Introducción a la Programación Funcional
Nota de clase 03: Recursión y Funciones de Orden Superior*

Karla Ramírez Pulido Manuel Soto Romero

2 de septiembre de 2020
Facultad de Ciencias UNAM

Una de las principales características de los lenguajes funciones, como Racket, es el uso de la recursión como método de repetición en funciones. Se dice que la definición de una función es recursiva cuando se hace uso de ésta en la definición misma, a esto se le conoce como autoreferencia. Una función recursiva *válida* consta de dos partes:

- Un conjunto de casos base, los cuales son casos simples donde la definición se da directamente, es decir, sin usar autoreferencia.
- Un conjunto de reglas recursivas donde se define un nuevo caso de la definición en términos de anteriores ya definidos.

A continuación se presentan algunas funciones sobre listas, para manipular las mismas se hace uso de recursión.

3.1. Recursión sobre listas

Ejemplo 3.1. 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

```
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)))))
```

Código 1: Longitud de una lista

□

Ejemplo 3.2. Definir una función `quita` tal que (`quita n l`) representa a la lista `l` sin sus primeros `n` elementos. Por ejemplo:

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

Solución

*Adaptación de las Notas de Laboratorio del Curso de Lenguajes de Programación.

```
;; Función que quita los primeros n elementos de una lista.
;; quita: number (listof a) → (listof a)
(define (quita n l)
   (if (zero? n)
       l
       (quita (sub1 n) (cdr l))))
```

Código 2: Lista sin los primeros n elementos

□

Ejemplo 3.3. 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 '(#\H #\o #\l #\a))    = '(#\H #\o)
```

Solución:

```
;; Función que toma los primeros n elementos de una lista.
;; toma: number (listof a) → (listof a)
(define (toma n l)
   (if (zero? n)
       '()
       (cons (car l) (toma (sub1 n) (cdr l)))))
```

Código 3: Primeros n elementos de una lista

□

Ejemplo 3.4. 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 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))))
```

Código 4: Indica si un elemento pertenece a una lista

□

3.2. Funciones de orden superior

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[1] `foldr` y `foldl`. A continuación se muestra la implementación de estas funciones y algunos ejemplos de uso.

[1] Pertenecientes a la llamada programación origami.

3.2.1. Función `map`

Esta función aplica a cada elemento de una lista la función que recibe como parámetro. La definición de esta función se aprecia en el Código 5.

```
;; Función que aplica una función a cada elemento de una lista.
;; map: procedure (listof any) → (listof any)
(define (map f l)
    (match l
        ['() '()]
        [(cons x xs) (cons (f x) (map f xs))]))
```

Código 5: Implementación de map

Ejemplo 3.5. 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 que obtiene el nombre del mes representado por el número
;; recibido como parámetro.
;; nombre-mes: number → string
(define (nombre-mes n)
   (cond
      [(equal? n 1)  "Enero"     ]
      [(equal? n 2)  "Febrero"   ]
      [(equal? n 3)  "Marzo"     ]
      [(equal? n 4)  "Abril"     ]
      [(equal? n 5)  "Mayo"      ]
      [(equal? n 6)  "Junio"     ]
      [(equal? n 7)  "Julio"     ]
      [(equal? n 8)  "Agosto"    ]
      [(equal? n 9)  "Septiembre"]
      [(equal? n 10) "Octubre"   ]
      [(equal? n 11) "Noviembre" ]
      [(equal? n 12) "Diciembre" ]
      [else (error 'nombre-mes "Mes inválido.")]))

;; Función que transforma una lista de números enteros en una lista
;; de cadenas que representan el mes correspondiente.
;; meses: (listof number) → (listof string)
(define (meses l)
   (map nombre-mes l))
```

Código 6: Función meses

La función `meses` consiste de aplicar `nombre-mes` a cada elemento de la lista recibida como parámetro.

□

3.2.2. Función `filter`

Esta función recibe un predicado y deja en la lista aquellos elementos que cumplan con el mismo. La definición de esta función se aprecia en el Código 7.

