v8.13

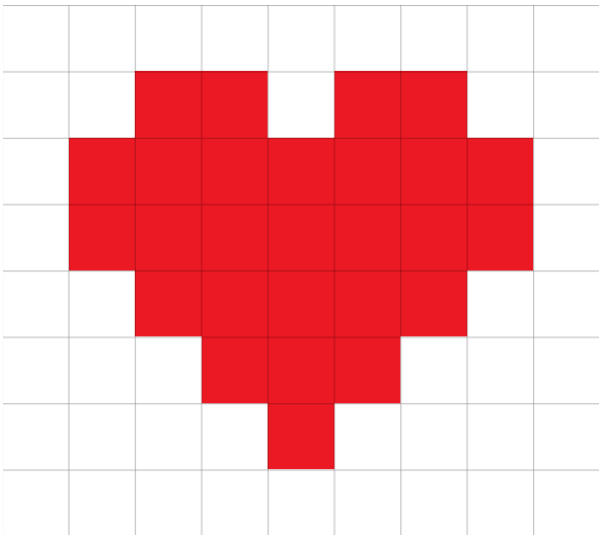
Trabajo Práctico

Programación I

1 Enunciado

Un **mapa de bits** (bitmap en inglés) representa una imágen 2D como una "grilla". El número de filas y de columnas de la grilla se corresponde con el ancho y el alto de la imagen (en píxeles). El valor almacenado en cada casilla de la grilla describe un pixel de la imagen, con una determinada profundidad de color.

Por ejemplo, considerar la siguiente imagen.



Esta imagen se puede representar con un bitmap de 9 píxeles de ancho por 8 píxeles de alto. Además, dado que la imagen tiene únicamente dos tonalidades: blaco y rojo, es suficiente con usar dos números enteros para distinguirlos. Por ejemplo, podemos usar el `0` cuando el pixel es de color blanco y el `1` cuando es rojo. De esta forma se obtiene el siguiente bitmap.

```
0 0 0 0 0 0 0 0 0
0 0 1 1 0 1 1 0 0
0 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0
0 0 1 1 1 1 1 0 0
0 0 0 1 1 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0
```

Este bitmap tiene una profundidad de color de 1 bit, ya que con 1 bit es posible distinguir $2^1 = 2$ valores diferentes. En el caso de que el bitmap tenga una profundidad de color de 2 bits, la cantidad de colores que podemos distinguir se duplica. Esto se debe a que con 2 bits podemos representar $2^2 = 4$ valores diferentes, que en sistema binario son: 00, 01, 10 y 11.

En la actualidad, la mayoría de los sistemas usan una profundida de color de 24 bits, que da lugar a 2^24 = 16777216 colores distintos.

Una forma de almacenar un bitmap en un archivo plano es convertirlo a una `string`. Para ello, se puede aplanar el bitmap en una única línea, concatenando cada una de sus filas (de arriba hacia abajo). Por ejemplo, para el bitmap anterior se obtiene la siguiente `string`:

```
(string-append "000000000"
               "001101100"
               "011111110"
               "011111110"
               "001111100"
               "000111000"
               "000010000"
               "000000000")
==
"000000000001101100011111110011111110001111100000111000000010000000000000"
```

Por lo tanto, necesitamos una string de largo 8*9=72 para almacenar este bitmap.

Como se puede observar, la string contiene largas subcadenas de `0`s y de `1`s. Surge entonces una optimización que consiste en reemplazar las subcadenas de valores iguales (de largo mayor a 1) por su valor concatenado con su recuento. Por ejemplo, la primer subcadena de 11 valores `0`, la representamos con `"011"`. La segunda subcadena de 2 valores `1`, la representamos con `"12"`. La tercera subcadena de 1 valor `0`, la representamos con `"0"` (omitimos el recuento en este caso), y así sucesivamente. Además, es necesario agregar un separador para identificar donde finaliza el recuento. Por ejemplo, separando las subcadenas con una `","`, resulta la siguiente representación.

```
"011,12,0,12,03,17,02,17,03,15,05,13,07,1,013"
```

