Introducción al paradigma funcional

Tradicionalmente la visión de la programación que se tiene por la gran mayoría de personas vinculadas como estudiante o profesionales al desarrollo de software solo tiene en cuenta la secuencia de instrucciones que se le debe dar al computador para que ejecute una tarea.

Una visión general de la programación debe incluir otros aspectos o modelos de programar una computadora, entonces, se debe definir la programación como el análisis y la solución de problemas mediante la descripción de valores, propiedades y métodos, el diseño de algoritmos correctos y eficientes y de estructuras de datos que soporten dicha solución.

De acuerdo con esta visión de la programación la visión tradicional solo forma una parte pequeña de una visión global, es decir que la programación imperativa, solo forma parte de un universo de posibilidades para el desarrollo de estas soluciones.

Introducción al paradigma funcional

Un paradigma de programación es un modelo básico de diseño e implementación de programas, un modelo que permite desarrollar programas conforme a ciertos principios o fundamentos específicos que se aceptan como válidos.

Un paradigma, es una colección de modelos conceptuales que juntos modelan el proceso de diseño, orientan la forma de pensar y solucionar los problemas y, por lo tanto, determinan la estructura final de un programa.

Los paradigmas de programación abarcan dos ramas principales: imperativa y declarativa, estas ramas abarcan los diferentes paradigmas de programación:

Introducción al paradigma funcional

Rama Imperativa (procedimentales)

En esta rama están los paradigmas que indican el modo de construir la solución, detallando paso a paso el mecanismo para la obtención de esta.

- Paradigma estructurado.
- Paradigma orientado a objetos

Rama Declarativa

En esta rama se describen lasa características que debe tener la solución, es decir que se debe obtener pero no como obtenerla.

- Paradigma funcional
- Paradigma lógico.

Introducción al paradigma funcional

Los lenguajes de programación imperativos evolucionaron desde los lenguajes ensambladores, por esto se basan en el conocimiento de una arquitectura física de las máquinas para operar sobre dicha arquitectura.

Los lenguajes imperativos dejan de lado los procesos de pensamiento de los programadores.

Por otro lado, los lenguajes de programación funcionales tratan el problema de la programación desde un punto de vista matemático, utilizando la noción de función como base para la construcción de los algoritmos y estructuras de datos.

En los lenguajes imperativos existen dos tipos de construcciones: las instrucciones de control y las expresiones. Las instrucciones de control permiten manejar explícitamente el flujo de control, y las expresiones son utilizadas para calcular valores.

Introducción al paradigma funcional

Los lenguajes imperativos poseen la característica de poder realizar modificaciones implícitas a la memoria de la computadora. A estas modificaciones se las conocen como "efectos laterales", ya que no son claramente visibles en el código del programa, y tienen consecuencias no deseables para el razonamiento de propiedades.

A modo de ejemplo de un programa en el paradigma imperativo:

```
Procedimiento factorial(x):
    n<-x;
    a<-1;
    mientras (n>0) hacer
        a<a*n;
        n<n-1;
    fin mientras
    retorne a;
fin procedimiento
```


Introducción al paradigma funcional

En matemáticas no existe la noción de **estado implícito** que puede ser modificado, haciendo innecesaria la presencia de instrucciones. En cambio existen valores (inmutables) que pueden ser expresados de maneras complejas mediante expresiones; el conjunto de valores conocidos conforman de esta manera un estado explícito, dando como resultado un lenguaje estático.

El cálculo de dichos valores se realiza mediante un proceso de reemplazo de subexpresiones que no tienen un orden preestablecido, dando como resultado un control implícito de operación.

Al no existir **efectos laterales**, dos expresiones sintácticamente iguales darán el mismo valor, propiedad conocida como **transparencia referencial**; esta propiedad es el pilar de la habilidad de razonamiento. Por ello, aumentar el nivel de abstracción mantiene la coherencia con el modelo subyacente.

