Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Práctica 1

Karla Ramírez Pulido karla@ciencias.unam.mx
J. Ricardo Rodríguez Abreu ricardo_rodab@ciencias.unam.mx
Manuel Soto Romero manu@ciencias.unam.mx

Fecha de inicio: 2 de febrero de 2018
Fecha de término: 22 de febrero de 2018
Semestre 2018-2

Objetivo

Que los estudiantes conozcan y utilicen las primitivas básicas del lenguaje de programación Racket a través de la primitiva `plai` (*Programming Languages Application and Interpretation)*, mismas que serán utilizadas a lo largo del semestre.

Antecedentes

En las sesiones de laboratorio se revisó el lenguaje de programación Racket y su ambiente de desarrollo integrado, del cual se elaboraron actividades disponibles en el repositorio del curso. En caso de no haber elaborado dichas actividades durante las sesiones de laboratorio, se recomienda revisarlas junto con el Anexo 1 de las notas del curso.

Repositorio

El material necesario para completar esta práctica se encuentra en el repositorio de *GitHub Classroom* del curso: `https://classroom.github.com/g/C2khfbxo`.

Desarrollo de la práctica

En equipos de **tres integrantes** completar las funciones faltantes del archivo `practica1.rkt` hasta que pasen todas las pruebas unitarias incluidas en el archivo `pruebas_practica1.rkt`[1] .

[1]Para tener derecho a calificación, el archivo debe ejecutarse sin errores.

Ejercicio 1.1 (0.5 pts.) Completar el cuerpo de la función `ecuacion-lineal` que resuelve la ecuación $Ax + B = 0$. El primer argumento de la función corresponde al valor de A y el segundo al valor de B.

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

```
> (ecuacion-lineal 8 1)
-1/8
```

Ejercicio 1.2 (0.5 pts.) Completar el cuerpo de la función `area-cilindro` que calcula el área de un cilindro dado el diámetro y la altura como primer y segundo parámetro respectivamente. Uar la primitiva `let` para evitar cálculos repetitivos.

Fórmula:

$$A = 2\pi r\,(r + h)$$

```
;; area-cilindro: number number -> number
(define (area-cilindro d h) ...)
```

```
> (area-cilindro 10 20)
786.37
```

Ejercicio 1.3 (1 pt.) Una persona conduce demasiado rápido, y un oficial de policía lo detiene. Completar el cuerpo de la función `tipo-multa` que calcula el tipo de multa que recibirá el conductor. Si la velocidad está entre 60 o menos el resultado es `'sin-multa`. Si la velocidad está entre 61 y 80, el resultado es `'multa-pequeña`. Si la velocidad es de 81 o más, el resultado es `'multa-grande`. A menos que sea cumpleaños del conductor: en ese día, la velocidad a la que conduce será cinco veces más alta.

```
;; tipo-multa: number boolean -> symbol
(define (tipo-multa v c) ...)
```

```
> (tipo-multa 60 #f)
'sin-multa
```

Ejercicio 1.4 (1 pt.) Completar el cuerpo de la función **recursiva** `sdigito` que calcula el *s-dígito* de un número natural *n*. El s-dígito de un número natural se define como sigue:

- Si *n* consiste de un solo dígito, entonces su s-dígito es *n*.
- En otro caso, el s-dígito de *n* es igual al s-dígito de la suma de los dígitos de *n*.

Por ejemplo, el s-dígito de 984 se calcula como sigue:

$$sdigito\,(984) = sdigito\,(9+8+4) = sdigito\,(21) = sdigito\,(2+1) = sdigito\,(3) = 3.$$

No está permitido convertir el número a lista, cadena o a cualquier otro tipo de dato.

```
;; sdigito: number -> number
(define (sdigito n) ...)
```

```
> (sdigito 984)
3
```

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.

No está permitido convertir la cadena a lista, o a cualquier otro tipo de dato.

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

```
> (cuenta-pares ‘‘AxAxA’’)
3
```

Ejercicio 1.6 (1 pt.) Completar el cuerpo de la función *recursiva* `cuadrado` que construya una cadena que represente un cuadrado de longitud *n* para después, imprimirlo haciendo uso de la función display o alguna otra de impresión.

```
;; cuadrado: number -> string
(define (cuadrado n) ...)
```

```
> (display (cuadrado 10))
**********
*        *
*        *
*        *
*        *
*        *
*        *
*        *
*        *
**********
```

Ejercicio 1.7 (3 pts.) Completar el cuerpo de las siguientes funciones **recursivas** sobre listas.

1. Completar el cuerpo de la función **recursiva** `repeticiones` que calcula el número de repeticiones de cada elemento de una lista.