Cuyo largo es:

```
(string-length "011,12,0,12,03,17,02,17,03,15,05,13,07,1,013")
==
44
```

Por lo tanto, ahora necesitamos una string de 44 caracteres para almacenarla, en lugar de 72, lo que representa un factor de compresión de 72/44 = 1,64. Esto quiere decir que la imagen comprida ocupa 1,64 veces menos que la original. El factor es mayor mientras más cantidad y mayor longitud de este tipo de subcadenas tenga el bitmap.

Esta forma de compresión de datos se denomina **run-length encoding (RLE)**. Es muy eficiente para comprimir imágenes simples como iconos o trazos, aunque no se desempaña demasiado bien para fotografías.

El objetivo de este trabajo práctico es implementar un programa que comprima y descomprima imágenes usando RLE.

2 Colores RGBA

Ya sabemos que Racket tiene predefinidos diferentes colores, por ejemplo, `"red"`, `"pink"`, `"light brown"`, `"medium purple"`, `"dark yellow"`, etc. Aunque estos colores pueden parecer suficientes, no dejan de ser un número bastante acotado.

En realidad, Racket maneja colores RGBA, es decir, con una profundidad de color de 32 bits. Destina 3 canales de 8 bits para indicar la cantidad de rojo (R), verde (G) y azul (A) que tiene el color. Además, destina un canal "alfa" de 8 bits para indicar la opacidad del color (A), usado para generar transparencias o suavizar bordes. Un canal de 8 bits permite usar valores naturales del 0 a 255, donde 0 es 0%, 255 es 100%, y para los demás valores podemos hacer una regla de tres simple, por ejemplo 128 es 50.2%, y así sucesivamente.

Racket dispone de una estructura `color`, definida de la siguiente forma (no copiar esta definición, ya está en el paquete `2htdp/image`).

```
(define-struct color (red green blue alpha))
; color es [Natural, Natural, Natural, Natural]
; red: número de 0 a 255 con la cantidad de rojo
; green número de 0 a 255 con la cantidad de verde
; blue número de 0 a 255 con la cantidad de azul
; alpha número de 0 a 255 con la cantidad de opacidad
```

Así, por ejemplo, podemos construir el color amarillo mezclando rojo y verde.

```
(define AMARILLO (make-color 255 255 0 255))
```

Y podemos probar nuestro color dibujando un círculo.

```
(circle 20 "solid" AMARILLO)
```


También podemos usar la opacidad para aclarar la tonalidad de amarillo, por ejemplo al 50%.

```
(define AMARILLO-CLARO (make-color 255 255 0 128))
(circle 20 "solid" AMARILLO-CLARO)
```


Ejercicio 1. Forme el color rosa con RGBA y dibuje un círculo de ese color. Puede probar directamente desde DrRacket o buscar en internet qué colores mezclar para formarlo.

Ejercicio 2. Diseñe un predicado `color=?` que, dados dos colores RGBA, determine si son iguales.

Ejemplos:

```
(color=? (make-color 10 4 6 124) (make-color 10 4 6 124)) == #t
(color=? (make-color 10 4 6 124) (make-color 235 0 45 50)) == #f
```

Ejercicio 3. Diseñe una función `mascara-1bit` que convierta un color RGBA a una string. Esta función debe devolver `"0"` cuando el color es claro, y debe devolver un `"1"` en caso contrario.

Tomaremos el siguente criterio, un color es claro cuando la opacidad es inferior al 50% o la suma de los canales RGB es mayor a 650.

