Universidad Tecnológica Nacional

FACULTAD REGIONAL CORDOBA

PARADIGMAS DE PROGRAMACION

Unidad IV
Paradigma Funcional
Parte III

CONTENIDOS ABORDADOS

- Repaso de expresiones.
- Expresiones recursivas.
- Tipos compuestos:
  - Tuplas
  - Listas

Repaso: Expresiones Condicionales

- **if** exprBool
  then exprSi
  else exprNo
- **case** (expresión) **of**
  valor1 -> resultado1
  valor2 -> resultado2
  _ -> otroresultado
- *f x1... xn* | *condicion1* **=** *e1*
  | *condicion2* **=** *e2*
  .
  | *condicionm* **=** *em*
  | ***otherwise*** **=** *exp*

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.
- La recursión es una técnica que se basa en definir expresión en términos de ellas mismas. Si del lado derecho de una expresión aparece en algún punto la misma expresión que figura del lado izquierdo, se dice que la expresión tiene llamado recursivo, es decir **“se llama a sí misma”**.

Expresiones recursivas

- 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: Obtener el factorial de un número.

Formulación:

0! = 1

n! = n*n-1*n-2*n-3*…..*1

n! = n*(n-1)!

(n-1)!= (n-1) * (n-2)!

….

Término general:

Funciones:

Caso básico: factorial 0 = 1

Caso recursivo: factorial n = n * factorial (n-1)

Expresiones recursivas

Implementación de la función factorial:

- 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)
```

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

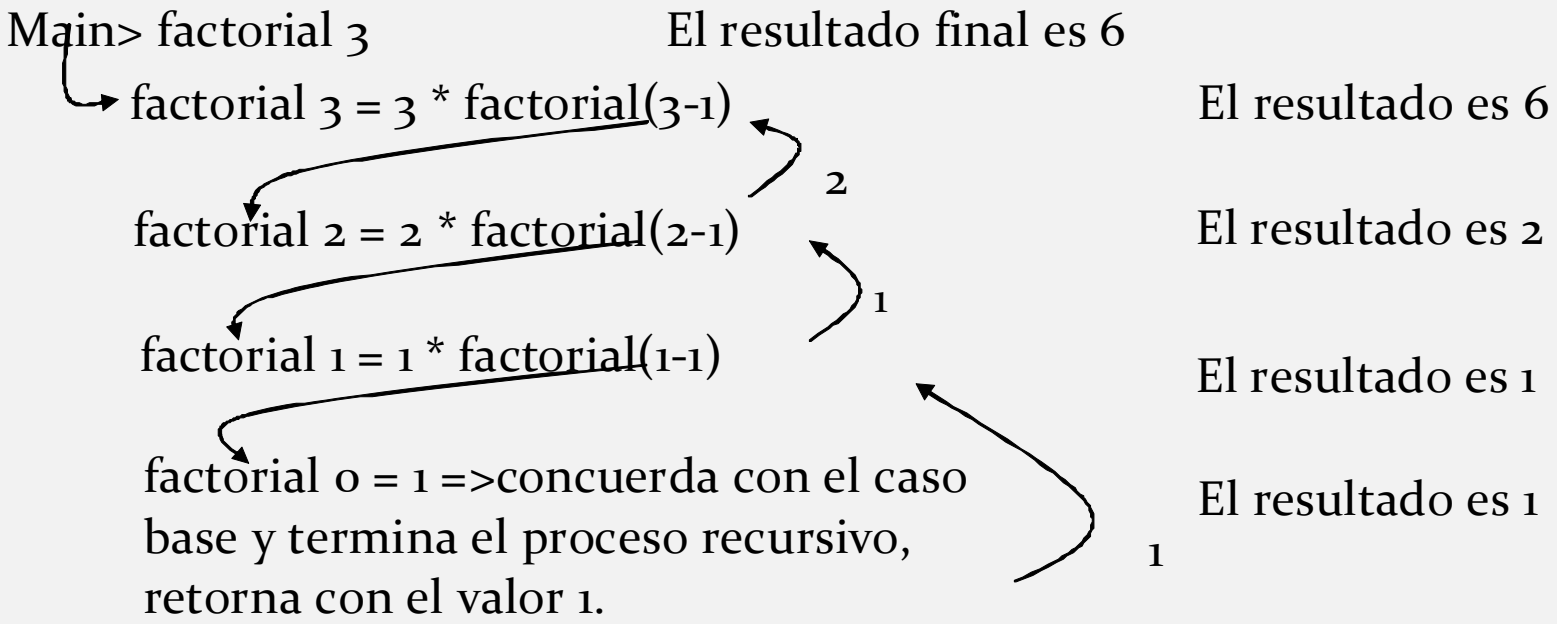
```
factorial::Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Expresiones recursivas

