcule

El sucesor de un numero entero

```
sucesor :: Integer → Integer

sucesor  x  =  x + 1
```

"sucesor" es el nombre de la función

- La primera línea es una declaración de tipos e indica que sucesor es una función de enteros en enteros
- La segunda línea es una ecuación, y proporciona la forma de cálculo del valor de esa función.

Declaración de una función ejemplo (II)

Ej:

```
Main> sucesor 3
4 :: Integer
Main> sucesor 3 * 4
16 :: Integer
Main> sucesor (3 * 4)
13 :: Integer
```

Siempre que no se especifiquen los paréntesis, se supondrá que la expresión viene en forma currificada

Reducción de Expresiones (I)

La labor de un evaluador es calcular el resultado que se obtiene al simplificar una expresión utilizando las definiciones de las funciones involucradas.

Ej: doble :: Integer → Integer

doble x = x + x

5 * doble 3

→ { por la definición de 'doble' }

5 * (3 + 3)

→ { por el operador + }

5 * 6

→ { por el operador * }

30

Reducción de Expresiones (II)

Una expresión se reduce sustituyendo, en la parte derecha de la ecuación de la función, los ***Parámetros Formales*** o argumentos por los que aparecen en la llamada(también llamados ***Parámetros Reales o Parámetros***).

Cuando una expresión no pueda reducirse más, se dice que está en ***Forma Normal***

Es importante el orden en el que se aplican las reducciones, y dos de los mas interesantes son:

Aplicativo y Normal

Orden Aplicativo

Se reduce siempre el término ***MAS INTERNO*** (el más anidado en la expresión). En caso de que existan varios términos a reducir (con la misma profundidad) se selecciona el que aparece más a la izquierda de la expresión.

Esto también se llama "*paso de parámetros por valor*", ya que ante una aplicación de una función, se reducen primero los parámetros de la función.

A los evaluadores que utilizan este tipo de orden, se les llama "***estrictos o impacientes***"

Orden Normal

Consiste en seleccionar el término ***MÁS EXTERNO*** (el menos anidado), y en caso de conflicto el que aparezca más a la izquierda de la expresión.

Esta estrategia se conoce como "*paso de parámetro por nombre o referencia"*, ya que se pasan como parámetros de las funciones expresiones en vez de valores.

A los evaluadores que utilizan el orden normal se les llama ***"no estrictos"***.

Una de las características más interesantes es que este orden *es normalizante.*

Evaluación PEREZOSA o LENTA (Lazy) (I)

No se evalúa ningún elemento en ninguna función hasta que no sea necesario

Las listas se almacenan internamente en un formato no evaluado

La evaluación perezosa consiste en utilizar paso por nombre y recordar los valores de los argumentos ya calculados para evitar recalcularlos.

También se denomina *estrategia de pasos de parámetros por necesidad.*

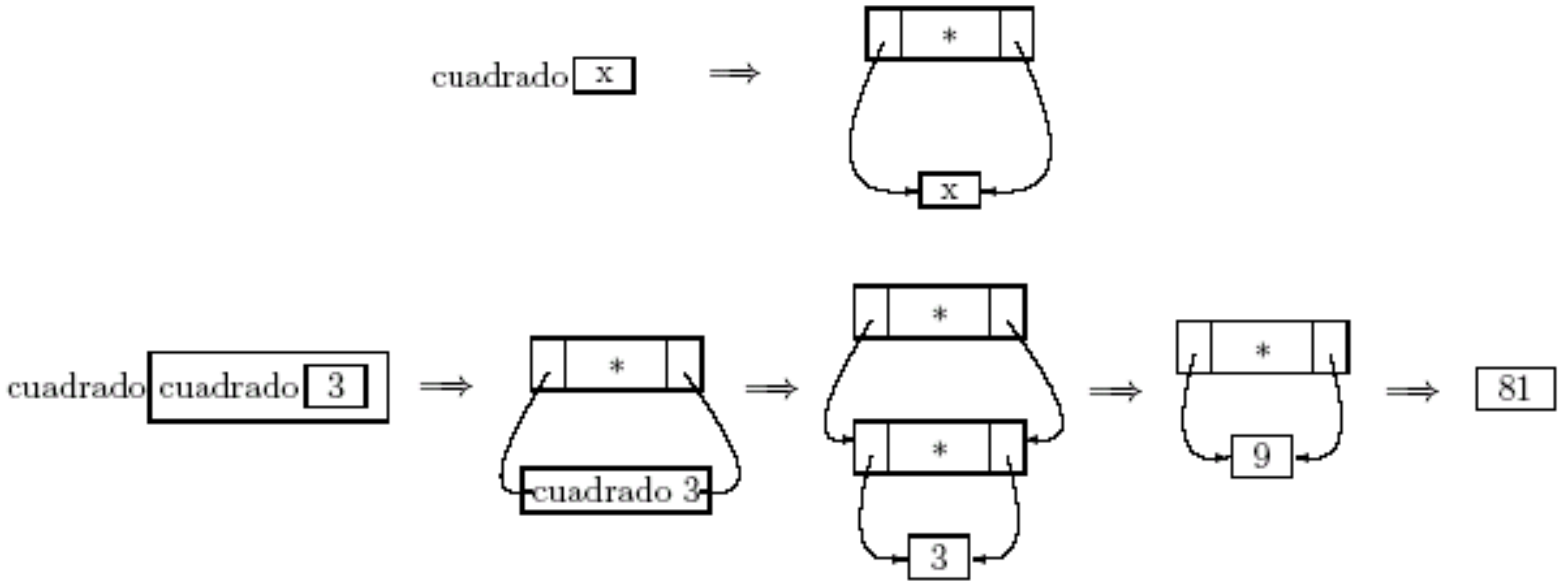
Con una estrategia no estricta de la expresión doble (doble 3), la expresión (3 + 3) se calcula dos veces.

