John, l'artista

Pràctica de Programació Funcional

Paradigmes i Llenguatges de Programació

Curs 22/23

Iván Navarrete Rojo, Codi UdG: u1972808
Óscar Molina Muñoz, Codi UdG: u1972843

Índex

Índex 1
1. Descripció del problema 2
2. Codi comentat 2
3. Particularitats del codi 11
4. Implementacions 13
5. Referències de bibliografia 17

1. Descripció del problema

Hem creat en Haskell un programa per dibuixar per pantalla mitjançants comandes, hem optimitzat l'entrada de comandes, hem implementat el canvi de color del llapis i la possibilitat de fer branques. Per tant, hem pogut generar patrons en forma de fractals tal i com es demanaven.

2. Codi comentat

Artist.hs

```
module Artist where

import UdGraphic
import Test.QuickCheck
import Debug.Trace

-- Problema 1

separa :: Comanda -> [Comanda]
separa (com1 :#: com2) = separa com1 ++ separa com2 -- Concatena
separa Para = [] -- Cas base
separa com = [com] -- Cas base

-- Problema 2

ajunta :: [Comanda] -> Comanda
ajunta [] = Para -- Cas base
ajunta [com] = com :#: Para -- Cas base
ajunta (com : l) = com :#: ajunta l

-- Problema 3

-- Si les llistes que retorna "separa" son iguals llavors totes
dues comandes són iguals
prop_equivalent :: Comanda -> Comanda -> Bool
prop_equivalent com1 com2 = separa com1 == separa com2

-- Es mira si l'ajunta i separa funciona correctament
prop_split_join :: Comanda -> Bool
prop_split_join c = prop_equivalent (ajunta (separa c)) c

conteComposta :: [Comanda] -> Bool
conteComposta [] = False
conteComposta ((x :#: y):xs) = True
conteComposta (x:xs) = conteComposta xs
```

```
-- Es mira si el separa funciona correctament quitant els "Para" i
les funcions compostes amb ":#:"
prop_split :: Comanda -> Bool
prop_split c = let llista = separa c
               in not(elem Para llista) && not(conteComposta
llista)

-- Problema 4

-- Crea la copia de la comanda amb una funció recursiva
copia :: Int -> Comanda -> Comanda
copia 1 com = com
copia m com = let n = m-1
              in com :#: copia n com

-- Problema 5

-- Per definir el pentagon s'utilitza el predicat "copia"
pentagon :: Distancia -> Comanda
pentagon d = copia 5 (Avança d :#: Gira 72)

-- Problema 6

-- De la mateixa forma que amb el pentagon, es defineix amb el
predicat "copia"
poligon :: Distancia -> Int -> Angle -> Comanda
poligon d n a = copia n (Avança d :#: Gira a)

prop_poligon_pentagon :: Bool
prop_poligon_pentagon = prop_equivalent (poligon 1 5 72) (pentagon
1)

-- Problema 7

-- Es defineix l'"espiral" recursivament de forma que cada
iteració gira "a" graus
espiral :: Distancia -> Int -> Distancia -> Angle -> Comanda
espiral d 1 _ a = (Avança d :#: Gira a)
espiral d m pas a = let n=m-1
                        sd=d+pas
                    in (Avança d :#: Gira a :#: espiral sd n pas
a)

-- Problema 9

-- Fa el mateix que l'ajunta original, amb la diferència que no
s'afegeix l'últim "Para", i a més s'ajunten els "Gira" junts i
"Avança" junts en un de sol
```

```
ajuntaOpt :: [Comanda] -> Comanda
ajuntaOpt [] = Para -- Cas base
ajuntaOpt [com] = if(com == Para || com == Avança 0 || com == Gira
0) -- Cas base
                  then
                      Para
                  else
                      com
ajuntaOpt (Avança d:Avança e:l) = ajuntaOpt (Avança (d+e):l)
ajuntaOpt (Gira a:Gira b:l) = ajuntaOpt (Gira (a+b):l)
ajuntaOpt (com : l) = if(com == Para || com == Avança 0 || com ==
Gira 0)
                      then
                          ajuntaOpt l
                      else
                          com :#: ajuntaOpt l

-- Fa recursió fins que l'"optimitza" no té cap canvi nou, i
llavors és la comanda optimitzada
optimitza :: Comanda -> Comanda
optimitza c = if(c/=copt) then
                optimitza(copt)
              else
                copt
            where
                copt = ajuntaOpt (separa c)

-- Gramàtica de fractals
-- Problema 10

triangle :: Int -> Comanda
triangle n = Gira 90 :#: fTriangle n

fTriangle :: Int -> Comanda
fTriangle 0 = Avança 30
fTriangle n = fTriangle (n-1) :#: Gira 90 :#: fTriangle (n-1) :#:
Gira (-90) :#: fTriangle (n-1) :#: Gira (-90) :#: fTriangle (n-1)
:#: Gira 90 :#: fTriangle (n-1)

-- Problema 11

fulla :: Int -> Comanda
fulla n = CanviaColor blau :#: fFulla n

fFulla :: Int -> Comanda
fFulla 0 = CanviaColor vermell :#: Avança 30
```

```
fFulla n = gFulla (n-1) :#: Branca (Gira (-45) :#: fFulla (n-1))
:#: Branca (Gira 45 :#: fFulla (n-1)) :#: Branca(gFulla (n-1) :#:
fFulla (n-1))

gFulla :: Int -> Comanda
gFulla 0 = Avança 30
gFulla n = gFulla (n-1) :#: gFulla (n-1)

-- Problema 12

hilbert :: Int -> Comanda
hilbert n = lHilbert n

lHilbert :: Int -> Comanda
lHilbert 0 = Para
lHilbert n = Gira 90 :#: rHilbert (n-1) :#: Avança 30 :#: Gira
(-90) :#: lHilbert (n-1) :#: Avança 30 :#: lHilbert (n-1) :#: Gira
(-90) :#: Avança 30 :#: rHilbert (n-1) :#: Gira 90

rHilbert :: Int -> Comanda
rHilbert 0 = Para
rHilbert n = Gira (-90) :#: lHilbert (n-1) :#: Avança 30 :#: Gira
90 :#: rHilbert (n-1) :#: Avança 30 :#: rHilbert (n-1) :#: Gira 90
:#: Avança 30 :#: lHilbert (n-1) :#: Gira (-90)

-- Problema 13

fletxa :: Int -> Comanda
fletxa n = fFletxa n

fFletxa :: Int -> Comanda
fFletxa 0 = Avança 30
fFletxa n = gFletxa (n-1) :#: Gira 60 :#: fFletxa (n-1) :#: Gira
60 :#: gFletxa (n-1)

gFletxa :: Int -> Comanda
gFletxa 0 = Avança 30
gFletxa n = fFletxa (n-1) :#: Gira (-60) :#: gFletxa (n-1) :#:
Gira (-60) :#: fFletxa (n-1)

-- Problema 14

branca :: Int -> Comanda
branca n = CanviaColor blau :#: gBranca n

gBranca :: Int -> Comanda
gBranca 0 = CanviaColor vermell :#: Avança 30
```

```
gBranca n = fBranca (n-1) :#: Gira (-22.5) :#:
Branca(Branca(gBranca (n-1)) :#: Gira 22.5 :#: gBranca (n-1)) :#:
Gira 22.5 :#: fBranca (n-1) :#: Branca(Gira 22.5 :#: fBranca (n-1)
:#: gBranca (n-1)) :#: Gira (-22.5) :#: gBranca (n-1)

fBranca :: Int -> Comanda
fBranca 0 = Avança 30
fBranca n = fBranca (n-1) :#: fBranca (n-1)
```

