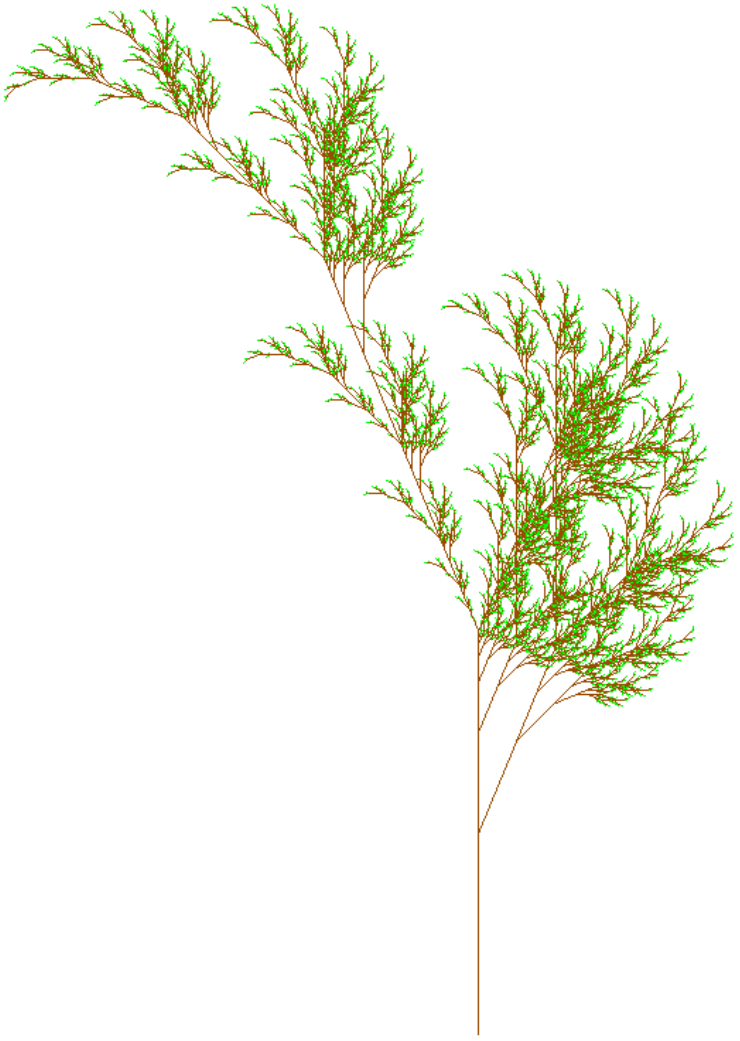
John the artist 🎨
Pràctica de Programació Funcional

Paradigmes i Llenguatges de Programació

Curs 22/23



Jordi Carmona Codinach, u1972798
Daniel Tur Zurita, u1972799

Girona 24 de Maig de 2023

ÍNDEX

Descripció del problema resolt

En aquesta pràctica hem après a utilitzar una gramàtica que funciona a partir de Comandes i a representar diverses figures a partir d'aquesta.
Totes les funcionalitats obligatòries que es demanaven a l'enunciat estan fetes, les possibles millores d'aquesta pràctica podrien ser implementar l'opció CanviPunta, implementar algun altre Fractal més complexa amb una anidament o gramàtica superior o també definir el tipus gramàtica per poder representar de forma simbólica una gramàtica de fractals.

Link al github del projecte:
https://github.com/JordiCarmonaCodinach/PracticaHaskellJohnTheArtist

Tipus de dades

En aquesta pràctica hem utilitzat varis tipus de dades:

- **Ln:** Representació d'una línia a partir del tipus de Llapis a utilitzar, el punt inicial i el punt final.
- **Angle:** Tipus de dades alias del tipus Float.
- **Distància:** Tipus de dades alias del tipus Float.
- **Comanda:** Representació de diferents ordres, on les possibles ordres són: avançar (Avança) una distància, girar (Gira) un angle, seqüenciar ordres (:#:), parar (Para), cambiar el color del llapis (CanviaColor) i branca per generar altres ordres i tornar al punt inicial (Branca) .
- **Pnt**: Tipus de dades que representa un punt 2D format per dos Float.
- **Llapis:** Tipus de dades que representa un color RGB a partir de tres Float.

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

type Angle     = Float

type Distancia = Float

data Comanda   = Avança Distancia
               | Gira Angle
               | Comanda :#: Comanda
               | Para
               | CanviaColor Llapis
               | Branca Comanda

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

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

Problemes

Problema 1 (Separa)

El següent codi mostra l'implementació de la funció **separa**, que a partir d'una Comanda dona una llista de Comandes amb cada tipus de comanda a partir de la comanda inicial separades per una coma, això s'aconsegueix amb una crida recursiva a la funció on hi han 4 casos:

- Cas Avança: retornem la mateixa comanda en forma de llista.
- Cas Gira el mateix que el cas Avança però amb la comanda Gira.
- Cas quan hi ha més d'una Comanda: es fa una crida recursiva a la funció i concatena els resultats.
- Cas Para: retorna una llista buida, ja que Para no fa res:

```
separa :: Comanda -> [Comanda]
separa (Avança d) = [Avança d]
separa (Gira a) = [Gira a]
separa (p :#: q) = separa p ++ separa q
separa Para = []
```

Exemple d'execució:

```
*Main UdGraphic Artist> separa (Avança 3 :#: Gira 4 :#: Avança 7 :#: Para)
[Avança 3.0,Gira 4.0,Avança 7.0]
```

Problema 2 (Ajunta)

El següent codi mostra l'implementació de la funció **ajunta**, que a partir d'una llista de comandes separades per comes, ajunta totes les comandes en una sola, això s'aconsegueix amb una crida recursiva de la funció on hi han 3 casos:

- Cas base: quan la llista està buida retorna Para
- Cas una sola comanda: retorna la Comanda de la llista concatenant amb una crida recursiva amb paràmetre una llista buida, així s'escriu un Para al final.
- Cas varies comandes, retorna la comanda i fa una crida recursiva de la resta de comandes.

