Universidad Tecnológica Nacional
Facultad Regional Córdoba

Paradigmas de Programación

UNIDAD NRO. 4
PARADIGMA FUNCIONAL (PARTE II)

CONTENIDOS ABORDADOS

- Repaso
- Sintaxis de Haskell (Continuación)
  - Definiciones locales: expresiones let y where.
  - Disposición de código: Layout
- Expresiones recursivas
- Tipos compuestos: listas

BLOQUE 1: REPASO

PARADIGMA FUNCIONAL

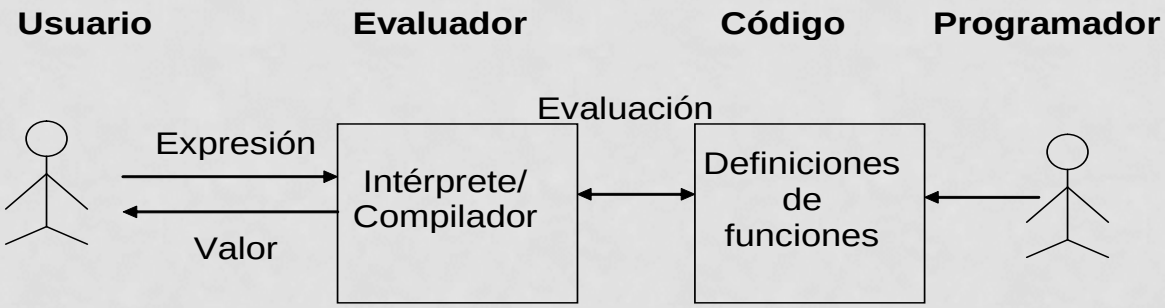
- El paradigma funcional es un paradigma declarativo que se basa en un modelo matemático de **composición de funciones**. Ejemplos:

  doble(doble(5))=20

  doble(mayor(5,20))=40

- Lenguajes asociados: Lisp, ML,**Haskell** entre otros.

Funciones + Control = programa



SINTAXIS DE HASKELL

- **Tipos de datos: Primitivos**: Int, Integer, String, Char y Bool.

- **Conectores Lógicos**: **&&** (and), || (or), **/=** (distinto) y **not** (negación). Actúan en forma corto circuitada como en Python.

- Los **operadores** son funciones que se escriben entre sus dos argumentos en lugar de precederlos. Esta notación se llama infija de un operador. Ejemplo 3 <= 4.

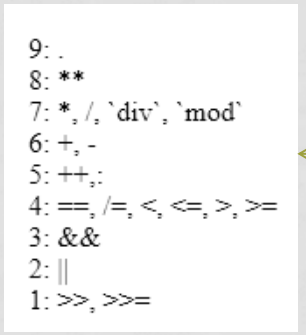
- Cuando se utilizan símbolos de operadores debe tenerse en cuenta:

  - La **precedencia**
  - La **asociatividad**

SINTAXIS DE HASKELL

- **Operadores:**
  - **La precedencia:** Por ejemplo: "2 * 3 + 4" podría interpretarse como "(2 * 3) + 4" o como "2 * (3 + 4)"
  - Cada operador tiene asignado un valor de precedencia (un entero entre 0 y 9). En el ejemplo anterior se comparan los valores de precedencia (en este caso (+) y (*) tienen el valor 6 y 7 lo que efectúa primero la multiplicación).

  - **La asociatividad:** en el caso de tener operadores con el mismo nivel de precedencia. Ejemplo: "1 – 2 - 3" puede ser considerado como "(1 - 2) – 3" resultando -4 o como "1 – (2 - 3)" resultando 2. La regla de asociatividad que aplica Haskell es Asociatividad a izquierda=> "(1 - 2) – 3" => 4



Valor de precedencia en Haskell

SINTAXIS DE HASKELL

Operadores:

- **Operadores relacionales**: >=, <=, <, >, == (igualdad).
- **Operadores aritméticos**: +, *, / (división no entera), ^ potencia.