UdGraphic.hs

```
module UdGraphic (
    Comanda(..),
    Distancia,
    Angle,
    Llapis(..), blau, vermell,
    display,
    execute
    )
    where

import qualified Graphics.Rendering.OpenGL as GL
import Graphics.UI.GLUT hiding (Angle)
import Data.IORef
import Data.List
import Control.Monad( liftM, liftM2, liftM3 )
import System.Random
import Test.QuickCheck

infixr 5 :#:

-- Punts

data Pnt = Pnt Float Float
  deriving (Eq,Ord,Show)

instance Num Pnt where
  Pnt x y + Pnt x' y'  =  Pnt (x+x') (y+y')
  Pnt x y - Pnt x' y'  =  Pnt (x-x') (y-y')
  Pnt x y * Pnt x' y'  =  Pnt (x*x') (y*y')
  fromInteger          =  scalar . fromInteger
  abs (Pnt x y)        =  Pnt (abs x) (abs y)
  signum (Pnt x y)     =  Pnt (signum x) (signum y)

instance Fractional Pnt where
  Pnt x y / Pnt x' y'  =  Pnt (x/x') (y/y')
```

```
  fromRational         =  scalar . fromRational

scalar :: Float -> Pnt
scalar x  =  Pnt x x

scalarMin :: Pnt -> Pnt
scalarMin (Pnt x y)  =  scalar (x `min` y)

scalarMax :: Pnt -> Pnt
scalarMax (Pnt x y)  =  scalar (x `max` y)

dimensions :: Pnt -> (Int,Int)
dimensions (Pnt x y)  =  (ceiling x, ceiling y)

lub :: Pnt -> Pnt -> Pnt
Pnt x y `lub` Pnt x' y'  =  Pnt (x `max` x') (y `max` y')

glb :: Pnt -> Pnt -> Pnt
Pnt x y `glb` Pnt x' y'  =  Pnt (x `min` x') (y `min` y')

pointToSize :: Pnt -> Size
pointToSize (Pnt x y) = Size (ceiling x) (ceiling y)

sizeToPoint :: Size -> Pnt
sizeToPoint (Size x y) = Pnt (fromIntegral x) (fromIntegral y)

-- Colors

data Llapis = Color' GL.GLfloat GL.GLfloat GL.GLfloat
            | Transparent
            deriving (Eq, Ord, Show)

pencilToRGB :: Llapis -> GL.Color3 GL.GLfloat
pencilToRGB (Color' r g b)  =  GL.Color3 r g b
pencilToRGB Transparent  =  error "pencilToRGB: transparent"

blanc, negre, vermell, verd, blau :: Llapis
blanc   = Color' 1.0 1.0 1.0
negre   = Color' 0.0 0.0 0.0
vermell = Color' 1.0 0.0 0.0
verd    = Color' 0.0 1.0 0.0
blau    = Color' 0.0 0.0 1.0

-- Lines

data Ln = Ln Llapis Pnt Pnt
  deriving (Eq,Ord,Show)
```

```
-- Window parameters

theCanvas :: Pnt
theCanvas  =  Pnt 800 800

theBGcolor :: GL.Color3 GL.GLfloat
theBGcolor = pencilToRGB blanc



-- Main drawing and window functions

display :: Comanda -> IO ()
display c = do
  initialDisplayMode $= [DoubleBuffered]
  initialWindowSize  $= pointToSize theCanvas
  getArgsAndInitialize
  w <- createWindow "pencilcil Graphics"
  displayCallback $= draw c
  reshapeCallback $= Just (\x -> (viewport $= (Position 0 0, x)))
  --actionOnWindowClose $= ContinueExectuion
  draw c
  mainLoop

draw :: Comanda -> IO ()
draw c = do clear [ColorBuffer]
            loadIdentity
            background
            toGraphic $ rescale $ execute c
            swapBuffers

toGraphic :: [Ln] -> IO ()
toGraphic lines  = sequence_ (map f lines)
  where
  f (Ln pencil startP endP)  =
    GL.color (pencilToRGB pencil) >>
    GL.renderPrimitive GL.LineStrip (toVertex startP >> toVertex
endP)

background :: IO ()
background = do GL.color theBGcolor
                GL.renderPrimitive GL.Polygon $ mapM_ GL.vertex
                      [GL.Vertex3 (-1) (-1) 0,
                       GL.Vertex3   1  (-1) 0,
                       GL.Vertex3   1    1  0,
                       GL.Vertex3 (-1)   1 (0::GL.GLfloat) ]
```

```
toVertex (Pnt x y)  =  GL.vertex $ GL.Vertex3
 (realToFrac x) (realToFrac y) (0::GL.GLfloat)



-- Definició de les comandes per moure el llapis

type Angle     = Float
type Distancia = Float
data Comanda   = Avança Distancia
               | Gira Angle
               | Comanda :#: Comanda
               | Para
               | CanviaColor Llapis
               | Branca Comanda
                deriving (Eq,Show)


-- Problema 8
-- Pas de comandes a lines a pintar per GL graphics

-- Tipus "EstatLlapis" auxiliar per determinar l'estat del llapis
en cada comanda
data EstatLlapis = EstatLlapis Llapis Angle Pnt

-- Predicat auxiliar "polar" que ens calcula la desviació amb
l'angle
polar :: Angle -> Pnt
polar a = Pnt (cos rad) (sin rad)
    where
        rad = a * pi / 180

-- Predicat auxiliar "executeLn" que ens retorna les línies i es
va actualitzant l'estat del llapis
executeLn :: Comanda -> EstatLlapis -> ([Ln], EstatLlapis)
executeLn (Avança d) (EstatLlapis color angle inici) = (if color
== Transparent -- Si el color del llapis és "Transparent" llavors
no es dibuixa res
                                        then []
                                        else [Ln color inici
final], EstatLlapis color angle final) -- Altrament, traça la
línia amb totes les dades proporcionades
                                    where
                                        final = inici + scalar d
* polar angle -- Càlcul del punt final amb la desviació
corresponent (si es que en té)
```

