PROGRAMACIÓN FUNCIONAL

Aplicación de conceptos: Listas

Listas

- Definición de listas
- Funciones sobre listas
- Propiedades de las listas
- Secuencias aritméticas
- Listas por comprensión

Definición de listas

- Dado un tipo cualquiera a, definimos inductivamente al conjunto [a] con las siguientes reglas:
  - [ ] :: [a]
  - si x :: a y xs :: [a]
    entonces (x:xs) :: [a]
- Notación:

$$[\ x_1, x_2, x_3\ ] = (x_1 : (x_2 : (x_3 : [\ ])))$$

Funciones sobre listas

- Siguiendo el patrón de recursión

```
len :: [a] -> Int
len [ ] = 0
len (x:xs) = 1 + len xs

append :: [a] -> [a] -> [a]
append [ ] ys = ys
append (x:xs) ys = x : (append xs ys)

(++) = append
```

Funciones sobre listas

- Sin seguir el patrón de recursión

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs

null :: [a] -> Bool
null [ ] = True
null (x:xs) = False
```

Funciones sobre listas

- Más funciones siguiendo el patrón de recursión

```
sum :: [Int] -> Int
sum [ ] = 0
sum (n:ns) = n + sum ns

prod :: [ Int ] -> Int
prod [ ] = 1
prod (n:ns) = n * prod ns
```

  - ¿por qué puede definir (sum [ ]) y (prod [ ]) de esta manera?

Funciones sobre listas

- Más funciones siguiendo el patrón de recursión

```
upperl :: [Char] -> [Char]
upperl [ ] = [ ]
upperl (c:cs) = (upper c) : (upperl cs)

novacias :: [ [ a ] ] -> [ [ a ] ]
novacias [ ] = [ ]
novacias (xs:xss) = if null xs then novacias xss
                                else xs : novacias xss
```

Funciones sobre listas

- Siguiendo otro patrón de recursión

```
maximum :: [ a ] -> a
maximum [ x ] = x
maximum (x:xs) = x `max` maximum xs

last :: [ a ] -> a
last [ x ] = x
last (x:xs) = last xs
```

  - ¿puede establecer cuál es el patrón?
  - ¿por qué (maximum [ ]) no está definida?

Funciones sobre listas

- Otras funciones

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

insert :: a -> [ a ] -> a
insert x [ ] = [ x ]
insert x (y:ys) = if x <= y then x : (y : ys)
                            else y : (insert x ys)
```

Secuencias aritméticas

- Notación especial para escribir listas
  - Para elementos de la clase Enum
    - con operaciones toEnum y fromEnum
    - enteros, flotantes, caracteres, booleanos
  - Ejemplos:
    - [1..10] = [1,2,3,4,5,6,7,8,9,10]
    - [10..1] = [ ]
    - ['a'..'e'] = ['a','b','c','d','e',]
    - [False .. True] = [False,True]

Secuencias aritméticas

- Pueden tener un paso mayor o menor a uno
  - Ejemplos:
    - [2,4..10] = [2,4,6,8,10]
    - [10,9..1] = [10,9,8,7,6,5,4,3,2,1]
    - ['a','d'..'z'] = ['a','d','g','j','m','p','s','v','y']
    - [True,False .. True] = [True]

Secuencias aritméticas

- Definición

[a..b] = enumFromTo a b

[a,b..c] = enumFromThenTo a b c

```
enumFromTo a b = enumFromStepTo a 1 b

enumFromThenTo a b c = enumFromStepTo a (b -a) c

enumFromStepTo :: Int -> Int -> [Int]
enumFromStepTo a s b =
        if (s > 0 && a>b) || (s <= 0 && a<b)
         then [ ]
         else a : enumFromStepTo (a+s) s b
```

Listas por comprensión

- Notación especial para escribir listas
  - Similar a la notación de conjuntos por comprensión.
  - ¡El orden importa!
  - Ejemplos:
    - [x^2 | x <- [1..10], even x]

      = [4,16,36,64,100]
    - [(i,j) | i <- [1..3], j <- ['a','b']]

      = [(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]

Listas por comprensión

- Definición:

  $[\ e \mid qual_1\ ,\ \ldots\ ,\ qual_n\ ]$

  donde
  - e es una expresión cualquiera
  - $qual_i$ es un calificador que puede ser
    - un generador de la forma x <- lexp, siendo x una variable y lexp una expresión de tipo [a]
    - un test de la forma bexp, siendo bexp una expresión de tipo Bool

Listas por comprensión

- Observaciones
  - Las variables de un generador pueden aparecer:
    - en la expresión e
    - en cualquier lexp ó bexp a la derecha de su generador
  - Las variables NO pueden aparecer definidas por más de un generador
  - El orden de los calificadores es importante
  - Puede no haber ningún calificador

Listas por comprensión

- Significado (informal)
  - [e |] = [e]
  - [e | True, q_2, ..., q_k] = [e | q_2, ..., q_k]
  - [e | False, q_2, ..., q_k] = [ ]
  - [e | x <- [a_1,a_2,...,a_n], q_2, ..., q_k] =
    [e{a_1/x} | q_2{a_1/x}, ..., q_k{a_1/x}]
    ++ [e{a_2/x} | q_2{a_2/x}, ..., q_k{a_2/x}]
    ++ ...
    ++ [e{a_n/x} | q_2{a_n/x}, ..., q_k{a_n/x}]
- (**Notación:** e{v/x} significa e donde x fue reemplazado por v)

Listas por comprensión

- Ejemplo
  - [x^2 | x <- [1..6], even x]

    = [1^2 | even 1] ++ [2^2 | even 2]
    ++ [3^2 | even 3] ++ [4^2 | even 4]
    ++ [5^2 | even 5] ++ [6^2 | even 6]

    = [1^2 | False] ++ [2^2 | True]
    ++ [3^2 | False] ++ [4^2 | True]
    ++ [5^2 | False] ++ [6^2 | True]

    = [ ] ++ [2^2] ++ [ ] ++ [4^2] ++ [ ] ++ [6^2]

    = [4, 16, 36]

Listas por comprensión

- Ejemplo
  - [(i,j) | i <- [1..3], j <-['a','b']]

    = [(1,j) | j <-['a','b']]
    ++ [(2,j) | j <-['a','b']]
    ++ [(3,j) | j <-['a','b']]

    = ([(1,'a') |] ++ [(1,'b') |])
    ++ ([(2,'a') |] ++ [(2,'b') |])
    ++ ([(3,'a') |] ++ [(3,'b') |])

    = [(1,'a'), (1,'b'), (2,'a'), (2,'b'), (3,'a'), (3,'b')]

Listas por comprensión

- Otros ejemplos
  - spaces n = [' ' | i <- [1..n]]
  - divisores n = [d | d <- [1..n], d `divide` n]
  - d `divide` n = n `mod` d == 0
  - primo n = divisores n == [1,n]
  - prodCart xs ys = [(x,y) | x <- xs, y <- ys]

Resumen

- Distintas formas de escribir listas
- Funciones sobre listas
- Propiedades sobre listas