Introducción a Haskell

Introducción

- Programación funcional es un paradigma de programación;
- En el paradigma imperativo, un programa es una secuencia de instrucciones que cambian celdas en la memoria;
- En el paradigma funcional, un programa es un conjunto de definiciones de funciones que aplicamos a valores;
- En los lenguajes funcionales las funciones son entidades de 1ra clase, es decir, pueden ser usadas como cualquier otro objeto: pasadas como parámetro, devueltas como resultado o incluso almacenadas en estructuras de datos

Introducción

- Al elevar las funciones de nivel, los lenguajes ganan una gran flexibilidad, capacidad de abstracción y modularización en el procesamiento de datos;

- Los lenguajes funcionales proveen un alto nivel de abstracción, lo que hace que los programas sean más concisos, más fáciles de entender y más rápidos de desarrollas que los imperativos.

- Podemos programar en un estilo funcional en muchos lenguajes Ejemplos: Scheme, ML, O'Caml, Haskell, F#, Scala.

Introducción

Quicksort en **Java**

```
public class Quicksort {
  public static void qsort(double[] a) {
    qsort(a, 0, a.length - 1);
  }
  public static void qsort(double[] a,
                           int left, int right) {
      if (right <= left) return;
      int i = partition(a, left, right);
      qsort(a, left, i-1);
      qsort(a, i+1, right);
  }
  private void swap(double[] a, int i, int j) {
    double tmp = a[i]; a[j] = a[j]; a[i] = tmp;
  }
```

```
  private static int partition(double[] a,
                               int left, int right) {
      int i = left;
      int j;
      for(j=left+1; j<=right; ++j) {
          if(a[j] < a[left]) {
            ++i;
            swap(a, i, j);
          }
        }
      swap(a, i, left);
      return i;
  }
}
```

Quicksort en **Haskell**

```
qsort [] = []
qsort (x:xs) = qsort xs1 ++ [x] ++ qsort xs2
      where xs1 = [x' | x'<-xs, x'<=x]
            xs2 = [x' | x'<-xs, x'>x]
```

Introducción

- Ventajas
  - Programas más concisos;
  - Más cercano de una especificación matemática;
  - Excelente modularidad (polimorfismo, orden superior, *lazy evaluation*);
  - Prácticamente todo componente es reutilizable (función);
  - Demostraciones de corrección usando pruebas matemáticas;
  - El orden de ejecución no afecta los resultados.

Introducción

- Desventajas
  - Compiladores/interpretadores más complejos;
  - Difícil prever los costos de ejecución (tiempo/espacio);
  - Algunos algoritmos son más eficientes cuando implementados de forma imperativa.

Introducción (Historia)

- 1930s Alonzo Church desenvuelve el cálculo-λ, un formalismo matemático para expresar computación usando funciones;
- 1950s Inspirado en el cálculo-λ, John McCarthy desenvuelve LISP, una de los primeros lenguajes de programación (no era tipado y puro);
- 1970s Robin Milner desenvuelve ML, el primer lenguaje funcional con polimorfismo e inferencia de tipos;
- 1970s–1980s David Turner desenvuelve varios lenguajes que emplean *lazy evaluation*, culminando en el lenguaje comercial *MirandaTM*

Introducción (Historia)

- 1987 Un comité académico inicia el desenvolvimiento de *Haskell*, un lenguaje funcional estandarizado y abierto;

- 2003 Publicación de *Haskell* 98, una definición estandarizado del lenguaje;

- 2010 Publicación del standard del lenguaje *Haskell* 2010.

Haskell

- Un lenguaje funcional puro de uso genérico;
- Nombrada en homenaje al matemático americano Haskell B. Curry (1900–1982);
- Concebida para enseñanza y también para el desarrollo de aplicaciones reales;
- Resultado de más de veinte años de investigación por una comunidad de base académica muy activa;
- Implementaciones abiertas y libremente disponibles:

http://www.haskell.org

Haskell (Aplicación)

Utilizaciones en *backend* de aplicaciones *web*:

- Bump mover ficheros entre *smartphones*
  http://devblog.bu.mp/haskell-at-bump
- Janrain plataforma de *user management*
  http://janrain.com/blog/
- Chordify extracción de acordes musicales http://chordify.net

Más ejemplos:

http://www.haskell.org/haskellwiki/Haskell_in_industry

Haskell (Compiladores)

Hugs

- Un interpretador interactivo de Haskell;
- Suporta Haskell 98 y bastantes extensiones;
- Para aprendizaje y desenvolvimiento de pequeños programas;
  Disponible en http://www.haskell.org/hugs

Haskell (Compiladores)

Glasgow Haskell Compiler (GHC)

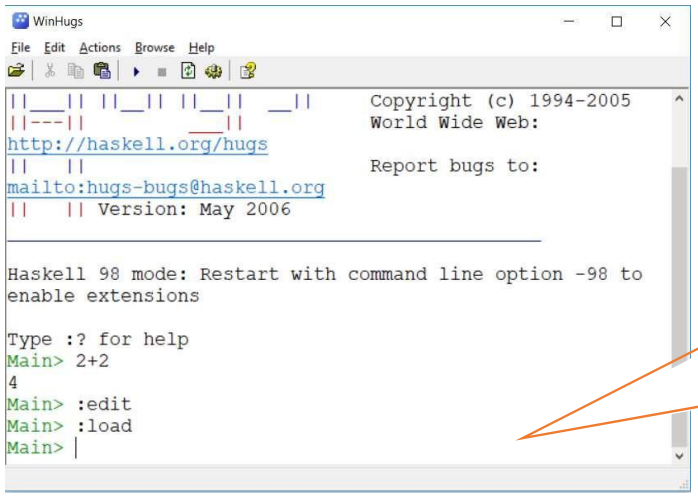
- Compilador que genera código-máquina nativo;
- Suporta Haskell 98, Haskell 2010 y bastantes extensiones;
- Optimización de código, interfaces a otros lenguajes, *profilling*, gran conjunto de bibliotecas, etc;
- Incluye también el interpretador ghci (alternativo a Hugs)
  Disponible en http://www.haskell.org/ghc