També hi ha la funció **ajunta'**, que fa el mateix que la funció ajunta però sense posar un "Para" al final, aquesta variació de la implementació és necessària en els exercicis posteriors.

```
ajunta :: [Comanda] -> Comanda
ajunta []     = Para
ajunta [c]    = c :#: ajunta []
ajunta (c:cs) = c :#: ajunta cs

ajunta' :: [Comanda] -> Comanda
ajunta' [c]    = c
ajunta' (c:cs) = c :#: ajunta' cs
```

Exemple d'execució:

```
*Main UdGraphic Artist> ajunta [Avança  3, Gira 4, Avança 7]
Avança 3.0 :#: Gira 4.0 :#: Avança 7.0 :#: Para
*Main UdGraphic Artist> ajunta' [Avança  3, Gira 4, Avança 7]
Avança 3.0 :#: Gira 4.0 :#: Avança 7.0
```

Problema 3 (Equivalent)

En el següent codi es la implementació de la funció **prop_equivalent**, **prop_separa_ajunta** i **prop_separa**, on:

- **Prop_equivalent** compara si dos Comandes quasevols son iguals, retorna cert si és cert, altrament retorna fals. Per poder utilitzar l'operador == hem sobrecarregat l'operador Eq del tipus Comanda on comparem els valors de les dos comandes, ja siguin Avança, Gira, dos comandes concatenades (utilitza la recursivitat de la pròpia funció Eq), Para, CanviaColor, Branca (utilitza també la pròpia recursivitat), i qualsevol altre cosa es False.

```
prop_equivalent :: Comanda -> Comanda -> Bool
prop_equivalent c1 c2 = separa c1 == separa c2

instance Eq Comanda where
  (Avança d1) == (Avança d2) = d1 == d2
  (Gira a1) == (Gira a2) = a1 == a2
  (p1 :#: q1) == (p2 :#: q2) = p1 == p2 && q1 == q2
  Para == Para = True
  (CanviaColor l1) == (CanviaColor l2) = l1 == l2
  (Branca c1) == (Branca c2) = c1 == c2
  _ == _ = False
```

- **Prop_separa_ajunta** comprova que donada una Comanda qualsevol, si primer la separa i el seu resultat després l'ajunta, aquesta comanda resultant és equivalent a la comanda inicial. Això ho fa utilitzant les funcions Separa, Ajunta i prop_equivalent prèviament definides.
-

```
prop_separa_ajunta :: Comanda -> Bool
prop_separa_ajunta c = prop_equivalent c (ajunta (separa c))
```

- **Prop_separa** comprova que donada una Comanda qualsevol, si se l'hi aplica la funció separa, aquesta no conté cap Para ni cap comanda composta, això ho aconseguim comprovant que cada una de les comandes que retorna la funció separa només poden ser o Avança o Gira, qualsevol altre cosa serà False.

```
prop_separa :: Comanda -> Bool
prop_separa c = all esComanda (separa c)
  where
    esComanda (Avança _) = True
    esComanda (Gira _) = True
    esComanda _ = False
```

Exemple d'execució:

```
*Main UdGraphic Artist> quickCheck(prop_equivalent(Para :#: Avança 10) (Avança 10))
+++ OK, passed 1 test.
*Main UdGraphic Artist> quickCheck prop_separa_ajunta
+++ OK, passed 100 tests.
*Main UdGraphic Artist> quickCheck prop_separa
+++ OK, passed 100 tests.
```

Problema 4 (Copia)

El següent codi mostra l'implementació de la funció **copia**, on l'ùnic que fem és que donat un enter **n** i una Comanda **c** retornem la comanda **c** replicada **n** cops, per fer-ho utiltizem la funció replicate de Haskell que crea una llista de longitud donada pel primer argument i els elements que tenen el valor del segon argument, i després ajuntada amb la funció ajunta.

```
copia :: Int -> Comanda -> Comanda
copia n c = ajunta (replicate n c)
```

Exemple d'execució:

```
*Main UdGraphic Artist> copia 3 (Avança 10 :#: Gira 120)
Avança 10.0 :#: Gira 120.0 :#: Avança 10.0 :#: Gira 120.0 :#: Avança 10.0 :#:
Gira 120.0 :#: Para
```

Problema 5 (Pentagon)

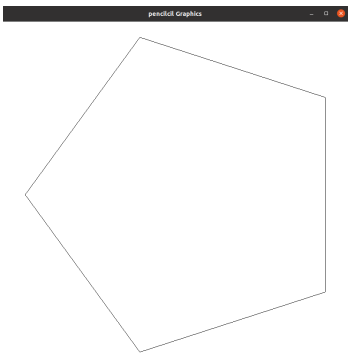
El següent codi mostra l'implementació de la funció **pentagon**, on donat un enter **n** generem una Comanda que serà la codificació d'un pentàgon, on s'utilitza la funció prèviament definida copia, per a generar 5 comandes iguals on en cada una serà un Avança de n i una Gira de 72, ja que l'angle del pentàgon té 72 graus.

```
pentagon :: Distancia -> Comanda
pentagon d = copia 5 (Avança d :#: Gira 72)
```

