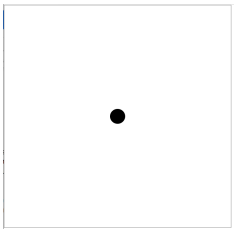
1a PART: Haskell bàsic

Exercici 1.- Editeu un fitxer amb aquest programa i executeu-lo amb run-main passant-li com argument el nom del fitxer. El resultat és un contingut SVG que podreu visualitzar amb el navegador.

Codi:

```
import Drawing

myDrawing :: Drawing
myDrawing = solidCircle 1

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

Sortida:



Per compilar i executar les pràctiques utilitzarem l'scrip WEBprofe/usr/bin/run-main:

$ run-main fitxer_codi.hs > /public_html/practica1/sortida1.svg

El fitxer sortida.svg es pot visualitzar amb qualsevol navegador que suporti SVG introduint la URL http://soft0.upc.edu/~ldatusrXX/practica1/sortida1.svg

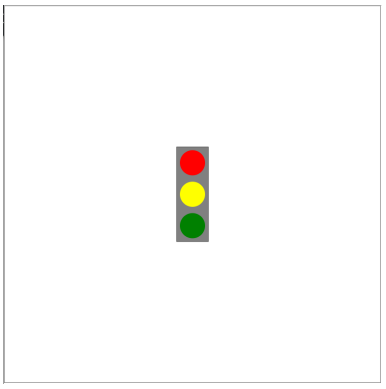
Definició de funcions

Exercici 2.- Realitzeu un programa que generi el dibuix d'un semàfor centrat a l'origen amb 3 llums de colors vermell, groc i verd. Definiu una funció lightBulb que dibuixi un llum, amb 2 paràmetres corresponents al color i la posició en l'eix Y; i realitzeu el programa usant aquesta funció.

Codi:

```
import Drawing

trafficLight :: Drawing
lightBulb:: Color-> Double-> Drawing
lightBulb color positionY = colored color (translated 0
                                    (positionY)(solidCircle 1))
trafficLight = rectangle 2.5 7.5 <> colored gray
                (solidRectangle 2.5 7.5) <> lightBulb red 2.5 <>
                lightBulb yellow 0 <> lightBulb green (-2.5)

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

Sortida:



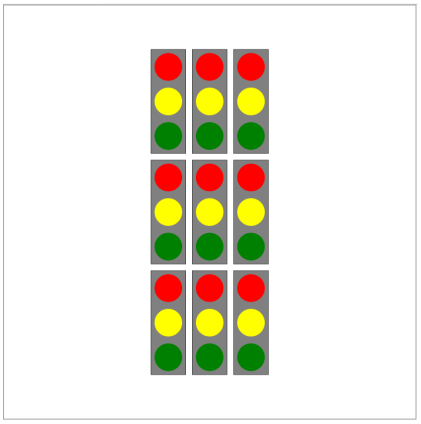
Funcions recursives

Exercici 3.- Realitzeu un programa que dibuixi una array de 3 x 3 semàfors.

Codi:

```
import Drawing

trafficLight :: Drawing
lightBulb :: Color -> Double-> Drawing
lightBulb color positionY = colored color (translated 0
                                    (positionY)(solidCircle 1))
trafficLight = translated 0 0 (rectangle 2.5 7.5) <> translated
          0 0 (colored gray (solidRectangle 2.5 7.5)) <>
          lightBulb red 2.5 <> lightBulb yellow 0 <>
          lightBulb green (-2.5)

nLights :: Int -> Drawing
nLights 0 = blank
nLights n = trafficLight <> translated 3 0 (nLights (n-1))

myDrawing :: Drawing
myDrawing = translated (-3) 0 (nLights 3) <> translated (-3) 8
          (nLights 3) <> translated (-3) (-8) (nLights 3)

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

Sortida:



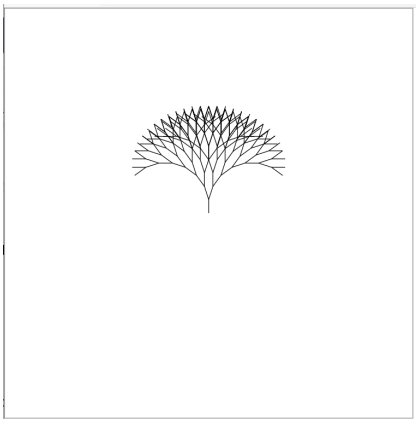
Exercici 4

a.- Realitzeu un programa que dibuixi l'arbre de la figura de 8 nivells.

Codi:

```
import Drawing

branch :: Drawing
branch = polyline [(0,0), (0,1)]

tree :: Int -> Drawing
tree 0 = branch
tree n = branch <> translated 0 1 (rotated (pi/10) (tree (n-1)))
      <> translated 0 1 (rotated (-pi/10) (tree (n-1)))

myDrawing = tree 7

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

Sortida:

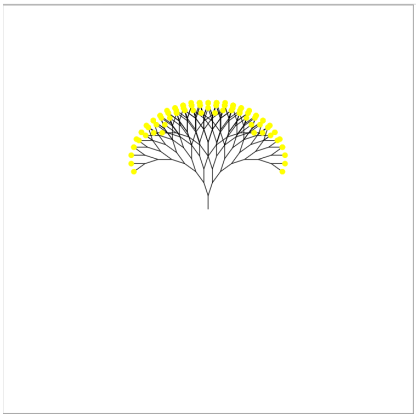


b.- Modifiqueu el programa de manera que dibuixi petites flors (simples cercles grocs) en les branques (veieu el dibuix resultant).

Codi:

```
import Drawing

branch :: Drawing
branch = polyline [(0,0), (0,1)]

tree :: Int -> Drawing
tree 0 = branch
tree n = branch <> translated 0 1 (rotated (pi/10) (tree
        (n-1))) <> translated 0 1 (rotated (-pi/10) (tree (n-1)))

flower :: Drawing
flower = colored yellow (solidCircle 0.20)

flowers :: Int -> Drawing
flowers 0 = flower
flowers n = translated 0 1 (rotated (pi/10) (flowers(n-1)))
            <> translated 0 1 (rotated (-pi/10) (flowers(n-1)))
myDrawing = tree 7 <> flowers 8

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

Sortida:



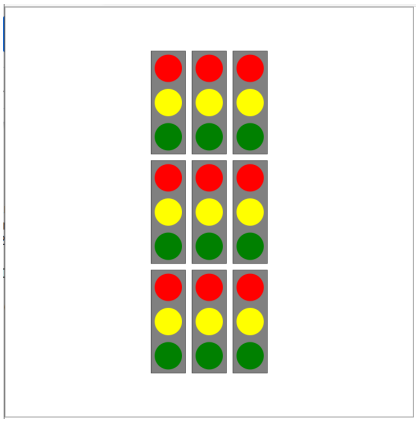
Funcions d'ordre superior

Exercici 5.- Completeu la funció repeatDraw i executeu el programa anterior.

Codi:

```
import Drawing

trafficLight :: Drawing
lightBulb :: Color -> Double -> Drawing
lightBulb color posY = colored color (translated (0) (posY)
                                (solidCircle 1))
trafficLight = (rectangle 2.5 7.5) <> colored gray
                (solidRectangle 2.5 7.5) <> lightBulb red 2.5 <>
                lightBulb yellow 0 <> lightBulb green (-2.5)

repeatDraw :: (Int -> Drawing) -> Int -> Drawing
repeatDraw thing 0 = blank
repeatDraw thing n = repeatDraw thing (n-1) <> thing n

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

light :: Int -> Int -> Drawing
```

Sortida:



```
light r c = translated (3 * fromIntegral c - 6) (8 * fromIntegral r
            - 16) trafficLight

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

Exercici 6.- Resoleu el següent problema:

trafficLights :: [(Double, Double)] -> Drawing

(Donada una llista de n punts dibuixar n semàfors situats en les corresponents posicions, usant les funcions foldMap i trafficLight).

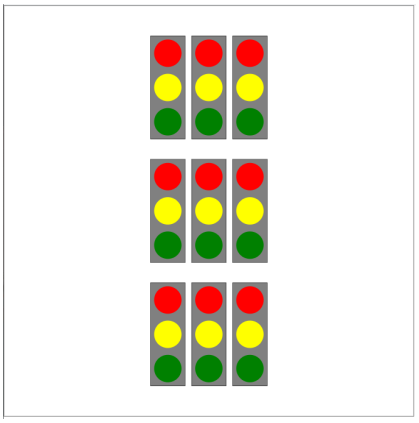
Observació: El tipus llista pertany a la classe Foldable i el tipus Drawing pertany a la classe Monoid. En el monoid Drawing l'operació (<>) és la composició de figures i l'element neutre és blank. Aleshores el mateix problema (del exercici 3) es pot resoldre trivialment aplicant la funció trafficLights a la llista dels 9 punts corresponents.

