Lenguajes de Programación, 2022-1
Nota de clase A: Introducción a Racket

Karla Ramírez Pulido Manuel Soto Romero

20 de septiembre de 2021
Facultad de Ciencias UNAM

Expresiones en Racket

Conociendo Racket

Racket es el lenguaje de programación que se usa a lo largo de estas notas de clase. A continuación se listan algunas de sus principales características:

- Es un dialecto[1] de Lisp y un descendiente de Scheme por lo que hereda muchas características sintácticas y semánticas de estos lenguajes, por ejemplo el uso de paréntesis para delimitar expresiones y notación prefija.
- Es un lenguaje de programación que combina varios estilos de programación, principalmente el funcional, y es orientado a la creación de lenguajes de programación.
- Racket incluye, al igual que Lisp, muchos dialectos en su núcleo, para los fines de este curso, se hará uso, principalmente, del dialecto `plai` (*Programming Languages: Application and Interpration*) que incluye primitivas para definir intérpretes de manera sencilla.

Interacción con Racket

Para interactuar con Racket, es recomendable hacerlo a través de su Ambiente de Desarrollo Integrado[2] DrRacket, pues provee herramientas de resaltado de código, numeración de líneas, resaltado de paréntesis y otras herramientas visuales útiles especialmente cuando se está aprendiendo a usar este lenguaje, sin embargo, también es posible interactuar con Racket a través de la línea de comandos y un editor de texto.

DrRacket

DrRacket se compone de dos áreas, llamadas *área de definiciones* y *área de interacciones* ubicadas en la parte superior e inferior de la ventana respectivamente. La Figura 1 muestra esta pantalla. La parte superior de DrRacket es dónde se escriben las definiciones de funciones y programas de Racket, por otro lado la parte inferior funciona como si fuera una calculadora, se escribe una expresión, se presiona la tecla **`[Intro]`** y se imprime una respuesta, este programa es conocido como REPL[3] o intérprete.

Adicional a las primitivas que trae por defecto el intérprete de Racket, es posible cargar en el ambiente de Racket un archivo de definiciones propias, esto puede hacerse escribiendo directamente en el área de definiciones o cargando un archivo con extensión **`.rkt`** indicando el dialecto que Racket debe usar en la primera línea. Para el caso de estas notas, la primera línea de todos los archivos **`.rkt`** que se generen será **`#lang racket`**. Una vez que se ha cargado un archivo en el área de definiciones de DrRacket **`[Archivo→Abrir]`** se procede a ejecutarlo a través del botón **`[Ejecutar]`**.

[1] Variante de un lenguaje de programación con reglas sintácticas y semánticas similares.
[2] IDE, Integrated Development Eviroment.
[3] Del inglés Read-Eval-Print Loop.