Exemple d'execució:

```
*Main UdGraphic Artist> pentagon 50
Avança 50.0 :#: Gira 72.0 :#: Avança 50.0 :#: Gira 72.0 :#: Avança 50.0 :#: Gi
ra 72.0 :#: Avança 50.0 :#: Gira 72.0 :#: Avança 50.0 :#: Gira 72.0 :#: Para
```

Representació gràfica del pentàgon (utilitzant display(pentagon 50)):



Problema 6 (Poligon)

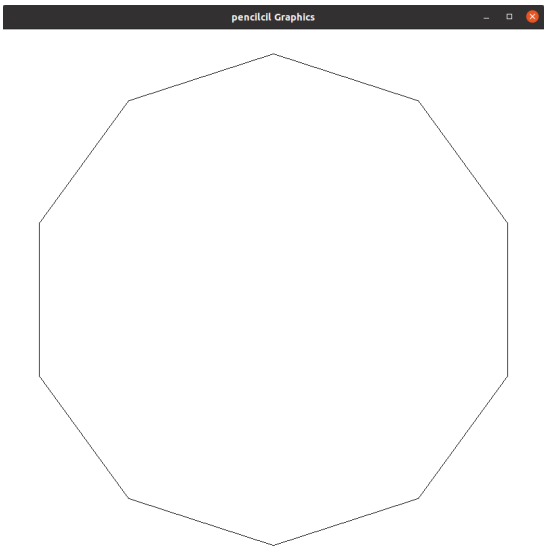
El següent codi mostra l'implementació de la funció **poligon**, on donada la mida de costat **d**, el nombre de costats **n**, i l'angle entre costats **a** genera **n** Comandes de la Comanda "Avança **d** i Gira **a**".
També s'ha implementat la funció **prop_poligon_pentagon**, on donada una distància **n**, es genera la codificació d'un pentàgon de **n** cares i d'un polígon de longitud **n**, 5 cares i angle 72, després es comprova si efectivament generen la mateix codificació (es comparen les 2 Comandes per veure si són iguals amb la funció prop_equivalent).

```
poligon :: Distancia -> Int -> Angle -> Comanda
poligon d n a = copia n (Avança d :#: Gira a)

prop_poligon_pentagon :: Distancia -> Bool
prop_poligon_pentagon d = prop_equivalent c1 c2
  where
    c1 = poligon d 5 72
    c2 = pentagon d
```

Exemple d'execució:

```
*Main UdGraphic Artist> poligon 50 5 72
Avança 50.0 :#: Gira 72.0 :#: Avança 50.0 :#: Gira 72.0 :#: Avança 50.0 :#: Gi
ra 72.0 :#: Avança 50.0 :#: Gira 72.0 :#: Avança 50.0 :#: Gira 72.0 :#: Para
*Main UdGraphic Artist> quickCheck prop_poligon_pentagon
+++ OK, passed 100 tests.
```

Representació gràfica del poligon (utilitzant display(poligon 50 10 36)):



Problema 7 (Espiral)

El següent codi mostra l'implementació de la funció **espiral**, on donada la mida del primer segment **d**, nombre de segments a dibuixar **n**, distància que s'augmenta per cada segment successiu **pas**, i l'angle de l'espiral **a**, generem **n** Comandes de forma recursiva, on en cada

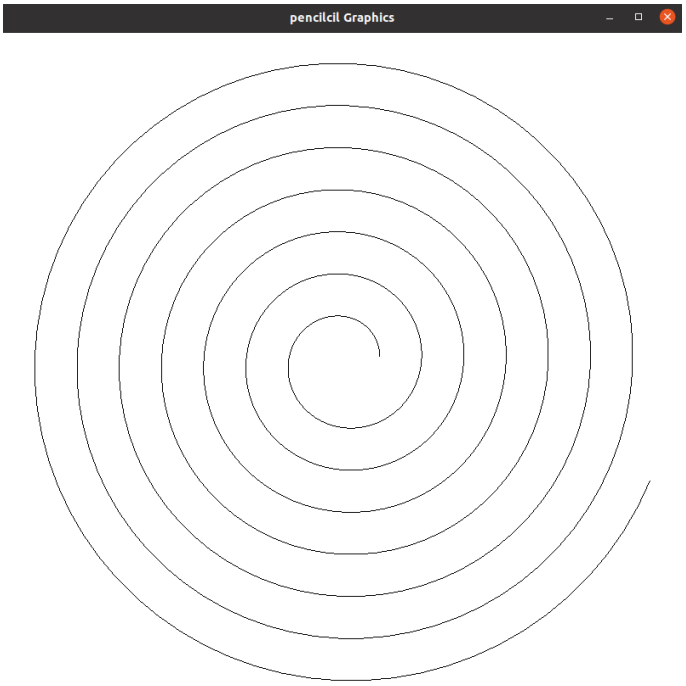
crida recursiva es disminueix 1 el paràmetre fins a arribar a 0 i s'augmenta **d** en **pas**, en cada crida en generarà una Comanda "Avança **d** :#: Gira **a** :#: "Crida Recursiva".

```
espiral :: Distancia -> Int -> Distancia -> Angle -> Comanda
espiral _ 0 _ _ = Para
espiral distancia n pas angle = Avança distancia :#: Gira angle :#:
espiral (distancia + pas) (n - 1) pas angle
```

Exemple d'execució:

```
*Main UdGraphic Artist> espiral 30 4 5 30
Avança 30.0 :#: Gira 30.0 :#: Avança 35.0 :#: Gira 30.0 :#: Avança 40.0 :#: Gi
ra 30.0 :#: Avança 45.0 :#: Gira 30.0 :#: Para
```

Representació gràfica de l'espiral (utilitzant display(espiral 30 500 0.5 5)):



Problema 8 (Execute)

El següent codi mostra l'implementació de la funció **execute**, on donada una Comanda c qualsevol retorna una llista de Ln, aquesta funció depen d'un altre funció auxiliar **execute'** que donada una Comanda **c** i una **tupla** formada per Llapis, un Angle i un Pnt.