La funció repeatDraw es pot implementar amb una ja donada per Haskell (foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m)

Codi:

```
import Drawing

trafficLight :: Drawing
lightBulb :: Color -> Double -> Drawing
lightBulb color posY = colored color (translated 0 (posY)
                       (solidCircle 1))
trafficLight = (rectangle 2.5 7.5) <> colored gray
               (solidRectangle 2.5 7.5) <> lightBulb red 2.5 <>
               lightBulb yellow 0 <> lightBulb green (-2.5)

lights :: (Double, Double) -> Drawing
lights (x,y) = translated (3 * x) y trafficLight

trafficLights :: [(Double, Double)] -> Drawing
trafficLights f = foldMap lights f

myDrawing :: Drawing
myDrawing = trafficLights [(-1,0),(0,0),(1,0),
                           (-1,9),(0,9),(1,9),
                           (-1,-9),(0,-9),(1,-9)]
main :: IO()
main= svgOf myDrawing
```

Sortida:



2a PART: Interacció i modificació d'estat

Per començar descarreguem el fitxer prog-2-fun.zip a la carpeta de practiques i el descomprimim.

```
~/ldatusrXX/practiques$ curl http://soft0.upc.edu/dat/practica1/prog-fun-2.zip > prog-fun-2.zip
~/ldatusrXX/practiques$ unzip prog-fun-2.zip
```

La biblioteca Drawing usada en aquesta pràctica ofereix la funció activityOf amb el següent tipus:

```
activityOf :: (Show state, Read state)
           => Double -> Double
           -> state
           -> (Event -> state -> state)
           -> (state -> Drawing)
           -> IO ()
```

Aquesta és una declaració polimòrfica, on state és una variable de tipus que fa referència a un tipus qualsevol corresponent a l'estat de l'aplicació. Hi ha el requeriment que aquest tipus pertanyi a les classes Show i Read.
La funció activityOf obté una aplicació CGI que mantindrà l'estat i processarà els esdeveniments generats des de la pàgina HTML que interacciona amb l'usuari.

Per probar l'aplicació:

1. Completar el mòdul.
2. Compilar/instalar el CGI (src/exemple.hs).
   ~/ldatusrXX/practiques/prog-fun-2$ bin/make-cgi src/exemple.hs
3. Visitar el CGI amb la URL http://soft0.upc.edu/~ldatusrXX/practica1/exemple.cgi i comprobar el funcionament.

Realització del joc

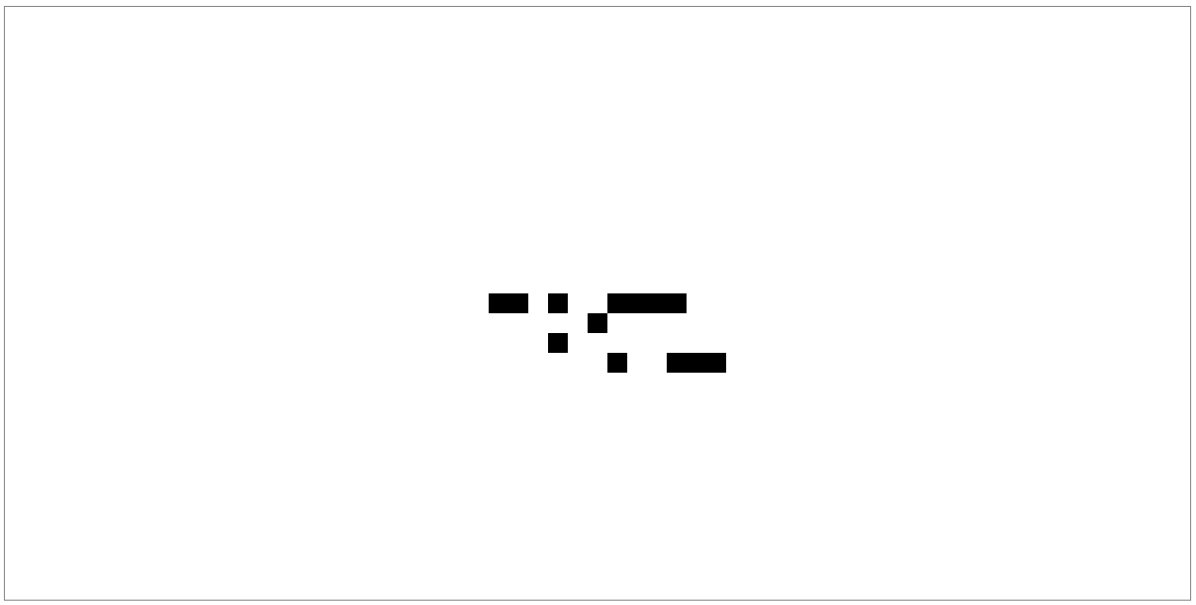
La pràctica consisteix en la realització de 5 pasos. En cadascun dels passos anirem introduint noves funcionalitats fins arribar a la versió final. El codi de cada pas estarà format pel mòdul principal Main (fitxers src/life-1.hs... src/life-4.hs) i els mòduls Life.Board i Life.Draw. Els mòduls Life.Board i Life.Draw són comuns a tots els passos i contenen la definició del taulell del joc i de la funció que el dibuixa respectivament. La definició del taulell del joc i de les seves funcions es dóna ja feta en el mòdul Life.Board (veieu en el projecte el fitxer src/Life/Board.hs).

Pas 1.- Life.Draw

Per començar creem el taulell del joc.

