Lab1: Programación Funcional en Haskell

Objetivos

La intención de este laboratorio es desafiarlos a resolver un problema computacional concreto y complejo utilizando los conceptos de la programación funcional, es decir, ofrecerles una experiencia de cómo abordar un problema desde el paradigma funcional.

En particular, nos interesa hacer foco sobre tres aspectos:

1. Separación entre “tipos” (constructores del tipo) y “funciones del tipo” que operan sobre ellos (operadores del tipo).
2. Definición de Tipos y Operadores Polimórficos.
3. Funciones como valores nativos del lenguaje, abstracciones de alto orden (funciones que toman y devuelven funciones).

Además, tenemos objetivos que son transversales a todos los labs del curso:

1. Cómo lidiar con un nuevo lenguaje de programación (Haskell) y un skeleton ofrecido por la cátedra.
2. Trabajar en equipo, dividiendo tareas y luego coordinarlas utilizando software de versionamiento (git).
3. Encontrar e interpretar correctamente documentación sobre librerías, poder adaptar ejemplos generales a su necesidad o problemática actual.

Lab1: Definiendo Nuestro Propio “Lenguaje de Descripción”

Proponemos la implementación de un pequeño lenguaje, no de programación sino descriptivo, es decir, un lenguaje que no cuenta con capacidad de cómputo (no hay noción de flujo de programa), solo sirve para describir “alguna cosa”. En la industria hay lenguajes descriptivos muy utilizados tales como html, latex, xml, etc, que nos permiten describir la estructura de un página web, o de un documento de texto con mucha notación matemática, o los datos de alguna base de datos, respectivamente.

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 específicos a cierto ámbito. La idea original de nuestro lenguaje está en el artículo de Peter Henderson, que recomendamos leer.

En particular, nuestro lenguaje descriptivo nos permitirá describir de manera muy simple figuras complejas a partir de:

1. Un conjunto de figuras básicas o atómicas , tales como triángulo, cuadrado, circulo, perrito, casita, etc).

2. Un conjunto de constructores que realizan ciertas transformaciones geométricas de una figura, tales como “rotar” una figura, “apilar” dos figuras, “juntar” dos figuras, etc.

Asi por ejemplo, la cadena “Rotar (Basica Triangulo)” será una expresión válida de nuestro lenguaje y ésta expresa de alguna manera una figura que consiste en un triángulo rotado.

Para definir un lenguaje necesitamos definir dos cosas:

1. **Su Sintaxis:** el conjunto de expresiones o fórmulas válidas del lenguaje, i.e, cómo vamos a escribir nuestras expresiones en el lenguaje.

2. **Su Semántica:** cómo vamos a interpretar cada expresión del lenguaje, i.e, definir la figura geométrica de cada expresión del lenguaje. Por ejemplo, para la expresión “Rotar (Basica Triangulo)” deberíamos ver en el monitor un triángulo rotado. La interpretación estará dada en nuestro caso por una función que transforma las expresiones del DSL en dibujos visibles en la pantalla.

Manos a la Obra

1. Implementar el DSL en Haskell, lo cual consiste de dos partes:
    a. Implementar la sintaxis del lenguaje, es decir definir cuáles serán las “expresiones válidas” del lenguaje.
    b. Implementar la semántica. Esto significa definir una función que toma una expresión del lenguaje y devuelve su respectivo dibujo por pantalla. Para esto deberán usar alguna biblioteca de gráficos. Recomendamos `Gloss`.

2. Utilizar el DSL para “expresar o dibujar 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. Obvio que pueden ponerse creativos y hacer todos los dibujos que quieran.

En las páginas siguientes tenemos más detalle de cada punto.

1.a Sintaxis del Lenguaje

Las expresiones válidas de nuestro lenguaje siguen la siguiente gramática:

```
Dibujo a :=
    | Basica a
    | Rotar <Dibujo a>
    | Rotar45 <Dibujo a>
    | Espejar <Dibujo a>
    | Apilar <Float> <Float> <Dibujo a> <Dibujo a>
    | Juntar <Float> <Float> <Dibujo a> <Dibujo a>
    | Encimar <Dibujo a>
```

Notar que las expresiones del lenguaje deben soportar polimórfismo, por ejemplo:

```
"Rotar (Apilar 1 1 (Basica 1) (Basica 2))"
es una expresion de Dibujo Int

"Rotar (Apilar 1 1 (Basica True) (Basica False))"
es una expresion de Dibujo Bool

Data MisBasicas = Triangulo | Rectangulo
"Rotar (Apilar 1 1 (Basica Triangulo) (Basica Rectangulo))"

es una expresion de Dibujo MisBasicas
```

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*.Por 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 el tipo de nuestras composiciones fantásticas 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 identidad, componer 1 vez es aplicar la función 1 vez, etc.
-- Componer negativamente es un error!
comp :: Int -> (a -> a) -> 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 funciones de alto orden para la manipulación de Dibujos.

