Lógica Computacional, 2019-2
Nota de laboratorio 2: Estructuras de datos

Manuel Soto Romero

13 de febrero de 2019
Facultad de Ciencias, UNAM

A diferencia de los lenguajes imperativos, en Haskell y en otros lenguajes funcionales, las estructuras de datos son definidas sin la posibilidad de poder alterar sus elementos debido a que el concepto de estado no está presente. Las principales estructuras de datos en Haskell son las tuplas y las listas y su manipulación consiste, en general, en procesarlas recursivamente y reconstruir la estructura hasta llegar al punto donde se debe realizar el cambio correspondiente. En esta nota se presenta la definición de estas estructuras y se presentan algunos mecanismos para representarlas y procesarlas.

1. Tuplas

Una tupla es una estructura de datos con las siguientes características:

- Sus elementos se delimitan por paréntesis y se separan por comas. Son estructuras de datos *heterogéneas*, es decir, sus elementos no necesariamente son del mismo tipo. Al número de elementos de una tupla se le conoce como *tamaño* de la tupla.

  ```
  (1,''Dos'',’3’,5.5)
  ```

- Las tuplas de tamaño 2 son conocidas como *pares*. Para acceder a los elementos de un par se tienen las funciones `fst` y `snd` que obtienen el primer y segundo elemento de la estructura respectivamente.

  ```
  (  1   ,   2   )
    fst     snd
  ```

- Al definir la firma de una función que recibe o regresa tuplas hay que especificar el tipo de cada entrada, delimitado por paréntesis y separando cada tipo por una coma.

  Tupla con números: `(Int, Float)`

  Si se requiere una tupla con elementos de cualquier tipo, pueden usarse *variables de tipo*. El tipo de estas variables se conoce[1] en tiempo de ejecución. A estas funciones se les llama *polimórficas*.

  ```
  (a, b)
  ```

- Algo importante es que, una vez que se establece el tamaño de una tupla, este no puede cambiar, es decir, no se puede construir una tupla de mayor tamaño a partir de otra.

[1] Instancia.

1.1. Vectores en $\mathbb{R}^2$

A continuación se define una biblioteca para trabajar con vectores cuyos elementos se encuentran en $\mathbb{R}^2$, se muestra el uso de pares y el uso de las respectivas funciones para acceder al primer y segundo elemento respectivamente.

Ejemplo 1. Definir una función `suma` tal que (`suma u v`) representa la suma de los vectores `u` y `v` denotados por pares. Por ejemplo:

```
suma (1,2) (3,4)          = (4,6)
suma (1.5,0) (0,1.5)      = (1.5,1.5)
suma (3.0,4.3) (2.3,5.2)  = (5.3,9.5)
```

Solución. Función `suma`:

```haskell
-- Función que suma dos vectores representados por pares.
suma :: (Float, Float) -> (Float, Float) -> (Float, Float)
suma u v = (fst u + fst v, snd u + snd v)
```

Listado de código 1: Suma de vectores

□

Ejemplo 2. Definir una función `productoPunto` tal que (`productoPunto u v`) representa el producto punto de los vectores `u` y `v` denotados por pares. Por ejemplo:

```
productoPunto (1,2) (3,4)          = 11.0
productoPunto (1.5,0) (0,1.5)      = 0.0
productoPunto (3.0,4.3) (2.3,5.2)  = 29.26
```

Solución. Función `productoPunto`:

```haskell
-- Función que realiza el producto punto de dos vectores representados
-- por pares.
productoPunto :: (Float, Float) -> (Float, Float) -> Float
productoPunto u v = fst u * fst v + snd u * snd v
```

Listado de código 2: Producto punto de vectores

□

Ejemplo 3. Definir una función `productoEscalar` tal que (`productoEscalar u k`) representa el producto del vector `u` denotado por un par por el escalar `k` de tipo flotante. Por ejemplo:

```
productoEscalar (1,2) 2        = (2.0,4.0)
productoEscalar (1.5,0) 3.0    = (3.5,0.0)
productoEscalar (3.0,4.3) 4.0  = (12.0,17.2)
```

Solución. Función `productoEscalar`:

```haskell
-- Función que realiza el producto punto de un vector representado por
-- un par y un escalar.
productoEscalar :: (Float, Float) -> Float -> (Float, Float)
productoEscalar u k = (fst u * k, snd u * k)
```

Listado de código 3: Producto punto de vectores

□

2. Listas

Las listas en Haskell se definen recursivamente como sigue:

Definición 1. Una lista se define como:

1. La lista vacía y se representa por `[]`.
2. Si `x` es un elemento de un conjunto A y `xs` es una lista con elementos en A, entonces `x:xs` es una lista también. A `x` se le llama cabeza y a `xs` el resto de la lista.
3. Son todas.

