Lenguajes de Programación

Introducción a Racket

Estructuras de Datos y Recursión

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

16 de febrero de 2018

Pares

La principal estructura de datos de Racket es el par. Es una estructura que almacena dos valores que no deben ser necesariamente del mismo tipo. Para construir un par se usa la función `cons`.

Pares

La principal estructura de datos de Racket es el par. Es una estructura que almacena dos valores que no deben ser necesariamente del mismo tipo. Para construir un par se usa la función `cons`.

```
> (cons 1 2)
```

Pares

La principal estructura de datos de Racket es el par. Es una estructura que almacena dos valores que no deben ser necesariamente del mismo tipo. Para construir un par se usa la función `cons`.

```
> (cons 1 2)
'(1 . 2)
```

Pares

La principal estructura de datos de Racket es el par. Es una estructura que almacena dos valores que no deben ser necesariamente del mismo tipo. Para construir un par se usa la función `cons`.

```
> (cons 1 2)
'(1 . 2)
```

Para acceder a los elementos del par se tienen las funciones `car` (*Contents of the Address part of Register number*) y `cdr` (*Contents of the Decrement part of Register number*).

Biblioteca para trabajar con vectores

Podemos usar pares para definir una pequeña biblioteca para trabajar con vectores.

Biblioteca para trabajar con vectores

Podemos usar pares para definir una pequeña biblioteca para trabajar con vectores.

```
;; Función que construye un vector.
;; mvector: number number -> number
(define(mvector x y)
    (cons x y))
```

Biblioteca para trabajar con vectores

Podemos usar pares para definir una pequeña biblioteca para trabajar con vectores.

```
;; Función que construye un vector.
;; mvector: number number -> number
(define(mvector x y)
    (cons x y))
```

Ejercicio

Definir funciones para *a) sumar dos vectores*, *b) multiplicar un vector por un escalar* y *c) realizar el producto punto de dos vectores*.

Biblioteca para trabajar con vectores

```
;; Función que calcula la suma de dos vectores.
;; suma: pair pair -> pair
(define(suma v1 v2)
    (cons (+ (car v1) (car v2)) (+ (cdr v1) (cdr v2))))

;; Función que calcula el producto por escalar de un
;; real y un vector.
;; producto-escalar: number pair -> pair
(define(producto-escalar k v)
    (cons (* k (car v)) (* k (cdr v))))

;; Función que calcula el producto punto de dos vectores
;; producto-punto: pair pair -> number
(define(producto-punto v1 v2)
    (+ (* (car v1) (car v2)) (* (cdr v1) (cdr v2))))
```

Listas

Los pares son usados para implementar listas pues su definición lo permite:

Definición (Lista)

Una lista es alguna de las siguientes:

- La lista vacía es una lista y se representa por `empty`, `null` o `'()`.
- Si `x` es un elemento de un conjunto cualquiera y `xs` es una lista, entonces `(cons x xs)` lo es también. Llamamos a `x` la cabeza de la lista y a `xs` el resto.
- Son todas.

Listas

Los pares son usados para implementar listas pues su definición lo permite:

Definición (Lista)

Una lista es alguna de las siguientes:

- La lista vacía es una lista y se representa por `empty`, `null` o `'()`.
- Si `x` es un elemento de un conjunto cualquiera y `xs` es una lista, entonces `(cons x xs)` lo es también. Llamamos a `x` la cabeza de la lista y a `xs` el resto.
- Son todas.

```
> (cons 1 (cons 2 (cons 3 (cons 4 empty))))
```

Listas

Los pares son usados para implementar listas pues su definición lo permite:

Definición (Lista)

Una lista es alguna de las siguientes:

- La lista vacía es una lista y se representa por `empty`, `null` o `'()`.
- Si `x` es un elemento de un conjunto cualquiera y `xs` es una lista, entonces `(cons x xs)` lo es también. Llamamos a `x` la cabeza de la lista y a `xs` el resto.
- Son todas.

