Parser Combinators

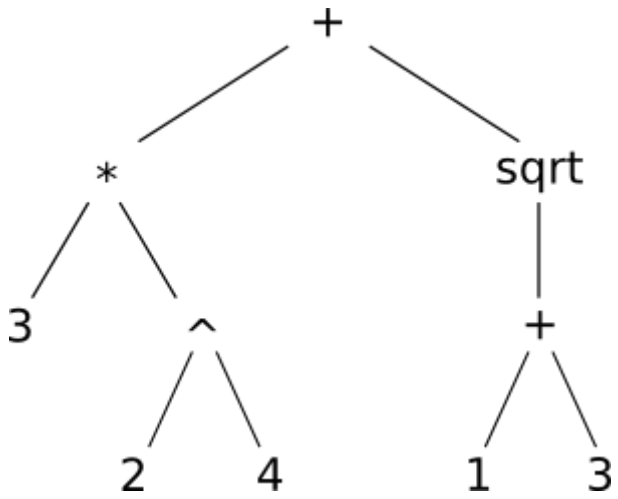
TAdeP - 2C 2020 - Trabajo Práctico Grupal: Objeto/Funcional

https://en.wikipedia.org/wiki/Parser_combinator

Introducción

Un parser es algo que toma como entrada un texto, y que como salida produce algo más fácil de interpretar o manipular en el lenguaje de programación que se esté usando, es una forma de traducir entre un lenguaje y otro.

Por ejemplo, un parser podría agarrar el texto “3 * 2 ^ 4 + sqrt(1 + 3)” y convertirlo en un árbol que tenga la siguiente forma:



Un **parser combinator** es una función de orden superior que recibe parsers como parámetro y retorna nuevos parsers. Esto va en línea con la filosofía de funcional que se basa en hacer componentes chiquitos que sirven como bloques de construcción, y a partir de combinar esos construir abstracciones más grandes.

IMPORTANTE: Este trabajo práctico debe implementarse de manera que se apliquen los principios del paradigma híbrido objeto-funcional enseñados en clase. No alcanza con hacer que el código funcione en objetos, hay que aprovechar las herramientas funcionales, poder

justificar las decisiones de diseño y elegir el modo y lugar para usar conceptos de un paradigma u otro.
Se tendrán en cuenta para la corrección los siguientes aspectos:

- Uso de Inmutabilidad vs. Mutabilidad.
- Uso de Polimorfismo paramétrico (Pattern Matching) vs. Polimorfismo Ad-Hoc.
- Aprovechamiento del polimorfismo entre objetos y funciones y Orden Superior.
- Uso adecuado del sistema de tipos.
- Manejo de herramientas funcionales.
- Cualidades de Software.
- Diseño de interfaces y elección de tipos.

Descripción General del Dominio

El objetivo de este trabajo práctico va a ser definir un lenguaje de propósito específico para describir imágenes. Luego, utilizando parser combinators vamos a escribir un parser que nos permita tomar una imagen descripta en nuestro lenguaje y transformarla en un modelo intermedio que podamos manipular.
Una vez teniendo ese modelo intermedio, vamos a interactuar con una API que les proveemos para efectivamente producir una imagen.

Primeros parsers

Para empezar, se pide implementar algunos parsers que podríamos usar para realizar parseos simples, como chequear por una cadena de caracteres en particular.
Cada uno de estos parsers tiene que recibir un string sobre el cual trabajar y retornar un resultado que puede ser un **error de parseo** o un **parseo exitoso** (que contiene el elemento parseado junto con la sección del string que no fue consumida).

anyChar: Lo primero que vamos a implementar va a ser un parser que lee cualquier carácter de un texto, pero solo uno, entonces, si a este parser le paso el texto “hola”, debería devolver un parseo exitoso con el carácter h como resultado. Sin embargo, si le paso un texto vacío (“”), debería fallar.

char: Ahora, queremos poder tener parsers que nos permitan decir qué carácter esperamos encontrar. Por ejemplo, queremos poder tener un parser del char ‘c’, al cuál si le paso como parámetro el texto “chau”, da un resultado exitoso con el valor ‘c’, pero si le paso el texto “hola”, debería fallar porque “hola” no empieza con c.

digit: debería devolver un resultado exitoso con el carácter parseado, si el carácter consumido es 0, 1, 2, 3, 4, 5, 6, 7, 8 o 9.