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

-- Funcion de fold para Dibujos a
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
```

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

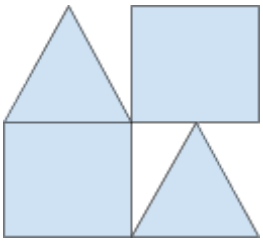
```haskell
-- `Pred a` define un predicado sobre figuras básicas. Por ejemplo,
-- `(== Triangulo)` es un `Pred TriOCuat` que devuelve `True` cuando la
-- figura es `Triangulo`.
type Pred a = a -> Bool

-- Dado un predicado sobre 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 (Basica x))` rota
-- todos los triángulos.
cambiar :: Pred a -> (a -> Dibujo a) -> Dibujo a -> Dibujo a

-- Alguna Basica satisface el predicado.
anyDib :: Pred a -> Dibujo a -> Bool

-- Todas las basica satisfacen el predicado.
allDib :: Pred a -> Dibujo a -> Bool

-- Hay 4 rotaciones seguidas.
esRot360 :: Pred (Dibujo a)


-- Hay 2 espejados seguidos.
esFlip2 :: Pred (Dibujo a)


data Superfluo = RotacionSuperflua | FlipSuperfluo

-- Chequea si el dibujo tiene una rotacion superflua
errorRotacion :: Dibujo a -> [Superfluo]

-- Chequea si el dibujo tiene un flip superfluo
errorFlip :: Dibujo a -> [Superfluo]

-- Aplica los chequeos rotacion y flip superfluo
-y acumula todos los errores en una lista y
- sólo devuelve la figura si no hubo ningún error.
checkSuperfluo :: Dibujo a -> Either [Superfluo] (Dibujo a)
```

2.b Semántica del Lenguaje

La semántica formal (es decir, la interpretación) de las figuras 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. (Más adelante hay más detalles sobre esto). Por ejemplo, si nuestras básicas son triángulos y rectángulos, vamos a poder escribir

```
Juntar 1 1
  (Apilar 1 1 (Basica Triangulo) (Basica Rectangulo))
  (Apilar 1 1 (Basica Rectangulo) (Basica Triangulo))
```

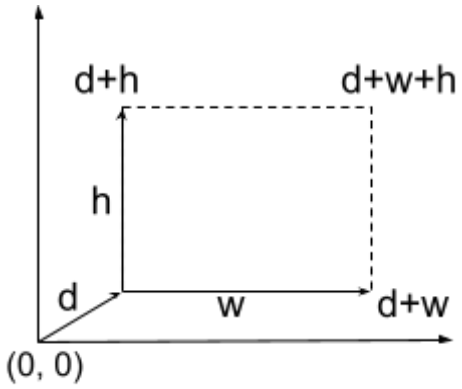
y eso luego lo interpretaremos por pantalla como



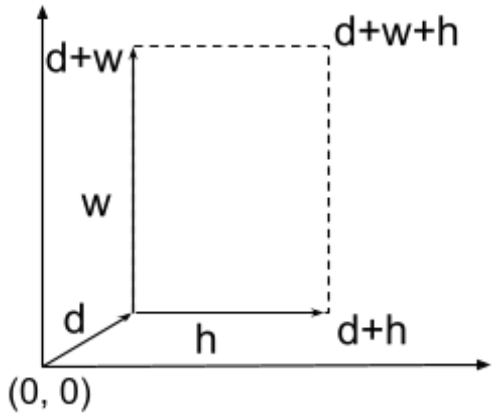
Se debe completar o modificar el archivo `Interp.hs` para que utilice una biblioteca para generar gráficos, e interprete las figuras en ésta. Se recomienda gloss pero pueden usar otra (ver `INSTALL.md` para instalar gloss).

