Lenguajes de Programación, 2018-1
Práctica 1: Introducción a Racket

Karla Ramírez Pulido J. Ricardo Rodríguez Abreu Manuel Soto Romero

Fecha de inicio: 9 de agosto de 2017
Fecha de entrega: 23 de agosto de 2017

Objetivos

- Aprender a usar el ambiente de desarrollo integrado, DrRacket, para escribir pequeños programas en el dialecto `plai` (*Programming Languages: Application and Interpretation*).
- Utilizar los tipos de datos básicos y funciones predefinidas de Racket.
- Definir funciones sencillas y otras más complejas que involucren el uso de condicionales y asignaciones locales.
- Usar las estructuras de datos del lenguaje y definir funciones recursivas sobre las mismas mediante el mecanismo de *apareamiento de patrones*[1].
- Utilizar funciones de orden superior sobre listas, conocer su implementación y usarlas en combinación con funciones anónimas (*lambdas*).

Descripción general

La práctica consiste en completar el cuerpo de varias funciones para que resuelvan determinados problemas. Cada función viene acompañada de distintas pruebas unitarias que deben pasar para saber que fueron completadas correctamente. No se permite modificar la firma de ninguna función ni usar primitivas que resuelvan directamente los ejercicios.

Archivos requeridos

El material de esta práctica consta de los siguientes archivos[2]:

- `practica1.rkt` archivo con las funciones a completar.
- `test-practica1.rkt` archivo con las pruebas unitarias de la práctica.

[1] Del inglés *Pattern Matching*.
[2] Los archivos pueden descargarse desde la página del curso `http://lenguajesfc.com`.

Desarrollo de la práctica

Ejercicio 1.1 (0.5 pts.) Completar el cuerpo de la función `rps` que toma dos números enteros positivos y eleva uno al otro, para luego sumar las raíces cuadradas de éstos, es decir, debe regresar $\sqrt{a^b} + \sqrt{b^a}$.

```
;; rps: number number -> number
(define (rps a b)
   ...)
```

```
> (rps 1 7)
3.6457513110645907
> (rps 2 9)
31.627416997969522
```

—

Ejercicio 1.2 (0.5 pts.) Completar el cuerpo de la función `area-heron` que encuentra el área de un triángulo dados sus lados, usando la fórmula de Herón. Usar la primitiva `let` para evitar cálculos repetitivos. Fórmula:

$$A = \sqrt{S\,(S-a)\,(S-b)\,(S-c)}$$

dónde S es el semiperímetro:

$$S = \frac{a+b+c}{2}$$

```
;; area-heron: number number number -> number
(define (area-heron a b c)
   ...)
```

```
> (area-heron 3 25 26)
36
> (area-heron 3 4 5)
6
```

—

Ejercicio 1.3 (1 pt.) Alicia y Bartolo quieren entrar a un antro, sin embargo, en la entrada hay un cadenero que sólo permite entrar a las parejas de acuerdo al estilo de su ropa. El estilo de la ropa se mide en una escala de 0 a 10. La respuesta del cadenero está dada de acuerdo a lo siguiente:

Si el estilo de alguno de los asistentes es de ocho o más, el cadenero responderá `'si`*, con la excepción de que si el estilo de alguno de los asistentes es de dos o menos, responderá* `'no`*. En otro caso, responderá* `'quiza`.

Completar el cuerpo del predicado `entra?` que determina si la pareja `a` `b` entrará al antro, usando condicionales.

```
;; entra?: number number -> symbol
(define (entra? a b)
   ...)
```

```
> (entra? 5 10)
'si
> (entra? 5 2)
'no
> (entra? 5 5)
'quiza
```

—

Ejercicio 1.4 (1 pt.) Completar el cuerpo de la función *recursiva* `apariciones` que calcula el número de apariciones de un dígito en un número entero positivo, recursivamente.

```
;; apariciones: number number -> number
(define (apariciones n m)
   ...)
```

```
> (apariciones 717 7)
2
> (apariciones 2 2)
1
> (apariciones 123 8)
0
```

—

Ejercicio 1.5 (1 pt.) Se dice que un par en una cadena *son dos caracteres idénticos, separados por un tercero*. Por ejemplo "`AxA`" es el par de "`A`". Los pares, además, pueden anidarse, por ejemplo, "`AxAxA`" contiene tres pares (dos de "`A`" y uno de "`x`").

Completar el cuerpo de la función *recursiva* `cuenta-pares` que calcula el número de pares de una cadena, recursivamente.

```
;; cuenta-pares: string -> number
(define (cuenta-pares c)
   ...)
```

```
> (cuenta-pares ‘‘axa’’)
1
> (cuenta-pares ‘‘axax’’)
2
> (cuenta-pares ‘‘axbx’’)
1
```

—

Ejercicio 1.6 (1 pt.) Completar el cuerpo de la función *recursiva* `piramide` que imprime una pirámide con `n` pisos haciendo uso de la función `display` o alguna otra de impresión.

```
;; piramide: number -> void
(define (piramide n)
   ...)
```

```
> (piramide 2)
 *
***
> (piramide 3)
  *
 ***
*****
> (piramide 10)
         *
        ***
       *****
      *******
     *********
    ***********
   *************
  ***************
 *****************
*******************
```

—

Ejercicio 1.7 (3 pts.) Completar el cuerpo de las siguientes funciones sobre listas. Para este ejercicio, se prohíbe hacer uso de funciones predefinidas de Racket que resuelvan directamente los ejercicios:

1. Completar el cuerpo de la función `arma-pares` que recibe dos listas y construye una nueva lista con listas de longitud dos, formadas a partir de los elementos de ambas listas.

```
;; arma-pares: list list -> (listof list)
(define (arma-pares lst1 lst2)
   ...)
```

```
> (arma-pares '(foo bar) '(10 20))
'((foo 10) (bar 20))
```

2. Completar el cuerpo de la función `lookup` que recibe una lista con elementos de la forma `'(id value)` y regresa el valor asociado al `id` que fue pasado como parámetro.

```
;; lookup: (listof list) -> a
(define (lookup id lst)
   ...)
```

```
> (lookup 'foo '((foo 10) (bar 20)))
20
```

3. Completar el cuerpo de la función `compara-longitud` que sin usar la función predefinida `length` compare la longitud de las listas `lst1 lst2`. Los valores de regreso deberán ser únicamente los símbolos `'lista1-mas-grande`, `'lista2-mas-grande` o `'listas-iguales`.

```
;; compara-longitud: list list -> symbol
(define (compara-longitud lst1 lst2)
   ...)
```

```
> (compara-longitud '(1 2 3) '(5 6 7 8 9 10))
'lista2-mas-grande
```

4. Completar el cuerpo de la función `entierra` que entierra el símbolo `nombre`, `n` número de veces. Es decir, se deberán anidar `n - 1` listas hasta que se llegue a la lista que tiene al símbolo `nombre`.

```
;; entierra: symbol number -> list
(define (entierra nombre n)
   ...)
```

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

5. Completar el cuerpo de la función `mezcla` que supone que las listas `lst1` `lst2` están ordenadas ascendentemente y construye una lista ordenada de manera ascendente con los elementos de ambas listas.

```
;; mezcla: list list -> list
(define (mezcla lst1 lst2)
   ...)
```

```
> (mezcla '(1 2 6 8 10 12) '(2 3 5 9 13))
'(1 2 2 3 5 6 8 9 10 12 13)
```

—

Ejercicio 1.8 (2 pts.) Completar los siguientes ejercicicios haciendo uso de las funciones de orden superior `map`, `filter`, `foldr` y/o `foldl`. Para este ejercicio se prohíbe definir funciones auxiliares, en caso de requerirlas, usar lambdas en combinación de asignaciones locales `let`, `let*` o `letrec`.

1. Completar el cuerpo de la función `binarios` que recibe una lista de números y regresa una nueva lista de cadenas que representan el número binario asociado a estos números.

```
;; binarios: (listof number) -> (listof string)
(define (binarios lst)
   ...)
```

```
> (binarios '(0 1 2 3 4))
'("0" "01" "010" "011" "100")
```

2. Completar el cuerpo de la función triangulares que recibe una lista de números y regresa una nueva conteniendo únicamente aquellos que son triangulares.

```
;; triangulares: (listof number) -> (listof number)
(define (triangulares lst)
   ...)
```

```
> (triangulares '(1 2 3 4 5 6))
'(1 3 6)
```

3. Completar el cuerpo de las funciones `intercalar` y `intercalal` que dada una lista y un símbolo, intercalan el símbolo entre los elementos de la lista dada usando `foldr` y `foldl` respectivamente.

```
;; intercalar: list symbol -> list
(define (intercalar lst s)
   ...)

;; intercalal: list symbol -> list
(define (intercalal lst s)
   ...)
```

```
> (intercalar '(1 2 3) '*)
'(1 * 2 * 3)
> (intercalal '(1 2 3) '*)
'(1 * 2 * 3)
```

Puntaje total: *10 puntos*

Entrega

El único archivo que se debe enviar al ayudante de laboratorio es:

- `practica1.rkt`

Recordando seguir los lineamientos de entrega de prácticas especificados en la sección correspondiente de la página del curso: `http://lenguajesfc.com/lineamientos.html`.

Puntos extra

Escoger alguno de los siguientes ejercicios y resolverlo **individualmente**[3]:

Punto extra 1.1 (1 pt.) Buscar y leer el artículo *Why Functional Programming Matters?* de J. Hughes y escribir un ensayo de al menos dos cuartillas. El ensayo debe incluir: Título, Introducción, Desarrollo, Conclusiones y Bibliografía. No se tomará en cuenta ningún ensayo que no esté debidamente citado e incluya las referencias correspondientes. El ensayo deberá entregarse de forma impresa a más tardar el 23 de agosto de 2017 durante la sesión de laboratorio.

Punto extra 1.2 (1 pt.) Definir una función recursiva `triangulo-pascal` que imprima los primeros `n` pisos del Triángulo de Pascal haciendo uso de la función `display` o alguna otra de impresión. El ejercicio debe enviarse a más tardar el 23 de agosto de 2017 en un archivo `cuenta_p1.rkt` donde `cuenta` es el número de cuenta del alumno, con el asunto `[LDP-Extra P1]`.

```
;; triangulo-pascal: number -> void
(define (triangulo-pascal n)
   ...)
```

```
> (triangulo-pascal 5)
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

[3]Pueden enviarse ambos ejercicios para ser revisados, sin embargo, sólo uno será tomado en cuenta.

Referencias

Algunas referencias de consulta:

[1] Karla Ramírez, Manuel Soto, Ricardo Rodríguez, *Notas de laboratorio del curso de Lenguajes de Programación*, Semestre 2018-1, Facultad de Ciencias, UNAM. Disponibles en: [`http://lenguajesfc.com/notas.html`].

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

[3] Eric Tánter, *PREPLAI: Scheme y Programación Funcional*, Primera edición, 2014. Disponible en: [`http://users.dcc.uchile.cl/~etanter/preplai/`] (Consultado el 4 de agosto de 2017).

[4] 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 4 de agosto de 2017).

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