Memòria pràctica 2

DAT QT2019

Alba Mendez

14 d'octubre de 2019

Part I.
Introducció

En aquesta pràctica es preten fer una breu introducció a Haskell, i a la programació funcional pura en general.

Es comença explicant què significa la programació funcional, què significa que sigui *pura* i què significa que sigui *avaluada per necessitat*, com en el cas de Haskell.

Llavors es presenta l'entorn de programació que es farà servir. Es disposa d'un mòdul `Drawing` que ofereix una API senzilla per pintar formes a través de fitxers SVG. També es disposa d'un script `executaMain` que compila i executa el fitxer de codi Haskell que li passem, generant els continguts del fitxer SVG en la seva sortida estàndard.

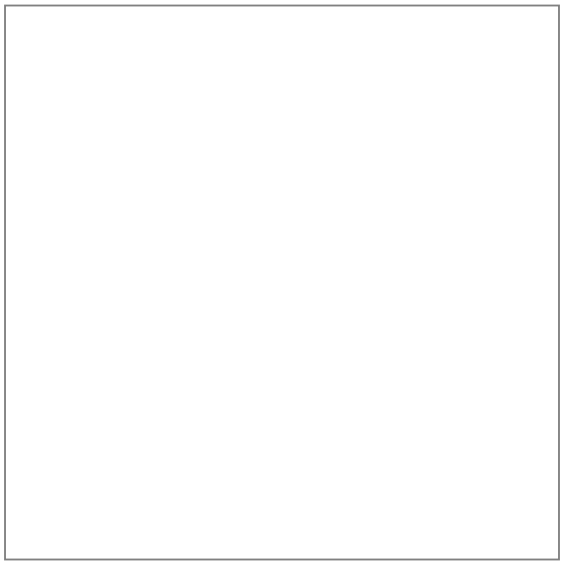


Figura 1: Resultat del primer exercici.

Part II.
Haskell bàsic

En aquest apartat s'explica com definir funcions, com funcionen les definicions i alguns operadors.

Exercici 1. Com a primer exercici es demana compilar un programa senzill amb el mòdul `Drawing`:

```
import Drawing

myDrawing :: Drawing
myDrawing = blank

main :: IO ()
main = svgOf myDrawing
```

El compilem i executem fent `executaMain ex1.hs > ex1.svg`. Llavors comprovem que, efectivament, es produeix un dibuix en blanc (fig. 1).

A continuació s'introdueixen algunes de les funcions de les que disposem en el mòdul `Drawing`, notablement `solidCircle`, `rectangle`, `colored`, `translated` i l'operador de composició: `<>`

Exercici 2. Es demana realitzar un programa que generi un semàfor amb tres llums de colors vermell, groc i verd. Hem de fer-ho mitjançant una funció `trafficLight` que dibuixi un llum a partir de dos paràmetres, el color i la posició en l'eix Y.

La implementació d'aquesta funció no té gaire misteri:

```
lightBulb (y, c) = colored c $ translated 0 y $ solidCircle 1
```

Abans de continuar, he vist necessari disposar d'una funció per composar una llista de formes. A aquesta funció li he anomenat `stack`, i es pot implementar fàcilment fent un fold:

```
stack :: Foldable f => f Drawing -> Drawing
stack = foldr (<>) blank
```

També he pensat que aniria bé una funció per mapejar una llista amb una funció i després, composar les formes resultants. L'anomenaré `dmap`:

```
dmap :: (x -> Drawing) -> [x] -> Drawing
dmap f l = stack $ f <$> l
```

Amb aquestes dues utilitats, escriure la resta del programa és tan fàcil com:

```
lights = [ (-2.5, green), (0, yellow), (2.5, red) ]
frame = rectangle 2.5 7.5 <> (colored gray $ solidRectangle 2.5 7.5)
trafficLight = dmap lightBulb lights <> frame

myDrawing = trafficLight
```

El resultat final es pot veure a la fig. 2, i el codi complert es troba a la pàgina 10.