Introducción al paradigma funcional

Ahora veamos el ejemplo anterior expresado respetando un modelado dentro del paradigma funcional:

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

Vemos una forma de implementar una solución que esta mas cercana al razonamiento del programador y no una secuencia de instrucciones orientadas más hacia la arquitectura de una máquina que a un razonamiento matemático.

Paradigma Funcional

Existen dos grandes categorías de lenguajes funcionales: los funcionales puros y los híbridos.

La diferencia entre ambos estriba en que los lenguajes funcionales híbridos son menos dogmáticos que los puros, al admitir conceptos tomados de los lenguajes imperativos, como las secuencias de instrucciones o la asignación de variables.

En contraste, los lenguajes funcionales puros tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial, algo que no se cumple siempre con un lenguaje funcional híbrido.

Paradigma Funcional

En resumen podemos decir que:

- El paradigma de programación funcional trata la computación como la evaluación de funciones matemáticas y evita los estados y datos mutables.
- El paradigma de programación funcional hace énfasis en la aplicación de funciones, en contraste con el estilo de programación imperativo que pone el énfasis en los cambios de estado.
- Los cálculos se ven como una función matemática que hace 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.
- Un programa funcional es una expresión simple que es ejecutada por evaluación de la expresión. El foco se centra en **QUÉ** va a ser computado, no en **CÓMO** va a serlo.

Expresiones y valores

La noción de expresiones es central en la programación funcional. La característica más importante de la notación matemática es que una expresión se usa para denotar o describir un valor. En otras palabras, el significado de una expresión es su valor y no existen efectos laterales ocultos.

Funciones

Una función en programación funcional tiene dos partes importantes, la declaración, donde se indica el nombre de la función, los tipos de parámetros que recibe y el tipo de retorno de la función (la declaración puede ser implícita) y la definición o definiciones de la función.

Por ejemplo veamos en el lenguaje Haskell como queda definida la función factorial de los ejemplos anteriores.

Funciones

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

En donde en negrillas se hace la declaración de la función factorial y el resto son las definiciones de la función.

Estas funciones se guardan en un archivo de texto plano que define un script que será cargado por Haskell.

En estos scripts se pueden definir módulos de funciones.

Reducción de expresiones

El computador evalúa una expresión reduciéndola a su forma equivalente más simple e imprimiendo su resultado. Vamos a ve un ejemplo para describir este proceso.

Vamos a reducir la siguiente expresión:

```
cuadrado (3 + 4)
```

Pero antes recordemos como estaría definida la función cuadrado:

```
cuadrado::Int->Int
cuadrado x = x * x
```


Reducción de expresiones

Una forma sería:

```
cuadrado (3+4)
    cuadrado 7      (+)
    7 * 7           (cuadrado)
    49              (*)
```

Esta forma de evaluar una función recibe el nombre de evaluación por valor, evaluación ansiosa o impaciente y consiste en reducir primero las expresiones que manejan valores para luego pasarlas por valor a las funciones.

Reducción de expresiones

Otra forma sería:

```
cuadrado (3+4)
    (3 + 4) * (3 + 4)     (cuadrado)
    7 * (3 + 4)           (+)
    7 * 7                 (+)
    49                    (*)
```

Esta otra forma recibe el nombre de evaluación por nombre o evaluación perezosa o lenta (lazy), de esta forma primero se hacen los remplazos correspondientes a las definiciones de las funciones y tan solo se evalúan cuando ya es necesario. No importa la forma que utilizamos, siempre el resultado final es el mismo. El proceso para evaluar una expresión básicamente es simple: sustituir y simplificar usando reglas primitivas y reglas definidas por el programador en forma de definiciones.