string: este caso es parecido a **char**, porque necesita saber qué string es el que esperamos parsear, pero se diferencia a los parsers que aparecieron hasta ahora porque consume

tantos caracteres como tenga el string esperado, y en el caso de éxito no debería tener un carácter como resultado, si no un string.
Por ejemplo, queremos poder obtener un parser con el string esperado “hola”, tal que si lo usamos para parsear “hola mundo!”, de un resultado exitoso con el valor “hola”, pero si lo usasemos para intentar parsear “holgado” falle.

integer: debería parsear un número entero, que se representa como una sucesión de dígitos que puede estar o no empezada por un **-**. El resultado de este parser debería ser un número entero, no el string parseado!

double: debería parsear un número decimal y devolverlo como un double. Lo que queremos es soportar decimales que se escriban como enteros que pueden estar seguidos por un punto (.) y una sucesión de dígitos (la parte decimal). El resultado de este parser debe ser un double en caso de éxito.

Combinators

Ahora que ya tenemos algunos parsers, se requiere implementar los combinadores de los mismos, con los cuales podemos crear parsers más complejos.

OR Combinator

<|>: este es el primer combinador propiamente dicho, a partir de dos parsers crea uno que trabaja de la siguiente manera: si el primer parser retorna un resultado, ese es el resultado, si no, intenta con el segundo parser y retorna lo que retornaría este último. Un ejemplo de creación de un parser usando este combinador sería:

```
val aob = char('a') <|> char('b')
```

Si le pasamos “arbol” a aob, debería poder parsear ‘a’, y si le pasasemos “bort” a aob debería poder parsear ‘b’.

Concat Combinator

<>: un parser combinator que secuencia dos parsers. Es decir, crea uno nuevo que primero parsea lo que parsearía el primero, y usando el resto del texto aplica el segundo parser.
Esperamos que el valor que devuelva en caso exitoso tenga una tupla con los dos valores parseados.
Por ejemplo:

```
val holaMundo = string("hola") <> string("mundo")
```

Si a holaMundo le pasasemos “holamundo”, debería producir un resultado exitoso con los valores “hola” y “mundo” en una tupla. Si le pasásemos “holachau”, debería fallar, porque o parsea todo o no parsea nada.

Rightmost Combinator

~>: (primerParser **~>** segundoParser) debería requerir que ambos parsers se hagan de manera secuencial (primero primerParser y luego segundoParser) pero me da sólo el resultado de segundoParser.

Leftmost Combinator

<~: (primerParser **<~** segundoParser) parsea en el mismo orden que el anterior, pero me da el resultado de primerParser.

Separated-by Combinator

sepBy: toma dos parsers: un parser de contenido y un parser separador, parsea 1 o más veces el parser de contenido (similar a la cláusula de kleene+) pero entre cada una aplica el parser separador.
Ejemplo:

```
val numeroDeTelefono = integer.sepBy(char('-'))
```

debería funcionar si le paso “4356-1234” pero no si le paso “hola-chau”. Si le paso “1234 5678” el resultado debería ser 1234 como valor parseado y “ 5678” resto por parsear.

Parsers parte II

Ahora queremos definir operaciones sobre los parsers para obtener parsers más complejos.

satisfies: A partir de un parser y una condición, nos permite obtener un parser que funciona sólo si el parser base funciona y además el elemento parseado cumple esa condición.

opt: convierte en opcional a un parser. Es decir, el nuevo parser siempre debería dar un resultado exitoso, pero si el parser original hubiese fallado, el resultado no contendrá ningún valor y no consumirá ningún caracter del input.
Ejemplo:

```
val talVezIn = string("in").opt
// precedencia parsea exitosamente las palabras "infija" y "fija"
val precedencia = talVezIn <> string("fija")
```

Si a precedencia le pasasemos “fija”, debería devolver una tupla con un valor vacío y con el valor “fija”, porque talVezIn no habría consumido ningún carácter del texto original.

*: la clausura de Kleene se aplica a un parser, convirtiéndolo en otro que se puede aplicar todas las veces que sea posible o 0 veces. El resultado debería ser una lista que contiene todos los valores que hayan sido parseados (podría no haber ninguno).

+: es como la clausura de Kleene pero requiere que el parser se aplique al menos UNA vez.

map: Dada una función de transformación y un parser, retorna un nuevo parser que parsea lo mismo que el original y convierte el valor parseado utilizando la función recibida.