Ejemplo: Obtener el factorial de un número.

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

Consulta:



Tipos de Datos compuestos

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

- Describiremos:

  - Tuplas
  - Listas

Tuplas

- Una tupla es un dato compuesto, es una sucesión de elementos, donde el tipo de cada elemento puede ser distinto. El tamaño de una tupla es fijo.
- Aridades:
  - La aridad de una tupla es el número de elementos.
  - La tupla de aridad 0, (), es la tupla vacía.
  - No están permitidas las tuplas de longitud 1.

- Ejemplos:

  (False,True) :: (Bool,Bool)

  (False,'a',True) :: (Bool,Char,Bool)

Tuplas

- Las tuplas son útiles cuando una función tiene que devolver más de un valor, por ejemplo:

```
--devuelve el anterior y posterior de un número
antPos:: Integer -> (Integer,Integer)
antPos x = (x - 1, x + 1)

Main> antPos 2
(1,3) :: (Integer,Integer)
```

Tuplas

- Dada una tupla (String, Float), las funciones para retornar los valores individuales son:
  - Devuelve el primer elemento:

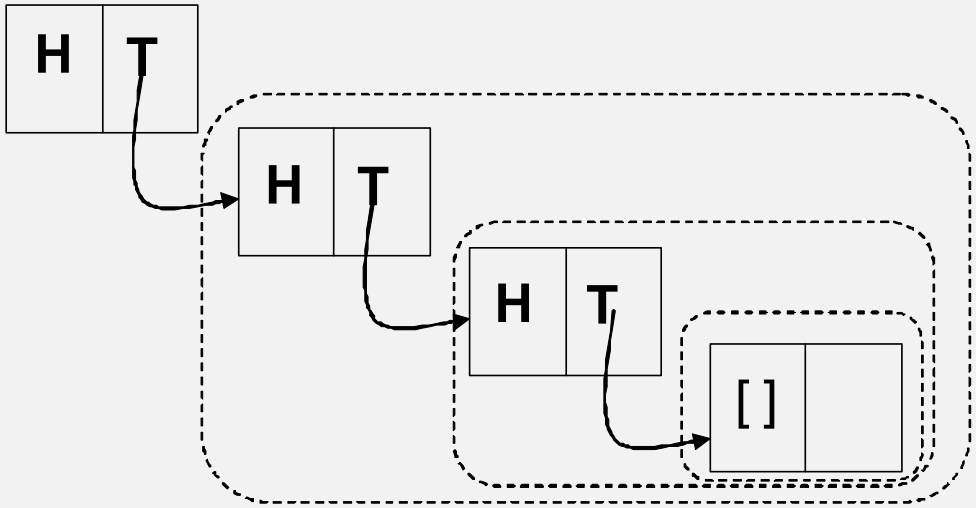
    ```
    fst:: (String, Float) -> String
    fst (x, y) = x
    ```

  - Devuelve el segundo elemento:

    ```
    last:: (String, Float) -> Float
    last (x, y) = y
    ```

Listas

- Son estructuras de datos que almacenan y manipulan conjunto de elementos del mismo tipo.
- El primer componente de la lista se llama *cabeza (Header)* y el segundo *cola (Tail).*



- Por definición son recursivas, el segundo término es una lista, que a su vez posee cabeza y cola, y así se repite la construcción hasta que se encuentra la lista vacía [].

