Lenguajes de Programación, 2019-2
Nota de laboratorio 1: Introducción a Racket

Manuel Soto Romero

7 de febrero de 2019

Facultad de Ciencias, UNAM

1. Conociendo Racket

Racket es el lenguaje de programación que se usa a lo largo de estas notas de laboratorio, 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 multiparadigma, principalmente funcional, orientado a la creación de lenguajes de programación. Se enfatiza en estas notas el uso de aspectos puros de la programación funcional, al evitar que ocurran *efectos secundarios*, garantizando así que se cumpla el principio de *transparencia referencial*.
- Racket incluye, al igual que Lisp, muchos dialectos en su núcleo, para los fines de este curso, se hará uso del dialecto `plai` (*Programming Languages: Application and Interpration*) que incluye primitivas para definir intérpretes de manera sencilla.

1.1. 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.

1.1.1. 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.

[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.

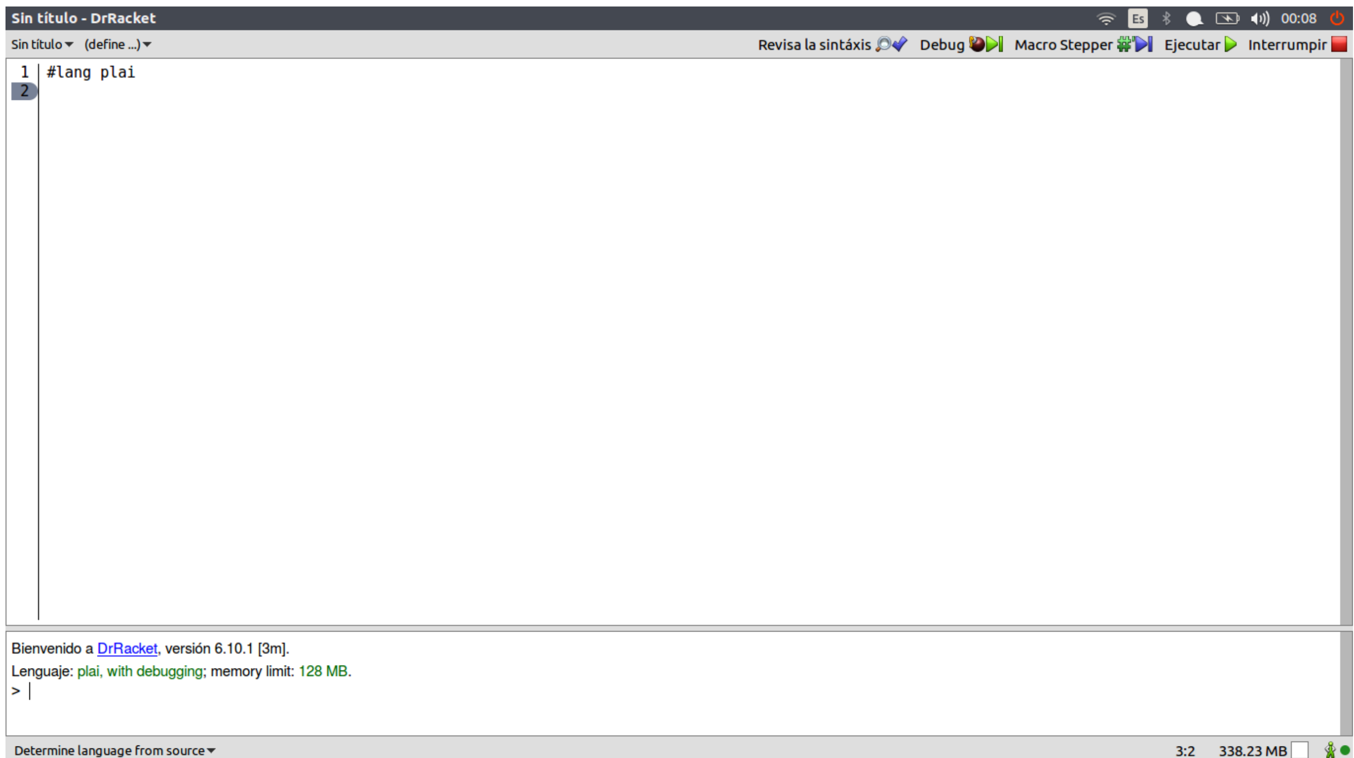

Figura 1: Pantalla principal de DrRacket

Adicional al intérprete de Racket, es posible cargar en el ambiente de Racket un archivo de definiciones, esto puede hacerse escribiendo directamente en el área de definiciones o cargando un archivo con extensión `.rkt` indicando el dialecto que Racket debe usar como primera línea. Para el caso de estas notas, la primera línea de todos los archivos `.rkt` que se generen será `#lang plai`. 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]`.