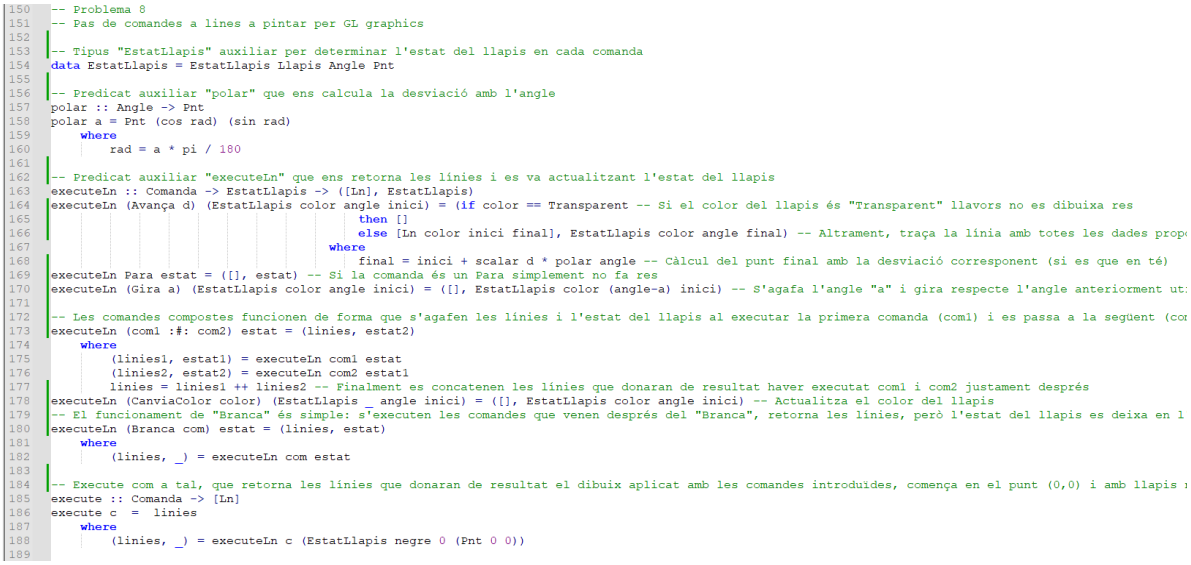
```
executeLn Para estat = ([], estat) -- Si la comanda és un Para
simplement no fa res
executeLn (Gira a) (EstatLlapis color angle inici) = ([],
EstatLlapis color (angle-a) inici) -- S'agafa l'angle "a" i gira
respecte l'angle anteriorment utilitzat

-- Les comandes compostes funcionen de forma que s'agafen les
línies i l'estat del llapis al executar la primera comanda (com1)
i es passa a la següent (com2)
executeLn (com1 :#: com2) estat = (linies, estat2)
    where
        (linies1, estat1) = executeLn com1 estat
        (linies2, estat2) = executeLn com2 estat1
        linies = linies1 ++ linies2 -- Finalment es concatenen les
línies que donaran de resultat haver executat com1 i com2
justament després
executeLn (CanviaColor color) (EstatLlapis _ angle inici) = ([],
EstatLlapis color angle inici) -- Actualitza el color del llapis
-- El funcionament de "Branca" és simple: s'executen les comandes
que venen després del "Branca", retorna les línies, però l'estat
del llapis es deixa en l'estat en que es va començar la "Branca"
executeLn (Branca com) estat = (linies, estat)
    where
        (linies, _) = executeLn com estat

-- Execute com a tal, que retorna les línies que donaran de
resultat el dibuix aplicat amb les comandes introduïdes, comença
en el punt (0,0) i amb llapis negre
execute :: Comanda -> [Ln]
execute c  =  linies
    where
        (linies, _) = executeLn c (EstatLlapis negre 0 (Pnt 0 0))

-- Rescales all points in a list of lines
--  from an arbitrary scale
--  to (-1.-1) - (1.1)

rescale :: [Ln] -> [Ln]
rescale lines | points == [] = []
              | otherwise    = map f lines
  where
  f (Ln pencil p q)  =  Ln pencil (g p) (g q)
  g p             =  swap ((p - p0) / s)
  points          =  [ r | Ln pencil p q <- lines, r <- [p, q] ]
  hi              =  foldr1 lub points
  lo              =  foldr1 glb points
  s               =  scalarMax (hi - lo) * scalar (0.55)
  p0              =  (hi + lo) * scalar (0.5)
```

```
  swap (Pnt x y)  =  Pnt y x


-- Generators for QuickCheck

instance Arbitrary Llapis where
    arbitrary  =  sized pencil
        where
          pencil n  =  elements
[negre,vermell,verd,blau,blanc,Transparent]


instance Arbitrary Comanda where
    arbitrary  =  sized cmd
        where
          cmd n  |  n <= 0     =  oneof [liftM (Avança . abs)
arbitrary,
                                          liftM Gira arbitrary ]
                 |  otherwise  =  liftM2 (:#:) (cmd (n `div` 2))
(cmd (n `div`2))
```