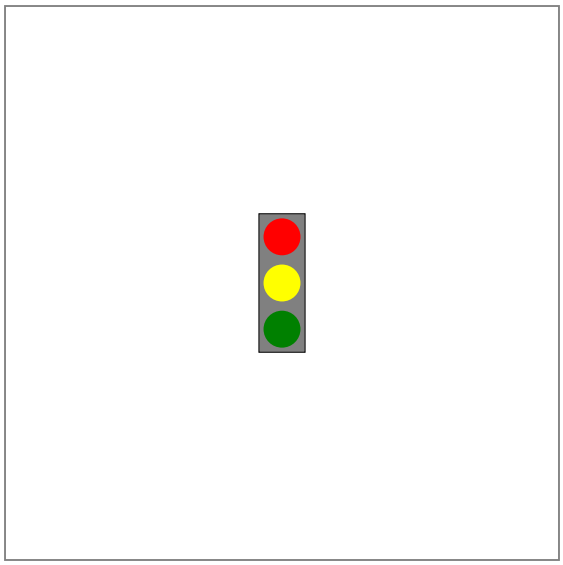


Figura 2: Resultat del segon exercici.

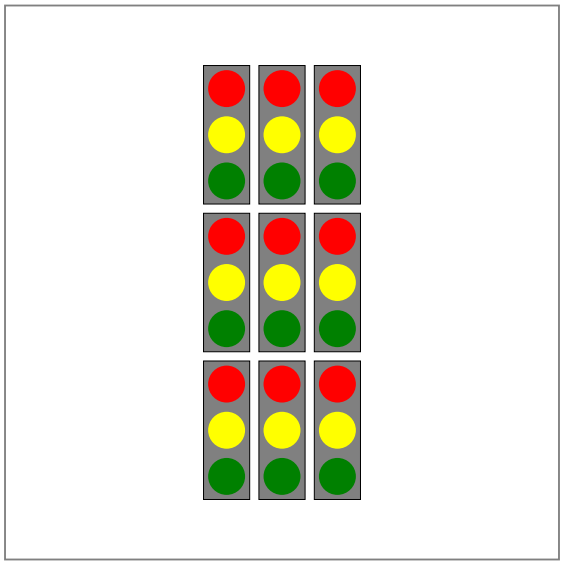


Figura 3: Resultat del tercer exercici.

Part III.
Més sobre funcions

1. Funcions recursives

En aquest apartat s'explica la necessitat de la recursivitat per expressar repeticions en un llenguatge funcional. S'explica com definir funcions recursives mitjançant pattern matching.

Exercici 3. Es demana dibuixar un array de 3×3 semàfors.

Només cal definir unes funcions utilitat, que repliquen objetes sobre un eix:

```
row xs fig = dmap (\x -> translated x 0 fig) xs
col ys fig = dmap (\y -> translated 0 y fig) ys
```

I combinar-les per obtenir l'array demanat:

```
myDrawing = col [-8, 0, 8] $ row [-3, 0, 3] $ trafficLight
```

El resultat final es pot veure a la fig. 3, i el codi complert es troba a la pàgina 11.

Exercici 4. Es demana ara dibuixar un arbre recursiu (veure resultat a la fig. 4a).

Per fer-ho, definirem una funció recursiva que dibuixa:

- una línia cap a dalt de l'origen, de longitud 1
- al final d'aquesta línia: dos subarbres, rotats 18° cap a cada sentit.

La funció queda així:

```
tree 0 = blank
tree n = let
    subtree = dmap (\s -> rotated (s*pi/2/5) $ tree (n-1)) [1, -1]
  in polyline [(0,0), (0,1)] <> translated 0 1 subtree

myDrawing = tree 8
```

El resultat es pot veure a la figura 4a.

Cal observar que aquesta definició **no** pinta una imatge simètrica, ja que una de les dues branques es pinta per sobre de l'altra. S'enten que això no és un problema, ja que la imatge que es donava com a enunciat era també així.

Es demana ara que l'arbre tingui 'flors' grogues al final de les branques. Això es pot aconseguir simplement canviant el cas zero de la funció recursiva:

```
tree 0 = colored yellow $ solidCircle 0.2
```

El resultat final es pot veure a la fig. 4b, i el codi complert es troba a la pàgina 12.