Observación. La función `:` es llamada *cons* y permite construir una lista dado un elemento y otra lista.

Algunas características particulares de las listas son:

- Sus elementos se delimitan por corchetes y se separan por comas. Son estructuras de datos *homogéneas*, es decir, todos sus elementos son del mismo tipo. Al número de elementos de una lista se le conoce como *longitud* de la lista.

  `[1,2,3,4]`

- Al definir la firma de una función que recibe o regresa listas hay que especificar el tipo de los elementos de la misma, delimitado por corchetes.

  Lista de números enteros: `[Int]`

  Si se requiere una lista con elementos de cualquier tipo, al igual que en las tuplas, pueden usarse variables de tipo.

  `[a]`

- A diferencia de las tuplas, sí es posible construir nuevas listas a partir de otras, ya sea mediante la función *cons* o mediante la concatenación de listas.

- La cadenas, son en realidad listas de caracteres, de manera que es equivalente escribir cadenas como `"Hola"` o como `['H','o','l','a']`, la primera forma es azúcar sintáctica[2] de la segunda.

[2] Modificación sintáctica para hacer una primitiva más fácil de leer o usar al usuario.

2.1. Representación de listas

Adicional a la notación de listas donde se delimitan los elementos por corchetes y se separan por comas, existen otras formas de definir listas como puede ser: mediante *rangos* o por *comprensión*.

2.1.1. Listas mediante rangos

Para definir listas de números pueden usarse rangos que indiquen algunos elementos de la lista. A continuación se presentan algunos ejemplos que pueden introducirse directamente en el intérprete.

Ejemplo 4. Listas mediante rangos de `n` a `m`.

```
Prelude> [1..5]
[1,2,3,4,5]
```

□

Ejemplo 5. Listas mediante rangos especificando los primeros y último elemento. Esta notación permite inferir algunos patrones sencillos.

```
Prelude> [2,4..10]
[2,4,6,8,10]
```

□

Ejemplo 6. Listas mediante rangos especificando el primer elemento. Esta forma de crear listas *pseudo infinitas* es útil cuando se trabaja con evaluación perezosa y se toma cada elemento de la lista conforme se vaya necesitando.

```
Prelude> let a = [1..]
Prelude> a !! 3
2
```

□

2.1.2. Listas por comprensión

Al igual que en Teoría de Conjuntos, en Haskell es posible definir listas mediante definiciones por comprensión a partir de los elementos de otras listas previamente construidas. Por ejemplo, el siguiente conjunto en matemáticas:

$$\{2x \mid x \in \{0,1,2,3,4,5\}\} = \{0,2,4,6,8,10\}$$

Puede representarse en Haskell, mediante listas por comprensión como se muestra en el Ejemplo 7.

Ejemplo 7. Lista de números pares del 0 al 10 definida por comprensión.

```
Prelude> [2*x | x <- [0,1,2,3,4,5]]
[0,2,4,6,8,10]
```

□

En el Ejemplo 7, la variable `x` obtiene cada elemento de la lista `[0,1,2,3,4,5]` y lo multiplica por dos. A la expresión completa `x <- [0,1,2,3,4,5]` se le conoce como *generador*. También es posible añadir condiciones y otros generadores o construir otras estructuras como pueden ser pares.

Ejemplo 8. Lista de pares tales que el producto de sus entradas siempre es un número par. Las entradas van del 0 al 3.

```
Prelude> [(x,y) | x <- [0..3],y <- [0..3], (mod (x*y) 2) == 0]
[(0,0),(0,1),(0,2),(0,3),(1,0),(1,2),(2,0),(2,1),(2,2),(2,3),(3,0),(3,2)]
```

□

2.2. Manipulación de listas

Algunas funciones predefinidas para manipular listas:

`head` Obtiene el primer elemento de una lista.

`tail` Obtiene el resto de la lista.

`null`	Indica si la lista recibida es vacía.
`length`	Obtiene la longitud de una lista.
`take`	Toma los primeros n elementos de una lista.
`drop`	Elimina los primeros n elementos de una lista.
`elem`	Indica si un elemento pertenece a una lista.