Evaluación PEREZOSA o LENTA (Lazy) (II)

Definimos la función cuadrado:

cuadrado :: Integer → Integer

cuadrado x = x * x



Tipos de datos elementales

- Booleanos (Bool, valores True y False)
- Enteros (Int e Integer)
- Caracteres (Char)
  - Entre comillas simples (Ej: ‘a’ ‘b’)
- Cadenas de caracteres (String)
  - Entre comillas doble “hola”
  - Equivale a una lista de Char
    - [‘h’,’o’,’l’,’a’] equivale a “hola”
  - Es posible utilizar operadores de listas para el tipo String como ++ “hola “ ++ “ a todos”

Definición de funciones (I)

- Las funciones en Haskell se definen en dos partes.
  - La primera identifica el nombre, el dominio y el intervalo de la función.
  - La segunda describe el significado de la función

```
max3 :: Int -> Int-> Int -> Int
max3 x y z
   | x >= y && x>=z    =x
   | y >= x && y>=z    =y
   | otherwise         =z
```

Definición de funciones (II). Ejemplos

```
Main> max3 4 7 5
7
```

```
Main> max3 'a' 'b' 'c'
ERROR - Type error in application
*** Expression     : max3 'a' 'b' 'c'
*** Term           : 'c'
*** Type           : Char
*** Does not match : Int
```

```
Main> max3 1 'a' 3
ERROR - Type error in application
*** Expression     : max 1 'a' 3
*** Term           : 'a'
*** Type           : Char
*** Does not match : a -> b
```

Definición de funciones (III)

- Si omitimos la primera línea, Haskell deriva la primera línea dándole en sentido más amplio posible.

```
max3 x y z
   | x >= y && x>=z    =x
   | y >= x && y>=z    =y
   | otherwise         =z
```

- Esta función max3 es un ejemplo de función polimórfica.
- Una función polimórfica es aquella que se aplica bien a argumentos de varios tipos.

Definición de funciones (IV). Ejemplos

```
Main> max3 'a' 'b' 'c'
'c'
```

```
Main> max3 4 7 5
7
```

```
Main> max3 [1,2,3] [3,4,5] [1,1,1]
[3,4,5]
```

```
Main> max3 1 'a' 3
ERROR - Cannot infer instance
*** Instance   : Num Char
*** Expression : max3 1 'a' 3
```

Definición de funciones (IV). Recursividad

- Podemos definir funciones en Hasekell utilizando recursividad

Factorial

```
fact :: Int -> Int
fact n
  | n == 0   = 1
  | n > 0    = n* fact(n-1)
```

Suma lista

```
suma_lista[]              = 0
suma_lista(cabeza:resto) = cabeza + suma_lista(resto)
```

Listas (I)

- La estructura de datos fundamental en Haskell son las listas.
- Podemos definirlas enumerando sus elementos o utilizando el símbolo especial “..”
- Ejemplo:

  Las definiciones [1,2,3,4,5] y [1.. 5] son equivalentes
- <- representa al símbolo matemático $\in$, que indica pertenencia a un conjunto

Listas (II). Programa

Fichero “dobla_lista.hs”

```
dobla_lista = [2*x |x <- [0 .. 10]]
```

• El programa dice literalmente “la lista de todos los valores 2*x, siendo n un elemento de la lista [0..10].

