

Laboratorio de programación funcional

Objetivos

Queremos que aprendan programación funcional, desde un aspecto práctico: cómo hacer para resolver un determinado problema con una mentalidad funcional. En particular, nos interesa hacer foco sobre dos aspectos:

1. Separación entre información (data) y procesamiento (funciones).
2. Funciones como valores, funciones de alto orden (funciones que toman funciones).

Además, tenemos otros dos objetivos que son horizontales a todos los labs:

1. Cómo lidiar con un nuevo lenguaje de programación. Si ya saben algo de Haskell ésta la tienen gratis en este lab. Si no se acuerdan mucho de pasadas experiencias, entonces seguramente tengan aquí ya un primer desafío.
2. LEER BIEN, encontrar documentación, instalar, etc.

Un lenguaje que vale más que mil dibujos

Este lab propone la implementación de un lenguaje pequeño específico para una tarea muy concreta: combinar dibujos básicos para crear diseños más interesantes. A este tipo de lenguajes se los suele conocer como DSL (Domain Specific Language: lenguaje de dominio específico) porque están pensados para eso: proveer abstracciones adecuadas para resolver problemas acotados a cierto ámbito. La idea original del lenguaje está en este [artículo](#) de Peter Henderson, que recomendamos leer.

Para definir un lenguaje necesitamos dos cosas: su sintaxis —cómo lo vamos a escribir—, y su semántica —cómo lo vamos a interpretar. Por ejemplo, en el lenguaje que vamos a ver vamos a *escribir* un operador de rotación, y lo vamos a *interpretar* como la rotación de una imagen. Esta interpretación estará dada en nuestro caso por una función que transforma los programas del DSL en dibujos en la pantalla. Otras interpretaciones son posibles, como escribir algo en un archivo, enviar paquetes por alguna interfaz de red, etc.

El laboratorio

Van a tener que:

1. Implementar el DSL en Haskell, lo cual consiste de dos partes:
 - a. Implementar la sintaxis del lenguaje, es decir definir cómo serán las “palabras” del lenguaje.
 - b. Implementar la semántica. Esto significa dar una función que toma una “palabra” y devuelve una interpretación, que en nuestro caso es un dibujo que responde a las instrucciones descritas en la palabra. Para eso deberán usar la biblioteca de gráficos [Gloss](#) (hay punto extra y agradecimiento eterno de Beta si usan otra biblioteca).
2. Utilizar el DSL para “decir” algo. En este caso queremos que reproduzcan una versión simplificada de la figura de Escher que se muestra en el artículo de Henderson, que consta de la superposición de una figura repetida y alterada hasta formar una composición compleja. Esta tarea incluirá las siguientes actividades:
 - a. Visualizar correctamente el gráfico de prueba que dibuja todas las operaciones sobre las figuras primitivas
 - b. Hacer una grilla de líneas horizontales y verticales
 - c. Reproducir el dibujo de Escher
 - d. Extender la interfaz de usuario del programa principal para que imprima en pantalla los nombres de todos los dibujos registrados.

Obvio que pueden ponerse creativos y hacer todos los dibujos que quieran!

3. Escribir tests, preferentemente usando herramientas de inteligencia artificial para generación de código.
4. Responder en forma detallada preguntas sobre el lab.

1. Implementar el lenguaje

Nuestro lenguaje está parametrizado sobre una colección de figuras básicas (representado por el no terminal `<Fig>`) y contiene instrucciones para hacer ciertas operaciones con las figuras básicas. Estas operaciones incluyen, rotar un dibujo, espejarlo, juntar horizontalmente un par de dibujos, apilar verticalmente una pareja de dibujos y superponer, o encimar, dos dibujos. La sintaxis de nuestro lenguaje va a estar definida por la siguiente gramática:

```
<Dibujo> ::= Figura <Fig> | Rotar <Dibujo> | Espejar <Dibujo>
| Rot45 <Dibujo>
| Apilar <Float> <Float> <Dibujo> <Dibujo>
| Juntar <Float> <Float> <Dibujo> <Dibujo>
| Encimar <Dibujo> <Dibujo>
```