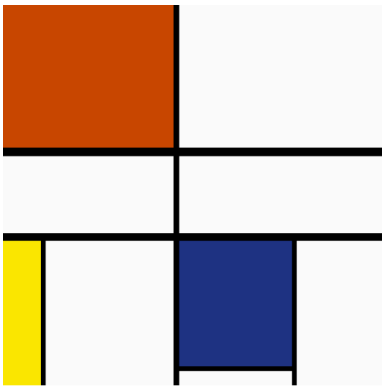
```
case class Persona(nombre: String, apellido: String)
val personaParser = (alphaNum.* <> (char(' ') ~> alphaNum.*))
          .map { case (nombre, apellido) => Persona(nombre, apellido) }
```

Caso práctico: Imágenes

Como comentamos antes, vamos a definir una especificación de un lenguaje que describe imágenes.
Una descripción de una imagen en nuestro lenguaje podría escribirse de esta manera:

```
escala[1.45, 1.45](
 grupo(
   color[0, 0, 0](
       rectangulo[0 @ 0, 400 @ 400]
   ),
   color[200, 70, 0](
       rectangulo[0 @ 0, 180 @ 150]
   ),
   color[250, 250, 250](
       grupo(
       rectangulo[186 @ 0, 400 @ 150],
       rectangulo[186 @ 159, 400 @ 240],
       rectangulo[0 @ 159, 180 @ 240],
       rectangulo[45 @ 248, 180 @ 400],
       rectangulo[310 @ 248, 400 @ 400],
       rectangulo[186 @ 385, 305 @ 400]
       )
   ),
   color[30, 50, 130](
       rectangulo[186 @ 248, 305 @ 380]
   ),
   color[250, 230, 0](
       rectangulo[0 @ 248, 40 @ 400]
   )
 )
)
```

En este caso, la descripción corresponde a una imagen que dibujada se vería así:



Como caso práctico vamos a escribir un parser que nos permita interpretar el lenguaje que definimos para luego poder manipularlo en nuestro código y hasta dibujar imágenes como la mostrada.

Requerimientos

Punto 1 - combinators

Implementar los parsers y parsers combinators explicados en el enunciado.

Punto 2 - parser de imágenes

Implementar un parser de imágenes que pueda convertir una descripción de una imagen a una representación que sea manipulable en código. Una forma de representar esa descripción es un AST.

Figuras

En nuestro lenguaje las imágenes se van a representar como una combinación de figuras más simples. Estas figuras son:

- triángulos, que se definen según 3 puntos donde un punto lo indicamos como x `@` y

```
triangulo[0 @ 100, 200 @ 300, 150 @ 500]
```

- rectángulo, que se define según sus vértice superior izquierdo e inferior derecho

```
rectangulo[0 @ 100, 200 @ 300]
```

- círculo, que se define según su centro y su radio

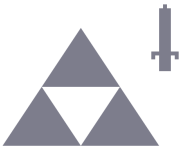
```
circulo[100 @ 100, 50]
```

También queremos poder definir grupos de figuras, donde las figuras dentro del grupo van entre parentesís separadas por coma como en el caso siguiente:

```
grupo(
   triangulo[200 @ 50, 101 @ 335, 299 @ 335],
   circulo[200 @ 350, 100]
)
```

Y los grupos deberían poder anidarse:

```
grupo(
    grupo(
       triangulo[250 @ 150, 150 @ 300, 350 @ 300],
       triangulo[150 @ 300, 50 @ 450, 250 @ 450],
       triangulo[350 @ 300, 250 @ 450, 450 @ 450]
    ),
    grupo(
       rectangulo[460 @ 90, 470 @ 100],
       rectangulo[430 @ 210, 500 @ 220],
       rectangulo[430 @ 210, 440 @ 230],
       rectangulo[490 @ 210, 500 @ 230],
       rectangulo[450 @ 100, 480 @ 260]
    )
)
```



Transformaciones

Finalmente, vamos a agregar transformaciones que se pueden hacer sobre nuestros elementos. Cada transformación afecta a un elemento a la vez (si queremos afectar a varios lo que hay que hacer es ponerlos en un grupo y afectar al grupo) , y las transformaciones que queremos implementar son:

- **color**, para la cual necesitamos definir un color (que vamos a representar como rgb, o sea con 3 valores que van del 0 al 255 para el rojo, verde y azul).

```
color[60, 150, 200](
    grupo(
       triangulo[200 @ 50, 101 @ 335, 299 @ 335],
       circulo[200 @ 350, 100]
    )
)
```