Listas

- Una lista es una colección de cero o más elementos todos del mismo tipo.
- Las expresiones de tipo lista se construyen con [] (que representa la lista vacía) y : (a:as es la lista que empieza con el elemento a y sigue con la lista as).
- También pueden escribirse entre corchetes, con los elementos separados por comas.
- Sintaxis para listas

  x1: (x2: (...(xn−1: (xn: [])))) ó [x1,x2,...xn−1,xn]
- Ejemplo:

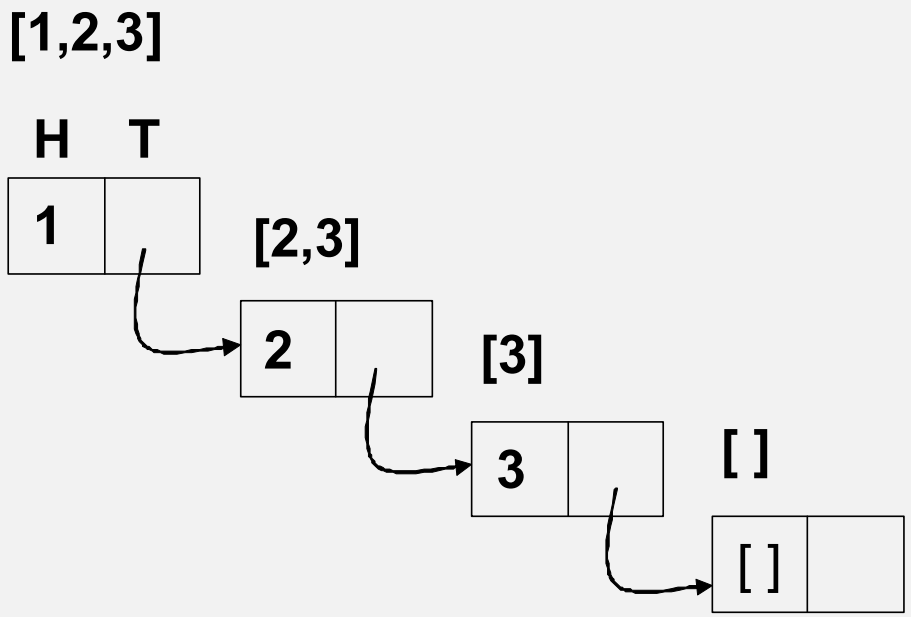
  1:(2:(3:([]))) ó [1,2,3]

Listas

Ejemplos:

- La siguiente lista de términos [1, 2, 3] se representa como:

  [1 : [2 : [3 : []]]] donde el termino constante 1 es la cabeza y la cola es la lista [[2 : [3 : []]]].



Listas

- **Definición general**: Si v1,v2,...,vn son valores con tipo t, entonces v1: (v2: (...(vn−1: (vn: [])))) es una lista con tipo [t].
- **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)
- Longitudes
  - La longitud de una lista es el número de elementos.
  - La lista de longitud 0, [], es la lista vacía.
  - Las listas de longitud 1 se llaman listas unitarias.