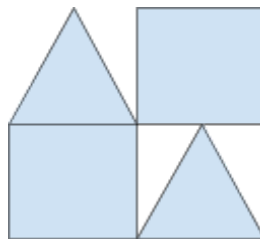
Usando este lenguaje podemos definir en Haskell funciones que combinan programas `<Dibujo>` para producir otros. Usualmente esas funciones se llaman combinadores. Por ejemplo, `rotar180 <Dibujo>` se define fácilmente como la composición de `Rotar` con `Rotar`.

La semántica formal (es decir, la interpretación) de las figuras básicas es una función que toma tres vectores a , b , c en \mathbb{R}^2 y produce una figura bi-dimensional donde a indica el desplazamiento del origen, b el ancho y c el alto.

Por ejemplo, si nuestras figuras básicas son triángulos y rectángulos, vamos a poder escribir

```
Juntar 1.0 1.0
  (Apilar 1.0 1.0 (Figura Triangulo) (Figura Rectangulo))
  (Apilar 1.0 1.0 (Figura Rectangulo) (Figura Triangulo))
```

y eso luego lo interpretaremos como



Como veremos, esta interpretación estará dada por dos funciones. Una será genérica, e interpretará los constructores del tipo `<Dibujo>`, rotando, apilando, etc las figuras básicas. Otra será construida específicamente para cada `<Fig>` que deseemos construir. Es decir, `<Dibujo>` va a ser *paramétrica* en `<Fig>`. En el ejemplo de arriba, `<Fig>` será (al menos) `Triangulo | Rectangulo`, y la función de interpretación específica será la encargada de dibujar un triángulo o rectángulo según corresponda.

1.a. Implementando la sintaxis

En el archivo `Dibujo.hs` vamos a definir el tipo principal para hacer dibujos, y funciones básicas para poder operar con dibujos.

1. Definir el lenguaje como un tipo de datos. Como no sabemos a priori qué figuras básicas tendremos, nuestro tipo `Dibujo` debe ser *polimórfico*. Como ejemplo, si vamos a hacer figuras que sólo tengan triángulos y rectángulos, podemos definir el tipo:

```
data TriORect = Triangulo | Rectangulo deriving (Eq, Show)
```

y luego definiremos nuestra composición fantástica como

```
type Fantastica = Dibujo TriORect
```

Nota: su tipo `Dibujo` también debe terminar con `deriving (Eq, Show)` para poder

comparar y mostrar en pantalla los constructores.

2. Vamos a modularizar el código del siguiente modo: en vez de exportar los constructores del tipo `Dibujo` vamos a exportar *funciones constructoras*. Esto nos permite desacoplar el tipo específico de la implementación. Nota: internamente dentro del archivo `Dibujo.hs` vamos a poder utilizar los constructores de tipo sin problema.
3. **Definir los siguientes combinadores.** Intenten no repetir código, usando funciones ya definidas o por definir. **Esta consideración se aplica al resto del trabajo.**

```
-- Composición n-veces de una función con sí misma. Componer 0 veces
-- es la función constante, componer 1 vez es aplicar la función 1 vez, etc.
-- Componer negativamente es un error!
comp :: (a -> a) -> Int -> a -> a

-- Rotaciones de múltiplos de 90.
r180 :: Dibujo a -> Dibujo a
r270 :: Dibujo a -> Dibujo a

-- Pone el primer dibujo arriba del segundo, ambos ocupan el mismo espacio.
(..) :: Dibujo a -> Dibujo a -> Dibujo a

-- Pone un dibujo al lado del otro, ambos ocupan el mismo espacio.
(///) :: Dibujo a -> Dibujo a -> Dibujo a

-- Superpone un dibujo con otro.
(^^^ :: Dibujo a -> Dibujo a -> Dibujo a

-- Dados cuatro dibujos los ubica en los cuatro cuadrantes.
cuarteto :: Dibujo a -> Dibujo a -> Dibujo a -> Dibujo a -> Dibujo a

-- Un dibujo repetido con las cuatro rotaciones, superpuestos.
encimar4 :: Dibujo a -> Dibujo a

-- Cuadrado con el mismo dibujo rotado i * 90, para i ∈ {0, ..., 3}.
-- No confundir con encimar4!
ciclar :: Dibujo a -> Dibujo a
```