Haskell

```
Prelude> :l "D:\\Haskell\\ejercicios\\dobla_lista.hs"
Main> print dobla_lista
[0,2,4,6,8,10,12,14,16,18,20]
```

Listas (III). Infinitas

- Podemos definir una lista infinita de la siguiente forma

```
[1..]
```

- Esta es una gran diferencia entre Haskell y otros lenguajes funcionales.
- Las listas infinitas y las funciones que calculan valores a partir de ellas son comunes en Haskell.
- Esto es posible gracias a la evaluación lenta o perezosa (lazy) de Haskell. No se evalúa ningún elemento en ninguna función hasta que no sea necesario
- Las listas se almacenan internamente en un formato no evaluado

Listas (IV). Tabla de operadores

Función	**Dominio**	**Explicación**
:	a->[a]->[a]	Añade un elemento al principio de la lista
++	[a]->[a]->[a]	Concatena dos lista
!!	[a]->Int->a	x!!n devuelve el enésimo elemento de la lista x
length	[a]->Int	Devuelve el número de elementos de una lista
head	[a] -> a	Primer elemento de una lista
tail	[a] -> a	Resto de una lista
take	Int -> [a]->[a]	Toma n elementos del principio de una lista
drop	Int -> [a] ->[a]	Saca n elementos del principio de una lista
reverse	[a] -> [a]	Invierte los elementos de una lista
zip	[a] -> [b] -> [(a,b)]	Crea una lista de pares
unzip	[(a,b]-> ([a], [b])	Crea un par de lista
sum	[Int] -> Int [Float] -> Float	Suma los elementos de una lista
product		Multiplica los elementos de una lista

Listas (V). Ejemplos operadores

```
8:[]                  -- resultado [8]
6:8:[]                -- resultado [6,8]
4:[6,8]               -- resultado [4,6,8]
head [1,2,3,4]        -- resultado 1
tail [1,2,3,4]        -- resultado [2,3,4]
head [[1],2,3,4]      -- ERROR - Cannot infer instance
[1,2] ++ [3,4]        -- resultado [1,2,3,4]
[1,2] ++ 3:[4]        -- resultado [1,2,3,4]
null []               -- resultado True
[1,2] == [1,2]
[1,2] == 1:[2]        -- resultado True
```

Listas (VI). Ejemplo pertenece

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece elemento lista
  |lista == []   = False
  |elemento == head lista = True
  |otherwise  = pertenece elemento (tail lista)
```

```
Main> :l"D:\\Haskell\\ejercicios\\miembro_listas.hs"
Main> pertenece 5 [1,2,3]
False
Main> pertenece 3 [1,2,3]
True
```

Flujo de control

- Las construcciones de flujo de control en Haskell son *guarded* e *if..then..else*

Con el comando *guarded:*

```
| x >= y && x>=z    =x
| y >= x && y>=z    =y
| otherwise         =z
```

Con el comando *if..then..else:*

```
if x >= y && x>=z then x
else if  y >= x && y>=z  then y
else z
```

Tuplas

- Una tupla en Haskell es un grupo de valores de distintos tipos encerrados entre paréntesis y separados por comas.

  (“Pedro”, 22, “636000111”)

Tuplas (II). Ejemplo

Ejemplo

```
type Entrada =  (Persona, Edad, Telefono)

type Persona = String

type Edad = Integer

type Telefono = String

type Listin = [Entrada]

encontrar :: Listin -> Persona -> [Telefono]

encontrar listin p = [telefono | (persona, edad, telefono) <- listin,
persona == p]

listin = [("Pedro", 22, "636777111"), ("Luis", 19, "647123123"),
("Alfonso", 22, "646099333")]
```

```
Main> encontrar listin "Alfonso"

["646099333"]

Main>
```

Hay dos modos de incluir comentarios en un programa:

Comentarios de una sola línea: comienzan por dos guiones consecutivos (--) y abarcan hasta el final de la línea actual:

```
f :: Integer → Integer
f x = x + 1 -- Esto es un comentario
```

Comentarios que abarcan más de una línea:

Comienzan por los caracteres {- y acaban con -} Pueden abarcar varias líneas y anidarse:

```
{- Esto es un comentario
de más de una línea -}
g :: Integer → Integer
g x = x - 1
```

Comparación de Patrones

Un patrón es una expresión como argumento en una ecuación.

Es posible definir una función dando más de una ecuación para ésta.

Al aplicar la función a un parámetro concreto la comparación de patrones determina la ecuación a utilizar.

Regla para la comparación de patrones

- Se comprueban los patrones correspondientes a las distintas ecuaciones en el orden dado por el programa, hasta que se encuentre una que unifique.
- 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.

Patrones constantes

Un patrón constante puede ser un número, un carácter o un constructor de dato.