1.1.2. Línea de comandos

Para ejecutar Racket vía terminal, basta con escribir el comando `racket` esto cargará automáticamente el intérprete (REPL) donde se pueden escribir expresiones para ser evaluadas.

```
$ racket
Welcome to Racket v7.0.
```

Para cargar un archivo con extensión `.rkt`, se debe abrir el intérprete de Racket y especificar el archivo con las definiciones correspondientes a través de la primitiva `enter!`.

```
$ racket
Welcome to Racket v7.0.
> (enter! ‘‘archivo.rkt’’)
‘‘archivo.rkt’’>
```

Para definir estos archivos `.rkt` puede usarse cualquier editor de textos como Emacs, Vi, Sublime Text, Atom, Gedit, o incluso DrRacket.

2. 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.

2.1. Tipos básicos

Booleanos (`boolean`)

Para representar booleanos, se tienen las constantes lógicas `#t` y `#f` para representar verdadero y falso respectivamente, sin embargo, se cuenta con una versión con azúcar sintáctica[4] `true` y `false`.

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

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`).

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

- 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
> 8324758923475824708454678924572094670 5
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 de 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
```

Existen, en Racket, otros tipos básicos, su representación y usos se discutirán más adelante.

2.2. Funciones predefinidas

Como se mencionó al principio de esta nota, Racket hace uso de paréntesis y usa notación prefija. Para usar una función sobre los tipos básicos, 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. Como puede apreciarse, 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. Por convención, en el nombre de las funciones de Racket, se separa cada palabra 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
```

3. 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.

3.1. 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 su contrato, o sea, su tipo (cuántos parámetros, de qué tipo, qué regresa) a través de los comentarios.
- Escribir la descripción de la función a través de los comentarios.
- Escribir pruebas unitarias asociadas a esta función, sobre los distintos posibles datos de entradas que pueda tener (casos significativos).
- Finalmente, implementar el cuerpo de la función.

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

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

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

Por otro lado, para definir pruebas unitarias, se hace uso, en estas notas, de la función `test` que provee el dialecto `plai`. Basta con conocer el valor esperado de la función con cierta entrada y compararlo con la llamada a la misma. Por ejemplo, si se hiciera una prueba unitaria para la función suma (+) con los parámetros 1 y 2, el valor esperado sería 3, por lo que la prueba unitaria quedaría como sigue:

```
(test (+ 1 2) 3)
```

Una prueba exitosa mostrará el mensaje `good`, mientras que una prueba incorrecta mostrará el mensaje `bad` en pantalla.

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.

3.2. Ejemplos de definición de funciones

Ejemplo 1. 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. Función `promedio-3`:

```
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))
5
6 (test (promedio-3 1 3 8)  4)
7 (test (promedio-3 -1 0 7) 2)
8 (test (promedio-3 -3 0 3) 0)
```

Listado de código 1: Promedio de tres números

□

Ejemplo 2. 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. Función `suma-monedas`:

```
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
5  ;;                    -> number
6  (define (suma-monedas a b c d e)
7     (+ (* a 0.5) (* b 1) (* c 2) (* d 5) (* e 10)))
8
9  (test (suma-monedas 0 0 0 0 1) 10)
10 (test (suma-monedas 0 0 8 0 3) 46)
11 (test (suma-monedas 1 1 1 1 1) 18.50)
```

Listado de código 2: Suma de monedas

□

Ejemplo 3. 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. Función `volumen-esfera`:

```
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)))
5
6 (test (volumen-esfera 10) 4188.7902)
```

Listado de código 3: Volumen de esfera

□

Ejemplo 4. 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. Función `area-circulo`:

```
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)))
5
6 (test (area-circulo 10) 78.53)
7 (test (area-circulo 4)  12.56)
8 (test (area-circulo 16) 201.06)
```

Listado de código 4: Área de círculo

□

4. Asignaciones locales

En el Ejemplo 4, se define una función que calcula el área de un círculo a partir de su diámetro. La línea 4 del Listado código 4, que implementa este código, muestra un cálculo repetido: la división del parámetro `d` entre 2. Este tipo de cálculo resulta ser inefiniciente, por lo que Racket provee de la primitiva `let` que permite definir variables con alcance local. La primitiva `let` tiene la siguiente sintaxis:

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

Esta primitiva permite asignar un nombre a expresiones a través de variables con alcance local. De esta forma, se obtiene una expresión más eficiente, en cuanto a la evaluación se refiere, debido a que el valor asociado a la variable o asignación local, sólo se calcula una vez.