```haskell
-- Gloss provee el tipo Vector y Picture.
type ImagenFlotante = Vector -> Vector -> Vector -> Picture
type Interpretacion a = a -> ImagenFlotante
```

Es importante entender el sistema vectorial que vamos a usar para graficar en pantalla. Los tres vectores, llamémosles *d, 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 desplazamiento *d* desde el origen, ancho *w,* y altura *h*:

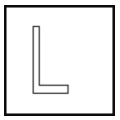
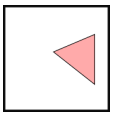


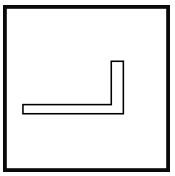
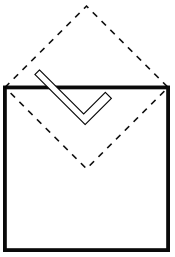
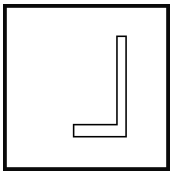
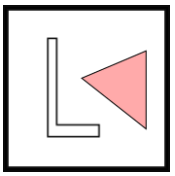
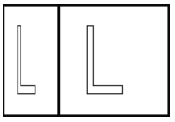
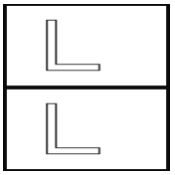
Nota: no necesariamente el espacio sea un rectángulo como el de la figura; variando los vectores podemos hacer espacios deformados. Esto es lo que los operadores del lenguaje van a variar para modificar la forma en que se debe dibujar un dibujo. Por ejemplo, el mismo gráfico base intercambiando los parámetros *h* y *w* nos da:



La semántica de cada operación de nuestro lenguaje está explicada en la tabla que figura a continuación, donde se debe entender a *func(f)(d, 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 *d, w, h*.

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

	figura		**figura**
f(d, w, h)		*g(d, w, h)*	

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

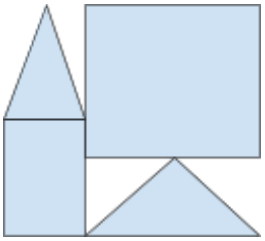
Se recomienda enfáticamente realizar dibujitos para comprender las operaciones.

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

```
Juntar 1.0 2.0
  (Apilar 1.0 1.0 (Basica Triangulo) (Basica Rectangulo))
  (Apilar 2.0 1.0 (Basica Rectangulo) (Basica 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).



```
-- Interpretaciones de los constructores de Dibujo

--interpreta el operador de rotacion
interp_rotar :: ImagenFlotante -> ImagenFlotante

--interpreta el operador de espejar
interp_espejar :: ImagenFlotante -> ImagenFlotante

--interpreta el operador de rotacion 45
interp_rotar45 :: ImagenFlotante -> ImagenFlotante

--interpreta el operador de apilar
interp_apilar :: Int -> Int -> ImagenFlotante -> ImagenFlotante ->
ImagenFlotante

--interpreta el operador de juntar
interp_juntar :: Int -> Int -> ImagenFlotante -> ImagenFlotante ->
ImagenFlotante

--interpreta el operador de encimar
interp_encimar :: ImagenFlotante -> ImagenFlotante -> ImagenFlotante

--interpreta cualquier expresion del tipo Dibujo a
interp :: Interpretacion a -> Dibujo a -> ImagenFlotante
```

2. Utilizar el lenguaje

2.a. Dibujos de prueba

Para probar que lo que hicieron es correcto en la carpeta Basica hay diferentes ejemplos de interpretaciones posibles de basicas y expresiones simples de dibujo como ejemplo para explorar.

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 (Main.hs), aunque entenderlo puede servirles para hacer algunas pruebas iniciales.

A tener en cuenta!!!

- 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 el código sea elegante: líneas de tamaño razonable, buen espaciado, consistencia.
- Que no haya errores de runtime; por ejemplo, por hacer: `head []` .

Para quienes desean ir más alla en el laboratorio pueden:

- Hacen un dibujo interesante, como una imagen fractal, un dibujo de escher.
- Extienden el lenguaje para expresar animaciones de dibujos.
- Agregan al lenguaje un operador para permitir modificar las dimensiones de un dibujo (u otras transformaciones, por ejemplo, rotar α grados).Entrega

Fecha de entrega: hasta Viernes 11/04/2025.

Recursos sobre Haskell

- Instalar Haskell, te recomendamos que sigas esos pasos.
- 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