- **escala**, la cual definimos según dos valores que representan el factor de escalado en x e y.

```
escala[2.5, 1](
      rectangulo[0 @ 100, 200 @ 300]
)
```

- **rotación**, la definimos con un valor que representa el ángulo de rotación en grados. El ángulo debería estar entre 0 y 359 inclusive. Si en la descripción dada el ángulo es mayor, queremos limitarlo a esos valores usando un ángulo equivalente.

```
rotacion[45](
      rectangulo[300 @ 0, 500 @ 200]
)
```

- traslación, la definimos con dos valores, uno que es el desplazamiento en x y otro el desplazamiento en y.

```
traslacion[200, 50](
      triangulo[0 @ 100, 200 @ 300, 150 @ 500]
)
```

Punto 3 - simplificando el AST

Una vez tenemos definido nuestro AST, queremos aplicar algunas simplificaciones sobre el mismo que nos devuelvan un AST equivalente (en el sentido de que describen la misma imagen) pero eliminando algunas cosas que son redundantes, como:

- Si tenemos una transformación de color aplicada a otra transformacion de color, debería quedar la de adentro.
- Si tenemos una transformación aplicada a todos los hijos de un grupo, eso debería convertirse en una transformación aplicada al grupo.

  ```
  grupo(
        color[200, 200, 200](rectangulo[100 @ 100, 200 @ 200]),
        color[200, 200, 200](circulo[100 @ 300, 150])
  )
  ```

  debería convertirse en algo equivalente a

  ```
  color[200, 200, 200](
     grupo(
        rectangulo[100 @ 100, 200 @ 200],
        circulo[100 @ 300, 150]
     )
  )
  ```

- Si tenemos una rotación, escala o traslación que contiene a otra transformación del mismo tipo, queremos reemplazarlas por la unión de las transformaciones. Para cada una de estas transformaciones, la forma de unirse es la siguiente:
    - Rotación: se suman los grados de ambas rotaciones. Por ejemplo:

```
rotacion[300](
      rotacion[10](
            rectangulo[100 @ 200, 300 @ 400]
      )
)
```

      debería quedar equivalente a lo que daría

```
rotacion[310](
      rectangulo[100 @ 200, 300 @ 400]
)
```

    - Escala: se multiplican los factores de escalado. Por ejemplo:

```
escala[2, 3](
      escala[3, 5](
            circulo[0 @ 5, 10]
      )
)
```

      debería quedar equivalente a

```
escala[6, 15](
      circulo[0 @ 5, 10]
)
```

    - Traslación: se suman los valores de las traslaciones. Por ejemplo:

```
traslacion[100, 5](
      traslacion[20, 10](
            circulo[0 @ 5, 10]
)
)
```

      debería quedar

```
traslacion[120, 15](
      circulo[0 @ 5, 10]
)
```

- Hay veces donde una rotación, escala o traslación no cambia de ninguna manera a al elemento al que se aplican, en esos casos queremos simplemente reemplazarlas por el elemento. Las transformaciones nulas que queremos borrar son:
    - rotación de 0 grados
    - escala de 1 en x, 1 en y
    - traslación de 0 en x, 0 en y

Punto 4 - interpretando el AST

Ahora que podemos convertir el texto en un AST y simplificar el mismo, lo que queremos es a partir de interpretar el AST dibujar la imagen. Para esto les proveemos una interfaz que permite dibujar gráficos a pantalla o a un archivo, y lo que es necesario implementar es un intérprete que tomando el AST interactúe con la interfaz como para dibujar la imagen.

Documentación de la interfaz para dibujar imágenes

El punto de entrada de esta interfaz es el objeto TADPDrawingAdapter que entiende tres mensajes: `forScreen`, `forImage` y `forInteractiveScreen` que dibujan una imagen en la pantalla, en un archivo, y en una pantalla interactiva que convierte un texto en imagen. Pueden elegir cualquiera de los 3 mensajes para probar que su programa funciona correctamente, no es necesario que lo implementen usando todos.

Los 3 mensajes se comportan similar respecto a que reciben una función por parámetro que trabaja con una instancia de la clase TADPDrawingAdapter.

El adapter que recibimos en la lambda funciona como un builder en el cual le mandamos un mensaje diciéndole qué dibujar y nos devuelve un nuevo adapter al cual seguir mandándole mensajes.