Ejemplo 5. Modificar la función `area-circulo` para que haga uso de la primitiva `let` y evite cálculos repetitivios.

Solución. Función `area-circulo2` usando `let`:

```
1 ;; Función que calcula el área de un círculo dado su diámetro.
2 ;; area-circulo2: number -> number
3 (define (area-circulo2 d)
4    (let ([r (/ d 2)])
5       (* pi r r)))
```

```
6
7 (test (area-circulo2 10) 78.53)
8 (test (area-circulo2 4)  12.56)
9 (test (area-circulo2 16) 201.06)
```

Listado de código 5: Área de círculo usando la primitiva let

□

Una limitante de las primitiva let, es la imposibilidad de definir variables en términos de otras definidas anteriormente. Por ejemplo, la siguiente expresión, lanza un error al escribirla en el intérprete de Racket:

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

La expresión lanza un error, pues al tratar de inicializar la variable `b`, no se conoce el valor para la variable `a`. Una forma de darle la vuelta a este tipo de expresiones es mediante la anidación de expresiones `let`, por ejemplo, la siguiente expresión es correcta y regresa `6` como resultado:

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

Sin embargo, esta forma de escribir expresiones puede llegar a ser tediosa para el usuario, por lo que Racket provee de una versión con azúcar sintáctica llamada `let*`:

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

5. Condicionales

De la misma forma que en matemáticas, en Racket pueden definirse funciones por partes de acuerdo a ciertas condiciones. Para establecer estas condiciones existen 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.

5.1. 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.

Ejemplo 6. Definir la función `valor-absoluto` tal que `(valor-absoluto x)` es el valor absoluto de un número entero. Por ejemplo:

```
(valor-absoluto 1729) = 1729
(valor-absoluto -265) = 265
```

Solución. Función `valor-absoluto`:

```
1 ;; Función que calcula el valor absoluto de un número entero.
2 ;; valor-absoluto: number -> number
3 (define (valor-absoluto x)
4    (if (< x 0)
5        (* x -1)
6        x))
7
8 (test (valor-absoluto 1729) 1729)
9 (test (valor-absoluto -265) 265)
```

Listado de código 6: Valor absoluto de un número

La condición se encuentra en la línea 4 `(< x 0)`, si el valor de entrada es menor que cero, entonces se multiplicará el mismo por `-1` (línea 5) y en caso contrario se regresa el valor tal cual (línea 6).

□

5.2. 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.

Ejemplo 7. Definir la función `nombre-mes` tal que `(nombre-mes n)` es el nombre del mes representado por el número entero `n`. Por ejemplo:

```
(nombre-mes 8)  = "Agosto"
(nombre-mes 10) = "Octubre"
(nombre-mes 11) = "Noviembre"
```

Solución. Función `nombre-mes`:

```
;; 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.")]))

(test (nombre-mes 8)  "Agosto")
(test (nombre-mes 10) "Octubre")
(test (nombre-mes 11) "Noviembre")
(test/exn (nombre-mes 25) "Mes inválido.")
```

Listado de código 7: Función que obtiene el nombre de un mes dado el número que lo representa

Las líneas 6 a 17 representan todas las posibles condiciones, mientras que la línea 18 representa el caso `else` en caso de que ninguna de las condiciones se cumpla. Si ninguna de las condiciones se cumple, se lanza un error indicando que se trata de un mes inválido.

Para lanzar errores, se usa la primitiva `error`, esta primitiva recibe un símbolo que indica el nombre de la función que está lanzando el error y una cadena que describe el error. Para validar este tipo de datos mediante pruebas unitarias se usa la primitiva `test/exn`, que recibe la descripción del error lanzado en lugar de un valor con el cual comparar.

□