4. **Definir esquemas para la manipulación de figuras básicas.**

```
-- Ver un a como un dibujo.
figura :: a -> Dibujo a

-- map para nuestro lenguaje.
mapDib :: (a -> Dibujo b) -> Dibujo a -> Dibujo b

-- Verificar que satisfaga la siguiente igualdad:
```

```

-- mapDib figura = id      (id es la función identidad).

-- Estructura general para la semántica (a no asustarse). Ayuda:
-- pensar en foldr y las definiciones de intro a la lógica
-- foldDib aplicado a cada constructor de Dibujo debería devolver el mismo
-- dibujo
foldDib :: (a -> b) -> (b -> b) -> (b -> b) -> (b -> b) ->
  (Float -> Float -> b -> b -> b) ->
  (Float -> Float -> b -> b -> b) ->
  (b -> b -> b) ->
  Dibujo a -> b

-- Extrae todas las figuras básicas de un dibujo.
figuras :: Dibujo a -> [a]

```

5. Usando los esquemas anteriores, es decir **no se puede hacer pattern-matching**, definir estas funciones en el archivo `Pred.hs`:

```

-- `Pred a` define un predicado sobre figuras básicas. Por ejemplo,
-- `(== Triangulo)` es un `Pred TriOCuat` que devuelve `True` cuando la
-- figura es `Triangulo`.
type Pred a = a -> Bool

-- Dado un predicado sobre figuras básicas, cambiar todas las que satisfacen
-- el predicado por el resultado de llamar a la función indicada por el
-- segundo argumento con dicha figura.
-- Por ejemplo, `cambiar (== Triangulo) (\x -> Rotar (Figura x))` rota
-- todos los triángulos.
cambiar :: Pred a -> (a -> Dibujo a) -> Dibujo a -> Dibujo a

-- Alguna figura satisface el predicado.
anyFig :: Pred a -> Dibujo a -> Bool

-- Todas las figuras satisfacen el predicado.
allFig :: Pred a -> Dibujo a -> Bool

-- Los dos predicados se cumplen para el elemento recibido.
andP :: Pred a -> Pred a -> Pred a

-- Algún predicado se cumple para el elemento recibido.
orP :: Pred a -> Pred a -> Pred a

```

1.b. La interpretación geométrica

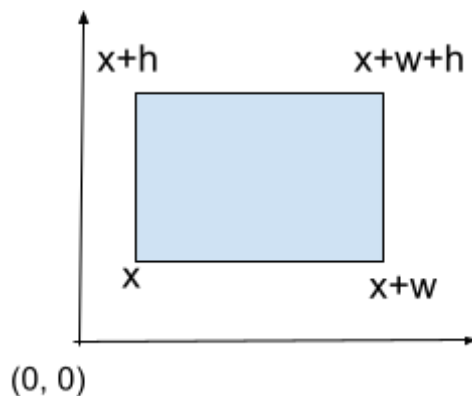
Se debe **completar o modificar el archivo `Interp.hs`** para que utilice una biblioteca para

generar gráficos, e interprete las figuras en ésta. Se recomienda [gloss](#) pero pueden usar otra (ver `INSTALL.md` para instalar gloss).

```
-- Suponemos que la biblioteca provee el tipo Vector y Picture.
type Output a = a -> Vector -> Vector -> Vector -> Picture

interp :: Output a -> Output (Dibujo a)
```