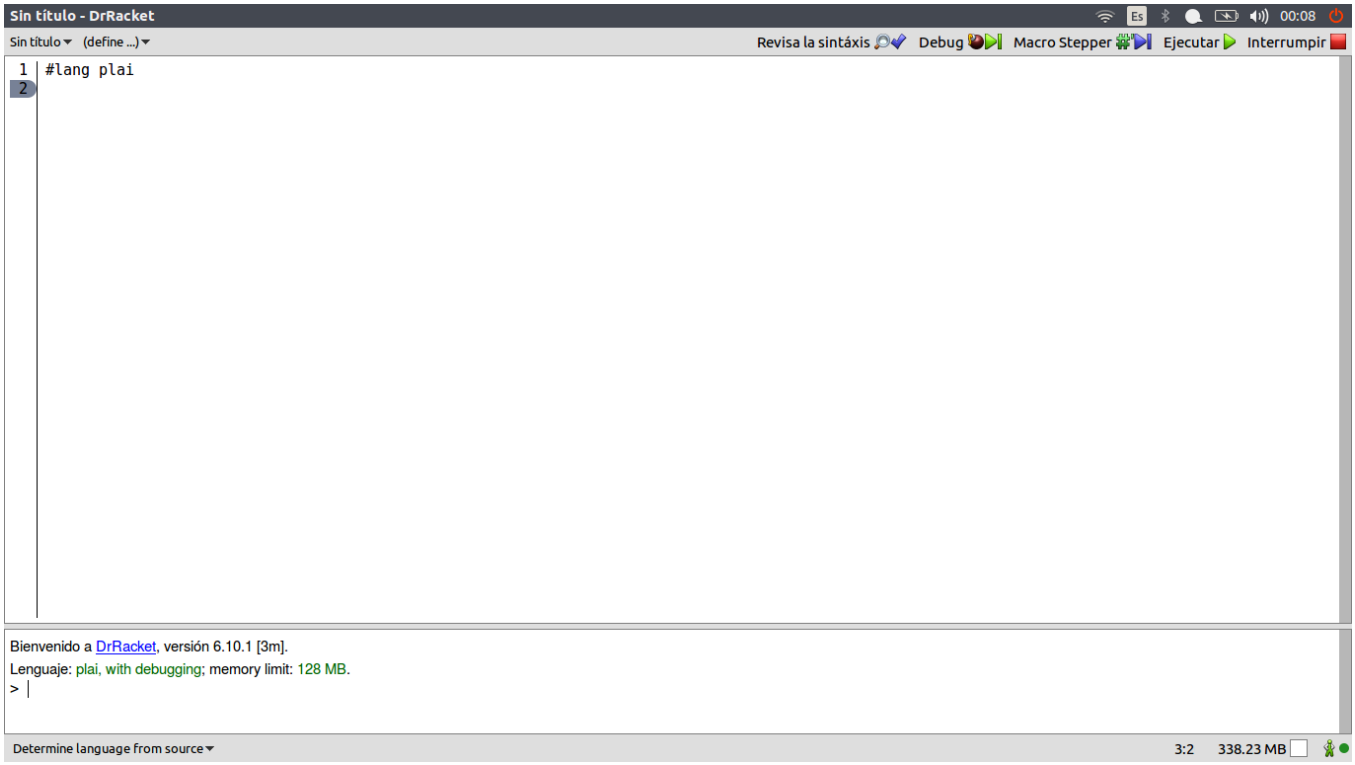

Figura 1: Pantalla principal de DrRacket

Tipos de datos primitivos

Para comenzar, se presentan los tipos de datos primitivos, se usa la forma interactiva (intérprete) para visualizar cómo es que se evalúan estas expresiones.

Tipos básicos

Booleanos (`boolean`)

Para representar a las constantes booleanas se tienen las formas **`#t`** y **`#f`** para representar verdadero y falso respectivamente, sin embargo, ambas cuentan con una versión con azúcar sintáctica[4] **`true`** y **`false`**.

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

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

Números (`number`)

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

- Números exactos: Enteros (**`integer`**), racionales (**`rational`**) y complejos (**`complex`**).
- Números inexactos: Flotantes (**`real`**) y complejos con flotantes.

Adicionalmente, en Racket no se tiene un límite para la longitud de números.

```
> -1
-1
> 7
7
> 1/7
1/7
> 1+7i
1 + 7i
> 5.5
5.5
> 5.25e8
5.28e8
> 832475892347582470845467892457209467O5
3247589234758247084546789245720946705
```

Caracteres

Para representar caracteres se usa la codificación Unicode. Los caracteres se representan anteponiendo los símbolos **`#\`**. Es posible usar el código Unicode correspondiente, por ejemplo **`#\u3123`**.

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

Cadenas

Las cadenas son agrupaciones de caracteres y 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 (*quote*). Se representan anteponiendo una comilla simple al inicio.

```
> 'manzana
'manzana
```

Funciones predefinidas

Como se mencionó al principio de esta nota, RACKET hace uso de paréntesis y usa notación prefija. Para usar una función, debe delimitarse la misma por paréntesis, el primer elemento después de abrir paréntesis "`(`" es el nombre de la función a aplicar, y se indican los argumentos de la función separando cada uno por espacios hasta cerrar paréntesis "`)`". A continuación se presentan algunos ejemplos.

Funciones lógicas

Se tienen funciones básicas de la lógica como son la negación, conjunción y disyunción.

```
> (not #t)
#f
> (and #t #t)
#t
> (and #t #t 5)
5
> (or #f #f)
#f
> (or #f 5 6)
5
```

Observación 0.1. Como puede apreciarse, en RACKET, todo lo que no sea explícitamente falso (`#f`) se considera verdadero.