```
;; Función que aplica filtra los elementos de una lista dada una
;; condición.
;; filter: (any -> boolean) (listof any) → (listof any)
(define (filter f l)
    (match l
        ['() '()]
        [(cons x xs)
            (cond
                [(f x) (cons x (filter f xs))]
                [else (filter f xs)])]))
```

Código 7: Implementación de filter

Ejemplo 3.6. 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 '((1 2 3) (3 4 5)) = '((3 4 5))
```

Solución:

```
;; Función que indica si una lista de tamaño 3 es una terna pitagórica.
;; terna-pitagorica?: list → boolean
(define (terna-pitagorica? l)
   (match l
      ['() #f]
      [(list u v w) (equal? (+ (expt u 2) (expt v 2)) (expt w 2))]))

;; Función que filtra las tuplas de tamaño 3 que cumplen con la
;; propiedad de ser una terna pitagórica.
;; ternasPitagoricas: (listof list) → boolean
(define (ternas-pitagoricas xs)
   (filter terna-pitagorica? xs))
```

Código 8: 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.

□

3.2.3. Funciones `foldr` y `foldl`

Estas funciones engloban un patrón es común, que consiste en aplicar una función mediante un recorrido 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):

```
;; Función que aplica una función a los elementos de una lista, de
;; forma encadenada a la derecha.
;; foldr: procedure any (listof any) → any
(define (foldr f v l)
    (match l
        ['() v]
        [(cons 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: procedure any (listof any) → any
(define (foldl f v l)
    (match l
        ['() v]
        [(cons x xs) (foldl f (f x v) xs)]))
```

Código 9: Implementaciones de foldr y foldl

Ejemplo 3.7. 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:

```
;; 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))
```

Código 10: Suman los elementos de una lista

La Figura 1, muestra, a la izquierda, el árbol de evaluación de la llamada a `suma-listar` con la lista `'(1 2 3)` y del lado derecho a `suma-listal` con la misma entrada.

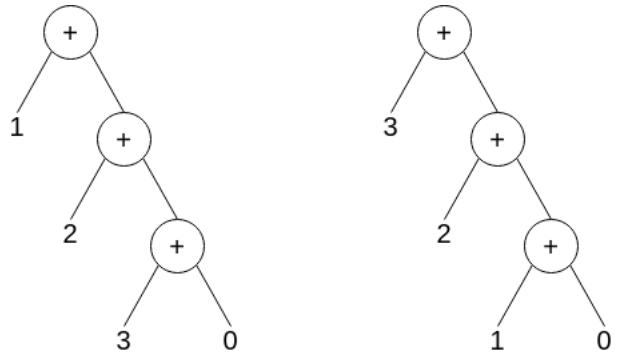

Figura 1: *Árboles de evaluación de suma-listar y suma-listal*

□

3.2.4. 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 λ de Alonzo 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 alguna de las siguientes:

```
(lambda (<parámetro1> ... <parámetroN>)
    <cuerpo>)

(λ (<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 (λ (x) (+ x 13)) '(1 2 3))
'(14 15 16)
> (filter (lambda (x) (zero? (modulo x 13))) '(1 2 13))
'(13)
```

Referencias

[1] Favio Miranda, Elisa Viso, *Matemáticas Discretas*, Las Prensas de Ciencias, Segunda Edición, 2015.

[2] Matthias Felleisen, Conrad Barski, et.al., *Realm of Racket*, No Starch Press, Primera Edición, 2013.

[3] Matthias Felleisen, Robert B. Findler, et.al., *How to Desing Programs*, MIT Press, Segunda Edición, 2019.

[4] Matthew Flatt, Robert B. Findler, et. al., *The Racket Guide*, Versión 7.5

[5] Éric Tanter, *PrePLAI: Scheme y Programación Funcional*, 2014.