Van a tener que entender qué hace `Output` e `interp`. Por lo pronto, es importante entender el sistema vectorial que vamos a usar para graficar en pantalla. Los tres vectores, llamémosles x , w , h , nos indican en qué espacio vamos a dibujar. El siguiente gráfico explica la interpretación gráfica, mostrando los cuatro puntos de un rectángulo con origen x , ancho w , y altura h :



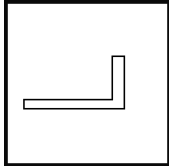
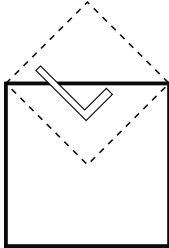
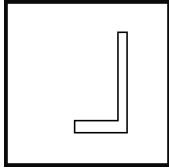
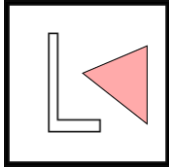
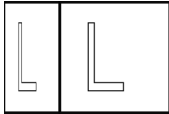
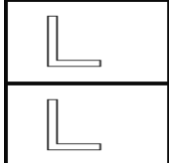
Nota: no necesariamente el espacio sea un rectángulo como el de la figura; variando los vectores podemos hacer espacios deformados. Esto es lo que los operadores del lenguaje van a variar para modificar la forma en que se debe dibujar una figura.

La semántica de cada operación de nuestro lenguaje está explicada en la tabla que figura a continuación, donde se debe entender a $func(f)(x, w, h)$ como el efecto de interpretar la función matemática $func$ (que se corresponde a uno de nuestros constructores) sobre la figura f , en los vectores x, w, h .

Supongamos que tenemos funciones f, g que producen la siguientes figuras:

figura

$f(x, w, h)$	
$g(x, w, h)$	

Operación	Semántica	Visualmente
$rotar(f)(x, w, h)$	$f(x+w, h, -w)$	
$rot45(f)(x, w, h)$	$f(x+(w+h)/2, (w+h)/2, (h-w)/2)$ ¡Notar que se va del espacio asignado por las coordenadas iniciales!	
$espejar(f)(x, w, h)$	$f(x+w, -w, h)$	
$encimar(f,g)(x, w, h)$	$f(x, w, h) \cup g(x, w, h)$	
$juntar(n, m, f, g)(x, w, h)$	$f(x, w', h) \cup g(x+w', r'*w, h)$ con $r'=n/(m+n), r=m/(m+n), w'=r*w$	
$apilar(n, m, f, g)(x, w, h)$	$f(x + h', w, r*h) \cup g(x, w, h')$ con $r' = n/(m+n), r=m/(m+n), h'=r'*h$	

Se recomienda fuertemente realizar dibujitos para comprender las operaciones.

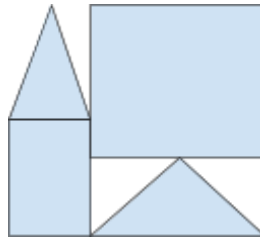
Para entender las proporciones en los números de Juntar y Apilar,

Juntar 1.0 2.0

(Apilar 1.0 1.0 (Figura Triangulo) (Figura Rectangulo))

(Apilar 2.0 1.0 (Figura Rectangulo) (Figura Triangulo))