Haskell (Compiladores)

Linux/Mac OS: ejecutar hugs o ghci

```
$ ghci
GHCi, version 6.8.3: http://www.haskell.org/ghc/
Loading package base ... linking ... done.
Prelude>
```

WinHugs



Utilizando el interpretador:

> 2+3*5 17

> (2+3)*5 25

> sqrt (3^2 + 4^2) 5.0

Haskell (Compiladores)

Algunos comandos básicos

Comando	Significado
:load	Carga un archivo para ejecución
:reload	Recarga modificaciones
:edit	Edita el archivo actual
:type <expr>	Muestra el tipo de una expresión
:help	Obtiene ayuda
:quit	Termina la sesión
:browse <module>	Muestra todas funciones de un módulo

Todo archivo(script) en Haskell debe tener la extensión .hs

Haskell (operadores y funciones aritméticas)

Operador/función	Significado
+	Adición
-	Subtracción
*	Multiplicacción
/	División
^	Potencia (exponente entero)
div	Cociente (división entera)
mod	Resto (división entera)
sqrt	Raiz cuadrada
==	Igualdad
/=	Diferencia
<, >, <=, >=	Comparaciones
&&, II, not	Operadores lógicos

Haskell (convenciones sintácticas)

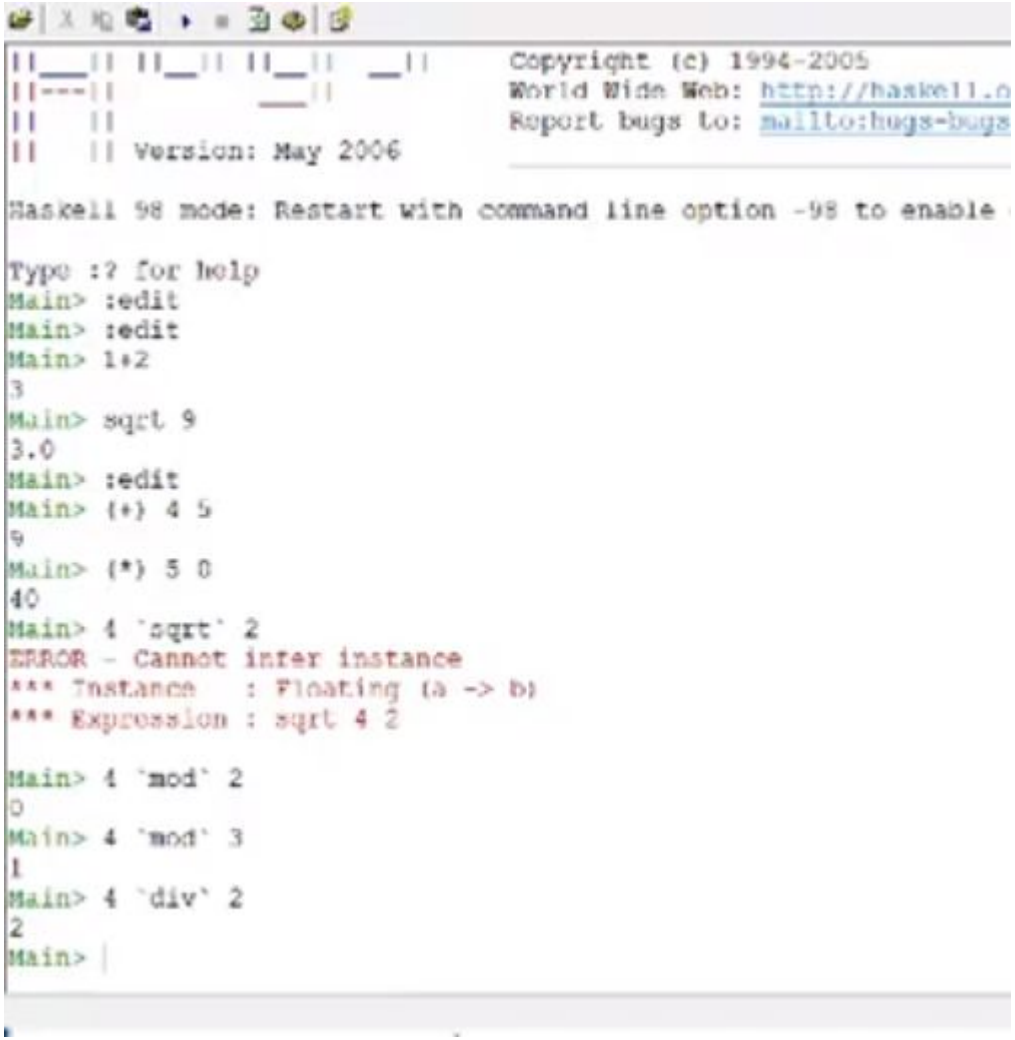
- Los argumentos de funciones son separados por espacios
- A función tiene mayor precedencia que cualquier operador