Listas

- **Ejemplos:**

```
[False,True] :: [Bool]--elementos booleanos
[1,2,3]:: [Int]          --elementos enteros
['a', 'a', 'a'] :: [Char]  --elementos char

-- tuplas(Bool,Int)
[(True,1), (False,2)]:: [(Bool,Int)]
--listas de enteros
[[1,2,3], [4], [5,6]]:: [[Int]]
--listas de char o String
["uno","dos"] :: [[Char]]
```

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: Funciones predefinidas

Algunas funciones útiles sobre listas:

- head xs devuelve el primer elemento de la lista.
- tail xs devuelve toda la lista menos el primer elemento.
- length xs devuelve la cantidad de elementos.
- sum xs devuelve la sumatoria de los valores de la lista.
- xs ++ys concatena ambas listas.
- last xs devuelve el último elemento de la lista.
- xs !! n devuelve el n-ésimo elemento de xs.
- elem x xs devuelve true si x es un elemento de xs.
- take n xs devuelve los n primeros elementos de xs.
- map function xs devuelve una lista con la misma cantidad de elementos, pero con la aplicación de la *function* a cada elemento de la lista *xs.*

Listas: Funciones predefinidas

Ejemplos:

```
Hugs>length [1,3,10]
3 :: Int
Hugs>sum [1,3,10]
14 :: Int

Hugs>[1,3,10] ++ [2,6,5,7]
[1, 3, 10, 2, 6, 5, 7] :: [Int]
Hugs>concat [[1], [2,3], [], [4,5,6]]
[1, 2, 3, 4, 5, 6] :: [Int]

Hugs>map fromEnum ['H', 'o', 'l', 'a']
[72,111,108,97] :: [Int]
```

Listas: Aplicaciones

Ejemplos usando funciones propias:

1. Retorna una lista de enteros:

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

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

2. Retorna la cabeza o cola de una lista:

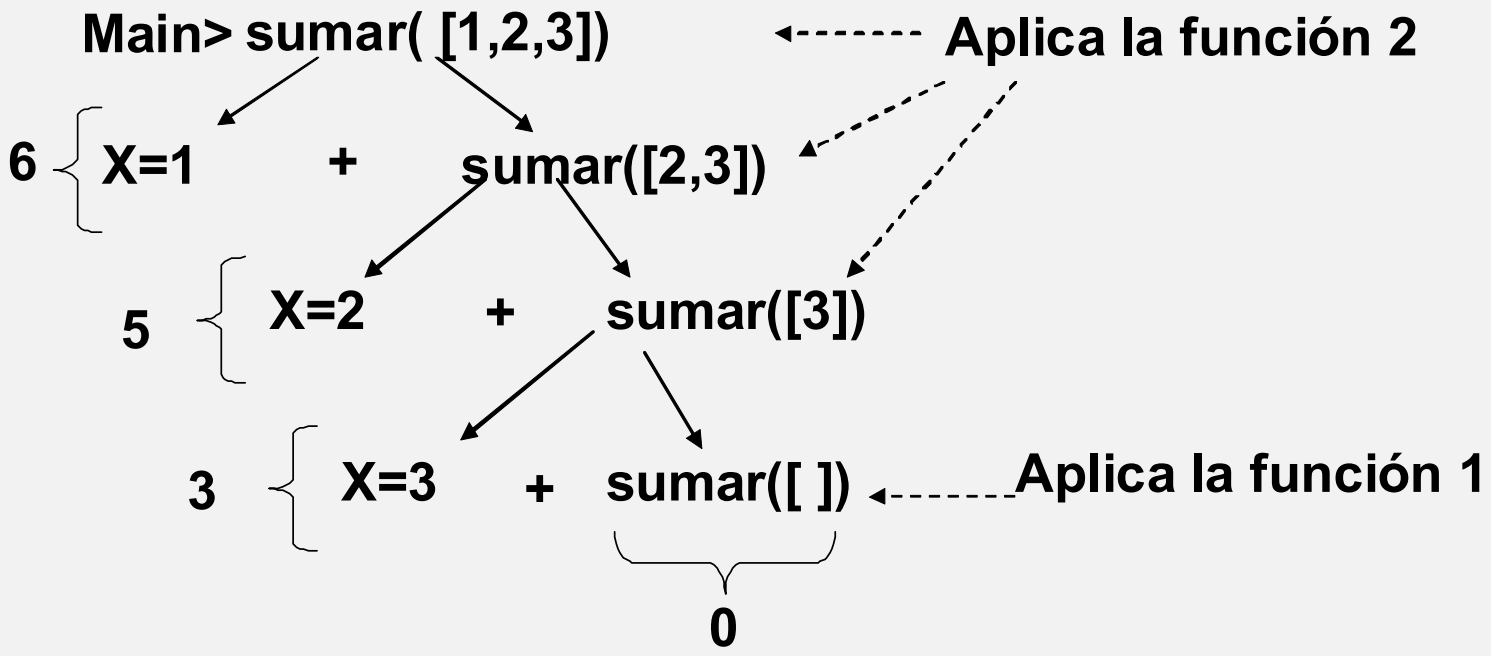
```
cabeza:: [Int] -> Int
cabeza (x:_) = x

cola:: [Int] -> [Int]
cola (_:xs) = xs
```