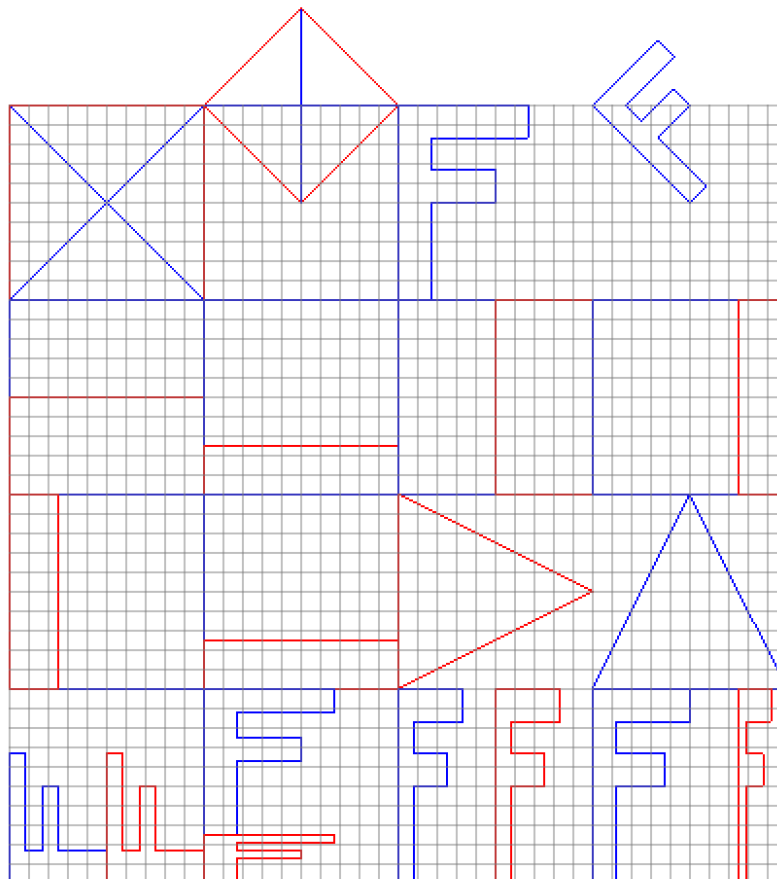
produce la siguiente figura. Es decir, la segunda columna ocupa el doble de tamaño que la primera, y en la segunda columna la primera figura (el cuadrado) ocupa el doble de tamaño que la segunda (el triángulo).



2. Utilizar el lenguaje

2.a. Gráfico de prueba

Para probar que lo que hicieron es correcto, les dejaremos un dibujo llamado `Dibujos/Feo.hs`, que representa el gráfico a continuación. Este dibujo es sólo una prueba arbitraria de las distintas operaciones.



La grilla gris de fondo es orientativa y se imprime en cada dibujo que hagan para facilitar el trabajo.

Ver el gráfico

Repasemos los tres componentes de nuestro lab: (i) el lenguaje, (ii) la interpretación geométrica, y (iii) los usos de nuestro lenguaje (por ahora sólo uno, el dibujo Feo). En ningún caso estamos produciendo ningún comportamiento, simplemente generamos valores de algún tipo. Es medio obvio que nos gustaría poder mostrar en la pantalla nuestros dibujos. Para eso necesitamos lidiar con la entrada/salida.

Como quizás ya saben, la forma en que se estructura la interacción en Haskell es a través de la mónada de IO. No nos preocupemos por qué es una mónada (para eso pueden hacer Conceptos Avanzados de Lenguajes de Programación cuando se dicte), nos basta con saber que la librería `gloss` nos ofrece una interfaz cómoda para eso.

Una ventaja (y desventaja...) de Haskell es la clara separación de responsabilidades: para resolver un problema en general debemos centrarnos en la solución *funcional* del mismo y lo más probable es que no necesitemos IO (excepto por cuestiones de eficiencia, quizás). Una vez que tenemos resuelto el problema (en nuestro caso los componentes que mencionamos más arriba), podemos armar un componente más para la IO.

En nuestro caso, lo que tenemos que realizar es utilizar la función apropiada de `gloss`:

```
display :: Display -> Color -> Picture -> IO ()
```

Hay dos alternativas para el argumento `Display`: una ventana (que podemos definir con `InWindow "titulo" (width, height) (x0, y0)`) o con pantalla completa (`FullScreen`). Para el `Color` de fondo se pueden pasar algunos colores predefinidos (los detalles no importan), y el último argumento es la figura a mostrar. El resultado es una *computación* en la mónada de IO. Para ejecutar nuestro programa debemos tener una función `main`. Por ejemplo, el siguiente programa muestra un círculo de tamaño 100 en una ventana de tamaño 200.

```
win = InWindow "Paradigmas" (200, 200) (0, 0)
main :: IO ()
main = display win white $ circle 100
```

Esto está solucionado en el código que les pasamos, aunque entenderlo puede servirles para hacer algunas pruebas iniciales.

