Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Anexo 1: Introducción a Racket

Karla Ramírez Pulido
karla@ciencias.unam.mx

Manuel Soto Romero
manu@ciencias.unam.mx

15 de febrero de 2018

Descripción

En esta nota se presentan los conceptos necesarios para que los estudiantes tengan un primer acercamiento con el lenguaje de programación Racket, que conozcan la estructura básica de un programa, que aprendan a ejecutarlos y que conozcan algunas primitivas básicas como los tipos básicos, la definición de funciones, el uso de estructuras de datos y la recursión.

Índice general

Índice de códigos

Índice de figuras

1.1 ¿Qué es Racket?

Racket es un lenguaje de programación de la familia de Lisp. Además de ser un lenguaje de propósito general, fue diseñado como una plataforma para la creación, diseño e implementación de lenguajes de programación. Racket incluye muchos dialectos en su núcleo para resolver determinados problemas. En estas notas se hará uso de la variante `plai` (*Programming Lenguages: Application and Interpretation*) que acompaña al libro del mismo nombre.

Racket es un lenguaje de programación funcional, lo cual quiere decir que todo es modelado mediante funciones y que además, éstas son tratadas como cualquier otro tipo de datos, esto es: pueden ser pasadas como parámetros, regresarse como valores en el resultado de alguna llamada a función e incluso almacenarse dentro de estructuras de datos.

Existen dos formas de escribir programas en Racket: (1) mediante el ambiente de desarrollo integrado DrRacket y (2) de la forma tradicional usando un archivo fuente y ejecutándolo en terminal. En estas notas se hará uso de DrRacket pues su ambiente gráfico es de gran utilidad para aquellos que se están adentrando en este maravilloso lenguaje.

Instalación de DrRacket

Para instalar DrRacket en sistemas operativos como Windows o MacOS basta con ingresar a la dirección `https://download.racket-lang.org/` y descargar la versión del instalador correspondiente. Una vez completada la descarga ejecutarlo y dejar las configuraciones por defecto.

La instalación en sistemas Linux basados en Debian como Ubuntu consiste en ejecutar los siguientes comandos en una terminal:

```
> sudo add-apt-repository ppa:plt/racket
> sudo apt update
> sudo apt install racket
```

Una vez instalado el ambiente de desarrollo, se procede a buscar el mismo en la lista de aplicaciones del sistema operativo y abrirlo para obtener la pantalla que se muestra en la Figura 1.

La parte de arriba del ambiente de desarrollo es llamada *área de definiciones* y es dónde se incluyen las funciones y definiciones necesarias para resolver un determinado problema. Por otro lado, la sección de abajo es llamada *área de interacción* y es dónde se ejecutan los programas escritos en el área de definiciones o las primitivas que incluye el lenguaje o dialecto en su núcleo.

Antes de comenzar a escribir programas, se cambian algunas configuraciones para aprovechar mejor el uso del ambiente de desarrollo.

Racket es un lenguaje que utiliza notación prefija y establece precedencia de operadores mediante paréntesis, por lo cual, la configuracion inicial que se modifica es la herramienta que permite el autocompletado de paréntesis, la numeración de líneas y delimitación del tamaño de las líneas de código. Para esto, se teclea `Ctrl + ;` e ir a la sección `Editando > General Editing` y seleccionar las casillas:

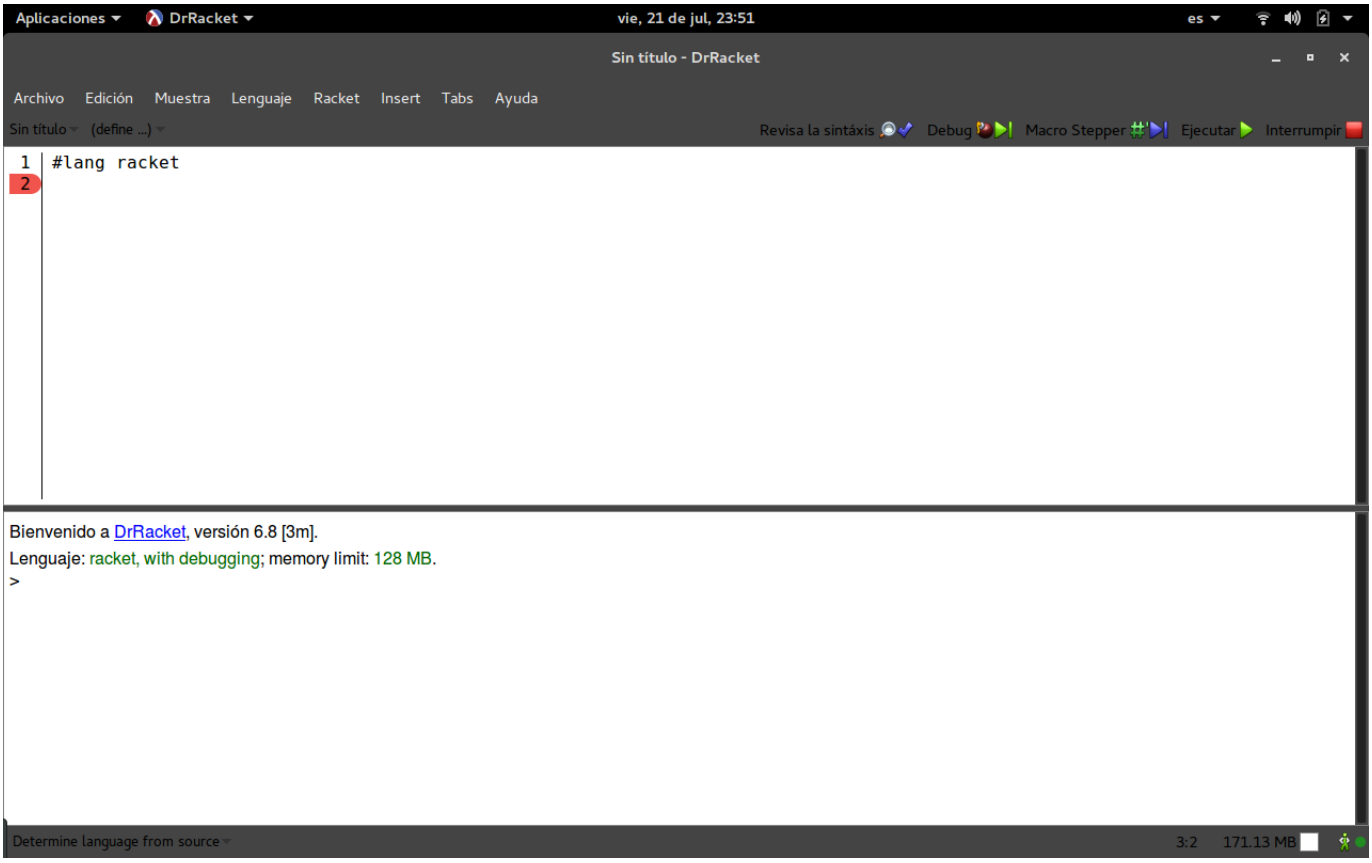

Figura 1: *Pantalla inicial de DrRacket*

- `Enable automatic parentheses`.
- `Show line numbers`.
- Asegurarse de que el valor de `Maximum character width guide` sea 102.

Esto se muestra en la Figura 2.

Ahora se configura el ambiente para que cargue el dialecto `plai` por defecto cada que se inicie la herramienta, para esto, al teclear `Ctrl + l` y escoger la opción de en `Mostrar Detalles`. Se podrá modificar el campo `Automatic #lang line` y escribir `#lang plai`. Esto se muestra en la Figura 3.

1.2 Primitivas de Racket

Antes de comenzar a definir funciones, se presentan los tipos básicos con los que cuenta el lenguaje, estos pueden probarse en el área de interacciones para comprender cómo se realiza la evaluación de estas expresiones.