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

Representación de listas

Existen otras dos formas de representar listas:

Representación de listas

Existen otras dos formas de representar listas:

```
> (cons (+ 1 2) (cons (+ 2 3) empty))
```

Representación de listas

Existen otras dos formas de representar listas:

```
> (cons (+ 1 2) (cons (+ 2 3) empty))
'(3 5)
```

Mediante la función `list`

Representación de listas

Existen otras dos formas de representar listas:

```
> (cons (+ 1 2) (cons (+ 2 3) empty))
'(3 5)
```

Mediante la función `list`

```
> (list (+ 1 2) (+ 2 3))
```

Representación de listas

Existen otras dos formas de representar listas:

```
> (cons (+ 1 2) (cons (+ 2 3) empty))
'(3 5)
```

Mediante la función `list`

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

Mediante el mecanismo de citado (`quote`**)**

Representación de listas

Existen otras dos formas de representar listas:

```
> (cons (+ 1 2) (cons (+ 2 3) empty))
'(3 5)
```

Mediante la función `list`

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

Mediante el mecanismo de citado (`quote`)

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

Representación de listas

Existen otras dos formas de representar listas:

```
> (cons (+ 1 2) (cons (+ 2 3) empty))
'(3 5)
```

Mediante la función `list`

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

Mediante el mecanismo de citado (`quote`)

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

Funciones recursivas

Racket también permite definir funciones recursivas.

Funciones recursivas

Racket también permite definir funciones recursivas.

```
;; Función que obtiene la suma de los dígitos de un número.
;; suma-digitos: number -> number
(define(suma-digitos n)
    (if(< n 10)
        n
        (+ (modulo n 10) (suma-digitos (quotient n 10)))))
```

Recursión y listas

Para definir funciones recursivas sobre listas, es preferente usar la técnica de *cazamiento de patrones* que se basa en el principio de *inducción estructural*. Para usar esta técnica, se usa la función `match`.

Recursión y listas

Para definir funciones recursivas sobre listas, es preferente usar la técnica de *cazamiento de patrones* que se basa en el principio de *inducción estructural*. Para usar esta técnica, se usa la función `match`.

```
;; Función que obtiene la longitud de una lista.
;; longitud: list -> number
(define(longitud lst)
    (match lst
        ['() 0]
        [(cons x xs) (+ 1 (longitud xs))]))
```

Funciones de orden superior

Una función de orden superior es aquella que recibe otra función y la usa para realizar algún cálculo. Las funciones más populares de este tipo son `map` y `filter`.

Funciones de orden superior

Una función de orden superior es aquella que recibe otra función y la usa para realizar algún cálculo. Las funciones más populares de este tipo son `map` y `filter`.

```
;; Función que dada una función y una lista, aplica la
;; función a cada elemento de la lista.
;; map: procedure list -> list
(define(map f lst)
    (match lst
        ['() lst]
        [(cons x xs) (cons (f x) (map f xs))]))
```

Funciones de orden superior

Una función de orden superior es aquella que recibe otra función y la usa para realizar algún cálculo. Las funciones más populares de este tipo son `map` y `filter`.

```
;; Función que dada una función y una lista, aplica la
;; función a cada elemento de la lista.
;; map: procedure list -> list
(define(map f lst)
    (match lst
        ['() lst]
        [(cons x xs) (cons (f x) (map f xs))]))

;; Función que dado un predicado y una lista, deja
;; los elementos de la lista que cumplen el predicado.
;; filter: procedure list -> list
(define(filter p lst)
    (match lst
        ['() lst]
        [(cons x xs)
            (cons (if (p x) (cons x (filter p xs)) (filter p xs)))]))
```

Funciones de orden superior

Otras funcione de este tipo son las funciones de plegado `foldr` y `foldl`.

Funciones de orden superior

Otras funcione de este tipo son las funciones de plegado `foldr` y `foldl`.

```
;; Función que aplica una función de forma encadenada a la derecha
;; a los elementos de una lista dado un caso base.
;; foldr : procedure any lst - > lst
(define(foldr f v lst)
    (match lst
        ['() v]
        [(cons x xs) (f x (foldr f v xs))]))
```