```
f :: Integer → Bool

f  1 = True

f  2 = False
```

La definición de la conjunción y disyunción de valores lógicos usa patrones constantes (True y False son dos constructores de datos para el tipo Bool)

Patrones para listas

Es posible utilizar patrones al definir funciones que trabajen con listas.

[ ]	sólo unifica con un argumento que sea una lista vacía.
[x], [x , y], etc.	sólo unifican con listas de uno, dos, etc. argumentos.
(x : xs)	unifica con listas con al menos un elemento

```
suma :: [Integer] → Integer

suma [ ] = 0

suma (x : xs) = x + suma xs
```

Patrones para tuplas

```
primero2 :: (Integer, Integer) → Integer
primero2 (x , y) = x
primero3 :: (Integer, Integer, Integer) → Integer
primero3 (x , y, z ) = x
```

Los patrones pueden anidarse.

```
sumaPares :: [(Integer, Integer)] → Integer
sumaPares [ ] = 0
sumaPares ((x , y) : xs) = x + y + sumaPares xs
```

Patrones aritméticos

Es un patrón de la forma (n + k), donde k es un valor constante natural.

```
factorial :: Integer → Integer
factorial 0 = 1
factorial (n + 1) = (n + 1) * factorial n
```

Patrones nombrados o seudónimos

Seudónimo o patrón alias para nombrar un patrón, y utilizar el seudónimo en vez del patrón en la parte derecha de la definición.

```
factorialn :: Integer → Integer

factorialn 0 = 1

factorialn m@(n + 1) = m * factorialn n
```

El patrón subrayado

Un patrón subrayado (_) unifica con cualquier argumento pero no establece ninguna ligadura.

```
longitud :: [Integer] → Integer

longitud [ ] = 0

longitud ( _ : xs) = 1 + longitud xs
```

Patrones y evaluación perezosa

La unificación determina qué ecuación es seleccionada para reducir una expresión a partir de la forma de los argumentos.

esVacia :: [a] → Bool

esVacia [ ] = True

esVacia (_ : _) = False

Para poder reducir una expresión como "esVacia ls" es necesario saber si ls es una lista vacía o no.

Patrones y evaluación perezosa

Definiendo “infinita”

```
infinita :: [Integer]

infinita = 1 : infinita
```

Haskell evalúa un argumento hasta obtener un número de constructores suficiente para resolver el patrón, sin obtener necesariamente su forma normal.

esVacia infinita

=> ! definición de infinita

esVacia (1 : infinita)

=> ! segunda ecuación de esVacia

False

Expresiones case

La sintaxis de esta construcción es:

```
case expr of
        patron1 → resultado1
        patron2 → resultado2
        ....
        patronn → resultadon
```

```
long :: [Integer] → Integer
        long ls = case ls of
        [ ] → 0
        _ : xs → 1 + long xs
```

La función error

Si intentamos evaluar una función parcial en un punto en el cual no está definida, se produce un error.

Con la función error podemos producir producir nuestro mensaje de error.

```
cabeza0 :: [Integer] → Integer

cabeza0 [ ] = error "lista vacía"

cabeza0 (x : ) = x


Main> cabeza0 [1, 2, 3]

1 :: Integer

Main> cabeza0 [ ]

Program error : lista vacía
```

Funciones a trozos

$$\text{absoluto} :: Z \rightarrow Z$$

$$\text{absoluto}(x) = \begin{cases} x \text{ si } x >= 0 \\ -x \text{ si } x < 0 \end{cases}$$

En Haskell la definición anterior puede escribirse del siguiente modo:

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

Es posible utilizar como guarda la palabra otherwise, que es equivalente al valor True.

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

Expresiones condicionales

Otro modo de escribir expresiones cuyo resultado dependa de cierta condición es utilizando expresiones condicionales.

```
if exprBool then exprSi else exprNo
```

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

```
Prelude> if 5 > 2 then 10.0 else (10.0/0.0)
10.0 :: Double
Prelude> 2 * if 'a' < 'z' then 10 else 4
20 :: Integer
```

Definiciones locales

A menudo es conveniente dar un nombre a una subexpresión que se usa varias veces.

```
raices :: Float → Float → Float → (Float, Float)
raices a b c
        | b ^ 2 - 4 * a * c >= 0    = ( (-b + sqrt(b ^ 2 - 4 * a * c))/(2 * a),
                                       (-b - sqrt(b ^ 2 - 4 * a * c))/(2 * a) )
        | otherwise = error "raíces complejas"
```

Podemos mejorar la legibilidad de la definición anterior definiendo variables locales.

En Haskell podemos usar para este propósito la palabra where.

```
raices :: Float → Float → Float → (Float , Float)
raices a b c
| disc >= 0 = ((-b + raizDisc)/denom; (-b - raizDisc)/denom)
| otherwise = error "raíces complejas"
        where
                disc =b ^ 2 - 4 * a * c
                raízDisc = sqrt disc
                denom =2 * a
```

Las definiciones locales “where” sólo pueden aparecer al final de una definición de función.

Para introducir definiciones locales en cualquier parte de una expresión se usa let e in.