Booleanos

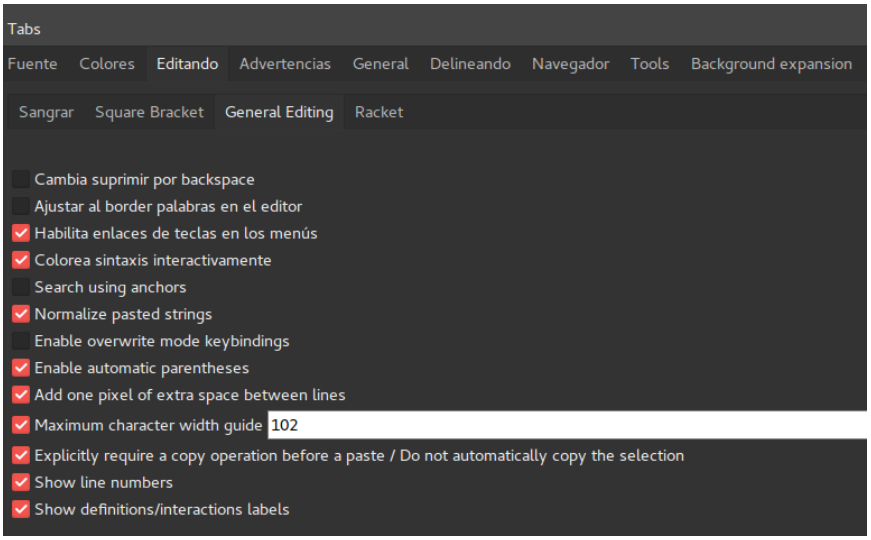

Figura 2: *Configuraciones de DrRacket*

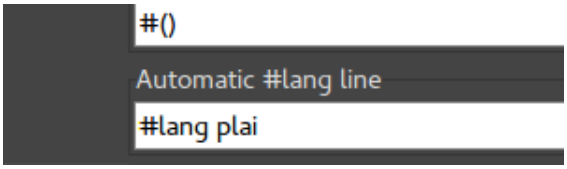

Figura 3: *Modificación al dialecto por defecto*

Para representar las constantes booleanas *verdadero* y *falso* se usa la notación `#t` y `#f`, aunque también existe una versión con azúcar sintáctica `true` y `false`.

```
> #t
#t
> #f
#f
> true
#t
> false
#f
```

Números

Se tienen dos tipos de números: *exactos* e *inexactos*. Los primeros son aquellos números para los cuales se conoce su valor a la perfección, mientras que los segundos son aquellos números que por su naturaleza no tienen un valor concreto, por ejemplo si tienen decimales. En este sentido, se tiene la siguiente clasificación:

- Exactos: Enteros, racionales y complejos.
- Inexactos: Flotantes y complejos con flotantes.

Además, no se tiene un límite en el tamaño de los números, pueden ser de cualquier longitud.

```
> -1
-1
> 7
7
> 1/7
1/7
> 1+7i
1 + 7i
> 5.5
5.5
> 5.25e8
5.25e8
> 27387276897958672387689576859678956247986527684762586754867285 3
27387276897958672387689576859678956247986527684762586754867285 3
```

Caracteres

Para representar caracteres se usa la codificación Unicode[1]. Los caracteres se representan precediendo los símbolos #\. Es posible tambien usar el código Unicode correspondiente, por ejemplo #\u3123.

```
> #\a
#\a
> #\E
#\E
```

Cadenas

Las cadenas, como en la mayoría de lenguajes, son agrupaciones de caracteres. Se delimitan por comillas dobles.

```
> ‘‘Hola mundo’’
‘‘Hola mundo’’
```

Símbolos

Son tratados como valores atómicos, por lo que su comparación es menos compleja a diferencia de una cadena. Su representación usa el mecanismo de citado (*quoting*) del que se habla en otra nota. Se delimitan por una comilla simple al inicio.