Codi a completar (fitxer src/Life/Draw.hs):

```
drawBoard board = foldMap (drawCell board) (liveCells board)

drawCell :: Board -> Pos -> Drawing
drawCell board (x,y) = translated (fromIntegral x) (fromIntegral y) (solidRectangle 1 1)
```



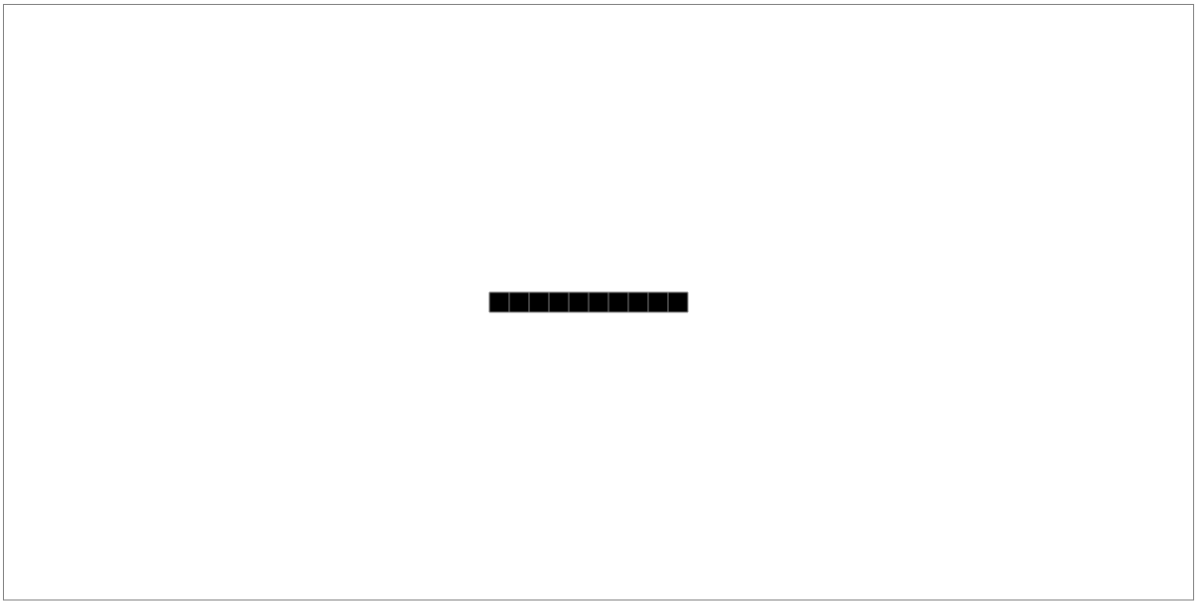
En el nou fitxer Main (life-1.cgi) podem apreta celes amb el ratolí. Si estan seleccionades (en negre) estan “vives”. Podem tornar-les a apretar perquè canviin a blanc. Per últim, si teclejem la tecla 'N' passem a la següent generació.

Pas 2.- Main

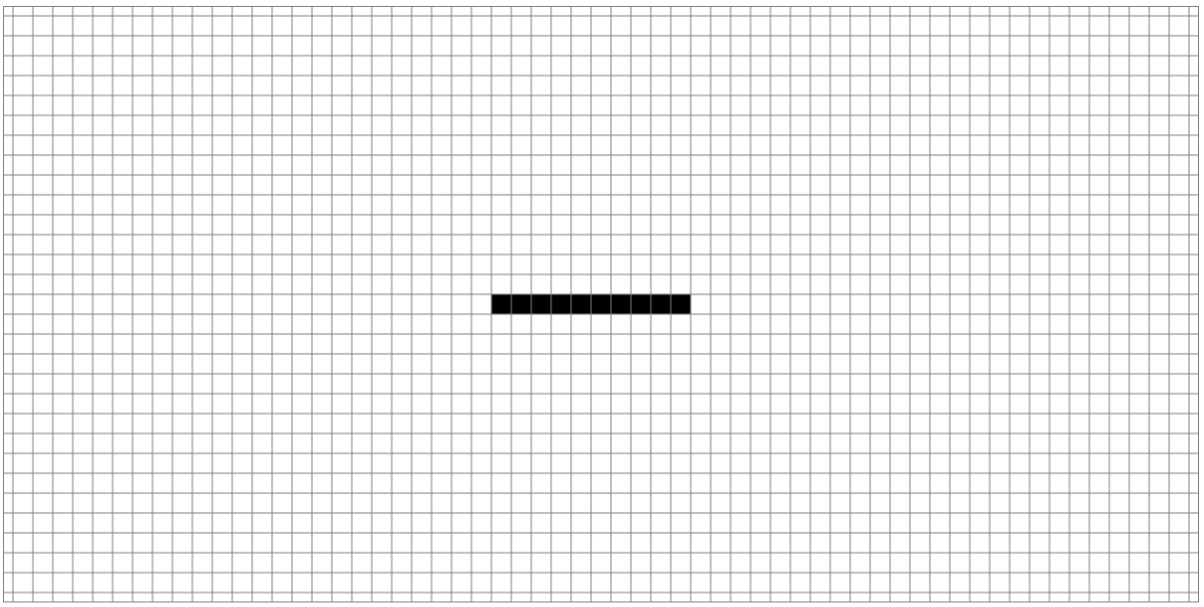
En el segon pas afegim un control per mostrar i ocultar una graella.