Haskell	Matemática
`f x`	$f(x)$
`f (g x)`	$f(g(x))$
`f (g x) (h x)`	$f(g(x), h(x))$
`f x y + 1`	$f(x, y) + 1$
`f x (y+1)`	$f(x, y + 1)$
`sqrt x + 1`	$\sqrt{x} + 1$
`sqrt (x + 1)`	$\sqrt{x + 1}$

Haskell (convenciones sintácticas)

- Un operador puede ser usado como una función escribiéndolo entre paréntesis;
- Recíprocamente: una función puede ser usada como operador escribiéndola entre crasas.

```
  (+) x  y   =  x +   y
  (*) 3  4   =  3 *   4
4 `mod`  2   =  mod   4 2
f x `div` n = div (f x) n
```



Haskell (convenciones sintácticas)

Identificadores

- Los nombres de funciones y argumentos deben comenzar por letras minúsculas y pueden incluir letras, dígitos, subguiones y apóstrofes:

```
fun1 x_2    y' fooBar
```

- Las siguientes palabras reservadas no puede ser usadas como identificadores:

```
case class data default deriving do else if import in
infix infixl infixr instance let module newtype of then
                        type where
```

Haskell (convenciones sintácticas)

Identación

- Todas las definiciones en un mismo ámbito deben comenzar en la misma columna:

```
a = 1

 b = 2

c = 3
```

ERRADO

```
  a = 1

b   = 2

  c = 3
```

ERRADO

```
a = 1

b = 2

c = 3
```

OK

- El orden de las definiciones no es relevante.

Haskell (convenciones sintácticas)

Comentarios

- Simples: comienzan por -- hasta el final de la linea
- Varias lineas: delimitados por {- y -}

```
-- función para sumar dos números
suma x y = x + y

{- función desactualizada
  calc x = x `mod` 2 -}
```

Haskell (Prelude)

El módulo *Prelude* contiene un gran conjunto de funciones pre-definidas:

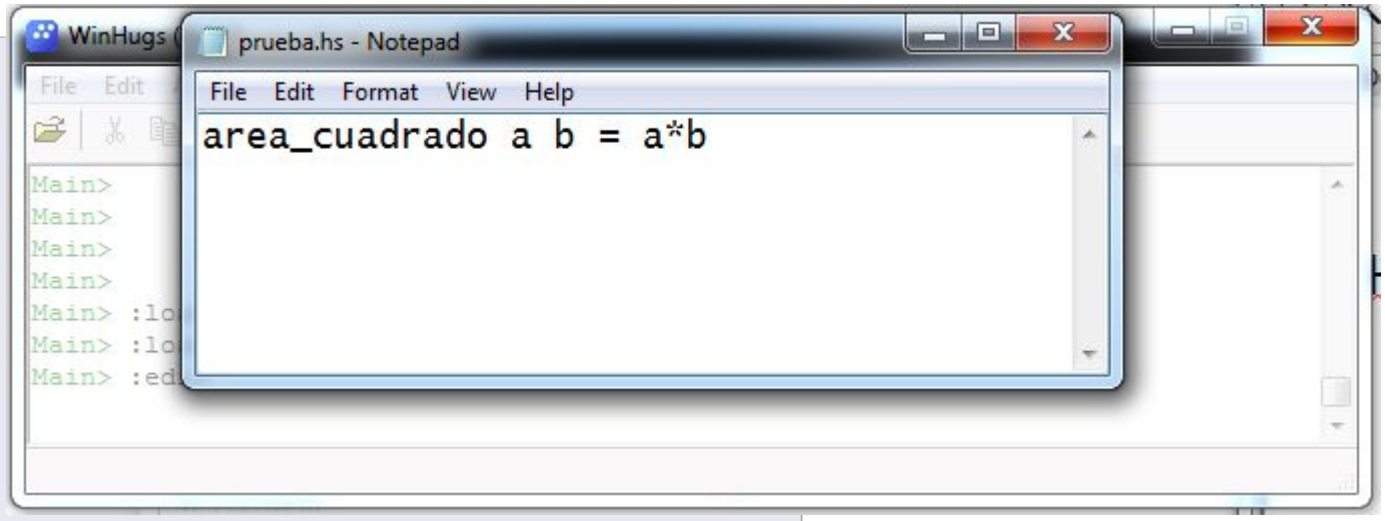
- operadores y funciones aritméticas;
- funciones genéricas sobre *listas y* muchas otras.

Prelude-standard es automáticamente cargado por el interpretador/compilador y puede ser usado en cualquier programa Haskell

```
>head [1,2,3,4] -- obtener el 1er elemento
>1
>tail [1,2,3,4] -- remover el 1er elemento
>[2,3,4]
```

Haskell (Definiendo funciones)

```
> :edit
```




```
> :load
> area_cuadrado 10 20
> 200
```

Haskell (Definiendo funciones)

```
funcionConstante = 12
volume_paralelepipedo b a l = b*a*l


suma x y = x + y


suma3 x y z =  suma (suma x y) z
media3 x y z = (suma3 x y z)/3
```

Ejercicio 01

```
 --1)  Haga una función  de la media de 4 números sin utilizar
 --     el operador más   (+), utilice apenas las funciones del slide anterior


--2) Haga una función para calcular la hipotenusa de un triángulo
--  rectángulo
```

Haskell (Definiciones locales)

Podemos hacer definiciones locales usando where a los operadores y funciones:

```
a = b+c
  where b = 1
        c = 2
d = a*2
```

La identación indica el ámbito de las declaraciones; también podemos usar agrupamiento explícito:

```
a = b+c
  where {b = 1;
         c = 2}
d = a*2
```

Ejercicio 02

```
--1) Haga una función sin parámetros de entrada y que defina tres
--  variables con valores fijos, la cual retorna “true” si los valores
--  son iguales.



--2) Haga una función para calcular el área de un triangulo con lados
--  “a”, “b” y “c” utilizando la formula de Heron, pero cree una
--  función local para calcular “S”.
```

$$A = \sqrt{s(s-a)(s-b)(s-c)} \qquad s = \frac{a+b+c}{2}$$

Haskell (Tipos de datos)

Un tipo es un nombre para una colección de valores relacionados

Escribimos `e :: T` para indicar que la expresión "*e*" admite el tipo *T*.