Funciones aritméticas

```
> (+ 1 2 3)
6
> (- 2/3 1/3)
1/3
> (* 2 3 6)
36
> (/ 10 2 2)
5/2
> (expt 5 2)
25
> (sqrt 81)
9
```

Manejo de cadenas

```
> (string-length "Manzana")
7
> (string-ref "Manzana" 3)
#\z
> (substring "Manzana" 1 3)
"an"
> (string-append "Man" "zana")
"Manzana"
```

Observación 0.2. Por convención, en el nombre de las funciones de RACKET, se separa cada palabra del identificador por un guión medio.

Predicados

A aquellas funciones que regresan verdadero (`#t`) o falso (`#f`) para verificar alguna propiedad, se les llama predicados, por convención, el nombre de estas funciones finaliza con un signo de interrogación (`?`).

```
> (number? 5)
#t
> (char? #\a)
#t
> (zero? 10)
#f
```

Conversores

Las funciones que realizan conversiones entre tipos de datos son llamadas conversores o transformadores y por convención se nombran poniendo una flecha (`->`) entre los tipos de datos que se convertirán.

```
> (inexact->exact 1.0)
1
> (exact->inexact 1)
1.0
> (string->symbol "Manzana")
'Manzana
```

Condicionales

Para establecer estas condiciones se tienen principalmente dos primitivas: **`if`** y **`cond`**. La primera primitiva es usada por lo general cuando se tiene una única condición mientras que la segunda se usa cuando se tienen dos o más condiciones.

Condicional `if`

La sintaxis del condicional **`if`** es la siguiente:

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

El primer valor **`<condición>`** debe ser una expresión booleana, el segundo valor **`<then-expr>`** se evaluará siempre que la condición sea verdadera, y el tercer valor **`<else-expr>`** se ejecutará cuando no lo sea.

```
> (if (< 10 2) 'manzana 'pera)
'pera
> (if (< 2 10) 'manzana 'pera)
'manzana
```

Condicional `cond`

La sintaxis del condicional **`cond`** es la siguiente:

```
(cond
   [<condición> <then-expr>]+
   [else <else-expr>]?)
```

Se tienen una serie de expresiones de la forma **`[<condición> <then-expr>]`** representando los posibles casos y el valor a devolver en caso de que se cumpla la condición. Opcionalmente se tiene un caso **`else`** que se evalúa cuando ninguna de las condiciones anteriores haya sido verdadera.

```
> (cond
    [(< 10 2) 'manzana]
    [(< 2 10) 'pera]
    [else 'fresa])
'pera
> (cond
    [(< 2 2) 'manzana]
    [(< 10 2) 'pera]
    [else 'fresa])
'manzana
> (cond
    [(< 10 2) 'manzana]
    [(< 20 10) 'pera]
    [else 'fresa])
'fresa
```

Listas

Las listas, en RACKET se definen recursivamente como sigue:

Definición 0.1. *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.*

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)
```

Representación de listas

Adicional a la notación de listas a través de **`cons`**, existen otras formas de definir listas como puede ser: mediante la función **`list`** o mediante el mecanismo de citado **`quote`**.

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 individualmente.

Ejemplo 0.1. Lista del 1 al 5 usando **`list`**.

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

□

Ejemplo 0.2. Lista de expresiones aritméticas con **`list`**.

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

□

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. Para definir listas con **`quote`**, basta con anteponer el símbolo `'`.

Ejemplo 0.3. Lista del 1 al 5 usando **`quote`**.

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

□

Ejemplo 0.4. Lista de expresiones aritméticas con **`quote`**.

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

□

Manipulación de listas

Algunas funciones predefinidas para manipular listas:

`empty?` Indica si la lista recibida es vacía.

```
> (empty? '(1 2 3))
#f
> (empty? '())
#t
```

`length` Obtiene la longitud de una lista.

```
> (length '(1 2 3))
3
> (length '())
0
```

`take` Toma los primeros n elementos de una lista.

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

`drop` Elimina los primeros n elementos de una lista.

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

`append` Concatena dos listas.