Reducción de expresiones

Veamos otro ejemplo, recordemos la función factorial:

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

Con esta definición en mente reduzcamos la siguiente expresión:

```
factorial 3
    3 * (factorial 2)                (factorial)
    3 * (2 * factorial 1)            (factorial)
    3 * (2 * (1 * factorial 0))      (factorial)
    3 * (2 * (1 * 1))                (factorial)
    3 * (2 * 1)                      (*)
    3 * 2                            (*)
    6                                (*)
```


Reducción de expresiones

Una expresión es canónica o está en forma normal si no puede reducirse. Algunas expresiones no pueden reducirse del todo.

Por ejemplo, si / es la operación de división, entonces la expresión 1 / 0 no puede reducirse.

Para estos casos existe un símbolo denominado **bottom (^)** que significa valor indefinido.

De esta manera, podemos decir que TODA expresión denota un valor.

Expresado en otras palabras en programación funcional siempre se espera que toda definición genere un valor y que este valor corresponda con la signatura de la declaración.

Reducción de expresiones

Haskell reduce las expresiones de manera perezosa y aunque a priori podemos decir que es una forma más larga de evaluación tiene la ventaja de poder manejar ciertas expresiones infinitas. Veamos un ejemplo de este caso, con las siguientes funciones:

```
infinito :: Integer
infinito = 1 + infinito

tres :: Integer->Integer
tres x = 3
```

Si elegimos reducirla haciendo evaluación por valor la evaluación no terminaría nunca, ocurriría lo siguiente:

```
tres infinito
    tres (1 + infinito)          (infinito)
    tres (1 + (1 + infinito)     (infinito)
    ...
```


Reducción de expresiones

Si elegimos hacerlo por nombre:

```
tres infinito
  3       (tres)
```


Tipos de datos

Existen dos clases de tipos:

- tipos básicos
- tipos compuestos o derivados.

Los tipos básicos son aquellos cuyos valores son primitivos, por ejemplo:

- Int: subconjunto de los enteros (normalmente, los enteros representables por una palabra del procesador): 2, 1...
- Integer: enteros (precisión absoluta)
- Char: caracteres: 'a','5', '\n', '\xF4'...
- Float/Double: números en punto flotante de simple/doble precisión: 3.14159, 2, 2.5e+2...
- String: cadenas de caracteres: "hola", …
- Bool: booleanos: True, False

Tipos de datos

Los tipos derivados o compuestos son aquellos cuyo valores se construyen de otros tipos. Por ejemplo:

- (Int, Char), el tipo de pares de valores donde la primer componente es un número y la segunda es un caracter.
- (Int->Int), el tipo de funciones cuyos argumentos son números y retornan otro número.
- [Int], una lista de números.

Cada tipo tiene asociado un cierto conjunto de operaciones.

Operadores

Cada tipo de dato tiene unos operadores asociados, recordemos que los operadores son trabajados como funciones, veamos la siguiente lista:

Booleanos
Se representan por el tipo "Bool" y contienen dos valores: "True" y "False". El estándar prelude incluye varias funciones para manipular valores booleanos: (&&),(||) y not.

- x && y es True si y sólo si x e y son True
- x || y es True si y sólo si x ó y ó ambos son True
- not x es el valor opuesto de x (not True = False,not False = True)
- También se incluye una forma especial de expresión condicional que permite seleccionar entre dos alternativas dependiendo de un valor booleano:

```
if expresion_booleana then x else y
```

- Si la expresión booleana expresion_booleana es True devuelve x, si es False, devuelve y. Obsérvese que una expresión de ese tipo sólo es aceptable si expresion_booleana es de tipo Bool y x e y son del mismo tipo.

Operadores

Enteros
Representados por el tipo "Int", se incluyen los enteros positivos y negativos tales como el 273, el 0 ó el 383. Como en muchos sistemas, el rango de los enteros utilizables está restringido.