Por ejemplo, esto dibuja dos rectángulos y un círculo:

```
TADPDrawingAdapter.forScreen { adapter =>
 adapter.rectangle((200, 200), (300, 400))
        .rectangle((200, 300), (500, 700))
        .circle((100, 200), 30)
}
```

Para dibujar figuras se le pueden mandar los siguientes mensajes a un adapter:

```
def circle(center: Point2D, radius: Double):
TadpDrawingAdapter
def rectangle(topLeft: Point2D, bottomRight: (Double,
Double)): TadpDrawingAdapter
def triangle(p1: Point2D, p2: Point2D, p3: Point2D):
TadpDrawingAdapter
Nota: Point2D es (Double, Double).
```

Si queremos aplicar transformaciones, lo que tenemos son mensajes para mandarles a partir de cuándo y hasta cuándo aplica cierta transformación.

Por ejemplo:

```
TADPDrawingAdapter.forScreen { adapter =>
 adapter.beginColor(Color.rgb(100, 100, 100))
        .rectangle((200, 200), (300, 400))
        .end()
        .triangle((100, 100), (0, 150), (200, 300))
}
```

En este ejemplo, el color sólo afecta al rectángulo.

Los mensajes que podemos usar para declarar dónde empieza una transformación son:

```
def beginTranslate(x: Double, y: Double): TadpDrawingAdapter
def beginScale(x: Double, y: Double): TadpDrawingAdapter
def beginRotate(degrees: Double): TadpDrawingAdapter
def beginColor(color: Color): TadpDrawingAdapter
```

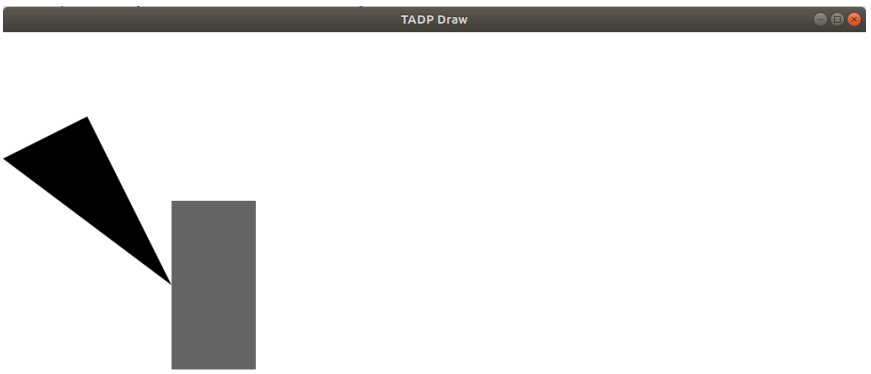
Importante: Las instancias de `TADPDrawingAdapter` son inmutables. Esto significa que al enviarle un mensaje, las modificaciones se aplican sobre un nuevo objeto y el viejo no debe usarse más. Es por eso que cada mensaje devuelve un nuevo `TADPDrawingAdapter`. Si se intenta usar de otra manera, los resultados pueden ser inesperados.

Ejemplos de uso de la interfaz:

forScreen:

```
TADPDrawingAdapter.forScreen { adapter =>
 adapter.beginColor(Color.rgb(100, 100, 100))
   .rectangle((200, 200), (300, 400))
   .end()
   .triangle((100, 100), (0, 150), (200, 300))
}
```

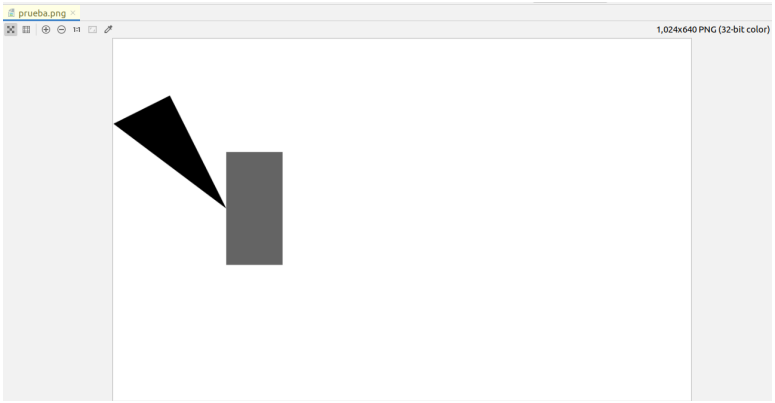
Al ejecutarse muestra esta ventana:



forImage:

```
TADPDrawingAdapter.forImage("prueba.png") { adapter =>
 adapter.beginColor(Color.rgb(100, 100, 100))
   .rectangle((200, 200), (300, 400))
   .end()
   .triangle((100, 100), (0, 150), (200, 300))
}
```

