Lenguajes de Programación, 2019-2
Nota de laboratorio 2: Estructuras de datos

Manuel Soto Romero

11 de febrero de 2019

Facultad de Ciencias, UNAM

A diferencia de los lenguajes imperativos, en Racket 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 Racket son los pares 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. Pares

Un par es una estructura de datos con las siguientes características:

- Son construidos mediante la función `cons`. Son estructuras de datos *heterogéneas*, es decir, sus elementos no necesariamente son del mismo tipo.

```
(cons 1 "a") = '(1 . "a")
```

- Para acceder a los elementos de un par se tienen las funciones `car`[1]y `cdr`[2] que obtienen el primer y segundo elemento de la estructura respectivamente.

```
'(  1  .  2  )
   car    cdr
```

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.

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

[1] Del inglés *Contents of the Address part of Register.*
[2] Del inglés *Contents of the Decrement part of Register.*

Solución. Función suma:

```
1 ;; Función que suma dos vectores representados por pares.
2 ;; suma: (pairof real) (pairof real) -> (pairof real)
3 (define (suma u v)
4    (cons (+ (car u) (car v)) (+ (cdr u) (cdr v))))
```

Listado de código 1: Suma de vectores

□

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

```
(producto-punto (cons 1 2)     (cons 3 4))     = 11.0
(producto-punto (cons 1.5 0)   (cons 0 1.5))   = 0.0
(producto-punto (cons 3.0 4.3) (cons 2.3 5.2)) = 29.26
```

Solución. Función producto-punto:

```
1 ;; Función que realiza el producto punto de dos vectores representados
2 ;; por pares.
3 ;; producto-punto: (pairof real) (pairofreal) -> real
4 (define (producto-punto u v)
5    (+ (* (car u) (car v)) (* (cdr u) (cdr v))))
```

Listado de código 2: Producto punto de vectores

□

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

```
(producto-escalar (cons 1 2)     2)   = '(2 . 4)
(producto-escalar (cons 1.5 0)   3.0) = '(3.5 . 0.0)
(producto-escalar (cons 3.0 4.3) 4.0) = '(12.0 . 17.2)
```

Solución. Función producto-escalar:

```
1 ;; Función que realiza el producto punto de un vector representado por
2 ;; un par y un escalar.
3 ;; producto-escalar: (pairof real) real -> (pairof real)
4 (define (productoEscalar u k)
5    (cons (* (car u) k) (* (cdr u) k)))
```

Listado de código 3: Producto por escalar de vectores

□

2. Listas

Las listas en Racket se definen recursivamente como sigue:

Definición 1. Una lista se define como:

1. La lista vacía y se representa por `empty`, `'()` o `null`.
2. Si `x` es un elemento de un conjunto cualquiera y `xs` es una lista, entonces `(cons 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 `cons` permite construir una lista dado un elemento de cualquier tipo y otra lista. En realidad son pares cuyo primer elemento es la cabeza de la lista y segundo elemento es otra lista, el resto.

Algunas características particulares de las listas son:

- Son estructuras de datos *heterogéneas*, es decir, sus elementos no son necesariamente del mismo tipo. Al número de elementos de una lista se le conoce como *longitud* de la lista.

  ```
  (cons 1 (cons 2 (cons 3 (cons 4 empty)))) = '(1 2 3 4)
  ```

- A diferencia de los pares, sí es posible construir nuevas listas a partir de otras, ya sea mediante la función `cons` o mediante la concatenación de listas (función `append`).

2.1. Representación de listas

Adicional a la notación de listas donde se delimitan los elementos por paréntesis y se separan por espacios, existen otras formas de definir listas como puede ser: mediante la función `list` o mediante el mecanismo de citado `quote`.

2.1.1. Listas mediante la función `list`

A diferencia de la función `cons`, `list` permite construir listas a partir de los elementos que la conforman, únicamente separando los mismos por espacios. Cada elemento de la lista es evaluado.

Ejemplo 4. Lista del 1 al 5 usando `list`.

```
> (list 1 2 3 4 5)
'(1 2 3 4 5)
```

□

Ejemplo 5. Lista de expresiones aritméticas con `list`.

```
> (list (+ 1 2 3) (* 2 3))
'(6 6)
```

□

2.1.2. Listas mediante `quote`

A diferencia de `cons` o list, el mecanismo de citado `quote`, no evalúa los elementos de una lista. Las primitivas de citado `quote`, tienen otras aplicaciones, por ejemplo en la definición de analizadores léxicos de los lenguajes de programación, más adelante en otra nota, se mencionan estas aplicaciones. Para definir listas con `quote`, basta con anteponer el símbolo `'`.

Ejemplo 6. Lista del 1 al 5 usando `quote`.

```
> '(1 2 3 4 5)
'(1 2 3 4 5)
```

□

Ejemplo 7. Lista de expresiones aritméticas con `quote`.

```
> '((+ 1 2 3) (* 2 3))
'((+ 1 2 3) (* 2 3))
```

□

2.2. Manipulación de listas

Algunas funciones predefinidas para manipular listas:

`empty?`	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.
`append`	Concatena dos listas.