A continuación se presentan algunas funciones para trabajar con listas, para manipular las mismas se hace uso de recursión. Como recordatorio, las funciones recursivas se definen de acuerdo a:

1. Un caso base que permite terminar con la recursión.
2. Un caso recursivo que permite descomponer el problema en partes más pequeñas[3].

Ejemplo 9. Definir una función `longitud` tal que (`longitud l`) representa la longitud de una lista. Se usan en este ejemplo, variables de tipo. Por ejemplo:

```
longitud []             = 0
longitud ['a']          = 1
longitud [5.0,1.3,2.7] = 3
```

Solución. Función `longitud`:

```
1 -- Función que obtiene la longitud de una lista.
2 longitud :: [a] -> Int
3 longitud l = if null l then 0 else 1 + longitud (tail l)
```

Listado de código 4: Longitud de una lista

□

Ejemplo 10. Definir una función `quita` tal que (`quita n l`) representa a la lista `l` sin sus primeros `n` elementos. Se usan en este ejemplo, variables de tipo. Por ejemplo:

```
quita 0 [1,2,3] = [1,2,3]
quita 2 []      = []
quita 2 "Hola"  = "la"
```

Solución. Función `quita`:

```
1 -- Función que quita los primeros n elementos de una lista.
2 quita :: Int -> [a] -> [a]
3 quita n l = if n == 0 then l else quita (n - 1) (tail l)
```

Listado de código 5: Lista sin los primeros n elementos

[3] Estrategia conocida como *Divide y vencerás*.

□

Ejemplo 11. Definir una función `toma` tal que (`toma n l`) representa los primeros `n` elementos de una lista `n`. Se usan en este ejemplo, variables de tipo. Por ejemplo:

```
toma 0 [1,2,3] = []
toma 2 []      = []
toma 2 "Hola"  = "Ho"
```

Solución. Función `toma`:

```
1 -- Función que toma los primeros n elementos de una lista.
2 toma :: Int -> [a] -> [a]
3 toma n l = if n == 0 then [] else (head l):(toma (n - 1) (tail l))
```

Listado de código 6: Primeros n elementos de una lista

□

Ejemplo 12. Definir una función `contiene` tal que (`contiene e l`) indica si el elemento `e` es elemento de la lista `l`. Se usan en este ejemplo, variables de tipo. Por ejemplo:

```
contiene 0 [1,2,3]    = False
contiene 2 []         = False
contiene 'o' "Hola"   = True
```

Observación 1. Para este ejemplo debe agregarse una restricción adicional, que indique que la lista debe tener elementos que puedan ser comparados por igual. Los tipos que cumplen con esta condición, pertenecen a una *clase* llamada `Eq` que debe indicarse antes de listar la lista de parámetros de la función (línea 3 del Listado de código 7).

Una *clase* representa a un grupo de tipos con características similares, a lo largo de estas notas se irán presentando diversas clases, conforme sean utilizadas. Para añadir una clase, deben usarse variables de tipo y se debe anteponer a la firma de la función, la clase a la que pertenecerá dicha variable seguida del símbolo `=>`.

Solución. Función `contiene`:

```
1 -- Función que indica si un elemento pertenece a una lista.
2 contiene :: Eq a => a -> [a] -> Bool
3 contiene e l = (head l) == e || (contiene e (tail l))
```

Listado de código 7: Indica si un elemento pertenece a una lista

□

3. Reconocimiento de patrones

Las funciones definidas en la sección 1.2 hacen uso de condicionales que verifican el valor de una expresión para poder realizar recursión. Sin embargo, en Haskell es posible realizar esto a través de la técnica conocida como *reconocimiento de patrones*[4] que en lugar de hacer verificaciones por valor, lo hace a través de la estructura de una expresión.

Con esta técnica se pueden reconocer patrones sobre números, tuplas, listas, entre otros. A continuación se definen algunas funciones que hacen uso de esta técnica.

Ejemplo 13. Definir una función `sumaDigitos` tal que (`sumaDigitos n`) representa la suma de los dígitos de un número entero. Se usa en este ejemplo, la técnica de reconocimiento de patrones, para el caso base se usa el patrón `0` y para el caso recursivo el patrón `n`. Por ejemplo:

```
sumaDigitos 0    = 0
sumaDigitos 8    = 8
sumaDigitos 1729 = 19
```

Solución. Función `sumaDigitos`:

```
1 -- Función que suma los dígitos de un número entero positivo.
2 sumaDigitos 0 = 0
3 sumaDigitos n
4    | n < 10 = n
5    | otherwise = (mod n 10) + (div n 10)
```