3. Particularitats del codi

Com a predicats més rellevants, analitzarem l'*execute*, *optimitza* i *fulla*.

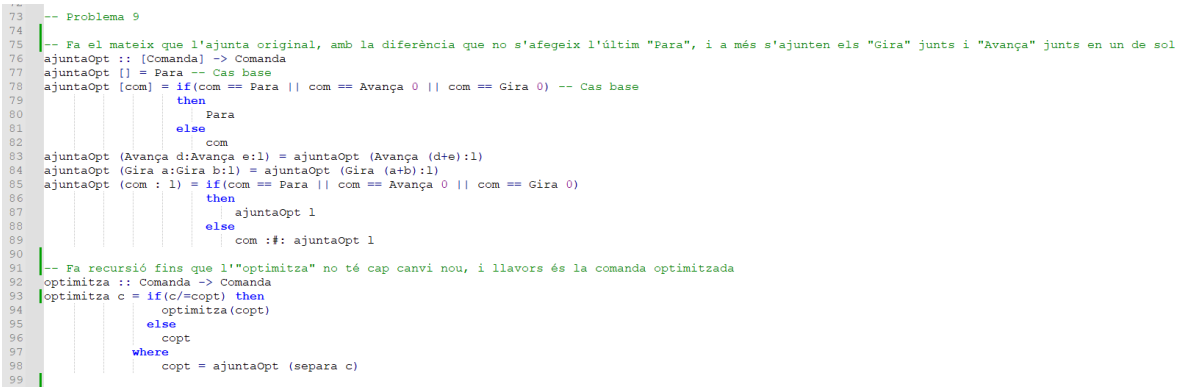
Respecte a l'*execute*, primerament s'ha de comentar que hem utilitzat un tipus de dada anomenat *EstatLlapis*, el qual es va guardant l'estat del llapis, és a dir, el seu color, l'angle que té i el punt en el que es troba. L'*execute*, a més, utilitza un predicat auxiliar immersiu anomenat *executeLn*, el qual serveix per obtenir les línies necessàries que són requerides per fer funcionar el predicat "*display*".
Aquest predicat *executeLn* itera de forma que llegeix comanda a comanda i va guardant-se les línies en ordre per ser posteriorment executades. En el propi codi està comentat cada funcionalitat de les comandes *Gira*, *Avança*, etc.. L'*executeLn* comença amb el llapis negre, amb angle 0 i en el punt (0,0).
Llavors, per poder calcular en quin punt ha d'anar a parar un *Avança* amb un determinat angle, tenim la funció auxiliar *polar*, la qual ens permet determinar la desviació del punt amb un determinat angle.



```haskell
-- Problema 8
-- Pas de comandes a lines a pintar per GL graphics

-- Tipus "EstatLlapis" auxiliar per determinar l'estat del llapis en cada comanda
data EstatLlapis = EstatLlapis Llapis Angle Pnt

-- Predicat auxiliar "polar" que ens calcula la desviació amb l'angle
polar :: Angle -> Pnt
polar a = Pnt (cos rad) (sin rad)
    where
        rad = a * pi / 180

-- Predicat auxiliar "executeLn" que ens retorna les línies i es va actualitzant l'estat del llapis
executeLn :: Comanda -> EstatLlapis -> ([Ln], EstatLlapis)
executeLn (Avança d) (EstatLlapis color angle inici) = (if color == Transparent -- Si el color del llapis és "Transparent" llavors no es dibuixa res
                                                        then []
                                                        else [Ln color inici final], EstatLlapis color angle final) -- Altrament, traça la línia amb totes les dades prop
                                                    where
                                                        final = inici + scalar d * polar angle -- Càlcul del punt final amb la desviació corresponent (si es que en té)
executeLn Para estat = ([], estat) -- Si la comanda és un Para simplement no fa res
executeLn (Gira a) (EstatLlapis color angle inici) = ([], EstatLlapis color (angle-a) inici) -- S'agafa l'angle "a" i gira respecte l'angle anteriorment ut

-- Les comandes compostes funcionen de forma que s'agafen les línies i l'estat del llapis al executar la primera comanda (com1) i es passa a la següent (co
executeLn (com1 :#: com2) estat = (linies, estat2)
    where
        (linies1, estat1) = executeLn com1 estat
        (linies2, estat2) = executeLn com2 estat1
        linies = linies1 ++ linies2 -- Finalment es concatenen les línies que donaran de resultat haver executat com1 i com2 justament després
executeLn (CanviaColor color) (EstatLlapis _ angle inici) = ([], EstatLlapis color angle inici) -- Actualitza el color del llapis
-- El funcionament de "Branca" és simple: s'executen les comandes que venen després del "Branca", retorna les línies, però l'estat del llapis es deixa en l
executeLn (Branca com) estat = (linies, estat)
    where
        (linies, _) = executeLn com estat

-- Execute com a tal, que retorna les línies que donaran de resultat el dibuix aplicat amb les comandes introduïdes, comença en el punt (0,0) i amb llapis
execute :: Comanda -> [Ln]
execute c  =  linies
    where
        (linies, _) = executeLn c (EstatLlapis negre 0 (Pnt 0 0))
```