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 8. Definir una función `longitud` tal que (`longitud l`) representa la longitud de una lista. 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: (listof a) -> number
3 (define (longitud l)
4    (if (empty? l)
5        0
6        (+ 1 (longitud (cdr l)))))
```

Listado de código 4: Longitud de una lista

□

Ejemplo 9. Definir una función `quita` tal que (`quita n l`) representa a la lista `l` sin sus primeros `n` elementos. Por ejemplo:

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

```
(quita 0 '(1 2 3))              = '(1 2 3)
(quita 2 '())                   = '()
(quita 2 '(#\H #\o #\l #\a))    = '(#\l #\a)
```

Solución. Función quita:

```
1 ;; Función que quita los primeros n elementos de una lista.
2 ;; quita: number (listof a) -> (listof a)
3 (define (quita n l)
4    (if (zero? n)
5        l
6        (quita (sub1 n) (cdr l))))
```

Listado de código 5: Lista sin los primeros n elementos

□

Ejemplo 10. Definir una función `toma` tal que (`toma n l`) representa los primeros `n` elementos de una lista `n`. Por ejemplo:

```
(toma 0 '(1 2 3))              = '()
(toma 2 '())                   = '()
(toma 2 '(#\H #\o #\l #\a))    = '(#\H #\o)
```

Solución. Función `toma`:

```
1 ;; Función que toma los primeros n elementos de una lista.
2 ;; toma: number (listof a) -> (listof a)
3 (define (toma n l)
4    (if (zero? n)
5        '()
6        (cons (car l) (toma (sub1 n) (cdr l)))))
```

Listado de código 6: Primeros n elementos de una lista

□

Ejemplo 11. Definir una función `contiene` tal que (`contiene e l`) indica si el elemento `e` es elemento de la lista `l`. Por ejemplo:

```
(contiene 0 '(1 2 3))              = #f
(contiene 2 '())                   = #f
(contiene #\o '(#\H #\o #\l #\a))  = #t
```

Solución. Función `contiene`:

```
;; Función que indica si un elemento pertenece a una lista.
;; contiene: a (lisotf a) -> boolean
(define (contiene e l)
   (or (equal? (car l) e) (contiene e (cdr l))))
```

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

□

3. Reconocimiento de patrones

Las funciones definidas en la subsección 2.2 hacen uso de condicionales que verifican el valor de una expresión para poder realizar recursión. Sin embargo, en Racket 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, pares, listas, entre otros. A continuación se definen algunas funciones que hacen uso de esta técnica. Para hacer reconocimiento de patrones, se hace uso de la primitiva `match`.

Ejemplo 12. Definir una función `suma-digitos` tal que (`suma-digitos 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:

```
(suma-digitos 0)    = 0
(suma-digitos 8)    = 8
(suma-digitos 1729) = 19
```

Solución. Función `suma-digitos`:

```
;; Función que suma los dígitos de un número entero positivo.
;; suma-digitos: number -> number
(define (suma-digitos n)
   (match n
      [0 0]
      [n (cond
            [(< n 10) n]
            [else (+ (modulo n 10) (quotient n 10))])]))
```

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

□

[4] Del inglés *pattern matching*.

Ejemplo 13. Definir una función `suma-lista` tal que (`suma-lista 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 `(cons x xs)` que representa a la lista con cabeza y resto. Por ejemplo:

```
(suma-lista '())       = 0
(suma-lista '(1))      = 1
(suma-lista '(1 2 3)) = 6
```

Solución. Función `suma-lista`:

```
1 ;; Función que suma los elementos de una lista.
2 ;; suma-lista: (listof number) -> number
3 (define (suma-lista l)
4    (match l
5       ['() 0]
6       [(cons x xs) (+ x (suma-lista xs))]))
```

Listado de código 9: Suma los elementos de una lista

□

Ejercicio 1. Definir una función `multiplica-lista` tal que (`multiplica-lista l`) representa el producto de los elementos de una lista. ¿Qué habría que cambiar a la función definida en el Listado de código 9?

4. Funciones de orden superior

Como se mencionó en la Nota de laboratorio 1, las funciones de Racket 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] Pertenecientes a la llamada programación origami.