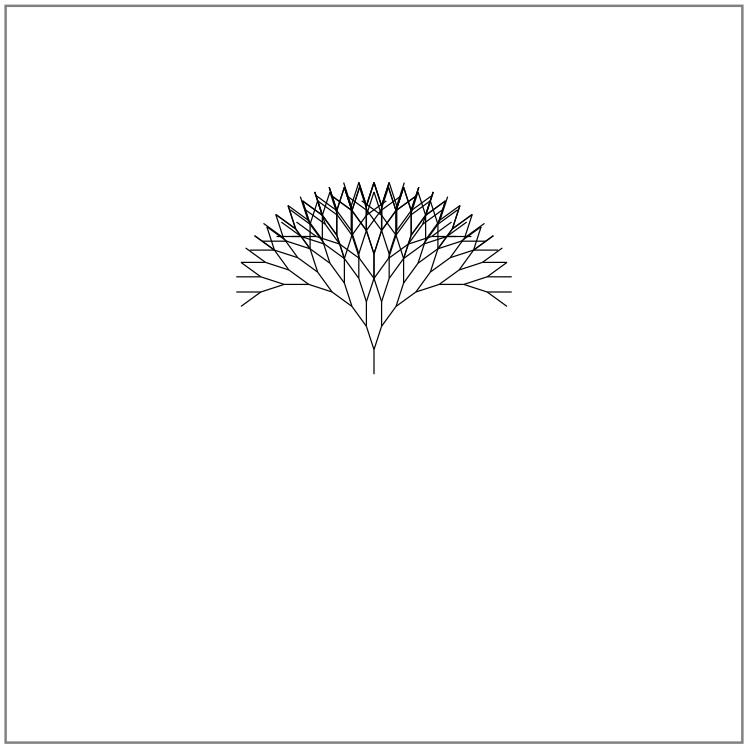
2. Funcions d'ordre superior

Ara s'introdueix el concepte de funció d'ordre superior (HOF) i la necessitat que solucionen. Més endavant s'introdueixen algunes funcions d'ordre superior incorporades a la llibreria estàndard de Haskell, i la classe *monoide*.

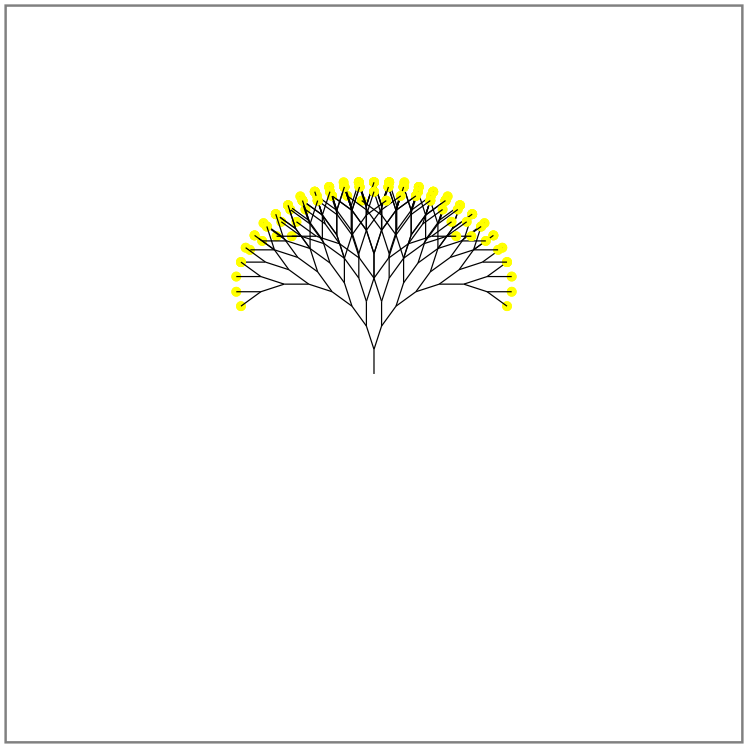
Exercici 5. Cal definir una funció `repeatDraw` que repetirà la funció `thing` n vegades, passant-li un enter de 1 a n en cada repetició.

La funció és només una versió especial de `dmap`:

```
repeatDraw :: (Int -> Drawing) -> Int -> Drawing
repeatDraw thing n = dmap thing [1..n]
```



(a) Arbre recursiu de vuit nivells.