FUNCIONES EN HASKELL

Sintaxis:

- **Definición de signatura:** La definición de tipo de una función, donde se especifican los tipos de los datos de entrada y salida que posee la función;
- **Implementación del cuerpo de la función**: Se especifica la descripción de la función para transformar los datos de entrada en los datos de salida.

nombre_funcion::tipo_argumento->tipo_resultado
nombre_funcion nombre_argumento=<implementacion>

FUNCIONES EN HASKELL

Ejemplo:

- **Función parcializada (curried):**

```
suma::Integer -> Integer -> Integer
suma x y  =  x + y
```

La invocación : suma 2 5 ==> 7

- **Función no parcializada (uncurried):**

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

La invocación : suma (2,5) ==> 7

MECANISMOS DE CONTROL HASKELL

Los mecanismos de **selección** se realizan mediante expresiones como:

- **Guardas:**

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

- **If then else:**

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

- **Case:**

```
paridad :: Int -> String
paridad x = case (x `mod` 2) of
                0 -> "par"
                1 -> "impar"
```

No brinda expresiones **iterativas**, una alternativa es la utilización de **recursividad**.

BLOQUE 2: HASKELL (PARTE II)

DEFINICIONES LOCALES

Expresiones Let.

- Las *expresiones* ***let*** de Haskell **son útiles cuando se requiere un conjunto de declaraciones locales**. Se introducen en un punto arbitrario de una expresión utilizando una expresión de la forma:

  let <decls> in <expr>

Ejemplo con una **declaración**:

```
Hugs> let x = 1 + 4 in x*x + 3*x + 1
41 :: Integer
```

Ejemplo con una **función**:

```
sumaLet::Int -> Int -> Int
sumaLet x y  = let z = 2 in z + x +y
```

DEFINICIONES LOCALES

Expresiones Where

- En algunos casos es conveniente establecer declaraciones locales en una ecuación con varias con guardas, lo que podemos obtener a través de una *cláusula where*:

```
f x y  |  y>z        =  ...
       |  y==z       =  ...
       |  y<z        =  ...
       where z = x*x
```

DEFINICIONES LOCALES

Expresiones Where Ejemplo

```
comparaCuadrado::Integer -> Integer -> String
comparaCuadrado x y   |   y>z  =  " el 2º es mayor que el cuadrado del 1º"
                      |   y==z  =  "el 2º es igual que el cuadrado del 1º"
                      |   y<z    =  "el 2º es menor que el cuadrado del 1º"
                     where z = x*x
```

Si evaluamos:

```
Main> comparaCuadrado 5 3
"el 2º es menor que el cuadrado del 1º" :: [Char]
```

DISPOSICIÓN DE CÓDIGO: LAYOUT

- Para marcar el final de una ecuación, una declaración, etc. se utiliza una sintaxis bidimensional denominada *espaciado (**layout**)* que se basa esencialmente en que las declaraciones están alineadas por columnas.

Reglas :

- El siguiente carácter de cualquiera de las palabras claves **(where, let, o case of)** es el que determina la columna de comienzo de declaraciones en las expresiones where, let, o case correspondientes.
- Es necesario asegurarse que la columna de comienzo dentro de una declaración está más a la derecha que la columna de comienzo de la siguiente cláusula.

DISPOSICIÓN DE CÓDIGO: LAYOUT

Ejemplos:

```
--usando layout
ejemplo1::Int
ejemplo1  = (a +a)
                    where a = 6   --asigna el valor 6
ejemplo2::Int
ejemplo2  = (a + b)
                where
                          a = 6
                    b = 5
```

DISPOSICIÓN DE CÓDIGO: LAYOUT

Ejemplos:

```
--no usando layout-Incorrecto!!
ejemplo2::Int
ejemplo2  = (a + b)
            where
                    a = 6
                        b = 5
```

La definición de la expresión anterior nos da el siguiente error:

ERROR file:.\prueba.hs:27 - Syntax error in input (unexpected `=')

BLOQUE 3: RECURSIÓN Y LISTAS

EXPRESIONES RECURSIVAS

- La recursión es una herramienta poderosa y usada muy frecuentemente en los programas en Haskell.
- Una expresión es recursiva cuando su evaluación (en ciertos argumentos) involucra el llamado a la misma expresión que se esta definiendo. En la definición recursiva, es necesario considerar dos momentos de evaluación:
  - **Evaluación del caso Básico**: Momento en que se detiene el proceso de evaluación, se obtiene una valorización de la expresión.
  - **Evaluación Recursiva:** Se evalúa la expresión actual, efectuando evaluaciones a si misma, hasta obtener la valorización de la expresión.