También se puede utilizar el tipo Integer que denota enteros sin límites superior ni inferior. En el estándar prelude se incluye un amplio conjunto de operadores y funciones que manipulan enteros:

- (+) suma.
- (*) multiplicación.
- (-) resta.
- (^) potenciación.
- negate menos unario (la expresión "x" se toma como "negate x")
- div división entera
- rem resto de la división entera.
- Siguiendo la ley: (x `div` y)*y + (x `rem` y) == x

Operadores

Enteros

- mod módulo, como rem sólo que el resultado tiene el mismo signo que el divisor.
- odd devuelve True si el argumento es impar
- even devuelve True si el argumento es par.
- gcd máximo común divisor.
- lcm mínimo común múltiplo.
- abs valor absoluto
- signum devuelve -1, 0 o 1 si el argumento es negativo, cero ó positivo, respectivamente.

Operadores

Flotantes
Representados por el tipo "Float", los elementos de este tipo pueden ser utilizados para representar fraccionarios así como cantidades muy largas o muy pequeñas.

Sin embargo, tales valores son sólo aproximaciones a un número fijo de dígitos y pueden aparecer errores de redondeo en algunos cálculos que empleen operaciones en punto flotante.

Un valor numérico se toma como un flotante cuando incluye un punto en su representación o cuando es demasiado grande para ser representado por un entero.

También se puede utilizar notación científica; por ejemplo 1.0e3 equivale a 1000.0, mientras que 5.0e2 equivale a 0.05.

Operadores

Flotantes

El estándar prelude incluye también múltiples funciones de manipulación de flotantes:

- pi
- exp
- log
- sqrt
- sin
- cos
- tan
- asin
- acos
- atan ....

Listas en programación funcional

Una lista es una estructura de datos que representa un conjunto de datos de un mismo tipo. Su declaración es muy simple, ejemplo:

- [Int]: Representa una lista de enteros [4,5,9,25,60]
- [Char]: Representa una lista de chars ['l','i','n','u','x']
- [Bool]: Representa una lista de valores booleanos [True, False, True]
- [String]: Representa una lista de strings ["buenas","noches"]
- [Float]: Representa una lista de flotantes [2.5,5.8,4.3,7.1]

Si a es un tipo cualquiera, entonces [a] representa el tipo de listas cuyos elementos son valores de tipo a. Es importante para el manejo de listas separar el primer elemento (cabeza), del resto(cola) de la siguiente manera(x:xs).

x representa la cabeza y xs la cola.

Listas en programación funcional

Hay varias formas de escribir expresiones de listas:

- La forma más simple es la lista vacía, representada mediante [].
- Las listas no vacías pueden ser construidas enunciando explícitamente sus elementos, por ejemplo [1,3,10] o añadiendo un elemento al principio de otra lista utilizando el operador de construcción (:). Estas notaciones son equivalentes:

  [1,3,10] = 1:[3,10] = 1:(3:[10]) = 1:(3:(10:[]))

- El operador (:) es asociativo a la derecha, de forma que 1:3:10:[] equivale a (1:(3:(10:[]))), una lista a cuyo primer elemento es 1, el segundo 3 y el último 10.

Listas en programación funcional

El standar prelude incluye un amplio conjunto de funciones de manejo de listas, por ejemplo:

- length xs devuelve el número de elementos de xs.
- xs ++ ys devuelve la lista resultante de concatenar xs e ys.
- concat xss devuelve la lista resultante de concatenar las listas de xss.
- map f xs devuelve la lista de valores obtenidos al aplicar la función f a cada uno de los elementos de la lista xs.

Listas en programación funcional

Ejemplos:

```
? length [1,3,10]
3
? [1,3,10] ++ [2,6,5,7]
[1, 3, 10, 2, 6, 5, 7]
? concat [[1], [2,3], [], [4,5,6]]
[1, 2, 3, 4, 5, 6]
? map fromEnum ['H', 'o', 'l', 'a']
[104, 111, 108, 97]
?
```

Obsérvese que todos los elementos de una lista deben ser del mismo tipo. La expresión ['a',2,False] no está permitida en Haskell

Listas en programación funcional

Secuencias*:*

La notación de las secuencias aritméticas permite generar una gran cantidad de listas útiles. Existen cuatro formas de expresar secuencias aritméticas:

[m..] Produce una lista (potencialmente infinita) de valores que comienzan con m y se incrementan en pasos simples.