Figura 1: Codi de'*execute* i predicats auxiliars

Respecte a l'*optimitza*, seguint una mica les indicacions de l'enunciat, hem fet un "*ajunta (separa c)*", amb la particularitat que hem canviat l'"*ajunta*" per un nou predicat anomenat *ajuntaOpt*. L'*optimitza* funciona de forma que es queda executant-se a si mateix fins que la comanda no canvia, és a dir, que està reduïda al màxim. Això passa gràcies al funcionament del predicat *ajuntaOpt*, el qual comprimeix els *Gira* junts i els *Avança* junts en un de sol, a més que es treu de comandes *dummy* tals com *Avança 0*, *Gira 0* o *Para*.



```haskell
-- Problema 9

-- Fa el mateix que l'ajunta original, amb la diferència que no s'afegeix l'últim "Para", i a més s'ajunten els "Gira" junts i "Avança" junts en un de sol
ajuntaOpt :: [Comanda] -> Comanda
ajuntaOpt [] = Para -- Cas base
ajuntaOpt [com] = if(com == Para || com == Avança 0 || com == Gira 0) -- Cas base
                  then
                      Para
                  else
                      com
ajuntaOpt (Avança d:Avança e:l) = ajuntaOpt (Avança (d+e):l)
ajuntaOpt (Gira a:Gira b:l) = ajuntaOpt (Gira (a+b):l)
ajuntaOpt (com : l) = if(com == Para || com == Avança 0 || com == Gira 0)
                      then
                          ajuntaOpt l
                      else
                          com :#: ajuntaOpt l

-- Fa recursió fins que l'"optimitza" no té cap canvi nou, i llavors és la comanda optimitzada
optimitza :: Comanda -> Comanda
optimitza c = if(c/=copt) then
                optimitza(copt)
              else
                copt
              where
                copt = ajuntaOpt (separa c)
```

Figura 2: Codi de *optimitza* i predicats auxiliars

Respecte a la *fulla*, a més de ser la segona gramàtica de fractals realitzada, utilitza la funcionalitat de *Branca* amb les comandes. Bàsicament, el que fa *Branca* és una vegada entra a executar les comandes de dins, al sortir retorna l'estat del llapis a l'estat amb el que es va entrar, així podent realitzar bifurcacions en el dibuix.

```haskell
-- Problema 11

fulla :: Int -> Comanda
fulla n = CanviaColor blau :#: fFulla n

fFulla :: Int -> Comanda
fFulla 0 = CanviaColor vermell :#: Avança 30
fFulla n = gFulla (n-1) :#: Branca (Gira (-45) :#: fFulla (n-1)) :#: Branca (Gira 45 :#: fFulla (n-1)) :#: Branca(gFulla (n-1) :#: fFulla (n-1))

gFulla :: Int -> Comanda
gFulla 0 = Avança 30
gFulla n = gFulla (n-1) :#: gFulla (n-1)
```