Execute crida a la funció execute' a partir de la comanda c i una tupla formada per la representació de llapis (negre), un angle 0 i un punt 0,0 que representarà l'inici del canvas.

Execute' es comporta d'una manera diferent depenent del tipus de Comanda entrat, on:

- **Avança:** donada una distància **d** i una **tupla** retorna una llista Ln amb una única línia amb el mateix llapis, el mateix punt inicial i el nou punt calculat i la **tupla** amb el mateix punt calculat, on el punt calculat es sumar la x a la **d** multiplicat pel cosinus de l'angle convertit a radiants i restar a la y **d** multiplicat per el sin de l'angle convertit a radiants.
- **Gira:** donat un angle **a** i una **tupla**, retorna una llista de Ln buida i la tupla igual però amb l'angle sumat a **a**.
- **Para:** donat un **Para** i una **tupla**, retorna una llista de Ln buida i la tupla igual.
- **CanviaColor:** donat un Llapis **llapis** i una **tupla** retorna una llista de Ln buida i la tupla igual però amb el nou Llapis **llapis.**
- **Branca:** donada una Branca **b** i una tupla **tupla**, retorna una llista de Ln amb una ùnica línia amb el resultat d'executar execute' recursivament amb els mateixos paràmetres i la mateixa **tupla** inicial, d'aquesta forma a l'acabar l'execució de la Branca tornarem al mateix Llapis, el mateix Angle i el mateix Pnt que al principi.
- **Comandes consecutives:** donades dues comandes consecutives **c** i **d** i una **tupla**, retornem una llista de Ln amb el resultat del primer paràmetre (llista de Ln) d'executar execute' **c tupla** concatenat amb el resultat del primer paràmetre (llista de Ln) d'executar execute' **d tupla** i el resultat de la tupla que retorna el segon execute' que hem fet (el que té **d** com a paràmetre).

Per a convertir els angles a Radiants utilitzem la funció toRadians que donat un Angle **a** retorna l'angle convertit a radiants multiplicant **a** per Pi i dividit entre 180.

Codi de l'implementació:

```
execute :: Comanda -> [Ln]
execute comanda = result
  where
    (result, (llapis,angle,pnt)) = execute' comanda (negre, 0, Pnt 0 0)

execute' :: Comanda -> (Llapis,Angle,Pnt) -> ([Ln], (Llapis,Angle,Pnt))
execute' (Avança d) (llapis, angle, pnt@(Pnt x y)) = ([Ln llapis pnt
nouPnt], (llapis, angle, nouPnt))
  where
    nouPnt = Pnt (x + d * cos (toRadians angle)) (y - d * sin (toRadians
angle))
execute' (Gira a) (llapis, angle, pnt) = ([], (llapis, (angle + a),
pnt))
execute' Para (llapis,angle,pnt) = ([], (llapis,angle,pnt))
execute' (CanviaColor llapis) (_, angle, pnt) =  ([],(llapis, angle,
pnt))
execute' (Branca b) (llapis,angle,pnt) = (ln, (llapis,angle,pnt))
```

```
  where
    (ln, (llapisBranca,angleBranca,pntBranca)) = execute' b
(llapis,angle,pnt)
execute' (c :#: d) (llapis,angle,pnt) = (lnC ++ lnD,
(dllapis,dangle,dpnt))
  where
    (lnC, (cllapis,cangle,cpnt)) = execute' c (llapis,angle,pnt)
    (lnD, (dllapis,dangle,dpnt)) = execute' d (cllapis,cangle,cpnt)

toRadians :: Angle -> Angle
toRadians angle = angle * pi / 180
```

Exemple d'execució:

```
*Main UdGraphic Artist> execute (Avança 30 :#: Para :#: Gira 10 :#: Avança 20)
[Ln (Color' 0.0 0.0 0.0) (Pnt 0.0 0.0) (Pnt 30.0 0.0),Ln (Color' 0.0 0.0 0.0)
(Pnt 30.0 0.0) (Pnt 49.696156 (-3.4729638))]
*Main UdGraphic Artist> execute (Avança 30 :#: Para :#: Gira 10 :#: Avança 20
:#: Gira (-15) :#: Para :#: Avança 10 :#: Para :#: Para)
[Ln (Color' 0.0 0.0 0.0) (Pnt 0.0 0.0) (Pnt 30.0 0.0),Ln (Color' 0.0 0.0 0.0)
(Pnt 30.0 0.0) (Pnt 49.696156 (-3.4729638)),Ln (Color' 0.0 0.0 0.0) (Pnt 49.69
6156 (-3.4729638)) (Pnt 59.658104 (-2.6014063))]
```

Problema 9 (Optimitza)

El següent codi mostra l'implementació de la funció **optimitza**, on donada una Comanda **c** retorna una Comanda simplificada al màxim, per fer-ho fem un crida recursiva en la funció optimitza, on si el resultat de "comandaOptimitzada" que es el resultat de simplificar la comanda amb la funció simplifica, el seu resultat seprar-ho amb la funcio separa i despres ajuntar-ho amb la funció ajunta' (ajunta' no posa un Para al final de la Comanda resultant) i després comprar-la amb la Comanda inicial **c**, si **no és igual** a la Comanda torna a cridar recursivament a la funció com a paràmetre la funció semi-optimitzada, altrament (és a dir, **si són iguals**) retorna finalment la Comanda simplificada al màxim.