- Si *e* :: *T*, entonces el resultado de *e* será un valor de tipo *T*.
- El interpretador verifica tipos indicados por el programador e infiere tipos omitidos.
- Los programas con errores de tipos son rechazados antes de la ejecución.

Haskell (Tipos de datos)

Una función hace corresponder valores de un tipo en valores de otro tipo

```
volume_paralelepipedo :: Int -> Int -> Int
volume_paralelepipedo b a l = b*a*l

suma :: Float -> Float -> Float
suma x y = suma x y

media3 :: Int -> Int -> Int -> Float
media3 x y z = (x + y + z)/3


suma :: (Int,Int) -> Int
suma (x,y) = x+y
```

Haskell (Tipos de datos)

Tipo	Descripción
Bool	valores lógicos - True, False
Char	caracteres simples - 'A', 'B', '?', '\n'
String	secuencias de caracteres - "Abba", "UB40"
Int	enteros de precisión fjja (32 o 64-bits) ej: 142, -1233456
Integer	enteros de precisión arbitraria (limitados por la memoria del computador)
Float	Punto flotante de precisión simple ej: 3.14154, -1.23e10
Double	Punto flotante de precisión doble

Haskell (Tipos de datos)

Una lista es una secuencia de tamaño variable de elementos de un mismo tipo.

```
[False,True,False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
```

En general: [*T*] es el tipo de listas cuyos elementos son de tipo *T*

Haskell (Tipos de datos)

Una tupla es una secuencia de tamaño fjjo de elementos de tipos posiblemente diferentes.

```
(42,'a') :: (Int,Char)
(False,'b',True) :: (Bool,Char,Bool)
```

En general: (*T*1,*T*2,...,*Tn*) es el tipo de tuplas con *n* componentes de tipos *Ti* para *i* de 1 a *n*.

Haskell (Tipos de datos)

Listas de tamaños diferentes pueden ser del mismo tipo.

Tuplas de tamaños diferentes tienen tipos diferentes.

```
['a'] :: [Char]

['b','a','b'] :: [Char]

('a','b') :: (Char,Char)

('b','a','b') :: (Char,Char,Char)
```

Los elementos de listas y tuplas pueden ser cualesquier valores, inclusive otras listas y tuplas.

```
[['a'], ['b','c']] :: [[Char]]
(1,('a',2)) :: (Int,(Char,Int))
(1, ['a','b']) :: (Int,[Char])
```

Haskell (Tipos de datos)

Observaciones:

- La lista vacia [] admite cualquier tipo de lista [*T*]
- La tupla vacia () es el único valor del *tipo unitario* ()
- No existen tuplas con apenas un elemento

Haskell (Tipos de datos)

Funciones polimórficas:

- Ciertas funciones operan con valores de cualesquier tipos; tales funciones admiten tipos con variables.
- Una función se dice polimorfa (“de muchas formas”) si admite un tipo con variables

```
length :: [a] -> Int
```

La función *length* calcula el tamaño de una lista de valores de cualquier tipo *a*.

Haskell (Tipos de datos)

Funciones polimórficas

- Al aplicar funciones polimorfas, las variables de tipos son automáticamente sustituidas por los tipos concretos:

```
> length   [1,2,3,4]              -- Int
4

> length   [False,True]           -- Bool
2

> length   [(0,'X'),(1,'O')]      --(Int,Char)
2
```

Las variables de tipo deben comenzar por una letra minúscula; es convencional usar *a*, *b*, *c*, . . .

Haskell (Tipos de datos)

Funciones polimórficas

- Ciertas funciones operan sobre varios tipos pero no sobre cualesquier tipos:

```
> sum [1,2,3]
6
> sum [1.5, 0.5, 2.5]
4.5
> sum ['a', 'b', 'c']
ERROR
>sum [True, False]
ERROR
```

Haskell (Tipos de datos)

Funciones polimórficas

- En estos casos el tipo más general de la función tiene *restricciones de clase*:

```
sum :: Num a => [a] -> a
```

- "Num a => ..." es una restricción de clase de la variable *a*.
- Indica que sum opera apenas sobre tipos *a* que sean numéricos.