Utiliza variable anónima, porque no necesita evaluar el argumento que no usa.

Listas: Aplicaciones recursivas

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: Aplicaciones recursivas

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

Listas: Aplicaciones recursivas

6. A partir de una lista de números enteros, contar cuantos elementos son pares:

```
pares:: [Int] -> Int          [2,4,5,8,3,1]   ➔ 3
pares [] = 0
pares (x : xs) = if ( (mod x 2) == 0 )
                    then 1 + pares xs
                    else pares xs
```

7. A partir de una lista de números enteros, contar cuantos elementos son mayores que un valor pasado por parámetro:

```
contarMayores:: [Int] -> Int-> Int
contarMayores [] _ = 0
contarMayores (x : xs) valor = if (x > valor)
                                then 1 +
(contarMayores xs valor)
                                else
(contarMayores xs valor)
```

Listas: Aplicaciones recursivas

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

- Usando el operador : para generar la lista.

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

- Usando el operador ++ para generar la lista.

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

Listas por comprensión

- Otra alternativa que permite Haskell para la generación de listas es mediante la definición por comprensión.

```
[ x | x <-[1..10]] [1,2,3,4,5,6,7,8,9,10]
```

función de cálculo para generar la lista

asigna a x un rango. Los rangos generan una lista que contiene una secuencia de elementos enumerables

- Ejemplo :

```
Hugs> [x | x <-[1..10]]
[1,2,3,4,5,6,7,8,9,10]
```

Listas por comprensión

- Otro ejemplo:

```
Hugs> [x*2| x <-[1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

- Las listas por comprensión pueden incluir la definición de condiciones para restringir los valores incluidos en la lista generada.
- Ejemplo: Genera solo elementos que su doble sea mayor o igual a 12, usando el rango 1..10

```
Hugs> [x*2| x <- [1..10], x >= 6]
[12,14,16,18,20]
```

Donde: `x >= 6` es la condición o guarda.

Listas por comprensión

- Ejemplos:

```
Hugs> [x*x | x <- [50..100]]
[2500,2601,2704,2809,2916,3025,3136,3249,3364,3481,360
0,3721,3844,3969,4096,4225,4356,4489,4624,4761,4900,50
41,5184,5329,5476,5625,5776,5929,6084,6241,6400,6561,6
724,6889,7056,7225,7396,7569,7744,7921,8100,8281,8464,
8649,8836,9025,9216,9409,9604,9801,10000]
Hugs> [x*x | x <- [50..100], x < 70]
[2500,2601,2704,2809,2916,3025,3136,3249,3364,3481,360
0,3721,3844,3969,4096,4225,4356,4489,4624,4761]
```

- Genera todos los números del 50 al 100 cuyo resto al dividir por 7 fuera 3

```
Hugs> [x | x <- [50..100], (mod x 7) == 3]
[52,59,66,73,80,87,94]
```