[1] Estándar de codificación de caracteres.

```
> 'manzana
'manzana
```

Por supuesto que esto no es lo único que se puede hacer con las primitivas del lenguaje. Racket incluye algunas funciones predefinidas para trabajar con estos tipos básicos. Como se mencionó antes, Racket es un lenguaje de notación prefija con paréntesis, por lo que para usar una función se usa la siguiente notación:

```
(función parámetro1 pararámetro2 ...)
```

Algunos ejemplos del uso de funciones predefinidas:

```
> (+ 1 2)
3
> (- 1 1/4)
3/4
> (+ 1 (- 3 4))
0
> (sqrt -1)
0+1i
> (or (< 5 4) (equal? 1 (- 6 5)))
#t
> (and (not (zero? 10)) (+ 1 2 3))
6
> (string-append "Ho" "la")
"Hola"
> (string-length "hola mundo")
10
> (substring "Apple" 3)
"le"
> (string->symbol "Apple")
'Apple
> (display "hola\nmundo")
hola
mundo
> (+ 1 "Hola")
+: contract violation
```

Observaciones

- Las funciones que regresan valores booleanos terminan con ? y son llamadas predicados.
- La primitiva and regresa la evaluación de la última expresión verdadero y considera verdadero todo aquello que no sea falso.

- Como convención, los nombres de función se escriben en minúsculas y se separa cada palabra con un guión.
- Como convención, las funciones que hacen conversiones de tipo se nombran poniendo un -> entre los tipos a convertir y son llamadas conversores.
- Racket es un lenguaje de tipo seguro, esto quiere decir que al invocar una función con argumentos de tipo equivocado, se generan errores.

Para conocer más funciones predefinidas, buscarlas en: `http://racket-lang.org/`

1.3 Definición de funciones

Además de las funciones que provee Racket, es posible definir funciones propias. Estas definiciones se deben colocar en el área de definiciones para posteriormente usarlas en el área de interacciones. Para definir una función se recomienda seguir los siguientes pasos:

1. Entender qué tiene que hacer la función.
2. Escribir la descripción de la función.
3. Escribir su contrato, (es decir, los tipos que recibe y regresa, y cuantos parámetros tiene).
4. Escribir las pruebas unitarias asociadas a estas funciones sobre los distintos posibles datos de entrada (casos significativos).
5. Finalmente, implementar el cuerpo de la función.

El contrato y la descripción se especifican mediante comentarios, los comentarios pueden ser de varias líneas, delimitados por #| y |# o de una línea delimitados por ;. Por ejemplo, la descripción y contrato de una función para calcular el área de un círculo dado su diámetro se muestra a continuación:

```
;; Función que calcula el área de un círculo dado su diámetro.
;; area: number -> number
```

Escribir las pruebas, obliga al programador a entender cómo se va a usar la función. Para definir pruebas unitarias se usa la primitiva `test` que tiene la siguiente sintaxis:

```
(test (<nombre-funcion> <parametro>*) <resultado-esperado>)
```

Por ejemplo, algunas pruebas unitarias para la función `area` usando la primitiva `test`.

```
(test (area 10) 78.53981633974483)
(test (area 4) 12.566370614359172)
(test (area 16) 201.06192982974676)
```

Para definir una función se usa la siguiente sintaxis:

```
(define (<nombre-funcion> <parametro>*)
   <cuerpo>)
```

Por ejemplo para calcular el área de un círculo se usa la fórmula:

$$A = \pi \cdot r^2$$

así, se obtiene la función `area`:

```
;; Función que calcula el área de un círculo dado su diámetro.
;; area: number -> number
(define (area diametro)
   (* pi (/ diametro 2) (/ diametro 2)))

(test (area 10) 78.53981633974483)
(test (area 4) 12.566370614359172)
(test (area 16) 201.06192982974676)
```

Código 1: *Área de un círculo*

Asignaciones locales

La función anterior es correcta, sin embargo, hay cálculos repetitivos en su definición. Para obtener el radio del círculo se divide el diámetro, pero se realiza dos veces, ¿no sería mejor si lo calculamos una vez y simplemente lo usamos?, para este tipo de situaciones, se usan asignaciones locales que permiten definir variables en una determinada región de un programa.