Haskell (Tipos de datos)

Funciones polimórficas

- Algunas clases son:

Clase	Descripción
Num	tipos *numéricos* (ej: Int, Integer, Float, Double)
Integral	tipos con *división entera* (ej: Int, Integer)
Fractional	tipos con *división fraccionaria* (ej: Float, Double)
Eq	tipos con *igualdad*
Ord	tipos con orden *total*

- Ejemplos:

```
(+) :: Num a => a -> a -> a
(/) :: Fractional a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
max :: Ord a => a -> a -> a
```

Haskell (Tipos de datos)

Funciones polimórficas

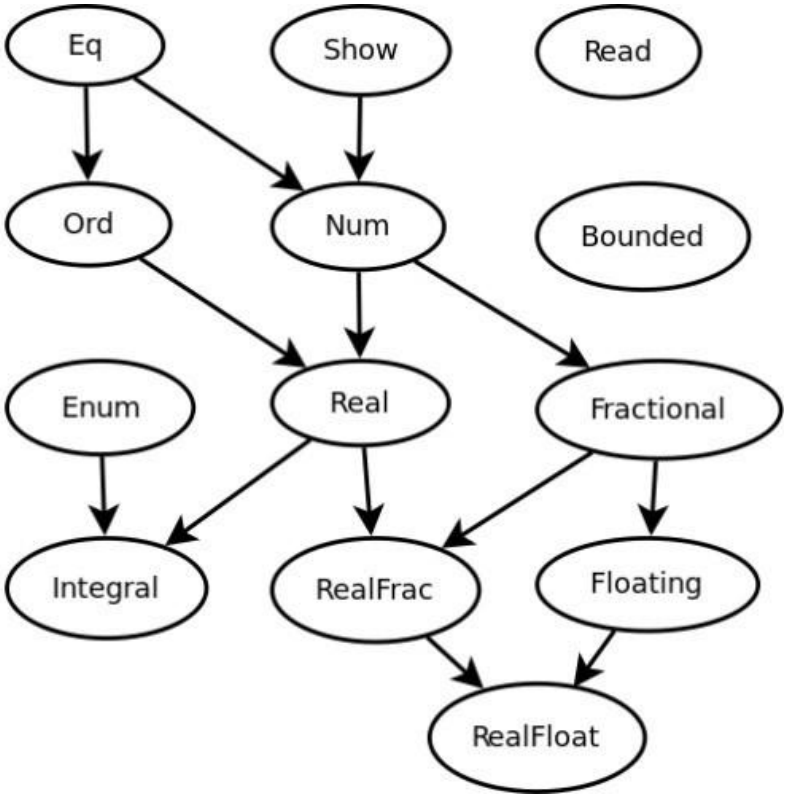
Algunas clases respetan una jerarquía:

- **Ord** es una subclase de **Eq**
- **Num** es una subclase de **Eq**
- **Fractional** e **Integral** son subclases de **Num**

Así, podemos usar:

== y **/=** con tipos en **Ord** o en **Num**

+, - y ***** con tipos en **Fractional** o en **Integral**



En algunos casos será necesario convertir elementos de una clase en otra, ejemplo: fromIntegral convierte cualquier tipo entero para cualquier otro tipo numérico.

Haskell (Expresiones condicionales)

Podemos expresar una condición con dos alternativas usando 'if . . . then . . . else . . . '.

```
abs :: Float -> Float
abs x = if x >= 0 then x else -x
```

Las expresiones condicionales pueden ser encadenadas:

```
signo :: Int -> Int
signo x = if x > 0 then 1
                   else if x == 0 then 0
                                  else -1
```

En Haskell, al contrario de C/C++/Java, la alternativa 'else' es obligatoria

Haskell (Expresiones condicionales)

Podemos usar guardas en vez de expresiones condicionales:

```
signo :: Int -> Int
signo x | x > 0     = 1
        | x == 0    = 0
        | otherwise = -1
```

- Testea las condiciones por el orden en el programa.
- Selecciona la primera alternativa verdadera.
- Si ninguna condición fuera verdadera: error de ejecución.
- La condición ‘otherwise’ es un sinónimo de True

Haskell (Expresiones condicionales)

Se puede usar múltiples ecuaciones para encontrar una solución:

```
or :: Bool -> Bool -> Bool
or False False = False
or True False = True
or False True = True
or True True = True
```

Se puede usar también correspondencia de patrones o variables anónimas para mejorar la solución arriba

```
or_v2 :: Bool -> Bool -> Bool
or_v2  False False = False
or_v2   _ _ = True
```

Ejercicios

1) Haga una función que calcule la distancia entre dos puntos

2) Haga una función para verificar si un año informado es bisiesto o no.

3) Defina una función que recibe tres números enteros representando, respectivamente, un día, un mes y un año y verifica si los números forman una fecha válida

4) Cree una función par::Int->Bool para verificar si un número es par o impar.

5) Escribe la función concepto :: Float -> Char que recibe una nota y devuelve el concepto correspondiente según las siguientes reglas:
   Puntuación inferior a 4 - Concepto E,
   Puntuación entre 4 y 5,99 - Concepto D,
   Puntuación entre 6 y 7,49 - Concepto C,
   Puntuación entre 7,5 y 8,99 - Concepto B y
   por encima de 9 - Concepto A.