La funció simplifica retorna una Comanda a partir d'una Comanda **c**, on depenent del tipus de comanda **c**:

- **Avança 0:** és equivalent a Para.
- **Gira 0:** és equivalent a Para.
- **Avança d1 :#: Avança d2 :#: cs:** és equivalent a la crida recursiva de Avança (d1+d2) :#: cs, on cs es la resta de la comanda.
- **Gira a1 :#: Gira a2 :#: cs:** és equivalent a la crida recursiva de Gira(a1+a2) :#: cs, on cs es la resta de la comanda.
- **c1 :#: c2:** és equivalent a la simplificació de les dos comandes consecutives simplificades prèviament (i cap altre al darrere) a partir de la funció simplificaAux.
- **Qualsevol altre tipus de Comanda (i cap altre al darrere):** és equivalent a ella mateixa.

La funció simplificaAux retorna una Comanda a partir d'una Comanda **c** i un altre Comanda **d**, on depenent del tipus de les dos comandes **c** i **d**:

- **Avança d1, Avança d2:** és equivalent a la crida recursiva de Avança (d1+d2).
- **Gira a1, Gira a2:** és equivalent a la crida recursiva de Gira(a1+a2).
- Qualsevol altre combinació de tipus de Comandes: són equivalents a elles mateixes.

```
optimitza :: Comanda -> Comanda
optimitza comanda = if comandaOptimitzada == comanda
                    then comandaOptimitzada
                    else optimitza comandaOptimitzada
  where
    comandaOptimitzada = ajunta'(separa(simplifica comanda))

simplifica :: Comanda -> Comanda
simplifica (Avança 0) = Para
simplifica (Gira 0) = Para
simplifica (Avança d1 :#: Avança d2 :#: cs) = simplifica (Avança (d1 +
d2) :#: cs)
simplifica (Gira a1 :#: Gira a2 :#: cs) = simplifica (Gira (a1 + a2) :#:
cs)
simplifica (c1 :#: c2) = simplificaAux (simplifica c1) (simplifica c2)
simplifica c = c
```

```haskell
simplificaAux :: Comanda -> Comanda -> Comanda
simplificaAux (Avança d1) (Avança d2) = Avança (d1 + d2)
simplificaAux (Gira a1) (Gira a2) = Gira (a1 + a2)
simplificaAux c1 c2 = c1 :#: c2
```

Exemple d'execució:

```
*Main UdGraphic Artist> optimitza (Avança 10 :#: Para :#: Avança 20 :#: Gira 3
5 :#: Avança 0 :#: Gira 15 :#: Gira (-50))
Avança 30.0
```

Problema 10 (Triangle)

El següent codi mostra l'implementació de la funció **triangle** (fractal), on donat un enter **n** genera una codificació d'un triangle de Sierpiński en forma de Comanda a partir de la següent gramàtica:

angle: 90
inici: +f
reescriptura: f → f+f-f-f+f

Aquesta funció és una funció recursiva on cada caràcter de la gramàtica que no sigui un + o un - (que són equivalents a les Comandes Gira 90 i Gira -90 respectivament), té una funció equivalent amb el mateix nom però amb el caràcter al davant (fTriangle en aquest cas), on en cada una d'aquestes tenen un cas base:

- Quan l'enter **n** és 0 retorna la Comanda CanviaColor color concatenat amb un Avança 10.