La primitiva `let` tiene la siguiente sintaxiss:

```
(let ([<identificador> <valor>]+)
   <cuerpo>)
```

Se recibe una lista de identificadores asociados a un valor, que pueden ser usados después en el cuerpo de la expresión.

Usando esta primitiva en la función `área`:

```
;; Función que calcula el área de un círculo dado su diámetro.
;; area: number -> number
(define (area diametro)
   (let ([r (/ diametro 2)])
      (* pi r r)))

(test (area 10) 78.53981633974483)
(test (area 4) 12.566370614359172)
(test (area 16) 201.06192982974676)
```

Código 2: *Área de un círculo con asignaciones locales*

Otra de estas primitivas es `let*` que permite usar identificadores anidados. Por ejemplo, en la siguiente expresión:

```
(let ([a 2] [b (+ a a)])
   b)
```

se genera un error, pues para definir `b` no se conoce el valor de `a`. Una solución sería anidar expresiones `let`:

```
(let ([a 2])
   (let ([b (+ a a)])
      b))
```

Sin embargo, mientras más compleja la expresión, es más tedioso hacer esta anidación. Para ello existe la versión con azúcar sintáctica[2] `let*`:

```
(let* ([a 2] [b (+ a a)])
   b)
```

También existe una variante llamada `letrec` que permite definir expresiones recursivas. Se revisan ejemplos de esta primitiva más adelante.

Si se quisiera definir una variable globalmente, basta con usar la primitiva `define`. Por ejemplo, si se especifica al inicio del programa `(define a 10)` la variable `a` tendrá un valor de `10` en todo el programa.

Condicionales

Es común que las funciones tomen decisiones mientras se ejecutan. En Racket existen, dos formas principales de tomar decisiones mediante condicionales.

Se tiene el típico `if` cuya sintaxis es:

[2]Azúcar sintáctica permite al programador escribir programas de forma dulce o amena.

```
(if <condición>
    <expresión-then>
    <expresión-else>)
```

La condición debe ser cualquier valor booleano. Es importante notar que las expresiones `if` siempre deben de ir acompañadas de un caso para `else`. Por ejemplo, a continuación se muestra una función que calcula el valor absoluto de un número:

```
;; Función que calcula el valor absoluto de un número.
;; absoluto: number -> number
(define (absoluto n)
   (if (< n 0) ; condición
       (* n -1) ; valor si la condición es verdadera
       n)) ; valor si la condición es falsa
```

Código 3: *Valor absoluto de un número*

Si el número ingresado es menor a 0, se multiplica por -1, en otro caso se regresa el mismo valor.

Otra forma de definir condiciones más complejas es usando `cond`:

```
(cond
   [<condición> <instrucción>* <regreso>]+
   [else <instrucción>* <regreso>]*)
```

Se verifican una o más condiciones y el caso para `else` es opcional. Si alguna condición es verdadera, se ejecuta la secuencia de instrucciones y se regresa el último valor. Por ejemplo, la función `mes`:

```
;; Función que regresa la representación en cadena de un mes
;; pasado como número.
;; mes: number -> string
(define (mes n)
   (cond
      [(= n 1) (display "Caso 1") "Enero")]
      [(= n 2) (display "Caso 2") "Febrero")]
      ...
      [else (error 'mes "Número inválido")]))
```

Código 4: *Valor de un mes*

La instrucción `else` es opcional y se usa en casos muy particulares, en este código se incluye a manera de ejemplo. La primitiva `error` muestra un error en la terminal, el símbolo que recibe como parámetro indica qué función lanzó el error.

1.4 Estructuras de datos

La principal estructura de datos de Racket es el *par*. Un par una estructura que almacena dos valores que no son necesariamente del mismo tipo. Para construir un par se usa la función `cons`. Por ejemplo:

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

Para acceder a los elementos del par se usan las funciones `car` (*Contents of the Address part of Register number*) y `cdr` (*Contents of the Decrement part of Register number*)[3].