Listado de código 8: Suma los dígitos de un número entero

□

Definición 2. A los números enteros u, v, w que satisfacen la ecuación diofantina de segundo grado $u^2 + v^2 = w^2$ se les llama *terna pitagórica.*

Ejemplo 14. Definir una función `esTernaPitagorica` tal que (`esTernaPitagorica x`) indica si los elementos de una tupla de tamaño 3 forman una terna pitagórica. Se usa en este ejemplo, la técnica de reconocimiento de patrones, para reconocer la tupla se usa el patrón `(u,v,w)`. Por ejemplo:

```
esTernaPitagorica (3,4,5) = True
esTernaPitagorica (1,2,3) = False
```

Solución. Función `esTernaPitagorica`:

```
1 -- Función que indica si una tupla de tamaño 3 forma una terna
2 -- pitagórica.
3 esTernaPitagorica :: (Int,Int,Int) -> Bool
4 esTernaPitagorica (u,v,w) = u^2 + v^2 == w^2
```

Listado de código 9: Indica si los elementos de una tupla forman una terna pitagórica

[4] Del inglés *pattern matching.*

□

Ejemplo 15. Definir una función `sumaLista` tal que (`sumaLista l`) representa la suma de los elementos de una lista. Se usa en este ejemplo, la técnica de reconocimiento de patrones, para reconocer el caso base se usa el patrón `[]` que representa a la lista vacía y para el caso base se usa el patrón `(x:xs)` que representa a la lista con cabeza y resto. Por ejemplo:

```
sumaLista []      = 0
sumaLista [1]     = 1
sumaLista [1,2,3] = 6
```

Solución. Función `sumaLista`:

```
1 -- Función que suma los elementos de una lista.
2 sumaLista :: Num a => [a] -> a
3 sumaLista [] = 0
4 sumaLista (x:xs) = x + sumaLista xs
```

Listado de código 10: Suma los elementos de una lista

Observación 2. La clase `Num` engloba números con operaciones básicas como son la suma, resta, multiplicación, entre otros. Los tipos de datos `Int`, `Integer`, `Float` y `Double` forman parte de esta clase.

□

Ejercicio 1. Definir una función `multiplicaLista` tal que (`multiplicaLista l`) representa el producto de los elementos de una lista. ¿Qué habría que cambiar a la función definida en el Listado de código 10?

4. Funciones de orden superior

Como se mencionó en la Nota de laboratorio 1, las funciones de Haskell actúan como cualquier otro valor en el lenguaje, por ejemplo, pueden pasarse como parámetro a otras funciones. Existen algunas funciones de este tipo para trabajar con listas, entre las que se encuentran: `map` y `filter` y las funciones de plegado[5] `foldr` y `foldl`. A continuación se muestra la implementación de estas funciones y algunos ejemplos de uso.

4.1. Función `map`

Esta función aplica a cada elemento de una lista la función que recibe como parámetro. Se define como sigue:

[5] Programación origami.

```haskell
-- Función que aplica una función a cada elemento de una lista.
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

Listado de código 11: Implementación de map

Ejemplo 16. Definir una función `meses` tal que (`meses 1`) transforma una lista de números enteros en una lista de cadenas que representan el mes correspondiente. Por ejemplo:

```
meses []      = []
meses [10,12] = ["Octubre", "Diciembre"]
```

Solución. Función `meses`:

```haskell
-- Función que obtiene el nombre del mes representado por el número
-- recibido como parámetro.
nombreMes :: Integer -> String
nombreMes n
   | n == 1  = "Enero"
   | n == 2  = "Febrero"
   | n == 3  = "Marzo"
   | n == 4  = "Abril"
   | n == 5  = "Mayo"
   | n == 6  = "Junio"
   | n == 7  = "Julio"
   | n == 8  = "Agosto"
   | n == 9  = "Septiembre"
   | n == 10 = "Octubre"
   | n == 11 = "Noviembre"
   | n == 12 = "Diciembre"
   | otherwise = error "Mes inválido"

-- Función que transforma una lista de números enteros en una lista
-- de cadenas que representan el mes correspondiente.
meses :: [Int] -> [String]
meses xs = map nombreMes xs
```

Listado de código 12: Transforma una lista de números enteros en una lista de cadenas que representan el mes correspondiente.

Se hace uso de la función `nombreMes` definida en la Nota de laboratorio 1 (líneas 1 a 17). La función `meses` consiste de aplicar `nombreMes` a cada elemento de la lista recibida como parámetro.

□

4.2. Función `filter`

Esta función recibe una función que verifica una propiedad y deja en la lista aquellos elementos que cumplan la misma. Se define como sigue:

```haskell
-- Función que aplica filtra los elementos de una lista dada una
-- condición.
filter :: (a -> Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs)
    | (f x) = x:(filter f xs)
    | otherwise = filter f xs
```

Listado de código 13: Implementación de filter

Ejemplo 17. Definir una función `ternasPitagoricas` tal que (`ternasPitagoricas 1`) filtra las tuplas de tamaño 3 que cumplen con la propiedad de ser una terna pitagórica. Por ejemplo:

```
ternasPitagoricas []                  = []
ternasPitagoricas [(1,2,3),(3,4,5)] = [(3,4,5)]
```

Solución. Función `ternasPitagoricas`:

```haskell
-- Función que filtra las tuplas de tamaño 3 que cumplen con la
-- propiedad de ser una terna pitagórica.
ternasPitagoricas :: [(Int,Int,Int)] -> Bool
ternasPitagoricas xs = filter esTernaPitagorica xs
```

Listado de código 14: filtra las tuplas de tamaño 3 que cumplen con la propiedad de ser una terna pitagórica

Se hace uso de la función `esTernaPitagorica` definida en la Listado de código 9. De esta forma, la función `ternasPitagoricas` consiste de filtrar usando la función `esTernaPitagorica` cada elemento de la lista recibida como parámetro.

□

4.3. Funciones `foldr` y `foldl`

La función `sumaLista` definida en el Listado de código 10, engloba un patrón muy común al recorrer una lista que consiste en ir aplicando de forma encadenada una función a los elementos de una lista, en este caso una suma. Este patrón es tan común, que existen dos funciones para realizar esta tarea, ya sea recorriendo de derecha izquierda (`foldr`) o de izquierda a derecha (`foldl`).

Estas funciones requieren de tres parámetros:

1. Una función a aplicar.
2. Un valor a regresar cuando se tenga un caso base (lista vacía).
3. La estructura sobre la cual se realizará el recorrido (lista).

Observación 3. Estas funciones se definen sobre distintas estructuras, la condición para poder usarlas es que pertenezcan a la clase `Foldable`.

A continuación se presenta la implementación de estas dos funciones (sobre listas):

```haskell
-- Función que aplica una función a los elementos de una lista, de
-- forma encadenada a la derecha.
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ v [] = v
foldr f v (x:xs) = f x (foldr f v xs)

-- Función que aplica una función a los elementos de una lista, de
-- forma encadenada a la izquierda.
foldl :: (a -> b -> b) -> b -> [a] -> b
foldl _ v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Listado de código 15: Implementaciones de foldr y foldl

Ejemplo 18. Definir dos funciones `sumaListaR` y `sumaListaL` tal que (`sumaListaR l`) y (`sumaListaL l`) suman los elementos de una lista a la derecha e izquierda respectivamente. Por ejemplo:

```
sumaListaR []      = 0
sumaListaL [1]     = 1
sumaListaR [1,2,3] = 6
```

Solución. Funciones `sumaListaR` y `sumaListaL`:

```haskell
-- Función que suma los elementos de una lista.
sumaListaR :: Num a => [a] -> a
sumaListaR xs = foldr (+) 0 xs

-- Función que suma los elementos de una lista.
sumaListaL :: Num a => [a] -> a
sumaListaL xs = foldl (+) 0 xs
```

Listado de código 16: Suman los elementos de una lista

□

4.4. Funciones anónimas (lambdas)

Otro tipo de función importante, en Haskell, son las *lambdas*. Este tipo de funciones son útiles cuando se desea usar una función con alcance local, esto es, que no esté disponible en todo el archivo de definiciones, sino únicamente dentro de la expresión donde son usadas. Debe su nombre a las funciones del Cálculo Lambda de Alonso Church.

Son utilizadas principalmente en combinación con otras funciones de orden superior como son `map` y `filter`, para realizar aplicaciones o filtros de funciones que no se volverán a usar. La sintaxis de una lambda es la siguiente:

```
\<parámetro1> ... <parámetroN> -> <cuerpo>
```

No se provee ningún nombre para la lambda, pues es una función anónima, se separan los parámetros por espacios y se indica un cuerpo. A continuación se presentan algunos ejemplos de uso de lambdas.

```
Prelude> (\x y -> x + y) 17 29
46
Prelude> map (\x -> x + 13) [1,2,3]
[14,15,16]
Prelude> filter (\x -> mod x 13 == 0) [1,2,13]
[13]
```