Funciones de orden superior

Otras funcione de este tipo son las funciones de plegado `foldr` y `foldl`.

```
;; Función que aplica una función de forma encadenada a la derecha
;; a los elementos de una lista dado un caso base.
;; foldr : procedure any lst - > lst
(define(foldr f v lst)
    (match lst
        ['() v]
        [(cons x xs) (f x (foldr f v xs))]))

;; Función que aplica una función de forma encadenada a la izquierda
;; a los elementos de una lista dado un caso base.
;; foldl : procedure any lst - > lst
(define(foldl f v lst)
    (match lst
        ['() v]
        [(cons x xs) (foldl f (f x v) xs)]))
```

Lambdas

Al usar funciones de orden superior, es normal que necesitemos una función que no esté definida en el núcleo de Racket y tengamos que definir una nueva función que no es usada mas que para aplicar la función de orden superior. En estos casos, realmente no importa el nombre de la función, sólo su comportamiento. Existen funciones de este tipo y son llamadas *lambdas*.

```
(lambda(<parámetro>*) <cuerpo>)
```

Lambdas

Al usar funciones de orden superior, es normal que necesitemos una función que no esté definida en el núcleo de Racket y tengamos que definir una nueva función que no es usada mas que para aplicar la función de orden superior. En estos casos, realmente no importa el nombre de la función, sólo su comportamiento. Existen funciones de este tipo y son llamadas *lambdas*.

```
(lambda(<parámetro>*) <cuerpo>)
```

```
> (lambda (x) (+ x 13))
```

Lambdas

Al usar funciones de orden superior, es normal que necesitemos una función que no esté definida en el núcleo de Racket y tengamos que definir una nueva función que no es usada mas que para aplicar la función de orden superior. En estos casos, realmente no importa el nombre de la función, sólo su comportamiento. Existen funciones de este tipo y son llamadas *lambdas*.

```
(lambda(<parámetro>*) <cuerpo>)
```

```
> (lambda (x) (+ x 13))
#<procedure>
```

Lambdas

Al usar funciones de orden superior, es normal que necesitemos una función que no esté definida en el núcleo de Racket y tengamos que definir una nueva función que no es usada mas que para aplicar la función de orden superior. En estos casos, realmente no importa el nombre de la función, sólo su comportamiento. Existen funciones de este tipo y son llamadas *lambdas*.

```
(lambda(<parámetro>*) <cuerpo>)
```

```
> (lambda (x) (+ x 13))
#<procedure>
> (map (lambda (x) (+ x 13)) '(1 2 3))
```

Lambdas

Al usar funciones de orden superior, es normal que necesitemos una función que no esté definida en el núcleo de Racket y tengamos que definir una nueva función que no es usada mas que para aplicar la función de orden superior. En estos casos, realmente no importa el nombre de la función, sólo su comportamiento. Existen funciones de este tipo y son llamadas *lambdas*.

```
(lambda(<parámetro>*) <cuerpo>)
```

```
> (lambda (x) (+ x 13))
#<procedure>
> (map (lambda (x) (+ x 13)) '(1 2 3))
'(14 15 16)
```

Lambdas y azúcar sintáctica

La definición de funciones antes vista, es realmente azúcar sintáctica para *lambdas*.

Lambdas y azúcar sintáctica

La definición de funciones antes vista, es realmente azúcar sintáctica para *lambdas*.

```
;; Función identidad.
;; identidad1: any -> any
(define (identidad1 x)
    x)
```

Lambdas y azúcar sintáctica

La definición de funciones antes vista, es realmente azúcar sintáctica para *lambdas*.

```
;; Función identidad.
;; identidad1: any -> any
(define (identidad1 x)
    x)

;; Función identidad.
;; identidad2: any -> any
(define identidad2
    (lambda (x) x))
```

Lambdas y asignaciones locales

Al definir una función se pueden definir funciones auxiliares dentro de su implementación.

Lambdas y asignaciones locales

Al definir una función se pueden definir funciones auxiliares dentro de su implementación.

```
;; Función que calcula el promedio de los elementos de una
;; lista.
;; promedio: list -> number
(define (promedio lst)
    (letrec (
        [suma (lambda (lst)
            (match lst
                ['() 0]
                [(cons x xs) (+ x (suma xs))])]
        [longitud (lambda (lst)
            (match lst
                ['() 0]
                [(cons x xs) (+ 1 (longitud xs))])])
        (/ (suma lst) (longitud lst))))
```