(b) Arbre recursiu de vuit nivells, amb flors.

Figura 4: Resultat del quart exercici.

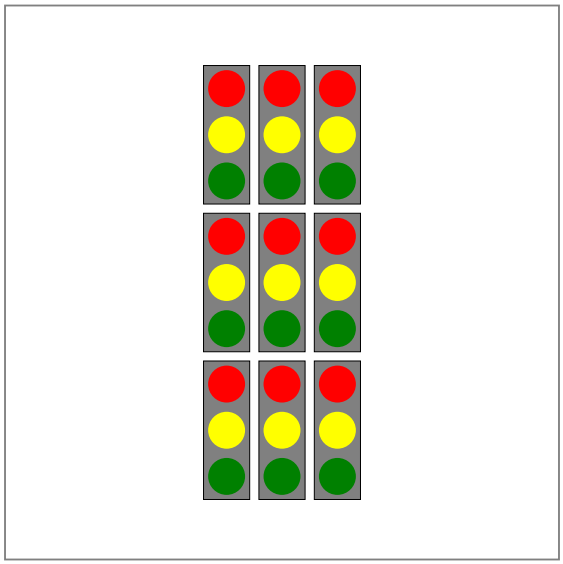


Figura 5: Resultat del cinqué exercici.

Ara l'exercici 3 es pot implementar així:

```
myDrawing = repeatDraw lightRow 3

lightRow :: Int -> Drawing
lightRow r = repeatDraw (light r) 3

light :: Int -> Int -> Drawing
light r c = translated (3 * fromIntegral c - 6) (8 * fromIntegral r -
  ↪  16) trafficLight
```

Comprovem que es produeix el mateix resultat (fig. 5). El codi complert es troba a la pàgina 13.

Ara s'introdueix la classe `Monoid m`, s'explica les operacions que implementa, i s'indica que `Drawable` implementa aquesta classe.

Sabent que els dibuixos són monoides, m'adono que he reinventat la roda... La meva funció `stack` fa el mateix que la funció estàndard `mconcat`, i que `dmap` fa el mateix que `foldMap`.

En altres paraules, l'exercici 2 també es pot escriure així:

```
trafficLight = foldMap lightBulb lights <> frame
```

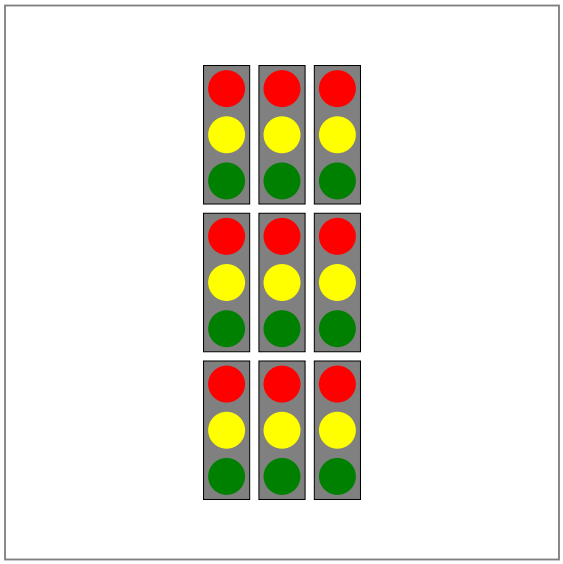


Figura 6: Resultat del sisé exercici.

Exercici 6. Es demana implementar una funció `trafficLights` que accepti una llista de punts `[(Double, Double)]` i dibuixi semàfors en cadascuna d'aquestes posicions, retornant el `Drawable` resultant.

Només cal aplicar `foldMap` parcialment:

```
trafficLights :: [(Double, Double)] -> Drawing
trafficLights = foldMap (\(x,y) -> translated x y trafficLight)
```

Amb això també podem reescriure l'exercici 3, només ens cal generar la llista de punts. Ho podem fer així:

```
points = concat (for [-8, 0, 8] \r -> (for [-3, 0, 3] \c -> (c, r)))
```

On la funció `for` és `flip fmap`. Llavors només cal pintar `trafficLights points`. El resultat final es pot veure a la fig. 6, i el codi complert es troba a la pàgina 14.