```
> (append '(h o) '(l a))
'(h o l a)
```

Definición de Funciones

Adicional a las funciones predefinidas sobre los tipos de datos básicos, es posible definir funciones propias. El diseño de funciones es un proceso que debe seguir una serie de pasos ordenados y bien especificados para garantizar su correcto funcionamiento, a continuación se presenta un método de definición de funciones y se presentan algunos ejemplos junto a su solución.

Método de definición de funciones

Cuando se define una función que cumple cierto objetivo, se recomienda seguir los siguientes pasos en el orden en que se indica[5]:

- Entender lo que la función tiene que hacer.
- Escribir la descripción de la función a través de los comentarios.
- Escribir su contrato, es decir, su tipo (cuántos parámetros recibe, de qué tipo son, cuál es el valor de regreso) a través de los comentarios.
- Escribir pruebas asociadas a esta función, sobre los distintos posibles datos de entradas que pueda tener (casos significativos).
- Finalmente, implementar el cuerpo de la función.

[5] `https://users.dcc.uchile.cl/~etanter/preplai/defun.html`

Para especificar el contrato y la descripción de la función, se usan comentarios. En Racket existen dos tipos de comentarios:

De una línea

```
;; Comentario de una línea
```

De varias líneas

```
#| Este es un comentario
   de varias líneas |#
```

Por otro lado, escribir pruebas, obliga al programador a definir desde un inicio cómo se va a usar la función. Para definir pruebas, simplemente se escriben casos significativos y se especifica qué deben devolver. Por ejemplo, sabemos que la suma de 1 con 2 es 3, por lo tanto se escribe la prueba:

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

Finalmente, para definir una función, se usa la siguiente sintaxis:

```
(define (<nombre> <parámetro>*)
   <cuerpo>)
```

Se debe especificar un nombre para la función, una secuencia de parámetros de entrada separados por espacios (puede no haber parámetros) y un cuerpo. Es importante recordar, que las definiciones de funciones deben realizarse dentro de un archivo con extensión **.rkt** o escribirlas directamente en el área de definiciones de DrRacket.

Ejemplos de definición de funciones

Ejemplo 0.5. Definir la función **promedio-3** tal que **(promedio-3 x y z)** es el promedio de los números **x**, **y** y **z**. Por ejemplo:

```
(promedio-3 1 3 8)  = 4
(promedio-3 -1 0 7) = 2
(promedio-3 -3 0 3) = 0
```

Solución:

```
1 ;; Función que calcula el promedio de tres números.
2 ;; promedio-3: number number number → number
3 (define (promedio-3 x y z)
4    (/ (+ x y z) 3))
```

Código 1: Promedio de tres números

□

Ejemplo 0.6. Definir la función **suma-monedas** tal que **(sumaMonedas a b c d e)** es la suma de los pesos correspondiente a monedas de 50 centavos, 1 peso, 2 pesos, 5 pesos y 10 pesos respectivamente. Por ejemplo:

```
(suma-monedas 0 0 0 0 1) = 10
(suma-monedas 0 0 8 0 3) = 46
(suma-monedas 1 1 1 1 1) = 18.50
```

Solución:

```
1 ;; Función que calcula la suma de los pesos correspondiente a monedas
2 ;; de 50 centavos, 1 peso, 2 pesos, 5 pesos y 10 pesos
3 ;; respectivamente.
4 ;; suma-monedas: number number number number number → number
5 (define (suma-monedas a b c d e)
6    (+ (* a 0.5) (* b 1) (* c 2) (* d 5) (* e 10)))
```

Código 2: Suma de monedas

□

Ejemplo 0.7. Definir la función **volumen-esfera** tal que **(volumen-esfera r)** calcula el volumen de una esfera de radio **r**. Por ejemplo:

```
(volumen-esfera 10) = 4188.7902
```

Solución:

```
1 ;; Función que calcula el volumen de la esfera de radio r.
2 ;; volumen-esfera: number → number
3 (define (volumen-esfera r)
4    (* 4/3 pi (expt r 3)))
```

Código 3: Volumen de esfera

□

Ejemplo 0.8. Definir la función **area-circulo** tal que **(area-circulo d)** calcula el área de un círculo de diámetro **d**. Por ejemplo:

```
(area-circulo 10) = 78.53
(area-circulo 4)  = 12.56
(area-circulo 16) = 201.06
```

Solución:

```
1 ;; Función que calcula el área de un círculo dado su diámetro.
2 ;; area-circulo: number → number
3 (define (area-circulo d)
4    (* pi (/ d 2) (/ d 2)))
```

Código 4: Área de círculo

□

Recursión y Funciones de Orden Superior

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.

Recursión sobre listas

Ejemplo 0.9. 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 5: Longitud de una lista

□

Ejemplo 0.10. 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

```
;; 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 6: Lista sin los primeros n elementos

□

Ejemplo 0.11. 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 7: Primeros n elementos de una lista

□

Ejemplo 0.12. 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 8: Indica si un elemento pertenece a una lista

□

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

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.

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

Código 9: Implementación de map

Ejemplo 0.13. 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

```
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
```

[6] Pertenecientes a la llamada programación origami.

```
;; 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 10: Función meses

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

□

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 11: Implementación de filter

Ejemplo 0.14. 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
```

```
11 (define (ternas-pitagoricas xs)
12    (filter terna-pitagorica? xs))
```

Código 12: 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.

□

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):

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

Código 13: Implementaciones de foldr y foldl

Ejemplo 0.15. 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:

```
1 ;; Función que suma los elementos de una lista.
2 ;; suma-listar: (listof number) → number
3 (define (suma-listar xs)
4    (foldr + 0 xs))
5
6 ;; Función que suma los elementos de una lista.
7 ;; suma-listal: (listof number) → number
8 (define (suma-listal xs)
9    (foldl + 0 xs))
```

Código 14: 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.

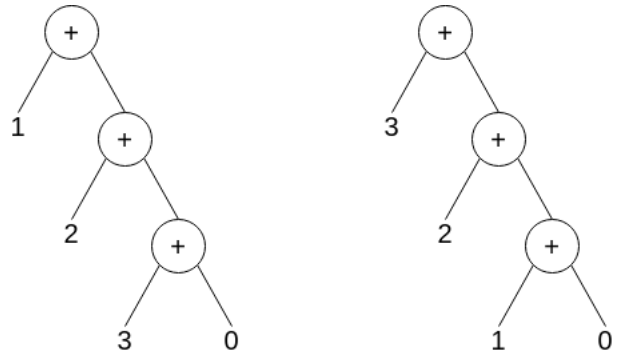

Figura 1: *Árboles de evaluación de suma-listar y suma-listal*

□

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)
```

Definición de Tipos de Datos

Adicional a los tipos de datos primitivos de Racket, es posible que el programador defina sus propios tipos de datos. La variante **plai** provee algunas primitivas que permiten al programador definir datos y manipularlos a través de funciones generadas al crear el dato o mediante la técnica de reconocimiento de patrones.

Definición de tipos

Para definir tipos de datos, **plai** provee la primitiva **define-type**, su sintaxis es la siguiente:

```
(define-type <NombreDelTipo>
   [<nombre-constructor> (<param1> <tipo1>?)*]+)
```

La definición de un tipo de dato se compone de:

- Un nombre para el tipo de dato. El nombrado de estos tipos usa notación Pascal, es decir, cada palabra del identificador inicia con una letra mayúscula.
- Un listado de constructores que a su vez se componen de:
  - Un nombre para el constructor. El identificador separa cada palabra por un símbolo de guión (-) y se escribe en minúsculas.
  - Un listado de parámetros que puede ser vacío. Cada parámetro tiene un identificador y un tipo de dato. El tipo de dato se especifica mediante predicados, por ejemplo: **number?**, **boolean?**, etcétera.

    (parametro tipo?)

Ejemplo 0.16. Definir un tipo de dato **Arbol** que representa árboles binarios.

Solución:

```
1 ;; Función que permite definir estructuras genéricas.
2 (define (any? a) #t)
3
4 (define-type Arbol
5    [hoja (elem any?)]
6    [nodo (elem any?) (izq Arbol?) (der Arbol?)])
```

Código 15: Definición del tipo ArbolBinario

Se aprecia en el Listado de código 1 que el tipo **Arbol** es recursivo, pues los parámetros de los constructores **izq** y **der** deben de ser del mismo tipo. Una vez definido el tipo de dato, pueden usarse los constructores **hoja** y **nodo** para construir árboles.