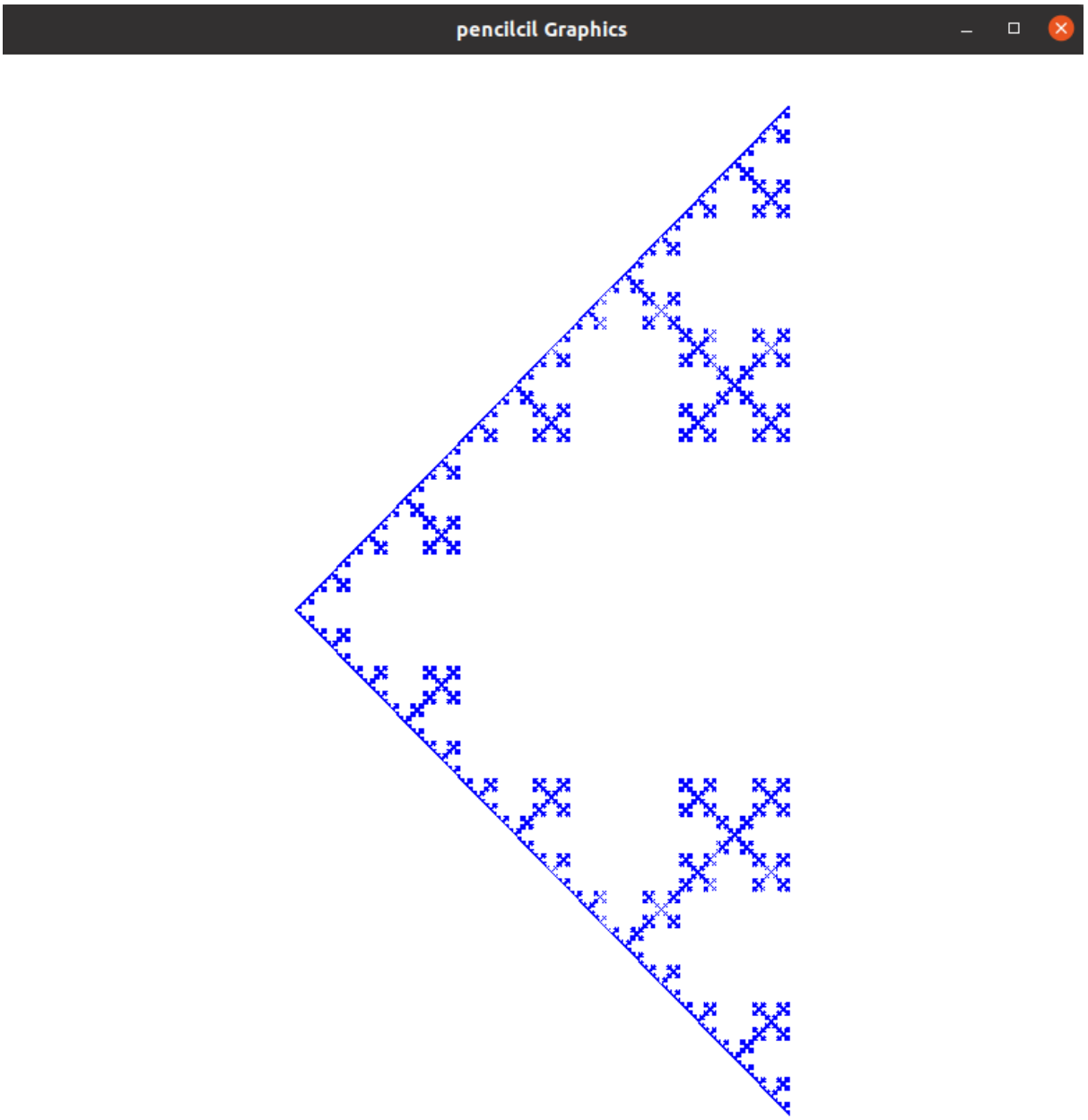
El cas recursiu és dóna quan l'enter **n** no és 0 (1 o més gran), en aquest cas s'expandeix recursivament segons la gramàtica (reescriptura) fins a arribar a n==0 en cada cas, per tant l'expansió (reescriptura) depèn de l'enter **n**.

```haskell
triangle :: Int -> Comanda
triangle n = fTriangle n

fTriangle :: Int -> Comanda
fTriangle 0 = CanviaColor blau :#: Avança 10
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)
```

Exemple d'execució:

```
*Main UdGraphic Artist> triangle 1
Color Color' 0.0 0.0 1.0 :#: Avança 10.0 :#: Gira 90.0 :#: Color Color' 0.0 0.
0 1.0 :#: Avança 10.0 :#: Gira -90.0 :#: Color Color' 0.0 0.0 1.0 :#: Avança 1
0.0 :#: Gira -90.0 :#: Color Color' 0.0 0.0 1.0 :#: Avança 10.0 :#: Gira 90.0
:#: Color Color' 0.0 0.0 1.0 :#: Avança 10.0
```

Representació gràfica del triangle (utilitzant display(triangle 7)):



Problema 11 (Fulla)

El següent codi mostra l'implementació de la funció **fulla** (fractal), on donat un enter **n** genera una codificació d'una fulla en forma de Comanda a partir de la següent gramàtica:

angle: 45
inici: f
reescriptura: f → g[-f][+f][gf]
g → gg

Aquesta funció és una funció recursiva on cada caràcter de la gramàtica que no sigui un + o un - (que són equivalents a les Comandes Gira 45 i Gira -45 respectivament), té una funció equivalent amb el mateix nom però amb el caràcter al davant (fFulla i gFulla en aquest cas), on en cada una d'aquestes tenen un cas base:

- Quan l'enter **n** és 0 retorna la Comanda CanviaColor color concatenat amb un Avança 10.

El cas recursiu és dóna quan l'enter **n** no és 0 (1 o més gran), en aquest cas s'expandeix recursivament segons la gramàtica (reescriptura) fins a arribar a n==0 en cada cas, per tant l'expansió (reescriptura) depèn de l'enter **n**.