Codi a completar (fitxer src/life-2.hs):

```
handleEvent (KeyDown "G") game =
  case (gmGridMode game) of
    NoGrid -> setGmGridMode LivesGrid game
    LivesGrid -> setGmGridMode ViewGrid game
    ViewGrid -> setGmGridMode NoGrid game

handleEvent _ game =
  game

draw game =
  let minCorner = (round((viewWidth/2)*(-1)), round((viewHeight/2)*(-1)))
    maxCorner = (round(viewWidth/2), round(viewHeight/2))
  in
    case (gmGridMode game) of
      NoGrid -> drawBoard (gmBoard game)
      LivesGrid -> drawBoard (gmBoard game) <> drawGrid (minLiveCell (gmBoard game))
                    (maxLiveCell (gmBoard game))
      ViewGrid -> drawBoard (gmBoard game) <> drawGrid (minCorner) (maxCorner)
```



Time: 58.034
Pending events: 0
Last event: KeyUp "G"



Time: 4.917
Pending events: 0
Last event: KeyUp "G"

Amb la tecla “G” podem canviar el ViewGridMode.

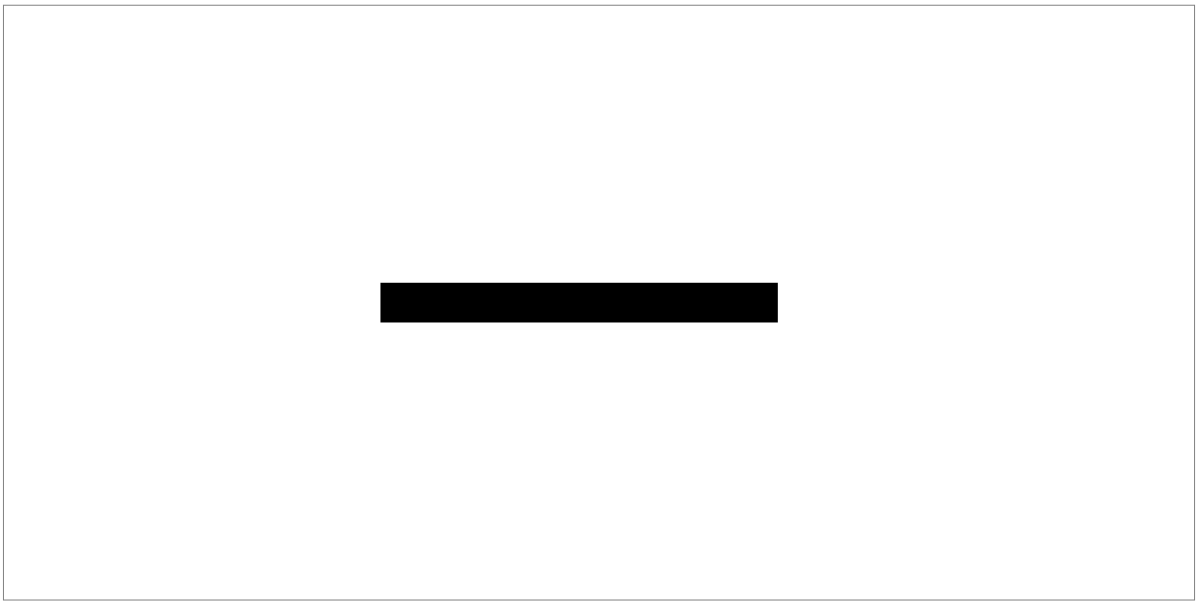
Pas 3.- Main

Ara introduirem controls per fer zoom i desplaçaments en el taulell. Per tant, hem de definir nous camps gmZoom i gmShift en l’estat del joc Game.