2.b. Grilla

El primer dibujo que van a tener que implementar de cero es una **grilla numerada de 8x8**, donde se imprime (0, 0) en el cuadrante de más arriba a la izquierda, y (7, 7) en el de más abajo a la derecha, donde en cada (i, j) i representa la fila y j la columna. Hint: para hacer la grilla vean y reutilicen el código de `Feo.hs`.

No se preocupen por la ubicación y el tamaño del texto (miren lo feo que está acá!), siempre

y cuando esté más o menos contenida en su cuadrante. Van a tener que investigar un poco de Gloss para hacerlo.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

2.c. Escher

La última tarea es **reconstruir el gráfico de Escher (con triángulos)**. Para eso se debe crear un módulo `Dibujos/Escher.hs` donde definen un sinónimo de tipos adecuado e implementan los siguientes combinadores, en función de la siguiente descripción de los dos primeros niveles:

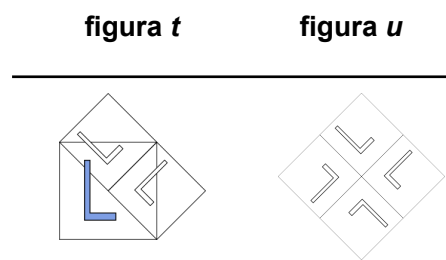
lado(1, f) = cuarteto(blank, blank, rot45(f), f)

lado(2, f) = cuarteto(lado(1, f), lado(1, f), rot45(f), f)

esquina(1, f) = cuarteto(blank, blank, blank, dibujo_u(f))

esquina(2, f) = cuarteto(esquina(1, f), lado(1, f), rot45(lado(1, f)), dibujo_u(f))

Para esto también necesitan las figuras *u* y *t* del paper de Henderson, que nosotros las generalizamos un poco, en azul se muestra la figura original.



Ya estamos cerca de completar el proceso, necesitamos un combinador para nueve piezas:

P	Q	R
S	T	U
V	W	X

Finalmente podemos definir:

$escher(n, f) = noneto(...)$, donde en **P** va $esquina(n, f)$ y en **Q** va $lado(n, f)$, el resto de las letras deben resolverlas ustedes.

```
-- Supongamos que eligen.
type Escher = Bool

-- El dibujoU.
dibujoU :: Dibujo Escher -> Dibujo Escher
dibujoU p = undefined

-- El dibujo t.
dibujoT :: Dibujo Escher -> Dibujo Escher
dibujoT p = undefined

-- Esquina con nivel de detalle en base a la figura p.
esquina :: Int -> Dibujo Escher -> Dibujo Escher
esquina n p = undefined

-- Lado con nivel de detalle.
lado :: Int -> Dibujo Escher -> Dibujo Escher
lado n p = undefined
```

```
-- Por suerte no tenemos que poner el tipo!  
noneto p q r s t u v w x = undefined  
  
-- El dibujo de Escher:  
escher :: Int -> Escher -> Dibujo Escher  
escher = undefined
```

2.c. Listar dibujos

Actualmente el programa principal toma el primer argumento y busca un dibujo con ese nombre. Si no hay, imprime el error. Extiendan la funcionalidad para que `si uno pasa --lista` imprima en pantalla los nombres de todos los dibujos registrados. **Bonus si además pregunta cuál mostrar** (y lo muestra en caso de estar).

3. Tests

Deben agregar tests en la carpeta `Tests`. Recomendamos fuertemente utilizar [GitHub Copilot](#) para hacerlos. El fin de esta recomendación es doble: por un lado, aliviar la tarea de crear los tests, y por otro, aprender los límites de las herramientas de Inteligencia Artificial.