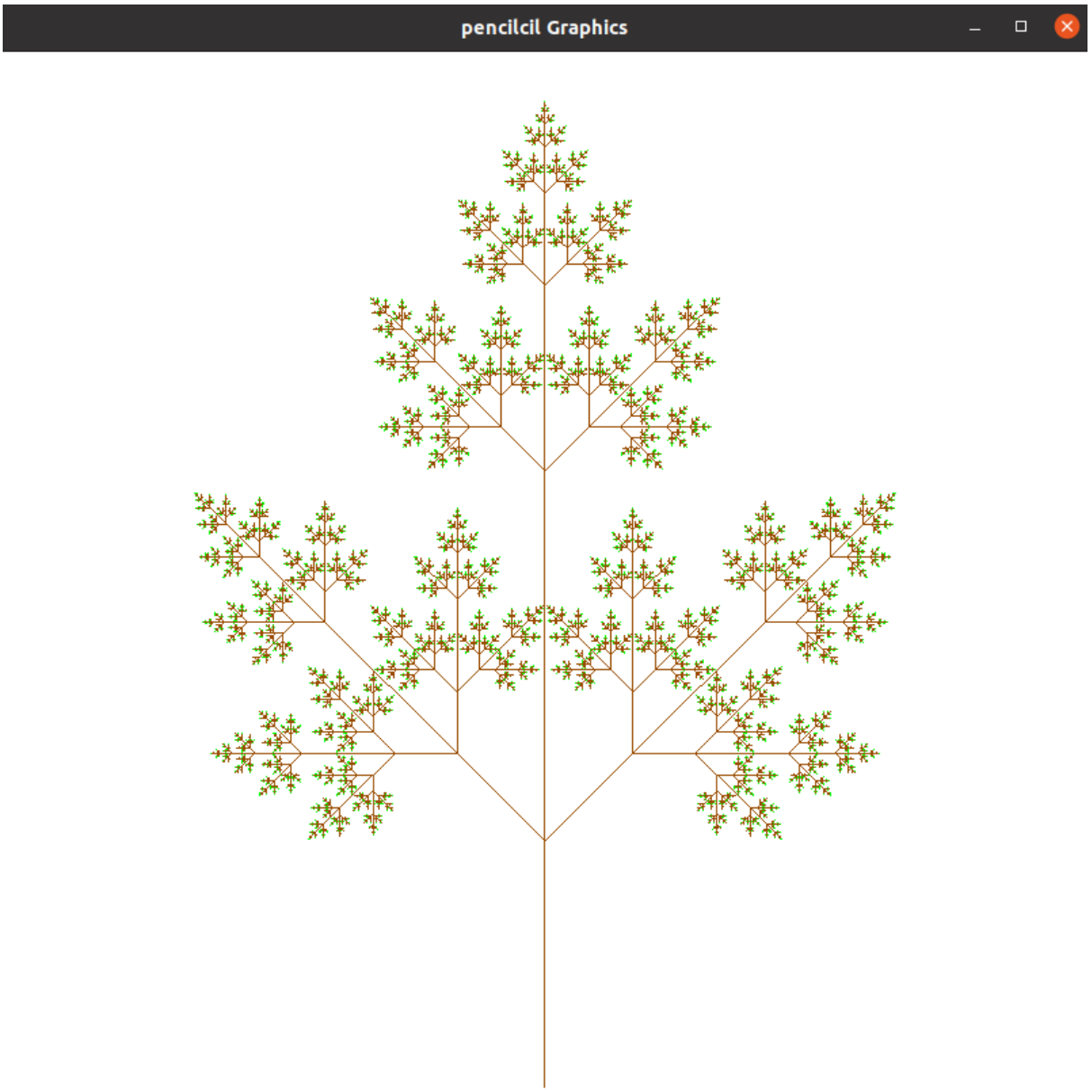
Aquesta funció a més a més utilitza conté branques, que ens serveixen per representar objectes a partir d'un cert punt i quan s'acaba la branca tornar a l'inici de la mateixa, en aquest cas tenim 3 branques i es criden en la funció fFulla amb la seva respectiva Comanda dins, primer la branca esquerra de l'arbre (Girem -45 per orientar la Branca), la segona a la dreta de l'arbre (Girem 45) i l'ùltim recta (Avançem i no girem).

```
fulla :: Int -> Comanda
fulla n = fFulla n

fFulla :: Int -> Comanda
fFulla 0 = CanviaColor verd :#: Avança 10
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 = CanviaColor marro :#: Avança 10
gFulla n = gFulla (n - 1) :#: gFulla (n - 1)
```

Exemple d'execució:

```
*Main UdGraphic Artist> fulla 1
Color Color' 0.6 0.3 0.0 :#: Avança 10.0 :#: Branca (Gira -45.0 :#: Color Colo
r' 0.0 1.0 0.0 :#: Avança 10.0) :#: Branca (Gira 45.0 :#: Color Color' 0.0 1.0
 0.0 :#: Avança 10.0) :#: Branca (Color Color' 0.6 0.3 0.0 :#: Avança 10.0 :#:
 Color Color' 0.0 1.0 0.0 :#: Avança 10.0)
```

Representació gràfica de la fulla (utilitzant display(fulla 9)):



Problema 12 (Hilbert)

El següent codi mostra l'implementació de la funció **hilbert** (fractal), on donat un enter **n** genera una codificació d'una corba de Hilbert en forma de Comanda a partir de la següent gramàtica:

angle: 90 inici: l
reescritura: l → +rf-lfl-fr+
r → -lf+rfr+fl

Aquesta funció és una funció recursiva on cada caràcter de la gramàtica que no sigui un + o un - (que són equivalents a les Comandes Gira 90 i Gira -90 respectivament), té una funció equivalent amb el mateix nom però amb el caràcter al davant (lHilbert, rHilbert i fHilbert en aquest cas), on en cada una d'aquestes tenen un cas base:

- Quan l'enter **n** és 0 retorna la Comanda CanviaColor color concatenat amb un Avança 10.