Ejemplos:

```
(mascara-1bit (make-color 240 230 190 255)) == "0"
(mascara-1bit (make-color 10 150 41 87)) == "0"
(mascara-1bit (make-color 255 0 0 255)) == "1"
```

3 Mapa de bits

Racket ya dispone de algunas funciones para trabajar con bitmaps. Por ejemplo, tenemos una función `image->color-list` que devuelve el bitmap de una imagen. Veamos primero cuál es su signatura.

```
image->color-list: Image -> Listof color
```

Observe que retorna una lista de colores, y no una string, para representar al bitmap. Sin embargo sigue la misma lógica que la mencionada al principio, es decir, la lista tiene tantos elementos como píxeles y están ordenados por fila. La única diferencia es que la lista guarda valores de tipo `color`, en lugar de `"0"` y `"1"`, permitiendo mayor profundidad de color.

Veamos por ejemplo lo que retorna cuando la llamamos sobre una imagen con un cuadrado rojo de 2x2 píxeles al lado de un cuadrado azul de 2x2 píxeles.

```
(define CUADRADOS
  (beside (square 2 "solid" "red")
          (square 2 "solid" "blue")))
(image->color-list CUADRADOS)
==
(list
 (make-color 255 0 0 255)
 (make-color 255 0 0 255)
 (make-color 0 0 255 255)
 (make-color 0 0 255 255)
 (make-color 255 0 0 255)
 (make-color 255 0 0 255)
 (make-color 0 0 255 255)
 (make-color 0 0 255 255))
```

Observar que los primeros 4 valores de la lista representan la primer fila del bitmap, donde los primeros 2 son los rojos y los últimos 2 son los azules. De la misma forma, los últimos 4 valores de la lista representan la segunda fila del bitmap.

Racket también dispone de la función `list-color->bitmap` que convierte una lista de colores RGBA en una imagen. Veamos cómo es su signatura.

```
color-list->bitmap: (Listof color) Number Number -> Image
```

Observe que además de la lista, también toma dos números, los cuales representan el ancho y alto de la imagen medidos en píxeles. Apliquemos por ejemplo esta función a la lista de colores que habíamos obtenido más arriba, que representaba a los cuadrados rojo y azul.

```
(color-list->bitmap (image->color-list CUADRADOS) 4 2)
```


Y si, son muy chicos, usemos la función `scale` para escalarlos 10 veces su tamaño.

```
(scale 10 (color-list->bitmap (image->color-list CUADRADOS) 4 2))
```

Por el momento seguiremos trabajando con bitmaps representados con listas de colores. Más adelante definiremos funciones para convertirlos en strings.

---

4 Compresión RLE

En esta sección nos ocuparemos de diseñar funciones que implementen la compresión RLE y su respectiva descompresión.

Primero vamos a definir una estructura que represente un par ordenado.

```
(define-struct Par (x y))
; Par es un (Any,Any)
; x es la primer componente del par
; y es la segunda componente del par
```

Ejercicio 4. Diseñe una función `comprimir` con la siguiente signatura.

```
comprimir: (Listof A) (A -> A -> Bool) -> Listof (Par (A ,Natural))
```

Esta función toma una lista `l` con elementos de tipo `A` y un predicado `eq` que toma dos elementos de tipo `A` y determina si son iguales. Debe aplicar la compresión RLE sobre `l` y retornar una lista de pares ordenados, donde la primer componente de cada par es un valor de la lista y la segunda es el número de veces que se repite.

Ejemplos:

```
(comprimir (list "A" "A" "A" "B" "B" "C" "A" "A") string=?)
 ==
(list (make-Par "A" 3) (make-Par "B" 2) (make-Par "C" 1) (make-Par "A" 2))
```

```
(comprimir (list 0 1 1 1 1) =)
 ==
(list (make-Par 0 1) (make-Par 1 4))
```

```
(define BLANCO (make-color 255 255 255 255))
(define NEGRO (make-color 0 0 0 255))
(comprimir (list BLANCO BLANCO BLANCO BLANCO NEGRO NEGRO) color=?)
 ==
 (list (make-Par BLANCO 4) (make-Par NEGRO 2))
```

Ejercicio 5. Diseñe una función `descomprimir` con la siguiente signatura.

```
descomprimir: Listof (Par (A ,Natural)) -> Listof A
```

Esta función debe ser la inversa de la función `comprimir`.