Part IV.
Codi complert

Exercici 2

```haskell
import Drawing

lights = [ (-2.5, green), (0, yellow), (2.5, red) ]
lightBulb (y, c) = colored c $ translated 0 y $ solidCircle 1

frame = rectangle 2.5 7.5 <> (colored gray $ solidRectangle 2.5 7.5)
trafficLight = dmap lightBulb lights <> frame

myDrawing :: Drawing
myDrawing = trafficLight

main :: IO ()
main = svgOf myDrawing

-- Utils

stack :: Foldable f => f Drawing -> Drawing
stack = foldr (<>) blank

dmap :: (x -> Drawing) -> [x] -> Drawing
dmap f l = stack $ f <$> l
```

Exercici 3

```haskell
import Drawing

lights = [ (-2.5, green), (0, yellow), (2.5, red) ]
lightBulb (y, c) = colored c $ translated 0 y $ solidCircle 1

frame = rectangle 2.5 7.5 <> (colored gray $ solidRectangle 2.5 7.5)
trafficLight = dmap lightBulb lights <> frame

myDrawing :: Drawing
myDrawing = col [-8, 0, 8] $ row [-3, 0, 3] $ trafficLight

main :: IO ()
main = svgOf myDrawing

-- Utils

stack :: Foldable f => f Drawing -> Drawing
stack = foldr (<>) blank

dmap :: (x -> Drawing) -> [x] -> Drawing
dmap f l = stack $ f <$> l

row xs fig = dmap (\x -> translated x 0 fig) xs
col ys fig = dmap (\y -> translated 0 y fig) ys
```

Exercici 4

```haskell
import Drawing

tree 0 = colored yellow $ solidCircle 0.2
tree n = let
    subtree = dmap (\s -> rotated (s*pi/2/5) $ tree (n-1)) [1, -1]
  in polyline [(0,0), (0,1)] <> translated 0 1 subtree

myDrawing :: Drawing
myDrawing = tree 8

main :: IO ()
main = svgOf myDrawing

-- Utils

stack :: Foldable f => f Drawing -> Drawing
stack = foldr (<>) blank

dmap :: (x -> Drawing) -> [x] -> Drawing
dmap f l = stack $ f <$> l
```

Exercici 5

```haskell
import Drawing

lights = [ (-2.5, green), (0, yellow), (2.5, red) ]
lightBulb (y, c) = colored c $ translated 0 y $ solidCircle 1

frame = rectangle 2.5 7.5 <> (colored gray $ solidRectangle 2.5 7.5)
trafficLight = dmap lightBulb lights <> frame

-- Array drawing

light :: Int -> Int -> Drawing
light r c = translated (3 * fromIntegral c - 6) (8 * fromIntegral r -
  ↪  16) trafficLight

lightRow :: Int -> Drawing
lightRow r = repeatDraw (light r) 3

myDrawing :: Drawing
myDrawing = repeatDraw lightRow 3

main :: IO ()
main = svgOf myDrawing

-- Utils

stack :: Foldable f => f Drawing -> Drawing
stack = foldr (<>) blank

dmap :: (x -> Drawing) -> [x] -> Drawing
dmap f l = stack $ f <$> l

repeatDraw :: (Int -> Drawing) -> Int -> Drawing
repeatDraw thing n = dmap thing [1..n]
```

Exercici 6

```haskell
import Drawing

lights = [ (-2.5, green), (0, yellow), (2.5, red) ]
lightBulb (y, c) = colored c $ translated 0 y $ solidCircle 1

frame = rectangle 2.5 7.5 <> (colored gray $ solidRectangle 2.5 7.5)
trafficLight = foldMap lightBulb lights <> frame

-- Array drawing

trafficLights :: [(Double, Double)] -> Drawing
trafficLights = foldMap (\(x,y) -> translated x y trafficLight)

points = concat (for [-8, 0, 8] \r -> (for [-3, 0, 3] \c -> (c, r)))

myDrawing :: Drawing
myDrawing = trafficLights points

main :: IO ()
main = svgOf myDrawing

-- Utils

for :: (Functor f) => f a -> (a -> b) -> f b
for = flip fmap
```