Haskell (Tuplas)

Las tuplas permiten la definición y el uso de tipos de datos heterogeneos bajo una estructura relacional. Las *tuplas son* representadas en scripts por listas de componentes separados por coma, entre paréntesis.

```
("JOSE",1.8,23)  o  (100,10.4,”Brasil”)
```

Prelude provee dos funciones para retornar respectivamente el primer y el segundo elemento:

```
fst :: (a,b) -> a   o   snd :: (a,b) -> b
```

Su implementación probablemente utiliza variables anónimas:

```
fst (x,_) = x  o  snd (_,y) = y
```

Haskell (Tuplas)

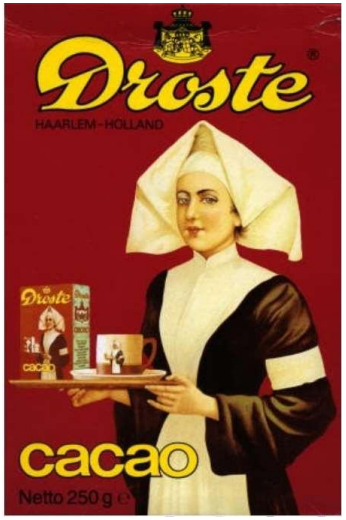
Se puede utilizar tuplas y la palabra reservada type para definir tipos de datos más complejos:

```
type Seq = String
type Nombres = (Seq, Seq, Seq, Seq)

--funciones constantes
f_nombres_estaciones :: Nombres
f_nombres_estaciones =
          ("Invierno","Otoño","Primavera","Verano")
```

Con uso de variables anónimas podemos buscar informaciones específicas en las uplas

```
primero :: Nombres ->Seq
primero (x, _, _, _) = x

segundo :: Nombres ->Seq
segundo (_, x, _, _) = x
```

Haskell (Recursión)



Se puede definir funciones usando otras previamente definidas.

Con todo, se puede definir una función por recurrencia, es decir, usando la propia función que estamos por definir; tales definiciones se dicen recursivas.

Ejemplo:

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

```
--factorial 5
--(factorial 4) * 5
--((factorial 3) * 4) * 5
--(((factorial 2) * 3) * 4) * 5
--((((factorial 1) * 2) * 3) * 4) * 5
--(((((factorial 0) * 1) * 2) * 3) * 4) * 5

--((((1 * 1) * 2) * 3) * 4) * 5
--(((1 * 2) * 3) * 4) * 5
--((2 * 3) * 4) * 5
--(6 * 4) * 5
--24 * 5
--120
```

Haskell (Recursión)

Alternativas de implementación:

Utilizando guardas:

```
factorial n | n==0 = 1
            | otherwise = n * factorial (n-1)
```

Utilizando condicional:

```
factorial n = if n==0 then 1 else n*factorial (n-1)
```

¿Por qué recursión?

- Exprimir la solución de un problema usando problemas semejantes pero de menor tamaño.
- Modelo universal de computación: cualquier algoritmo puede ser escrito usando funciones recursivas.
- Podemos demonstrar propiedades de funciones recursivas usando inducción matemática.

Haskell (Recursión)

Ejemplos:

N-esimo término de fibonnacci:

```
fibonacci :: Int -> Int  fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = (fibonacci (n-1)) + (fibonacci (n-2))
```

¿Cuántos números múltiplos de siete existen entre cero y un n-esimo número:

```
multiplo7 :: Int -> Int  multiplo7 7 = 1
multiplo7 n | n<=6 = 0
            | otherwise = 1 + multiplo7(n-7)
```

Ejercicio 04

--1) Cree una función recursiva para calcular la potencia de un número.

--2) Cree una función recursiva para decir si un número es par. Esa función debe retornar True o False.

--3) Cree funciones recursivas para informar el valor de una sumatoria dado un número N.

--4) Con base en la pregunta dos del Ejercicio 3 (slides anteriores), haga funciones recursivas que dado el "id" límite, se debe recorrer todos los datos de la base de datos del primeiro hasta ese valor final y retorne:

a) ¿Cuántas personas fueron seleccionadas en la base de datos

b) ¿Cuál es la menor edad de la base

c) ¿Cuál es la suma de la edad de las personas

d) ¿Cuál es la media de edad

e) cuántas personas están encima de la media de edad

5) sumaDigitos: recibe un número natural y retorna la suma de sus dígitos.
Ej.: sumaDigitos 3284 ==> 17

Haskell (Listas)

- Listas son colecciones de elementos:
  - en que el orden es significativa
  - elementos de un mismo tipo
  - posiblemente con elementos repetidos

- Una lista en Haskell
  - o es vacia [];
  - o es x:xs (x seguido de la lista xs)

```
[1, 2, 3, 4] == 1 : 2 : 3 : 4 : []
```

El operador **:** forma una nueva lista desde que que el último elemento sea una lista vacia.

Haskell (Listas)

- Se puede concatenar listas del mismo tipo con el operador ++

```
> [(1,'b'),(2,'c')] ++ [(3,'a'), (4,'w')]
[(1,'b'),(2,'c'),(3,'a'),(4,'w')]
```

- El tipo String en realidad es una lista del tipo char: [Char]

```
> ['O','l','a'] == "Hola"
True
```

- Algunas funciones de Prelude para manipular String

```
> words "El raton mordisqueo la ropa del rey de roma"
["El","raton","mordisqueo","la","ropa","del","rey","de","roma"]

> unwords  ["Un","plato","de","trigo","para","tres","tigres"]
"Un plato de trigo para tres tigres"
```

Haskell (Listas)

Se puede definir listas con secuencias aritméticas de la forma [a..b] donde a y b son números:

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]

> [1,3..10]
[1,3,5,7,9]

> [10,9..1]
[10,9,8,7,6,5,4,3,2,1]
```

Se puede definir listas infinitas:

```
take 10 [1,3..]
[1,3,5,7,9,11,13,15,17,19]

>  [1,3..] !! 4
9
```

Haskell (Listas)

La manipulación de las listas es muchas veces hecha a través de recursión separando la cabeza y cola:

```
productorio :: [Int] -> Int
productorio [] = 1
productorio (x:xs) = x * productorio xs


cantidad :: [a] -> Int  cantidad [] = 0
cantidad (_:xs) = 1 + cantidad xs
```

Ejercicio 05

```
--1) Cree una función  que retorne una  lista con todas las letras del alfabeto.