Ejemplo:

```
? [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, etc...
```


Listas en programación funcional

Secuencias*:*

La notación de las secuencias aritméticas permite generar una gran cantidad de listas útiles. Existen cuatro formas de expresar secuencias aritméticas:

[m..] Produce una lista (potencialmente infinita) de valores que comienzan con m y se incrementan en pasos simples.

Ejemplo:

```
? [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, etc...
```


Listas en programación funcional

Secuencias*:*

[m..n] Produce la lista de elementos desde m hasta n, con incrementos en pasos simples. Si m es menor que n devuelve la lista vacía.

Ejemplos:

```
? [-3..3]
[-3, -2, -1, 0, 1, 2, 3]
? [1..1]
[1]
? [9..0]
[]
```


Listas en programación funcional

Secuencias*:*

[m,m'..] Produce la lista (potencialmente infinita) de valores cuyos dos primeros elementos son m y m'. Si m es menor que m' entonces los siguientes elementos de la lista se incrementan en los mismos pasos. Si m es mayor que m' entonces los incrementos serán negativos. Si son iguales se obtendrá una lista infinita cuyos elementos serán todos iguales.

Ejemplos:

```
? [1,3..]
[1, 3, 5, 7, 9, 11, 13, etc...
? [0,0..]
[0, 0, 0, 0, 0, 0, 0, etc...
? [5,4..]
[5, 4, 3, 2, 1, 0, -1, etc...
```


Listas en programación funcional

Secuencias*:*

[m,m'..n] Produce la lista de elementos [m,m',..] hasta el valor límite n. Los incrementos vienen marcados por la diferencia entre m y m'.

Ejemplos:

```
? [1,3..12]
[1, 3, 5, 7, 9, 11]
? [0,0..10]
[0, 0, 0, 0, 0, 0, 0, etc...
? [5,4..1]
[5, 4, 3, 2, 1]
```


Listas en programación funcional

Secuencias*:*

Las secuencias no están únicamente restringidas a enteros, pueden emplearse con elementos enumerados como caracteres, flotantes, etc.

Ejemplos:

```
? ['0'..'9'] ++ ['A'..'Z']
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
? [1.2, 1.35 .. 2.00]
[1.2, 1.35, 1.5, 1.65, 1.8, 1.95]
```


Listas en programación funcional

Listas por comprehensión*:*

Son listas que no usan cabeza y cola, sino ciertos argumentos para definir los elementos que pertenecen a ella, de esta manera resolvemos problemas de una manera muy elegante y potente.

La notación de listas por comprehensión permite declarar de forma concisa una gran cantidad de iteraciones sobre listas. El formato básico de la definición de una lista por comprehensión es:

[<expr> | <cualif_1>, <cualif_2> . . . <cualif_n>]

Cada <cualif_i> es un cualificador.

Listas en programación funcional

Listas por comprehensión*:*

Cualificadores generadores: Un cualificador de la forma pat<-exp es utilizado para extraer cada elemento que encaje con el patrón pat de la lista exp en el orden en que aparecen los elementos de la lista.