El siguiente código implementa una pequeña biblioteca para trabajar con vectores cuyos elementos están en $\mathbb{R}^2$, usando pares, se usan las primitivas `car` y `cdr` para acceder a los elementos y aplicar la operación correspodiente:

```
#lang plai

#| Biblioteca para trabajar con vectores. |#

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

;; 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))))
```

Código 5: *Biblioteca de vectores*

[3]Se usan estos nombres en honor a las partes de la computadora IBM 704 dónde fue implementado originalmente Lisp.

Ejemplos de uso el código anterior

```
> (define v1 (mvector 1 7))
> (define v2 (mvector 2 9))
> (suma v1 v2)
'(3 . 16)
> (producto-escalar 2 v1)
'(2 . 14)
```

```
> (producto-punto v1 v2)
65
```

Por su naturaleza, los pares son usados para implementar otras estructuras de datos como, las listas, la estructura de datos más usada en los lenguajes de programación declarativos. Son implementadas usando pares pues su definición lo permite:

Definición 1.1 (Lista) Una lista es alguna de las siguientes[5]:

- 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.

De esta forma, para construir una lista se usa la función `cons`. En este caso el segundo elemento debe ser una lista, sea vacía o no.

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

Existen otras dos formas de representar listas: mediante la función `list` y mediante el símbolo de citado `'` (`quote`). La diferencia radica en que `cons` y `list` evalúan los elementos de la lista mientras que el símbolo `quote` mantiene los elementos sin evaluar, por ejemplo:

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

Más adelante se revisarán usos y beneficios de `quote`, por el momento momento basta conocer las distintas formas de representación.

1.5 Recursión

Además de las funciones y listas, otra característica importante de los lenguajes funcionales es el poder definir tareas que requieren repeticiones de forma recursiva. La sintaxis vista hasta el momento, no cambia. Definir funciones recursivas se hace de la misma manera que al definir otras funciones. Basta con indicar el caso base y el paso recursivo, Racket hará todo el trabajo.

Algunos ejemplos:

Una función que eleva un número `a` a otro `b` recursivamente.

```
;; Función que calcula la potencia de dos números.
;; potencia a b: number number -> number
(define (potencia a b)
   (if (zero? b)
       1
       (potencia a (sub1 b))))
```

Código 6: *Potencia*

```
> (potencia 5 0)
1
> (potencia 2 3)
8
```

Se suman los digitos de un número. Notar que para extrar los dígitos se usa la función `modulo` y para continuar procesando se usa la división entera.

```
;; 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)))))
```

Código 7: *Suma de los dígitos de un número*

```
> (suma-digitos 8)
8
> (suma-digitos 1729)
19
```

Para estructuras de datos como listas, es más común usar la técnica de *apareamiento de patrones* que en lugar de verificar condiciones trata de cazar el patrón de una estructura y en caso de lograrlo, regresa un valor. Esta técnica se en el principio de *inducción estructural.*

Para usar esta técnica se usa la primitiva `match`. Algunos ejemplos con listas:

La longitud de una lista basado en el caso para la lista vacía y el caso para la lista que tiene cabeza y resto.

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

Código 8: *Longitud de una lista*

```
> (longitud '())
0
> (longitud '(1 7 2 9))
4
```

Para invertir los elementos de una lista:

```
;; Función que obtiene la reversa de una lista
;; reversa: list -> list
(define (reversa lst)
   (match lst
      ['() lst]
      [(cons x xs) (append (reversa xs) (list x))]))
```

Código 9: *Reversa de una lista*

```
> (reversa '())
'()
> (reversa '(1 7 2 9))
'(9 2 7 1)
```

Para unir dos listas:

```
;; Función que obtiene la concatenación de dos listas
;; concatena: list list -> list
(define (concatena lst1 lst2)
   (match lst1
      ['() lst2]
      [(cons x xs) (cons x (concatena xs lst2))]))
```

Código 10: *Concatenación de listas*

```
> (concatena '() '(2 9))
'(2 9)
> (concatena '(1 7) '(2 9))
'(1 7 2 9)
```

Funciones de orden superior

Una función de orden superior es aquella que puede recibir otra función y usarla para realizar alguna tarea. Las funciones más populares de este tipo son `map`, `filter` y las funciones de plegado `foldr` y `foldl`. Todas estas son implementadas de forma recursiva, a continuación se muestra su implementación y usos[4].