EXPRESIONES RECURSIVAS

Ejemplo de la **función factorial**:

- 1. Utiliza una sola expresión, en donde la evaluación del caso básico y la evaluación recursiva están en una expresión if-then-else:

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

- 2. Utiliza dos expresiones, una para evaluar el caso básico y la otra para hacer la evaluación recursiva:

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

TIPOS DE DATOS COMPUESTOS

- Los tipos *compuestos*, son aquellos cuyos valores se construyen utilizando otros tipos, por ejemplo: listas, funciones y tuplas.

- A continuación describiremos:

  - **Listas**

LISTAS EN HASKELL

- Una lista **es una colección de cero o más elementos todos del mismo tipo.**

- **Definición general**: Si v1,v2,...,vn son valores con tipo t, entonces una forma de sintaxis es usando el operador "**:**" . Ejemplo: **1:2:3:[]**. Otra forma sintáctica es: **[1,2,3]**

- **Definición recursiva**: Una lista está compuesta por una cabeza y una cola que es una lista compuesta por los elementos restantes.
  - Lista vacía se denota con [ ].
  - El operador (:) permite dividir cabeza y cola: (x:xs)

LISTAS EN HASKELL

- **Ejemplos:**

```
[False,True] :: [Bool]            --elementos booleanos
[1,2,3]:: [Int]                          --elementos enteros
['a', 'a', 'a'] :: [Char]         --elementos char,
[[1,2,3], [4], [5,6]]:: [[Int]]   --listas de enteros
["uno","dos"] :: [[Char]]         --elementos listas de char o
                                         -- String
```

El tipo String es sinónimo de [Char], y las listas de este tipo se pueden escribir entre comillas: "uno" es lo mismo que ['u', 'n', 'o'].

LISTAS EN HASKELL. FUNCIONES ÚTILES

Función	Utilidad	Ejemplo
take n xs	Devuelve los "n" primeros elementos de xs.	Hugs> take 2 [1,2,3] **[1,2]**
drop n xs	Devuelve el resultado de sacarle a xs los primeros "n" elementos.	Hugs> drop 2 [1,2,3] **[3]**
head xs	Devuelve el primer elemento de la lista.	Hugs> head [1,2,3] **1**
tail xs	Devuelve toda la lista menos el primer elemento.	Hugs> tail [1,2,3] **[2,3]**
last xs	Devuelve el último elemento de la lista.	Hugs> last [1,2,3] **3**
init xs	Devuelve toda la lista menos el último elemento.	Hugs> init [1,2,3] **[1,2]**

LISTAS EN HASKELL. FUNCIONES ÚTILES

Función	Utilidad	Ejemplo
xs ++ ys	Concatena ambas listas.	Hugs> [1,2,3]++[4,5,6] **[1,2,3,4,5,6]**
xs !! n	Devuelve el n-ésimo elemento de xs.	Hugs> [1,2,3]!!2 **3**
elem x xs	Determina si "x" es un elemento de xs.	Hugs> elem 2 [1,2,3] **True**
length xs	Determina la cantidad de elementos de una xs.	Hugs> length [1,2,3] **3**

LISTAS: FUNCIONES PROPIAS

1. **Retorna una lista de enteros**:

```
lista:: [Integer]
lista = [1,2,3]

Main> lista
[1,2,3] :: [Int]
```