--2) Cree una función  que retorne una  lista con los números de 0 a 200 de
forma decreciente.

--3) Cree una función para retornar el inverso de una lista.

--4) Cree una función para obtener los “n” primeros elementos de una lista

--5) Cree una función para remover los “n” primeros elementos de una lista

--6) Cree una función que remueva el último elemento de una lista

--7) Cree una función que remueva los “n” últimos elementos de una lista

--8) Cree una función que remueva el n-ésimo elemento de una lista
```

Haskell (Listas - Comprensión)

- En matemática es usual definir conjuntos a partir de otro usando notación en comprensión. Ejemplo:

$$\{x^2 \ : \ x \in \{1,2,3,4,5\}\}$$

  El cual define el conjunto:

$$\{1,4,9,16,25\}$$

- En *Haskell* podemos definir una lista a partir de otra usando una notación semejante. Ejemplo:

```
> [x^2 | x<-[1,2,3,4,5]]

[1, 4, 9, 16, 25]
```

Haskell (Listas - Comprensión)

- Un término "*patrón*<-*lista*" se denomina generador:
  - determina los valores de las variables en el patrón;
  - determina el orden de los valores generados.

Ejemplo:

```
>[x | x <-[1..10] ]
 [1,2,3,4,5,6,7,8,9,10]
>[x | x <-[1,4..30] ]
 [1,4,7,10,13,16,19,22,25,28]
>[x | x <-[1,7..20] ]
 [1,7,13,19]
```

- Se puede usar múltiples generadores:

```
> [(x,y) | x<-[1,2,3], y<-[4,5]]
  [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Haskell (Listas - Comprensión)

- Orden entre generadores:

```
> [(x,y) | x<-[1,2,3], y<-[4,5]]          -- x primero
  [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
> [(x,y) | y<-[4,5], x<-[1,2,3]]          -- y primero
  [(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Las variables de los generadores posteriores cambian primero;
- Analogía: ciclos ‘for’ embutidos:

```
for(x=1; x<=3; x++)          for(y=4; y<=5; y++)
  for(y=4; y<=5; y++)   VS.    for(x=1; x<=3; x++)
     ...                          ...
```

Haskell (Listas - Comprensión)

Los generadores pueden depender de los valores *anteriores* pero no de los *posteriores.*

- Dependencias entre generadores:

```
> [(x,y) | x<-[1..3], y<-[x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

- Un ejemplo: la función *concat* (del prelúdio-padrón) concatena una lista de listas, ejemplo:

```
> concat [[1,2,3],[4,5],[6,7]]
[1,2,3,4,5,6,7]
```

Podemos definir usando una lista por comprensión:

```
concat :: [[a]] -> [a]
concat xss = [x | xs<-xss, x<-xs]
```

Haskell (Listas - Comprensión)

- Guardas

  Las definiciones en comprensión pueden incluir condiciones (designadas *guardas*) para filtrar los resultados.

Ejemplo: Conjunto de los enteros x, tal que x está entre 1 y 10 y x es par.

```
> [x | x<-[1..10], x`mod`2==0]
[2,4,6,8,10]
```

Divisores de un número entero positivo:

```
divisores :: Int -> [Int]
divisores n = [x | x <-[1..n], n`mod`x==0]
```

Haskell (Listas - Comprensión)

- Ejemplos:

```
--Un número es primo si es divisible por uno y por el
primo :: Int -> Bool
primo n = divisores n == [1,n]
```

```
--Listar todos los números primos entre 2 y N
primos :: Int -> [Int]
primos n = [x | x<-[2..n], primo x]
```

Combina dos listas en la lista de los pares de elementos correspondientes.

```
--Vamos a verificar si una lista está en orden
creciente
zipar (a:as) (b:bs) = (a,b) : zipar as bs
zipar _ _ = []
pares :: [a] -> [(a,a)]
pares xs = zipar xs (tail xs)

creciente :: Ord a => [a] -> Bool
creciente  xs = and [x<=x' | (x,x')<-pares xs]
```

Ejemplo:
>pares [1,2,3,4]
[(1,2),(2,3),(3,4)]

Usamos la función and de Prelude.

Haskell (Listas - Comprensión)

- Ejemplos:

```
--Busca un valor en una lista y obtiene todos sus índices
indices :: Eq a => a -> [a] -> [Int]
indices x ys = [i | (i,y)<-zip [0..n] ys, x==y]
            where n = length ys - 1
```

```
--Retornar cuantas letras de una string son minúsculas
minusculas :: String -> Int
minusculas txt = length [c | c<-txt, c>='a' && c<='z']
```

En el inicio del archivo digite:
Import Char

```
--Reescriba la función anterior con las funciones del módulo Char
minusculass :: String -> Int
minusculass cs = length [c | c<-cs, isLower c]
```

```
--Escriba una función para convertir un String para mayuscula
stringUpper :: String -> String
stringUpper cs = [toUpper c | c<-cs]
```

```
--Escriba una función para escribir un string con condición.
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

Ejercicio

```
--1) Cree una función que recibe dos parámetros: una lista de enteros y un  número
que representa la operación sobre la lista (1-lista de números pares y 2-  lista de
números impares). OBS: use las funciones odd y even.

--2) Cree una función que recibe una lista de tuplas del tipo (Int,Int) y retorne
una lista con la suma de las tuplas. Ej: [(1,2),(4,7)] => [3,11].