Implementación de la funcion `map`:

```
;; Función 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))]))
```

Código 11: *Implementación de map*

[4]No es necesario copiar la implementación, éstas se encuentran definidas en el núcleo de Racket

```
> (map add1 '(1 2 3 4))
'(2 3 4 5)
```

Implementación de la función `filter`:

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

Código 12: *Implementación de filter*

```
> (filter even? '(1 2 3 4))
'(2 4)
```

Implementación de la función `foldr`:

```
;; 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))]))
```

Código 13: *Implementación de foldr*

```
> (foldr + 0 '(1 2 3 4))
10
> (foldr cons empty '(1 2 3 4))
'(1 2 3 4)
```

Implementación de la función `foldl`:

```
;; 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)]))
```

Código 14: *Implementación de foldl*

```
> (foldl + 0 '(1 2 3 4))
10
> (foldl cons empty '(1 2 3 4))
'(4 3 2 1)
```

Funciones anónimas (lambdas)

Al usar funciones de orden superior, es normal que se necesite alguna función que no esté definida en el núcleo de Racket y se tenga que definir una nueva función que no es usada mas que para aplicar la función de orden superior. En estos casos, no importa el nombre de la función, sólo su comportamiento. Existen funciones de este tipo y son llamadas *funciones anónimas* o *lambdas*[5].

Para definir una lambda, se usa la siguiente sintaxis:

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

También se puede poner el símbolo λ directamente en Racket tecleando `Ctrl + Alt + \`.

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

Cuando se define una función, realmente siempre hay una lambda por detrás, sólo se le asigna un nombre usando azúcar sintáctica. Por ejemplo, estas funciones son equivalentes:

[5]Se toma este nombre del cálculo lambda. Los lenguajes funcionales están basados en el mismo.

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

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

Código 15: *Función identidad*

También podría usarse una lambda para definir funciones auxiliares dentro de una sola definición. Por ejemplo:

```
;; 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))))
```

Código 16: *Promedio de una lista*

Como puede inferirse, la primitiva `letrec` permite a los identificadores llamarse a sí mismos recursivamente. Lo que hace en este caso es asignarle una lambda a los identificadores `suma` y `longitud` para después usarlos en el cuerpo de la asignación local. Cabe resaltar que una función lambda es aplicada como cualquier otra función.

1.6 Referencias

[1] Rodrigo Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.

[2] Eric Tánter, *PREPLAI: Scheme y Programación Funcional*, Primera edición, 2014. Disponible en: [http://users.dcc.uchile.cl/~etanter/preplai/] (Consultado el 26 de julio de 2017).

[3] Matthias Felleisen, Robert Findler, Matthew Flatt, Shriram Krishnamurthi, *How to Design Programs*, Segunda Edición, The Mit Press, 2017. Disponible en: [http://www.ccs.neu.edu/home/matthias/HtDP2e/] (Consultado el 26 de julio de 2017).

[4] Matthias Felleisen, David Van Horn, Conrad Barski, *Realm Of Racket*, Primera edición, No Starch Press, 2013.

[5] Favio E. Miranda, A. Liliana Reyes, Lourdes del C. González, P. Selene Linares, *Notas del curso de Lógica Computacional*, Semestre 2017-2, Facultad de Ciencias, UNAM.

Nota: Esta es una versión preliminar de las notas de laboratorio y están en constante actualización por lo que agradecemos que en caso de detectar algún error o si se desea hacer alguna observación se envíe un correo electrónico a las direcciones `manu@ciencias.unam.mx` y `karla@ciencias.unam.mx` con el asunto `Nota A1: Lenguajes de Programación`.