2. **Retorna la cabeza de una lista:**

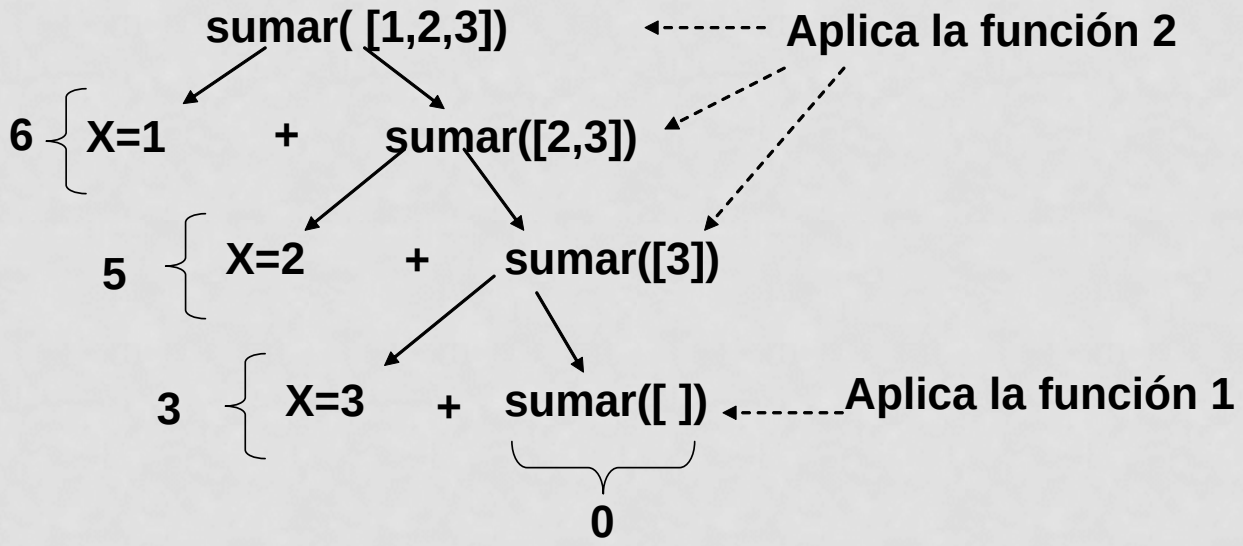
```
cabeza:: [Integer] -> Integer
 cabeza (x:_) = x
```

Utiliza variable anónima, porque no necesita evaluar la cola

LISTAS: FUNCIONES PROPIAS

3. **Con una lista de enteros retornar la suma de todos los elementos:**

```
sumar:: [Integer] -> Integer
sumar []= 0
sumar (x : xs) = x + sumar xs
```



LISTAS: FUNCIONES PROPIAS

4. **Con una lista de enteros retornar la cantidad de elementos:**

```
contar:: [Integer] -> Integer
contar []= 0
contar (x : xs) = 1 + contar xs
```

5. **Con una lista de enteros mostrar sus elementos:**

```
mostrar:: [Integer] -> String
mostrar [] = " "
mostrar (x : xs) = show(x) ++ " " ++ mostrar xs
```

LISTAS: FUNCIONES PROPIAS

6. **A partir de una lista de números enteros, mostrar en una cadena los números que sean pares:**

```
pares:: [Integer] -> String
pares [] = " "
pares (x : xs) = if  even x  then show(x)++ " " ++ pares xs
                             else pares xs
```

LISTAS: FUNCIONES PROPIAS

7. **Generar una lista de números descendentes, a partir de un valor inicial, el proceso termina cuando el numero sea 0:**

Usando el operador : para generar la lista.

```
generarLista:: Integer -> [Integer]
generarLista 0 = []
generarLista num = num : generarLista (num – 1)
```

Usando el operador ++ para generar la lista.

```
generarLista 0 = []
generarLista num = [num] ++ generarLista (num – 1)
```

ENCAJE DE PATRONES: PATTERN-MATCHING

- Es el proceso que evalúa los argumentos de una función, que está definida mediante más de una ecuación, para determinar cuál es la ecuación a aplicar.
- 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.
- Por ejemplo:

```
fact 0 = 1
fact n = n * fact (n-1)
```

ENCAJE DE PATRONES: PATTERN-MATCHING

Reglas 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.

ENCAJE DE PATRONES: PATTERN-MATCHING

Ejemplo:

```
length        :: [a] -> Integer
length []      =  0
length (x:xs)  =  1 + length xs
```

- El patrón [] "concuerda" (matches) o puede emparejarse con la lista vacía.
- El patrón x:xs. se podrá emparejar con una lista de al menos un elemento, instanciándose x a este primer elemento y xs al resto de la lista.
- Si la comparación tiene éxito, el miembro izquierdo es evaluado y devuelto como resultado de la aplicación. Si falla, se intenta la siguiente ecuación, y si todas fallan, el resultado es un error.