--3) Cree una función que recibe dos listas y retorne otra lista con la  combinación
de todos los elementos en pares. Ej [1,2] [3] => [(1,3),(2,3)]

--4) Cree una función que remueve un determinado caracter de un string

--5) Cree una función que retorne el código de los alumnos que tengan nota encima de
ocho, esa función debe manipular el retorno de la función: baseDeDatos = [ (1,
“André”, 10.0), (2, “Carlos”, 6.8), (3,”Maurício”, 7.0)]

--6) Cree una función que recibe una lista con varios nombres y verifica si algunos
de esos nombres comienzan con el parámetro de entrada N.

--7) Para calcular las combinaciones de notas para devolver el cambio durante un
  pago, podemos definir la función
```

Haskell (Funciones de alto orden)

- Una función es de orden superior si tiene un argumento que es una función o un resultado que es una función.

Ejemplo: el primer argumento de *twice es* una función:

Ahora funciones son consideradas con los mismos derechos de cualquier otro tipo, así ellas pueden ser pasadas por parámetro o pueden ser retornadas como resultado de otra función.

```
twice :: (a -> a) -> a -> a

twice f x = f (f x)

dobla x = 2 * x
Triplica x = 3 * x

>twice dobla 10
40

>twice triplica 10
90
```

Haskell (Funciones de alto orden)

Otro ejemplo:

```
app :: (a->b) -> (a,a) -> (b,b)
app f (x,y) = (f x, f y)

> app chr (65, 70)
('A','F’)

> app tan (65, 70)
(-1.4700382576,1.22195991813)
```

- Ventajas:
  - Permite definir patrones de computación comunes que pueden ser fácilmente reutilizados;
  - Facilita la definición de bibliotecas para dominios específicos.

Haskell (Funciones de alto orden)

Curring: aplicación de funciones parciales

La función mult abajo, puede ser entendida como teniendo dos argumentos de entrada y uno de retorno, pero en realidad mult es una función que recibe un argumento del tipo Int y devuelve una función de tipo (Int->Int).

```
mult :: Int -> Int -> Int
mult x y = x * y
```

≡ Int -> (Int -> Int)

En Haskell, todas las funciones son unarias!

mult 2 5 ≡ (mult 2) 5 :: Int

- Así, mult puede ser usada para generar nuevas funciones:

```
doblar = mult 2
triplicar = mult 3

> doblar 30
60
```

Haskell (Funciones de alto orden)

Los operadores infijados también pueden ser aplicados a apenas un argumento, generando así una nueva función.

Ejemplos:

```
(+) :: Integer -> Integer -> Integer
> (5+) 9
14
```

```
(<=) :: Integer -> Integer -> Bool
> (0<=) 10
True
```

```
 (*) :: Double -> Double -> Double
>(3*)7  21
```

Haskell (Funciones de alta orden)

Ejemplo de aplicación de high order functions

La mayoria de las definiciones sobre listas se encajan en tres casos:

- **folding** colocación de un operador entre los elementos de una lista;
- **filtering** filtra algunos elementos de la lista;
- **mapping** la aplicación de funciones a todos los elementos de la lista.

Haskell (Funciones de alto orden)

Folding

Vea que las funciones abajo tiene un patrón de computación:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

```
and [] = True
and (x:xs) = x && and xs
```

```
conc [] = []
conc (x:xs) = x ++ conc xs
```

Patrón, aplicamos un operador entre cada elemento de una lista

Para ese patrón existe la siguiente definición:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Utilizando la función de alto orden tenemos:

Retorno en el caso base

```
> foldr (+) 0 [1..5]
15
```

```
> foldr (&&) True [True,False]
False
```

```
concat l = foldr (++) [] l
> concat [“Hi, ”,”man”]
```

Haskell (Funciones de alto orden)

Mapping

Vea que las funciones abajo tienen un patrón de computación:

```
ft [] = []
ft (x:xs) =
      factorial x : ft xs
```

```
min [] = []
min (x:xs) =
        toLower x : min xs
```

```
trip [] = []
trip (x:xs) =
          (3*x) : trip xs
```

Patrón, aplicamos una función a cada elemento de la lista

Para ese patrón existe la siguiente definición:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

Utilizando la función de alto orden tenemos:

```
>map factorial [1..5]
 [1,2,6,24,120]
```

```
>map toLower [‘X’,’B’]
 “xb”
```

```
> map (3*) [1..5]
[3,6,9,12,15]
```

Haskell (Funciones de alto orden)

Filtering

Vea que las funciones abajo tienen un patrón de computación:

```
aprob [] = []
aprob (x:xs) = if x>=12
                  then x:aprob xs
                  else aprob xs
```

```
digit [] = []
digit  (x:xs)  |    isDigit x   = x:digit xs
               |    otherwise   = digit xs
```

Para ese patrón existe la siguiente definición:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
     | (p x)     = x : (filter p xs)
     | otherwise = filter p xs
```

Patrón, filtramos elementos de la lista de acuerdo con una función de criterio.

Utilizando la función de alto orden tenemos:

```
>filter isDigit "abc123cdf"
"123"
```

```
> filter (12<=) [4,18,12,10,18]
[18,12,18]
```

Ejercicios

```
--1) Usando funciones de alto-orden, cree una función o un comando en el prompt
que  receba una lista de strings y retorne una lista con el tamaño de cada string
de la  lista de entrada.

--2) Cree una función de alto orden llamada getLista, la cual recibe una función
de filtrado y una lista, el objetivo es retornar la lista de acuerdo con el
filtrado.
```