```
> (hoja 1)
(hoja 1)
> (nodo 1 (hoja 2) (hoja 3))
(nodo 1 (hoja 2) (hoja 3))
```

Observación 0.3. Para poder usar los constructores de un tipo de dato, deben aplicarse como cualquier otra función, esto es, delimitarse por paréntesis.

Observación 0.4. El predicado **any?** permite definir estructuras genéricas, heterogéneas al regresar **#t** con cualquier entrada.

□

Funciones predefinidas

Una vez creado un nuevo tipo de dato, **plai** provee funciones predefinidas para el tipo de dato que pueden usarse para definir funciones sobre el mismo. A continuación se listan estas funciones:

- Un predicado para el tipo de dato definido.

  Arbol?: **a** $\rightarrow$ ***boolean***

- Un predicado por cada uno de los constructores del tipo definido.

  hoja?: **a** $\rightarrow$ ***boolean***
  nodo?: **a** $\rightarrow$ ***boolean***

- Una función de acceso para cada uno de los parámetros de cada constructor del tipo definido.

  hoja-elem: Arbol $\rightarrow$ ***a***
  nodo-elem: Arbol $\rightarrow$ ***a***
  nodo-izq: Arbol $\rightarrow$ ***Arbol***
  nodo-der: Arbol $\rightarrow$ ***Arbol***

- Una función modificadora para cambiar el valor de cada uno de los parámetros de cada constructor del tipo definido.

  set-hoja-elem!: Arbol a $\rightarrow$ ***void***
  set-nodo-elem!: Arbol a $\rightarrow$ ***void***
  set-nodo-izq!: Arbol Arbol $\rightarrow$ ***void***
  set-nodo-der!: Arbol Arbol $\rightarrow$ ***void***

Observación 0.5. Por convención, las funciones modificadoras inician con la palabra **set** e incluyen un signo de admiración al final del identificador (**!**).

Observación 0.6. A lo largo de estas notas se evita el uso de funciones modificadoras, pues traen consigo efectos secundarios que violan el principio de transparencia referencial del estilo de programación funcional. Se usan las mismas para casos muy particulares.

Ejemplo 0.17. Definir la función **numero-hojas** tal que **(numero-hojas a)** es el número de hojas del árbol **a**. Por ejemplo,

```
(numero-hojas (hoja 1))                    = 1
(numero-hojas (nodo 1 (hoja 2) (hoja 3))) = 2
```

Solución:

```
1 ;; Función que obtiene el número de hojas de un árbol binario.
2 ;; numero-hojas: Arbol → number
3 (define (numero-hojas a)
4    (if (hoja? a)
5        1
6        (+ 1 (numero-hojas (nodo-izq a)) (numero-hojas (nodo-der a)))))
```

Código 16: Número de hojas de un árbol

□

Ejemplo 0.18. Definir la función **contiene?** tal que **(contiene? e a)** indica si el elemento e se encuentra en el árbol a.

```
(contiene? 1 (hoja 1))                    = #t
(contiene? 1 (hoja 2))                    = #f
(contiene? 3 (nodo 2 (hoja 1) (hoja 3)) = #t
```

Solución:

```
;; Predicado que indica si un elemento está contenido en un árbol.
;; contiene?: a Arbol → boolean
(define (contiene? e a)
   (cond
      [(hoja? a) (equal? e (hoja-elem a))]
      [(nodo? a)
         (or (equal? e (nodo-elem a))
             (contiene? e (nodo-izq a))
             (contiene? e (nodo-der a)))]))
```

Código 17: Indica si un elemento está contenido en un árbol

□

Reconocimiento de patrones

Al igual que con otros tipos de datos, puede usarse la técnica de reconocimiento de patrones sobre los tipos definidos mediante la primitiva **`define-type`** ya sea a través de la primitiva **`type-case`** como de la, ya estudiada, primitiva **`match`**. La sintaxis de ambas primitivas se presenta a continuación:

`type-case`:

```
(type-case <tipo> <identificador>
   [<nombre-constructor> (<parámetro>*) <expresión-regreso>]+
   [else <expresión-regreso>]?)
```

`match`:

```
(match <identificador>
   [(<nombre-constructor> <parámetro>*) <expresión-regreso>]+
   [else <expresión-regreso>]?)
```

Observación 0.7. Algunas observaciones sobre estas primitivas:

1. El caso **`else`** de **`type-case`** es opcional, a menos que no se realice el reconocimiento de todos los patrones del tipo de dato, en cuyo caso es obligatorio. Esta primitiva debe cubrir todos los casos posibles o lanzará un error.
2. El caso **`else`** de **`match`** es completamente opcional y no es necesario reconocer todos los patrones del tipo de dato.

Ejemplo 0.19. Modificar la función **`numero-hojas`** para que hagan uso de las primitivas **`type-case`** y **`match`.**

Solución:

```
;; Función que obtiene el número de hojas de un árbol binario.
;; numero-hojas1: Arbol → number
(define (numero-hojas1 a)
   (type-case Arbol a
      [hoja (x) 1]
      [nodo (x i d) (+ 1 (numero-hojas1 i) (numero-hojas1 d))]))

;; Función que obtiene el número de hojas de un árbol binario.
;; numero-hojas2: Arbol → number
(define (numero-hojas2 a)
   (match a
      [(hoja _) 1]
      [(nodo _ i d) (+ 1 (numero-hojas2 i) (numero-hojas2 d))]))
```

Código 18: Número de hojas de un árbol

□

Ejemplo 0.20. Modificar el predicado **`contiene?`** para que haga uso de **`type-case`** y **`match`**.

Solución:

```
;; Predicado que indica si un elemento está contenido en un árbol.
;; contiene1?: a Arbol → boolean
(define (contiene1? e a)
   (type-case Arbol a
      [hoja (x) (equal? e x)]
      [nodo (x i d)
         (or (equal? e x) (contiene1? e i) (contiene1? e d))]))

;; Predicado que indica si un elemento está contenido en un árbol.
;; contiene2?: a Arbol → boolean
(define (contiene2? e a)
   (match a
      [(hoja x) (equal? e x)]
      [(nodo x i d)
         (or (equal? e x) (contiene1? e i) (contiene1? e d))]))
```

Código 19: Indica si un elemento está contenido en el árbol

□

Ejemplo 0.21. Definir la función `aplana` mediante las primitivas `type-case` y `match` tal que `(aplana a)` es la lista de elementos del árbol a en *inorden*.

Solución:

```
;; Función que aplana un árbol binario.
;; aplana1: Arbol → (listof any?)
(define (aplana1 a)
   (type-case Arbol a
      [hoja (x) (list x)]
      [nodo (x i d) (append (list x) (aplana1 i) (aplana1 d))]))

;; Función que aplana un árbol binario.
;; aplana2: Arbol → (listof any?)
(define (aplana2 a)
   (match a
      [(hoja x) (list x)]
      [(nodo x i d) (append (list x) (aplana2 i) (aplana2 d))]))
```

Código 20: Lista de elementos de un árbol

□

Ejemplo 0.22. Definir la función `map-arbol` mediante las primitivas `type-case` y `match` tal que `(map-arbol f a)` es el árbol resultante de aplicar la función `f` a cada elemento del árbol `a`.

Solución:

```
;; Aplica una función a cada elemento de un árbol.
;; map-arbol1: procedure Arbol → Arbol
(define (map-arbol1 f a)
   (type-case Arbol a
      [hoja (x) (hoja (f x))]
      [nodo (x i d) (nodo (f x) (map-arbol1 f i) (map-arbol1 f d))]))