   ```
   ;; repeticiones: (listof any) -> (listof pair)
   (define (repeticiones lista) ...)
   ```

   ```
   > (repeticiones '(a a b a b c c a))
   '((a . 4) (b . 2) (c . 2))
   ```

2. Una forma de encontrar los números primos en un determinado rango es mediante la conocida *Criba de Eratóstenes*. El algoritmo consiste en ir tomando los números del rango y eliminar todos los números que sean múltiplos de éste. El algoritmo terminará cuando no se elimine ningún número.

   Por ejemplo, para encontrar los números primos del 2 al 20, se tiene la siguiente lista:

   ```
   '(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
   ```

   El primer paso consiste en eliminar todos aquellos números que sean múltiplos de dos, con lo cual quedaría la siguiente lista:

   ```
   '(2 3 5 7 9 11 13 15 17 19)
   ```

   Ahora, se pasa al siguiente número en la lista, en este caso el tres, y se repite el procedimiento:

   ```
   '(2 3 5 7 11 13 17 19)
   ```

   Para el siguiente paso, se deben eliminar los múltiplos de cinco, sin embargo no queda ningún múltiplo de este número en la lista con lo cual, termina el algoritmo y se concluye que los números primos del 2 al 20 son: 2, 3, 5, 7, 11, 13, 17 y 19.

   Completar el cuerpo de la función **recursiva** `criba-eratostenes` que encuenta los números primos en un rango de 2 a n usando la *Criba de Eratóstenes*.

```
;; criba-eratostenes: number number -> (listof number)
(define (criba-eratostenes n) ...)
```

```
> (criba-eratostenes 20)
'(2 3 5 7 11 13 17 19)
```

3. Un número puede representarse mediante el producto de números primos. Por ejemplo, el número 405 se puede representar como el producto de: $3^4 \times 5$.

   Para representar la descomposición en factores primos, suele usarse una tabla, a la izquierda se coloca el número a descomponer y a la derecha el número primo más pequeño por el que se puede dividir, en el siguiente renglón se coloca el resultado de la división del lado izquierdo y se busca el siguiente número primo a partir del número del renglón anterior. El proceso se detiene cuando la división resultante es uno.

405	3
135	3
45	3
15	3
5	5
1	

   Al tener números repetidos en la columna derecha se pueden representar como potencias.

   Completar el cuerpo de la función **recursiva** `descomposicion-primos` que toma un número y regresa una lista de pares con la descomposición en factores primos del mismo.

```
;; descomposicion-primos: number -> (listof pair)
(define (descomposicion-primos n) ...)
```

```
> (descomposicion-primos 405)
'((3 . 4), (5 . 1))
```

Ejercicio 1.8 (2 pts.) Completar los siguientes ejercicios 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 funciones anónimas (`lambda`) en combinación de asignaciones locales `let`, `let*` o `letrec`.

1. Completar el cuerpo de la función **recursiva** `binarios` que recibe una lista de números y regresa una lista de números binarios asociados a los números originales. No está permitido hacer ningún tipo de conversión.

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

```
> (binarios '(0 1 2 3 4))
'(0 1 10 11 100)
```

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

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

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

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

```
;; intercalar: (listof any) symbol -> (listof any)
(define (intercalar lista s) ...)

;; intercalal: (listof any) symbol -> (listof any)
(define (intercalal lista s) ...)
```

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

Puntos extra

Resolver **individualmente**.

Punto extra 1.1 (2 pts.) El *caracol* de una matriz, consiste en recorrer los elementos de una matriz en forma de caracol. Por ejemplo, el caracol de la matriz:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$

es

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

o en forma de lista:

```
'(1 2 3 4 5 10 15 20 25 24 23 22 21 16 11 6 7 8 9 14 19 18 17 12 13)
```

Definir la funcion **recursiva** `caracol` que recorre los elementos de una matriz de dimensión impar en forma de caracol. Se debe encontrar el elemento central y a partir de esta posición iniciar el caracol.

Para representar matrices se usarán listas de listas, por ejemplo, para la matriz:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

se usa la lista: `'((1 2 3) (4 5 6) (7 8 9))`.

El ejercicio debe enviarse a más tardar el 22 de febrero de 2018 en un archivo `cuenta_p1.rkt` donde cuenta es el número de cuenta del alumno, con el asunto `[LDP-Extra P1]` al correo `manu@ciencias.unam.mx`.

```
;; caracol: (listof (listof number)) -> (listof number)
(define (caracol matriz) ...)
```

```
> (caracol '((1 2 3) (4 5 6) (7 8 9)))
'(1 2 3 6 9 8 7 4 5)
```

Referencias

Algunas referencias de consulta:

[1] Karla Ramírez, Manuel Soto, *Notas de laboratorio del curso de Lenguajes de Programación*, Semestre 2018-2, 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.