Figura 3: Codi de *fulla* i predicats auxiliars

4. Implementacions

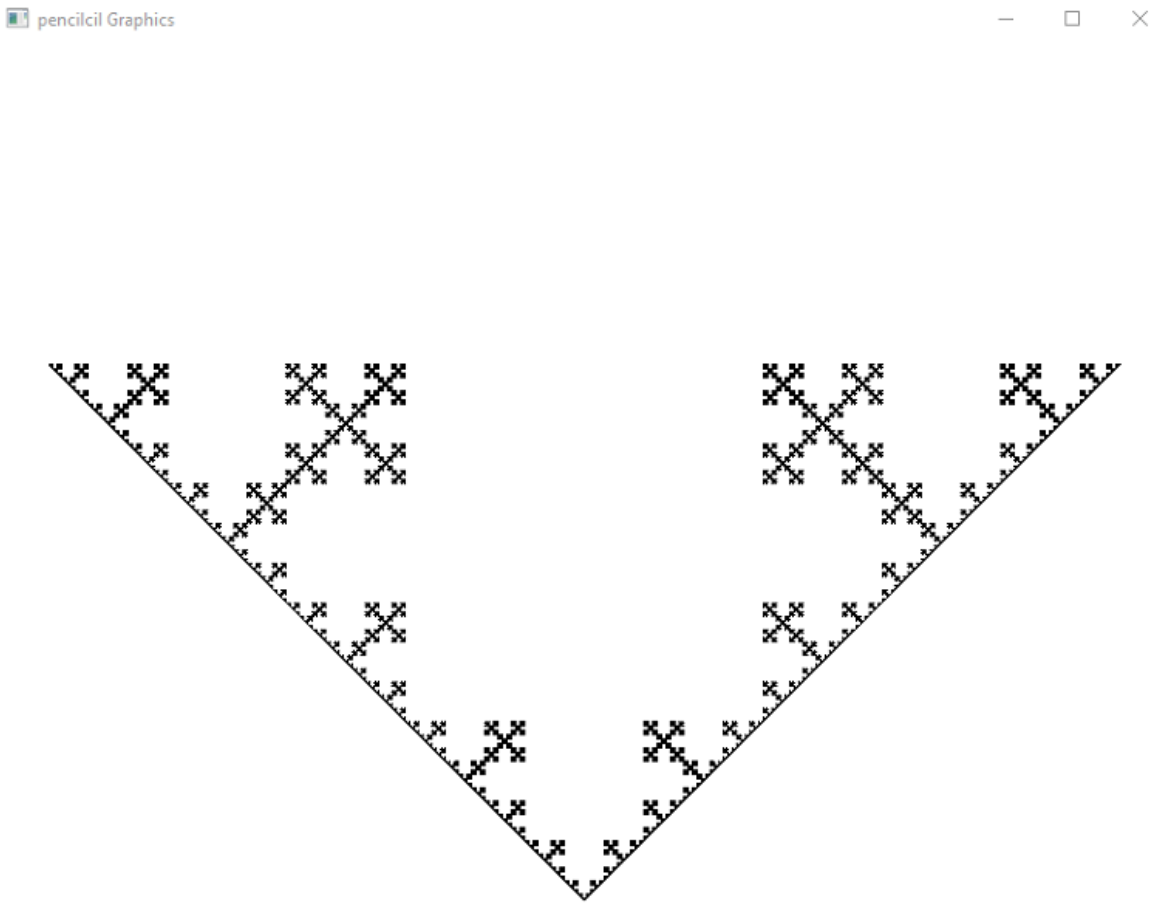


Figura 4: Resultat de la comanda `display (triangle 6)`

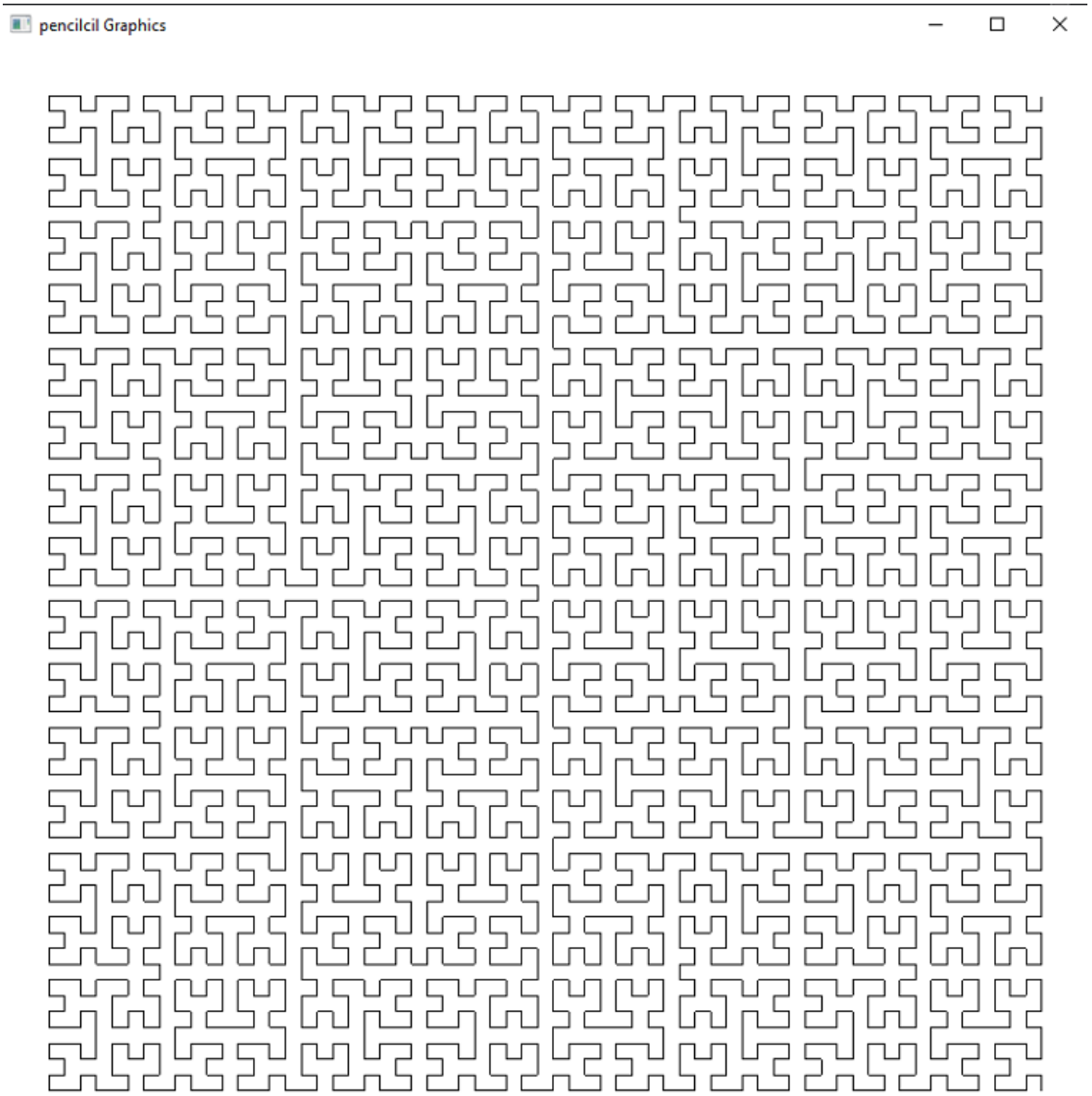


Figura 5: Resultat de la comanda `display (hilbert 6)`

