Introducción a la Programación Funcional
Nota de clase 01: Expresiones en Racket*

Karla Ramírez Pulido Manuel Soto Romero

31 de agosto de 2020
Facultad de Ciencias UNAM

1.1. 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.

1.1.1. 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.

1.1.2. 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]`.

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

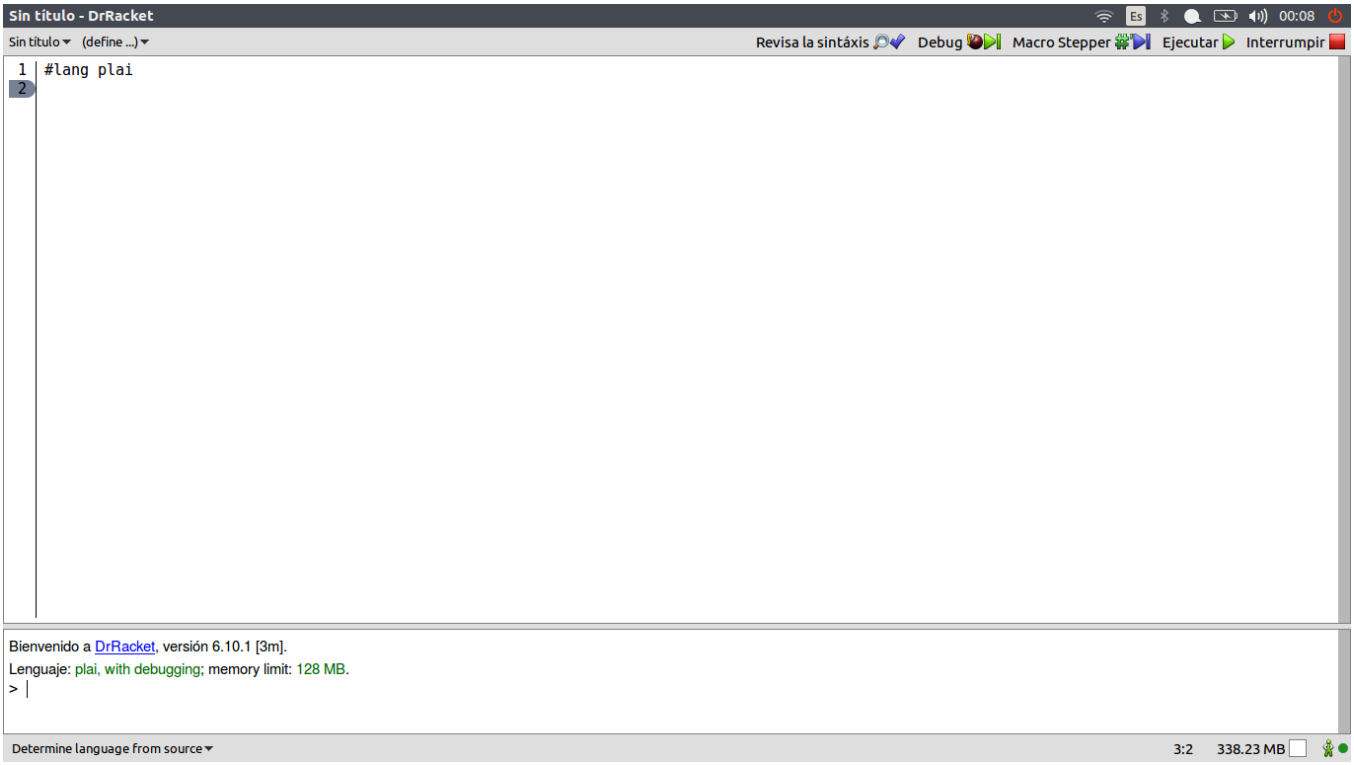

Figura 1: Pantalla principal de DrRacket

1.2. 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.

1.2.1. 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
```

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:

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

- 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
```

1.2.2. Funciones predefinidas

Como se mencionó al principio de esta nota, RACKET hace uso de paréntesis y usa notación prefija. Para usar una función, 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 1.1. 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 1.2. 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
```

1.3. 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.

1.3.1. 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.

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

1.3.2. 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.

```
> (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
```

1.4. Listas

Las listas, en Racket se definen recursivamente como sigue:

Definición 1.3. *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)
```

1.4.1. 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`.

1.4.2. 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 1.4. Lista del 1 al 5 usando `list`.

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

□

Ejemplo 1.5. Lista de expresiones aritméticas con `list`.

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

□

1.4.3. 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 1.6. Lista del 1 al 5 usando `quote`.

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

□

Ejemplo 1.7. Lista de expresiones aritméticas con `quote`.

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

□

1.5. 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)
```

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.