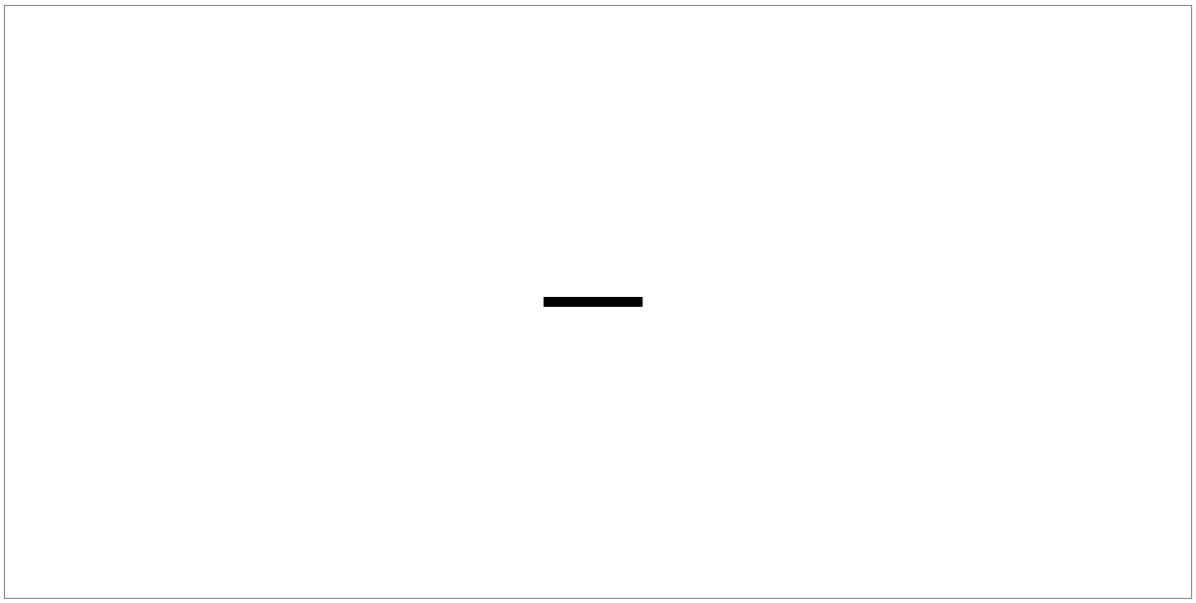
Codi a completar (fitxer src/life-3.hs):

```
handleEvent (KeyDown "G") game =
  case (gmGridMode game) of
    NoGrid -> setGmGridMode LivesGrid game
```

```
      LivesGrid -> setGmGridMode ViewGrid game
      ViewGrid -> setGmGridMode NoGrid game

handleEvent (KeyDown "I") game =
  if gmZoom game < 2 then setGmZoom (gmZoom game *2) game
  else game

handleEvent (KeyDown "O") game =
  if gmZoom game < 5 then setGmZoom (gmZoom game /2) game
  else game

handleEvent (KeyDown "ARROWUP") game =
  setGmShift (gmShift game ^-^ (1/gmZoom game)*^(0,5)) game

handleEvent (KeyDown "ARROWDOWN") game =
  setGmShift (gmShift game ^-^ (1/gmZoom game)*^(0,-5)) game

handleEvent (KeyDown "ARROWRIGHT") game =
  setGmShift (gmShift game ^-^ (1/gmZoom game)*^(5,0)) game

handleEvent (KeyDown "ARROWLEFT") game =
  setGmShift (gmShift game ^-^ (1/gmZoom game)*^(-5,0)) game

handleEvent _ game =
  game

pointToPos :: Point -> Game -> Pos
pointToPos p game =
  let (gx, gy) = (1/gmZoom game)*^p ^-^ gmShift game
  in (round gx, round gy)

draw game =
  let (x,y) = gmShift game
      minCorner = (round((viewWidth/2)*(-1)), round((viewHeight/2)*(-1)))
      maxCorner = (round(viewWidth/2), round(viewHeight/2))
  in scaled (gmZoom game) (gmZoom game) $ translated x y (drawBoard (gmBoard game)) <>
      case (gmGridMode game) of
        NoGrid -> blank
        LivesGrid -> drawGrid (minLiveCell (gmBoard game)) (maxLiveCell (gmBoard game))
        ViewGrid -> drawGrid (minCorner) (maxCorner)
```



Time: 9.14
Pending events: 0
Last event: KeyUp "I"