```
1  ;; Función que aplica una función a cada elemento de una lista.
2  ;; map: (a -> b) (listof a) -> (listof b)
3  (define (map f l)
4      (match l
5          ['() '()]
6          [(cons x xs) (cons (f x) (map f xs))]))
```

Listado de código 10: Implementación de map

Ejemplo 14. Definir una función `meses` tal que (`meses l`) 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`:

```
1  ;; Función que obtiene el nombre del mes representado por el número
2  ;; recibido como parámetro.
3  ;; nombre-mes: number -> string
4  (define (nombre-mes n)
5     (cond
6        [(equal? n 1)  "Enero"     ]
7        [(equal? n 2)  "Febrero"   ]
8        [(equal? n 3)  "Marzo"     ]
9        [(equal? n 4)  "Abril"     ]
10       [(equal? n 5)  "Mayo"      ]
11       [(equal? n 6)  "Junio"     ]
12       [(equal? n 7)  "Julio"     ]
13       [(equal? n 8)  "Agosto"    ]
14       [(equal? n 9)  "Septiembre"]
15       [(equal? n 10) "Octubre"   ]
16       [(equal? n 11) "Noviembre" ]
17       [(equal? n 12) "Diciembre" ]
18       [else (error 'nombre-mes "Mes inválido.")]))
19
20 ;; Función que transforma una lista de números enteros en una lista
21 ;; de cadenas que representan el mes correspondiente.
22 ;; meses: (listof number) -> (listof string)
23 (define (meses l)
24    (map nombre-mes l))
```

Listado de código 11: Función meses

Se hace uso de la función `nombre-mes` definida en la Nota de laboratorio 1 (líneas 4 a 18). La función `meses` consiste de aplicar `nombre-mes` a cada elemento de la lista recibida como parámetro.

□

4.2. Función `filter`

Esta función recibe un predicado y deja en la lista aquellos elementos que cumplan con el mismo. Se define como sigue:

```
1  ;; Función que aplica filtra los elementos de una lista dada una
2  ;; condición.
3  ;; filter: (a -> boolean) -> (listof a) -> (listof a)
4  (define (filter f l)
5      (match l
6          ['() '()]
7          [(cons x xs)
8              (cond
9                  [(f x) (cons x (filter f xs))]
10                 [else (filter f xs)])]))
```

Listado de código 12: Implementación de filter

Ejemplo 15. Definir una función `ternas-pitagoricas` tal que (`ternas-pitagoricas l`) filtra listas de longitud 3 que cumplen con la propiedad de ser una terna pitagórica. Por ejemplo:

```
(ternas-pitagoricas '())                          = '()
(ternas-pitagoricas '((list 1 2 3) (list 3 4 5)) = '((3 4 5))
```

Solución. Función `ternas-pitagoricas`:

```
1  ;; Función que indica si una lista de tamaño 3 es una terna pitagórica.
2  ;; terna-pitagorica?: list -> boolean
3  (define (terna-pitagorica? l)
4     (match l
5        ['() #f]
6        [(list u v w) (equal? (+ (expt u 2) (expt v 2)) (expt w 2))]))
7
8  ;; Función que filtra las tuplas de tamaño 3 que cumplen con la
9  ;; propiedad de ser una terna pitagórica.
10 ;; ternasPitagoricas: (listof list) -> boolean
11 (define (ternas-pitagoricas xs)
12    (filter terna-pitagorica? xs))
```

Listado de código 13: Filtra listas de longitud 3 que cumplen con la propiedad de ser una terna pitagórica

Se hace uso del predicado `terna-pitagorica?` definida en las líneas 3 a 6. De esta forma, la función `ternas-pitagoricas` consiste de filtrar usando la función `terna-pitagorica?` cada elemento de la lista recibida como parámetro.

□

4.3. Funciones `foldr` y `foldl`

La función `suma-lista` definida en el Listado de código 9, 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).

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

```
1  ;; Función que aplica una función a los elementos de una lista, de
2  ;; forma encadenada a la derecha.
3  ;; foldr: (a -> b -> b) b (listof a) -> b
4  (define (foldr f v l)
5      (match l
6          ['() v]
7          [(cons x xs) (f x (foldr f v xs))]))
8
9  ;; Función que aplica una función a los elementos de una lista, de
10 ;; forma encadenada a la izquierda.
11 ;; foldl: (a -> b -> b) b (listof a) -> b
12 (define (foldl f v l)
13     (match l
14         ['() v]
15         [(cons x xs) (foldl f (f x v) xs)]))
```

Listado de código 14: Implementaciones de foldr y foldl

Ejemplo 16. Definir dos funciones `suma-listar` y `suma-listal` tal que (`suma-listar l`) y (`suma-listal l`) suman los elementos de una lista a la derecha e izquierda respectivamente. Por ejemplo:

```
(suma-listar '())       = 0
(suma-listal '(1))      = 1
(suma-listar '(1 2 3)) = 6
```

Solución. Funciones `suma-listar` y `suma-listal`:

```
;; Función que suma los elementos de una lista.
;; suma-listar: (listof number) -> number
(define (suma-listar xs)
   (foldr + 0 xs))

;; Función que suma los elementos de una lista.
;; suma-listal: (listof number) -> number
(define (suma-listal xs)
   (foldl + 0 xs))
```

Listado de código 15: Suman los elementos de una lista

□

4.4. Funciones anónimas (lambdas)

Otro tipo de función importante, en Racket, 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:

```
(lambda (<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.

```
> (lambda (x y) (+ x y)) 17 29
46
> (map (lambda (x) (+ x 13)) '(1 2 3))
'(14 15 16)
> (filter (lambda (x) (zero? (modulo x 13))) '(1 2 13))
'(13)
```