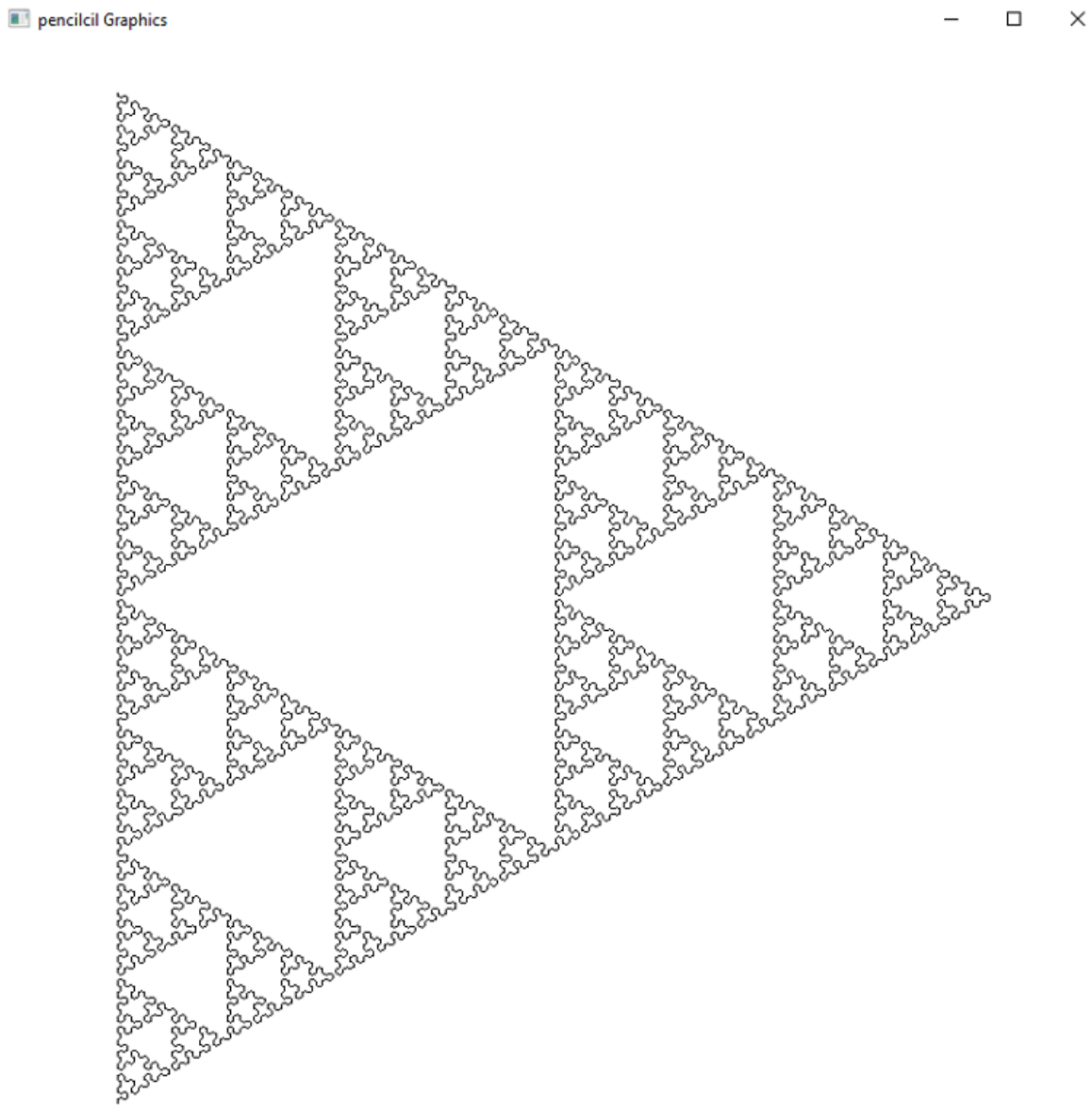


Figura 6: Resultat de la comanda `display (fletxa 8)`

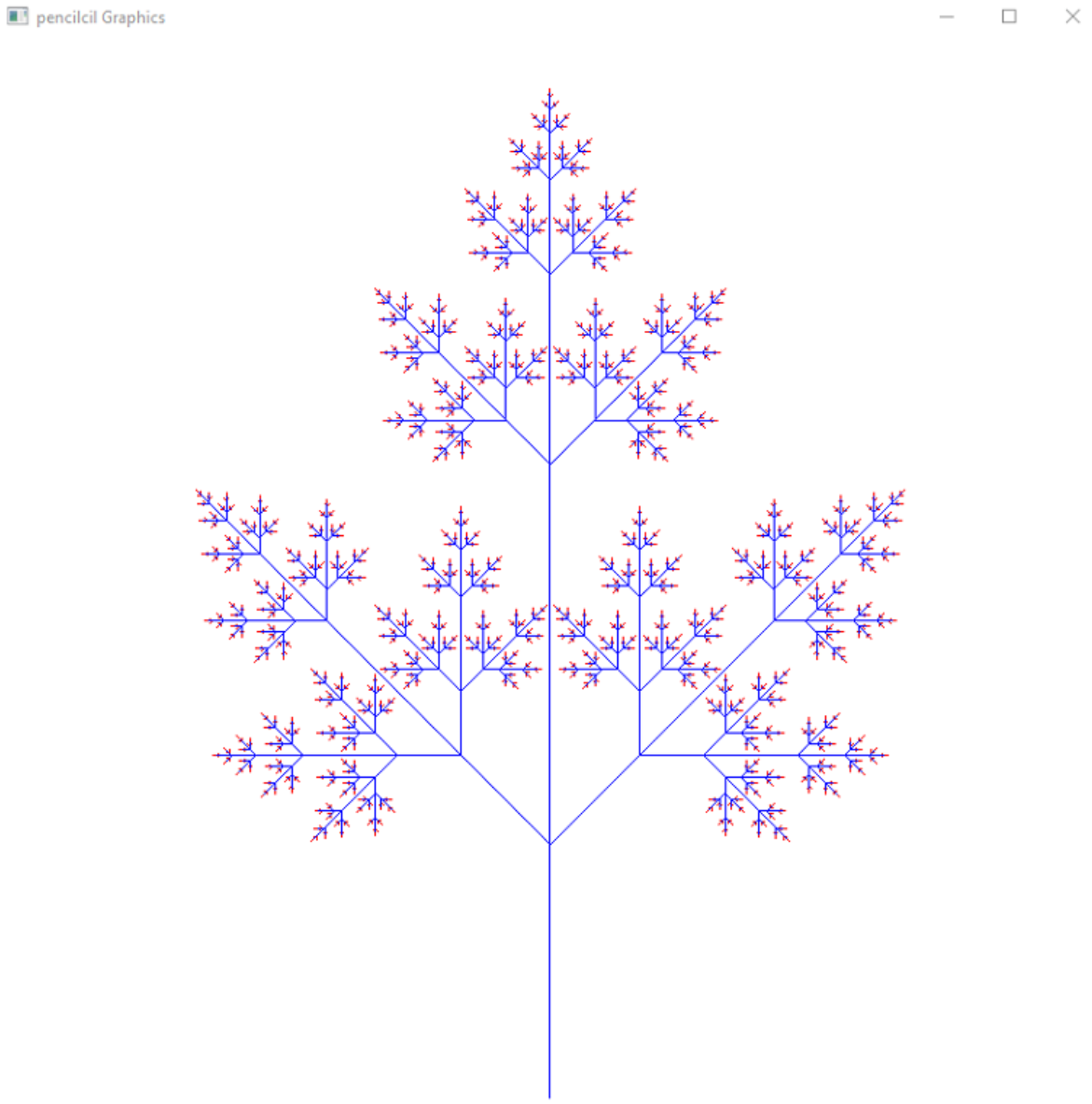


Figura 7: Resultat de la comanda `display (fulla 7)`

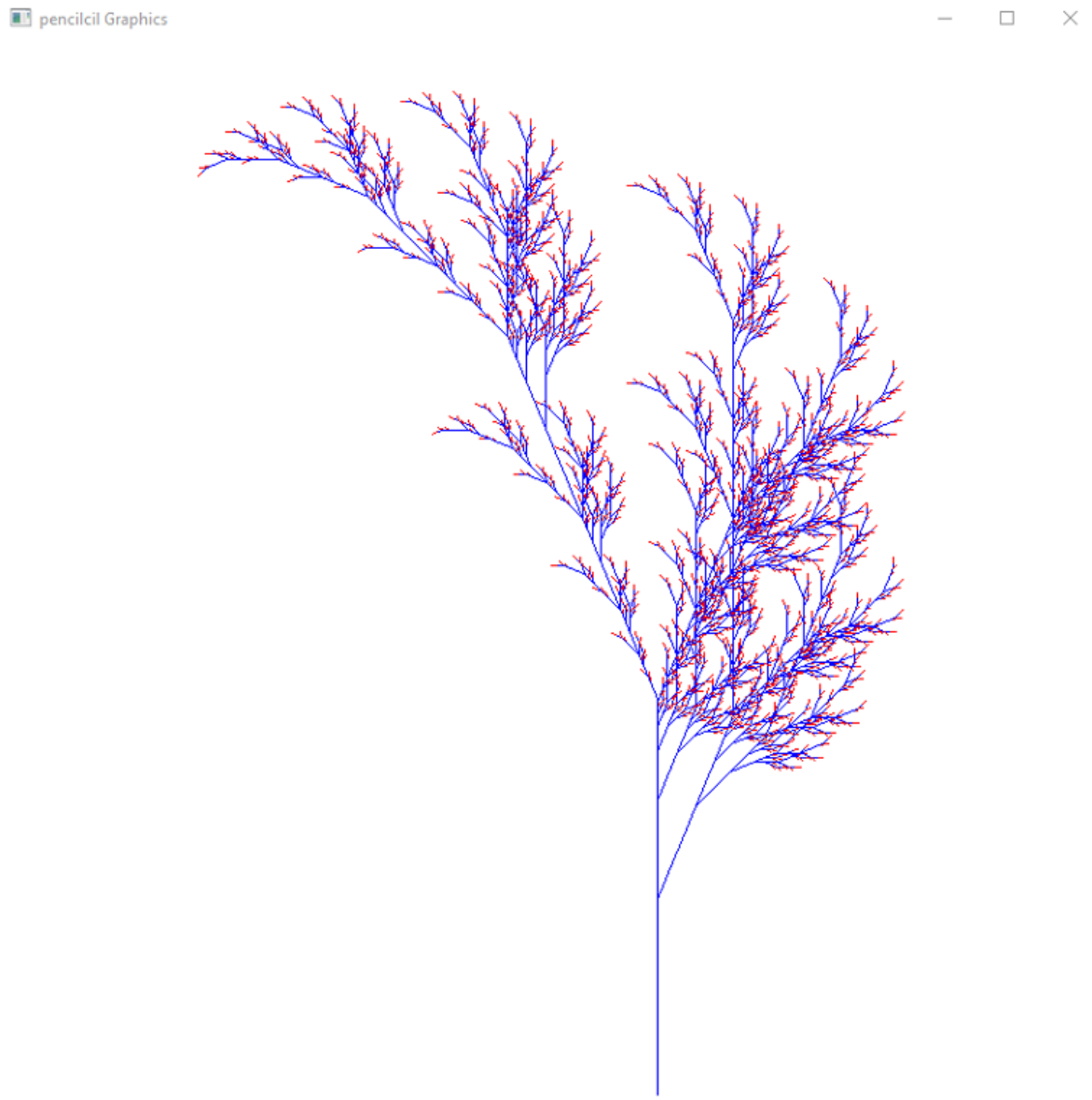


Figura 8: Resultat de la comanda `display (branca 6)`

5. Referències de bibliografia

Contribuents de la Viquipèdia. (2021). Corba de Hilbert. A Viquipèdia, l'enciclopèdia lliure. Recuperat el 20 de maig de 2023, de https://ca.wikipedia.org/wiki/Corba_de_Hilbert