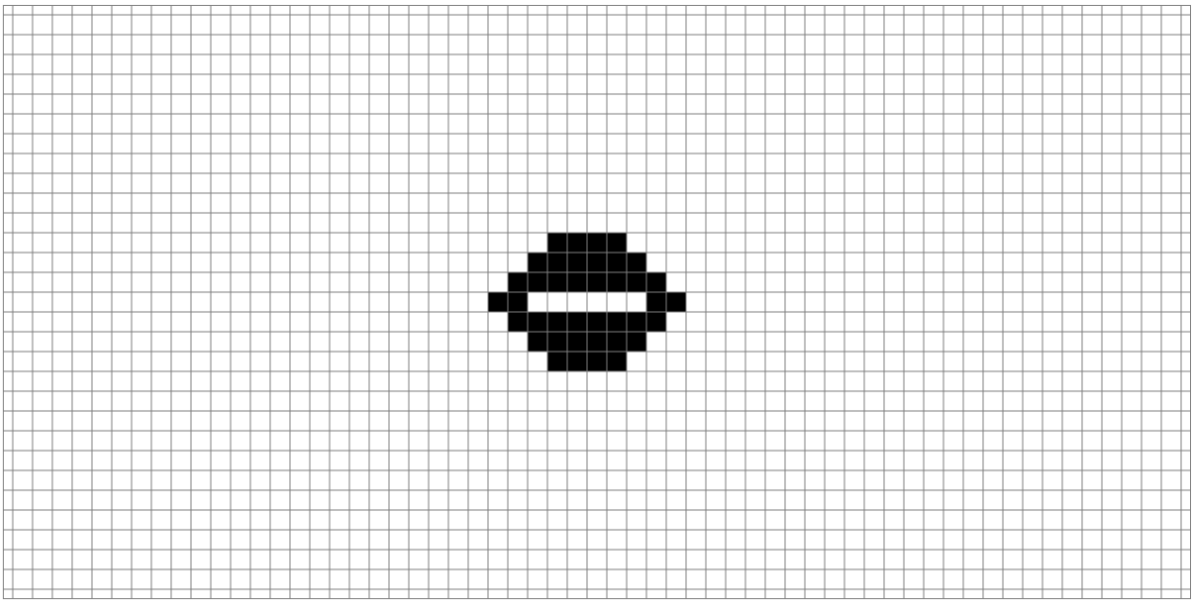
Time: 62.44
Pending events: 0
Last event: KeyUp "O"

Pas 4.- Main

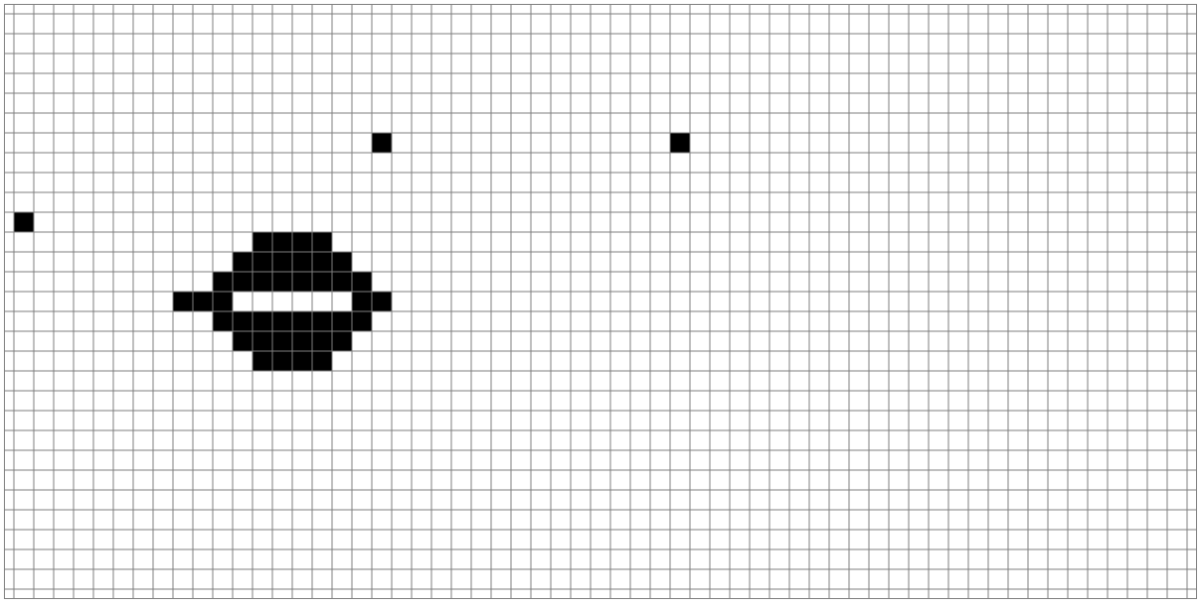
En el quart afegim el temps, evolució automàtica de generacions, i controls per poder canviar el mode i velocitat d'aquesta.

Codi a completar (fitxer src/life-4.hs):

```
--Afegim al codi anterior (src/life-3.hs)
handleEvent (KeyDown " ") game =
  setGmPaused (not $ gmPaused game) game

handleEvent (KeyDown "+") game =
  if gmInterval game/2 >= 0.125 then setGmInterval (gmInterval game /2) game
  else game

handleEvent (KeyDown "-") game =
  setGmInterval (gmInterval game*2) game

handleEvent (TimePassing dt) game =
  if gmPaused game == False then
    (if (gmElapsedTime game + dt >= gmInterval game)
      then setGmElapsedTime 0 (setGmBoard (nextGeneration $ gmBoard game) game)
    else setGmElapsedTime (gmElapsedTime game + dt) game)
  else game

handleEvent (MouseDown (x,y)) game =
  let pos = pointToPos (x,y) game
    board = gmBoard game
  in setGmBoard (setCell (not $ cellIsLive pos board) pos board) game
```



Teclejem l'espai i automàticament les cel·les evolucionen a la següent generació.
Amb el "+" i "-" podem canviar la velocitat amb la que aquestes ho fan.