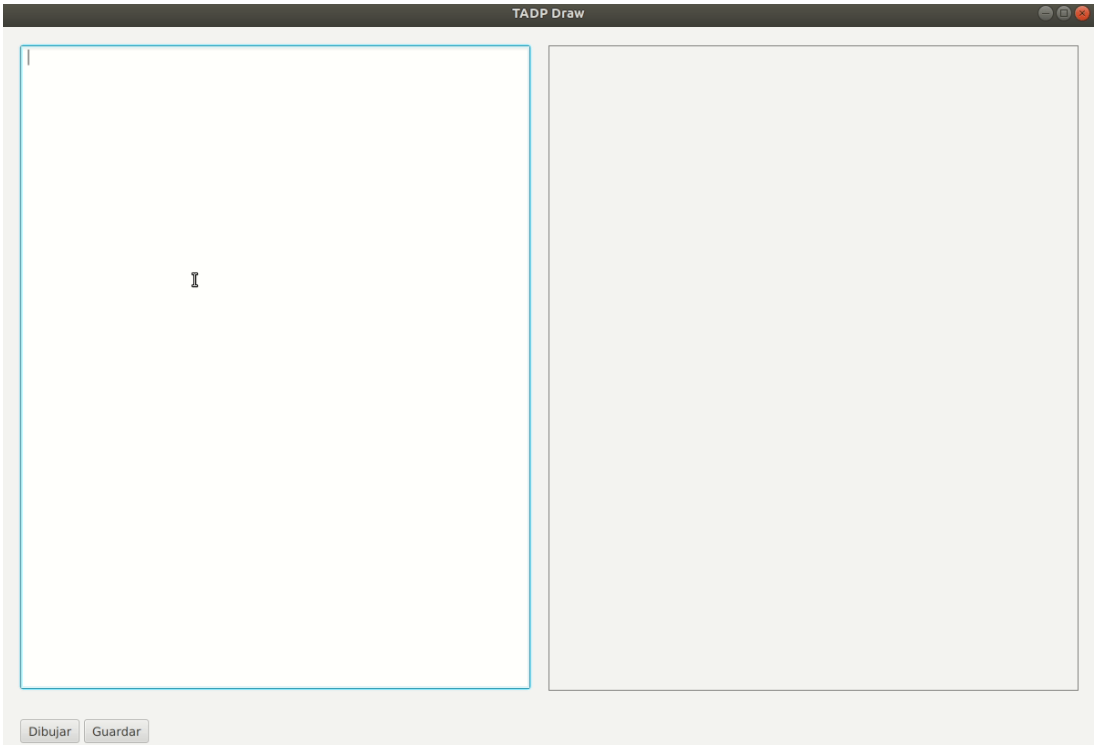
Al ejecutarse genera una imagen y la guarda como **prueba.png** en la carpeta **out** (que tiene que estar creada en la raíz del proyecto):



forInteractiveScreen:

```
TADPDrawingAdapter.forInteractiveScreen { (imageDescription, adapter) =>
 imageDescription match {
   case "triangulo" => adapter.triangle((100, 100), (0, 150), (200, 300))
   case "rectangulo" => adapter.beginColor(Color.rgb(100, 100, 100))
                               .rectangle((200, 200), (300, 400))
                               .end()
 }
}
```

Al ejecutar nos muestra una pantalla donde podemos escribir del lado izquierdo y apretando dibujar (o ctrl+Enter) dibujar algo en la pantalla de la derecha según la función que pasamos como parámetro. Pueden inyectar en esta función lo implementado en los puntos anteriores para que la pantalla pueda interpretar el lenguaje de imágenes.



Ejemplos de imágenes para probar

Algunas descripciones de imágenes que deberían ser válidas, y la imagen resultante para que puedan comparar:

- Composición C (la del ejemplo del principio)
- Corazón
- Murciélago
- Pepita
- Red
- Carpincho bostero: esta es muy pesada así que no se preocupen si no les anda por eso