Ejemplos:

```
(descomprimir (comprimir (list "A" "A" "A" "B" "B" "C" "A" "A") string=?))
 ==
(list "A" "A" "A" "B" "B" "C" "A" "A")
```

```
(descomprimir (comprimir (list 0 1 1 1 1) =))
 ==
(list 0 1 1 1 1)
```

```
(descomprimir (comprimir (list BLANCO BLANCO BLANCO BLANCO NEGRO NEGRO) color=?))
 ==
(list BLANCO BLANCO BLANCO BLANCO NEGRO NEGRO)
```

5 Factor de compresión

En este punto ya sabemos obtener el bitmap de una imagen, representado como una lista de colores. A continuación, vamos a convertirlos en strings y analizar el factor de compresión al aplicarles RLE. Para no complicar las cosas, vamos a convertir los colores RGBA en `"0"` o `"1"`. Es decir, pasamos de una profundidad de color de 32 bits a 1 bit.

Ejercicio 6. Diseñe una función `aplicar-mascara-1bit` que, dada una lista de colores RGBA, retorne la lista luego de aplicar la función `mascara-1bit` a cada uno de los elementos de la lista.

Ejemplos:

```
(aplicar-mascara-1bit (list (make-color 240 230 190 255)
                            (make-color 10 150 41 87)
                            (make-color 255 0 0 255)))
==
(list "0" "0" "1")
```

Ejercicio 7. Diseñe una función `lista->cadena` con la siguiente signatura.

```
lista->cadena: List (String) -> String -> String
```

Esta función toma una lista `l` de strings y una string `sep`. Retorna una string con la concatenación de cada uno de los elementos de `l` separados con `sep`.

Ejemplos:

```
(lista->cadena (list "0" "0" "1") ",") == "0,0,1"
(lista->cadena (list "0" "0" "1") "") == "001"
(lista->cadena (list "21" "de" "septiembre") " ") == "21 de septiembre"
(lista->cadena empty ",") == ""
```

Ejercicio 8. Diseñe una función `generar-cadena` que, dado un bitmap representado como una lista de `"0"` y `"1"`, retorne el mismo bitmap pero representado como una string de `"0"` y `"1"`.

Ejemplos:

```
(generar-cadena (list "0" "1" "1" "0" "1" "1" "1" "0" "0" "0" "0"))
```

```
  ==
  "01101110000"
```

Ejercicio 9. Diseñe una función `generar-cadena-rle` que, dado un bitmap representado como una lista de `"0"` y `"1"`, comprima la lista con RLE y la convierta a una string. Utilice la codificación mencionada al comienzo del trabajo práctico. Es decir, concatenando cada `"0"` y `"1"` con su número de repeticiones (siempre que sea mayor a 1) y agregando `","` para separar.

Ejemplos:

```
(generar-cadena-rle (list "0" "1" "1" "0" "1" "1" "1" "0" "0" "0" "0"))
 ==
 "0,12,0,13,04"
```

Ejercicio 10. Diseñe una función `factor-compresion` que, dada una imagen, retorne su factor de compresión. Recuerde que este factor se define como el cociente entre el largo de la string, que representa el bitmap de la imagen, y su largo luego de comprimirlo con RLE, siempre conviertiendo previamente el bitmap a una profundidad de 1 bit. Tendrá que usar todo lo que aprendió en este trabajo.

Ejemplos:

```
(check-within (factor-compresion ♥) 1.63 0.01)
```

```
(check-within (factor-compresion                  ) 16.37 0.01)
```

```
(check-within (factor-compresion                  ) 7.35 0.01)
```

```
(check-within (factor-compresion                  ) 4.85 0.01)
```