Seguim tenint les mateixes funcionalitats anteriors com moure i activar més cel les.

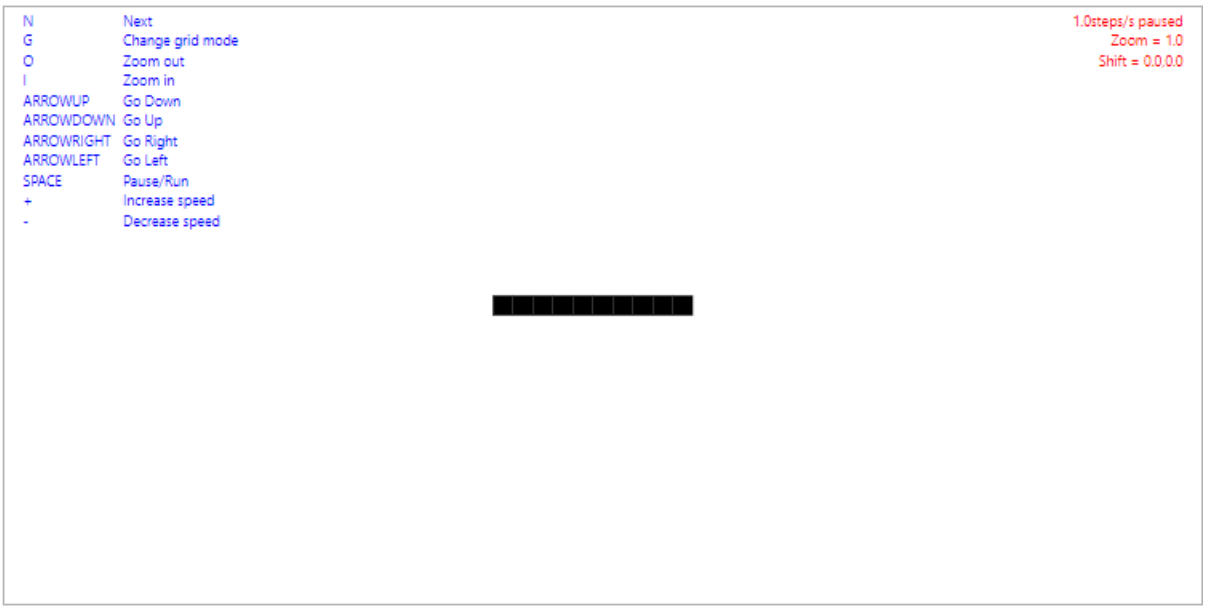
Pas 5.- Main

Per acabar modificarem la funció de dibuix per tal que mostri en el mode pausa una ajuda del joc i l'estat en tot moment.

Codi a completar (fitxer src/life-4.hs):

```
--Afegim al codi anterior (src/life-4.hs)
keys = ["N", "G", "O", "I", "ARROWUP", "ARROWDOWN", "ARROWRIGHT", "ARROWLEFT", "SPACE",
        "+", "-"]
values = ["Next", "Change grid mode", "Zoom out", "Zoom in", "Go Down", "Go Up", "Go Right", "Go Left",
        "Pause/Run", "Increase speed", "Decrease speed"]
```

```
aux :: Game -> [String]
aux g = [show (1/gmInterval g) ++ "steps/s" ++ (if gmPaused g) then "paused"
                                else ""),
     "Zoom = " ++ show (gmZoom g),
     "Shift = " ++ show (fst(gmShift g)) ++","++ show (snd(gmShift g))
]

line :: [String] -> String -> Drawing
line [] a = blank
line (l:ls) a = translated 0 (-1) ((atext a l) <> (line ls a))

draw game =
  let (x,y) = gmShift game
    minCorner = (round((viewWidth/2)*(-1)), round((viewHeight/2)*(-1)))
    maxCorner = (round(viewWidth/2), round(viewHeight/2))
  in scaled (gmZoom game) (gmZoom game) $ translated x y (drawBoard (gmBoard game)) <>
    case (gmGridMode game) of
      NoGrid -> blank
      LivesGrid -> drawGrid (minLiveCell (gmBoard game)) (maxLiveCell (gmBoard game))
      ViewGrid -> drawGrid (minCorner) (maxCorner)
    <>
    (if gmPaused game then
      (translated (-viewWidth/2 + 1) (viewHeight/2) (colored blue (line keys startAnchor)) <>
      translated (-viewWidth/2 + 6) (viewHeight/2) (colored blue (line values startAnchor)))
    else blank)
    <>
    translated (viewWidth/2 - 1) (viewHeight/2) (colored red (aux game) endAnchor)))
```



Amb l'espai mostrem el menú d'ajuda, ja que es veu en el mode "pausa".