;; Aplica una función a cada elemento de un árbol.
;; map-arbol2: procedure Arbol → Arbol
(define (map-arbol2 f a)
   (match a
      [(hoja x) (hoja (f x))]
      [(nodo x i d) (nodo (f x) (map-arbol1 f i) (map-arbol1 f d))]))
```

Código 21: Aplica una función a cada elemento del árbol

Ejercicios

1. Transforma las siguientes expresiones aritméticas en expresiones de RACKET y pruébalas en el *área de interacciones*. Anota el resultado de evaluar cada expresión.

   (a) $(4 \times 7) - (13 + 5)$

   (b) $(3 \times (4 + (-5 - 3)))$

   (c) $(2.5 \div (5 \times (1 \div 10)))$

   (d) $5 \times ((537 \times (98.3 \div (375 - (2.5 \times 153)))) + 255)$

2. Transforma las siguientes fórmulas en expresiones de RACKET y pruébalas en el *área de interacciones*.

   (a) Ecuación general de segundo grado:

   $$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad y \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

   con $a = 3$, $b = 6$ y $c = 2$.

   (b) Distancia entre dos puntos:

   $$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

   con $x_1 = 5$, $x_2 = -4$, $y_1 = -3$ y $y_2 = 6$.

   (c) Teorema de Pitágoras:

   $$c = \sqrt{a^2 + b^2}$$

   con $a = 3.7$ y $b = 5.4$.

   (d) Evaluación de polinomios:

   $$y = 2x^3 - 4x^2 + 8x - 2$$

   con $x = 6$.

3. Traduce la siguiente oración a una expresión en RACKET usando el condicional `if` y pruébala en la *ventana de interacciones* con `a = 1729` y luego con `a = 1836`. El resultado debe ser una cadena.

   Si el residuo de la división de un número es un número a entre 2 es 0, decimos que es un número par, de lo contrario es impar.

4. Dada una constante **a** con un valor entre 1 y 12, dar una expresión en RACKET usando cond para que dependiendo del valor de **a**, regrese una cadena con el mes equivalente. Por ejemplo, si tenemos **(define a 10)**, el condicional debe regresar **"Octubre"**.

5. Indica la salida de cada una de las siguientes expresiones y verifica el resultado en DRRACKET.

   (a) `(cons (+ 1 2) (cons (+ 3 4) (cons (+ 5 6) empty)))`

   (b) `(list (+ 1 2) (+ 3 4) (+ 5 6))`

   (c) `'((+ 1 2) (+ 3 4) (+ 5 6))`

   (d)
   ```
   (define a '(1 2 3))
   (define b '(4 5 6))
   (+ (first a) (first b))
   ```

6. Definir la función **area-total** que dada la generatriz y el diámetro de la base de un cono circular recto, calcular el área total del mismo.

$$At = \pi rg + \pi r^2$$

7. Definir el predicado **decremental?** que dados cuatro números indica si se encuentran ordenados de forma decremental.

8. Definir la función **pares?** que dada una lista, indica si todos los elementos contenidos en ésta son pares.

9. Definir la función **multiplica** que dada una lista, multiplica todos los elementos contenidos en la misma.

10. Definir la función *recursiva* **(entierra s n)** que dado un símbolo lo *entierra* n número de veces. Es decir, se deberán anidar $n-1$ listas hasta que se llegue a la lista que tiene como único elemento al símbolo correspondiente. Por ejemplo:

    ```
    (entierra 'foo 5) => (((((foo)))))
    ```

11. Definir la función *recursiva* **(reemplaza s t l)** que dada una lista de símbolos, reemplaza las apariciones del símbolo **s** por **t**. Por ejemplo:

    ```
    (reemplaza 'foo 'goo '(foo foo goo hoo)) => '(goo goo goo hoo)
    ```

12. Definir la función *recursiva* **(listoftype? p l)** que verifica que todos los elementos de una lista sean del tipo indicado por el predicado **p**. Por ejemplo:

    ```
    (listoftype? symbol? '(s a 2)) => #f
    ```

13. Definir un tipo de dato persona que sea representado mediante el nombre, edad, peso y estatura. Una vez definido el tipo de dato, definir una función **(imc p)** que dada una persona calcule el índice de masa corporal de la misma. Por ejemplo:

    ```
    (define p1 (persona "Juan" 19 65.0 1.75))
    (imc p1) => 21.224489795918366
    ```

Referencias

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

[2] Matthias Felleisen, Robert B. Findler, et.al., *How to Desing Programs*, MIT Pres, 2019.

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

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

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