Ejemplos:

```
? [x | x <- [1 .. 12]]
[1,2,3,4,5,6,7,8,9,10,11,12]
? [x+x | x <- [1 .. 12]]
[2,4,6,8,10,12,14,16,18,20,22,24]
? [x*x | x <- [1..10]]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```


Listas en programación funcional

Listas por comprehensión*:*

Filtros generadores: Una expresión de valor booleano, el significado de una lista por comprensión con un único filtro podría definirse como:

[e |condicion] = if condition then [e] else []

Esta forma de lista por comprehensión resulta útil en combinación con generadores.

Ejemplos:

```
? [x*x |x<-[1..10], even x ]
[4,16,36,64,100]
? [x | x <- [1..10], mod x 5 ==0]
[5,10]
```


Listas en programación funcional

Listas por comprehensión*:*

Si aparecen varios cualificadores hay que tener en cuenta que:

- Las variables generadas por los cualificadores posteriores varían más rápidamente que las generadas por los cualificadores anteriores:

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

- Los cualificadores posteriores podrían utilizar los valores generados por los anteriores:

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


Listas en programación funcional

Listas por comprehensión*:*

- Las variables definidas en cualificadores posteriores ocultan las variables definidas por cualificadores anteriores. Las siguientes expresiones son listas definidas por comprehensión de forma válida. Sin embargo, no es una buena costumbre reutilizar los nombres de las variables ya que dificulta la comprensión de los programas.

```
? [ x | x<-[[1,2],[3,4]], x<-x ]
[1, 2, 3, 4]

? [ x | x<-[1,2], x<-[3,4] ]
[3, 4, 3, 4]
```


Listas en programación funcional

Listas por comprehensión*:*

- Un cambio en el orden de los cualificadores puede tener un efecto directo en la eficiencia. Los siguientes ejemplos producen el mismo resultado, pero el primero utiliza más reducciones debido a que repite la evaluación de "even x" por cada valor posible de "y".

```
? [ (x,y) | x<-[1..3], y<-[1..2], even x ]
[(2,1), (2,2)]
(110 reductions, 186 cells)

? [ (x,y) | x<-[1..3], even x, y<-[1..2] ]
[(2,1), (2,2)]
(62 reductions, 118 cells)
```


Listas en programación funcional

Algunos ejemplos:

- Sumar los elementos de una lista. En este caso, el caso base es que la lista se encuentre vacía y devuelve 0, mientras tanto que siga sumando los elementos con la operación recursiva.

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

Main> sumar [5,4,7,8]
24
```


Listas en programación funcional

Algunos ejemplos:

- Invertir una lista: El operador Ord me sirve para indicar que representa a cualquier tipo de dato.

```
invertir::Ord a=>[a]->[a]
invertir [ ] = [ ]
invertir (x:xs) = (invertir xs)++[x]

Main> invertir [5,4,7,8]
[8,7,4,5]
```


Listas en programación funcional

Algunos ejemplos:

- Comparar si 2 listas son iguales:

```
igualLista:: Eq a => [a]->[a]->Bool
igualLista l1 l2 = l1 == l2

Main> igualLista ["Hola","Mundo"] ["Mundo","Hola"]
False
```


Listas en programación funcional

Algunos ejemplos:

- Comprobar si una lista esta ordenada: En este caso ‘y’ representa al 2do elemento de la lista.

```
lista_ordenada::Ord a=>[a]->Bool
lista_ordenada [] = True
lista_ordenada [_] = True
lista_ordenada (x:y:xs) = (x<=y) && lista_ordenada
(y:xs)

Main> lista_ordenada ['a','b','c','d']
True
```


Listas en programación funcional

Algunos ejemplos:

- Mostrar el elemento que se encuentra en cierta ubicación: Como en un array el 1er elemento esta en la ubicación 0.

```
mostrar_ubicacion::Ord a=>[a]->Int->a
mostrar_ubicacion l n = l!!n

Main> mostrar_ubicacion [15,25,26,28] 2
26
```


Listas en programación funcional

Algunos ejemplos:

- Mayor elemento de una lista:

```
mayor::[Int]->Int
mayor (x:xs)
    | x > mayor(xs) = x
    | otherwise = mayor(xs)

Main> mayor [78,24,56,93,21,237,46,74,125]
237
```


Listas en programación funcional

Algunos ejemplos:

- Contar cuantos elementos pares hay en una lista. Estamos diciendo que x pertenece a la lista y además debe cumplir la condición de ser par. Como en varios lenguajes el length cuenta los elementos.

```
contarpares::[Int]->Int
contarpares []=0
contarpares lista= length [x | x <- lista, mod x 2 ==0]

Main> contarpares [5,4,7,8]
2
```


Listas en programación funcional

Algunos ejemplos:

- Devolver los cuadrados de una lista:

```
cuadrados::[Int]->[Int]
cuadrados [ ] = [ ]
cuadrados l = [x*x| x <- l]

Main> cuadrados [1..10]
[1,4,9,16,25,36,49,64,81,100]
```


Listas en programación funcional

Algunos ejemplos:

- Devolver una lista de numeros primos de 1 a n: Para ello debemos crear nuestra función para saber si un numero es primo o no y después la aplicamos a la lista por comprensión:

```
divisible::Int->Int->Bool
divisible x y = (mod x y) ==0

divisibles::Int->[Int]
divisibles x = [y | y <-[1..x],divisible x y]

esPrimo::Int->Bool
esPrimo n = length (divisibles n) <=2
```


Listas en programación funcional

Algunos ejemplos:

```
primos::Int->[Int]
primos n = [x | x <-[1..n],esPrimo x]

Main> primos 100
[1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,
71,73,79,83,89,97]
```


Listas en programación funcional

Funciones útiles en listas

!!: Retorna el elemento ubicado en la posición n, empezando desde cero.

```
Main> ["maritza","celeste","nadia","maria","julia"]!!1
"celeste"
```

head: Retorna el primer elemento de la lista.

```
Main> head [11,1,1985,22,8,2007]
11
```

last: Retorna el último elemento de la lista.

```
Main> last [11,1,1985,22,8,2007]
2007
```


Listas en programación funcional

Funciones útiles en listas

tail: Retorna todos los elementos menos el primero.

```
Main> tail [11,1,1985,22,8,2007]
[1,1985,22,8,2007]
```

init: Retorna todos los elementos menos el último.

```
Main> init [11,1,1985,22,8,2007]
[11,1,1985,22,8]
```

length: Retorna el número de elementos de la lista.

```
Main> length [11,1,1985,22,8,2007]
6
```


Listas en programación funcional

Funciones útiles en listas

take: Retorna los primeros n elementos de la lista.

```
Main> take 2 [11,1,1985,22,8,2007]
[11,1]
```

drop: Retorna los elementos de la lista, excepto los n primeros.

```
Main> drop 2 [11,1,1985,22,8,2007]
[1985,22,8,2007]
```

takeWhile : Más potente que take pues puede retornar ciertos tipos de datos indicados.

```
Main> takeWhile (<=15) [1..30]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```


Listas en programación funcional

Funciones útiles en listas

dropWhile: Más potente que drop pues puede retornar ciertos tipos de datos indicados.

```
Main> dropWhile (<=15) [1..30]
[16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]
```

reverse: Invierte una lista.

```
Main> reverse [11,1,1985,22,8,2007]
[2007,8,22,1985,1,11]
```

concat: Toma ciertos elementos o listas y las retorna en una sola lista.

```
Main> concat ["open","source","solutions"]
"opensourcesolutions"
```


Listas en programación funcional

Funciones útiles en listas

words: Retorna una lista de strings de acuerdo a los espacios en blanco de un string.

```
Main> words ” I like to use Debian GNU/Linux”
["I","like","to","use","Debian","GNU/Linux"]
```

unwords: Retorna un string de una lista de strings.

```
Main> unwords ["I","like","to","use","Debian","GNU/Linux"]
“I like to use Debian GNU/Linux”
```

elem: Retorna si un elemento esta o no en la lista

```
Main> elem ‘t’ ['a','f','r','h','t']
True
```


Listas en programación funcional

Funciones útiles en listas

notElem: Lo opuesto a elem.

```
Main> notElem 't' ['a','f','r','h','t']
False
```