Debe haber al menos `tests para los módulos Pred y Dibujo`. Copilot sugiere [HUnit](#), y es una buena opción.

4. Preguntas

Las siguientes preguntas deben ser respondidas correctamente, con el mayor grado de precisión y claridad que puedan.

1. ¿Por qué están separadas las funcionalidades en los módulos indicados? Explicar detalladamente la responsabilidad de cada módulo.
2. ¿Por qué las figuras básicas no están incluidas en la definición del lenguaje, y en vez es un parámetro del tipo?
3. ¿Qué ventaja tiene utilizar una función de `fold` sobre hacer pattern-matching directo?

Qué se evalúa y puntos extras

- No se evaluarán proyectos que no se puedan compilar. La idea es que ningún grupo llegue a este punto al momento de la entrega: pregunten temprano para evitar esto. **Hint:** no intenten compilar al final. Deben committear seguido; asegúrense que cada commit compile.
- Que la elección de los tipos de datos sea la adecuada; en programación funcional esto es clave.
- Que se comprendan los conceptos de funciones de alto orden y la forma en que se combinan funciones.
- Que se haga buen reuso de funciones, es decir, que no reinventen una solución cada vez que se presente un mismo problema.
- Que se pueda adaptar fácilmente a otros usos; en algún momento, antes de la entrega, liberaremos un archivo `Basica/Feo.hs` que use `Dibujo.hs` e `Interp.hs` que les permita testear.
- Que el código sea elegante: líneas de tamaño razonable, buen espaciado, consistencia.

Se consiguen puntos extras si:

- **Hacen otra figura interesante, como una imagen fractal**, explicando cada paso de su construcción y de dónde sacaron la idea.
- **Extienden el lenguaje para indicar animaciones de figuras**. Hagan esto en un branch separado y comenten en el `README.md` de la branch `main` que lo hicieron.
- **Agregan** al lenguaje **un operador para permitir modificar las proporciones de una imagen**. Luego deben quitar las proporciones de los operadores `Apilar` y `Juntar`, dejando `apilar` y `juntar` tal cual como están ahora (es decir, deben usar el nuevo operador).

Entrega

Repositorio

Fecha de entrega: hasta el jueves 13/04/2023 a las 23:59:59 (hora de Argentina).

Deberán crear un tag indicando el release para corregir:

```
$ git tag -a lab-1 -m 'Algún mensaje lindo' && git push --tags
```

Si no está el tag, se corrige hasta la hora de entrega. Tampoco se consideran commits posteriores al tag. Pero pueden corregir el tag.

Qué debe haber en el repositorio

El contenido mínimo del repositorio debería ser el siguiente:

README.md	Un readme breve donde comentan su experiencia y responden las preguntas. Indicar acá si usan otra biblioteca
Dibujo.hs	Tipo de datos para <Dibujo> y todas las funciones relacionadas
Pred.hs	Funciones de predicados
Interp.hs	Interpretación geométrica de los dibujos
Dibujos/Escher.hs	Definición de combinadores, elección de tipo para instanciar <code>Dibujo</code> , definición de la interpretación de todas las figuras básicas
Dibujos/Grilla.hs	La grilla numerada
Dibujos/<Nombre>.hs	Si se copan y hacen otros diseños, que estén en el directorio <code>Dibujos</code>
Main.hs	Programa principal
Tests/TestDibujo.hs	Test del módulo <code>Dibujo</code>
Tests/TestPred.hs	Test del módulo <code>Pred</code>

Recursos sobre Haskell

- [Learn you a Haskell...](#) (Nota: tiene algún que otro comentario o ejemplo políticamente incorrecto y estúpido.)
- [Aprende Haskell... \(traducción del anterior\)](#).
- [Real World Haskell](#).
- [Buscador de funciones por tipo](#).
- [Guía de la sintaxis de Haskell](#).
- [Documentación de gloss](#).
-