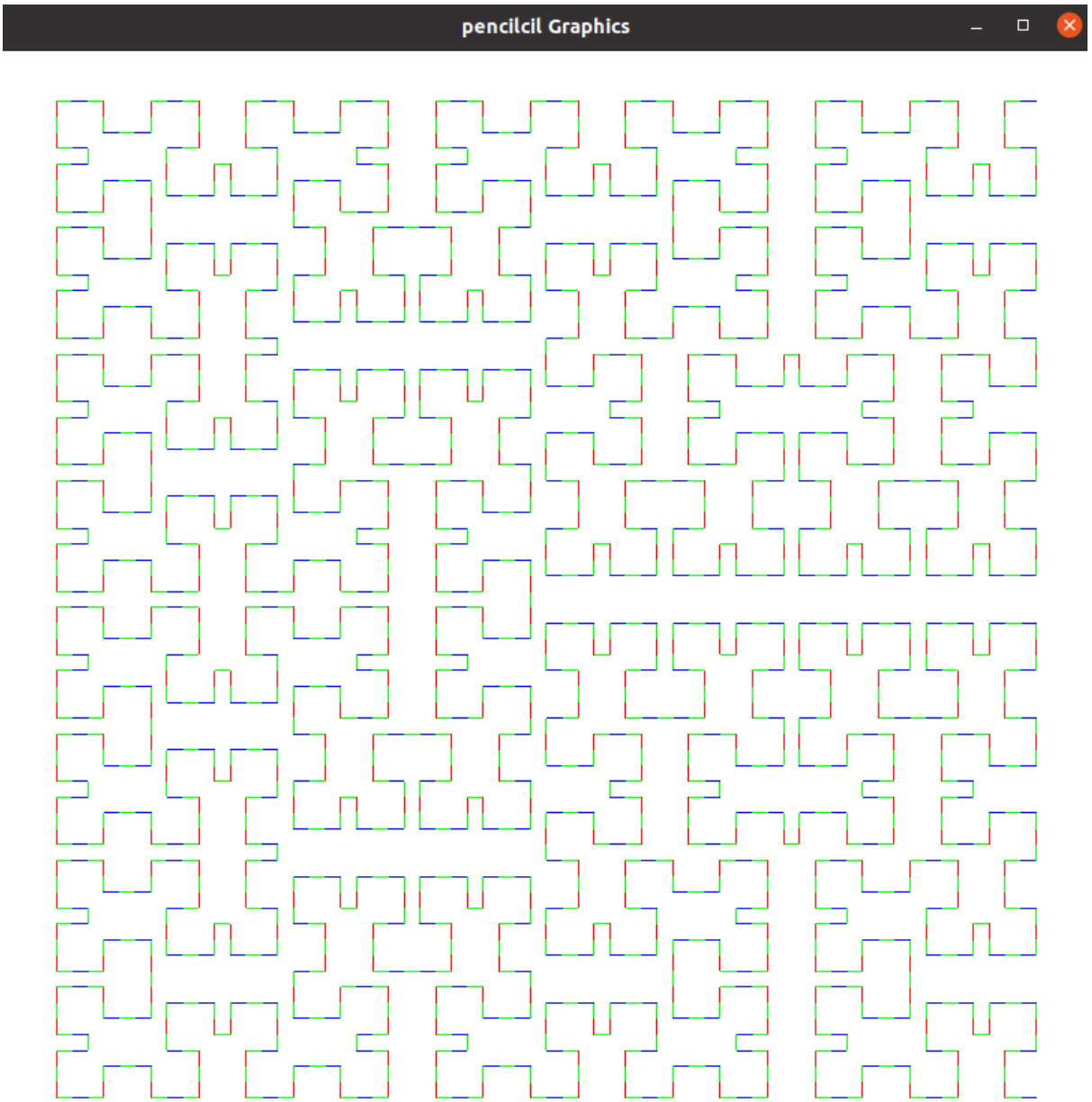
El cas recursiu és dóna quan l'enter **n** no és 0 (1 o més gran), en aquest cas s'expandeix recursivament segons la gramàtica (reescriptura) fins a arribar a n==0 en cada cas, per tant l'expansió (reescriptura) depèn de l'enter **n**.

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

lHilbert :: Int -> Comanda
lHilbert 0 = CanviaColor vermell :#: Avança 10
lHilbert n = Gira 90 :#: rHilbert (n - 1) :#: fHilbert :#: Gira (-90)
:#: lHilbert (n - 1) :#: fHilbert :#: lHilbert (n - 1) :#: Gira (-90)
:#: fHilbert :#: rHilbert (n - 1) :#: Gira 90

rHilbert :: Int -> Comanda
rHilbert 0 = CanviaColor blau :#: Avança 10
rHilbert n = Gira (-90) :#: lHilbert (n - 1) :#: fHilbert :#: Gira 90
:#: rHilbert (n - 1) :#: fHilbert :#: rHilbert (n - 1) :#: Gira 90 :#:
fHilbert :#: lHilbert (n - 1) :#: Gira (-90)

fHilbert :: Comanda
fHilbert = CanviaColor verd :#: Avança 10
```

Exemple d'execució:

```
*Main UdGraphic Artist> hilbert 1
Gira 90.0 :#: Color Color' 0.0 0.0 1.0 :#: Avança 10.0 :#: Color Color' 0.0 1.
0 0.0 :#: Avança 10.0 :#: Gira -90.0 :#: Color Color' 1.0 0.0 0.0 :#: Avança 1
0.0 :#: Color Color' 0.0 1.0 0.0 :#: Avança 10.0 :#: Color Color' 1.0 0.0 0.0
:#: Avança 10.0 :#: Gira -90.0 :#: Color Color' 0.0 1.0 0.0 :#: Avança 10.0 :#
: Color Color' 0.0 0.0 1.0 :#: Avança 10.0 :#: Gira 90.0
```

Representació gràfica de la corba de hilbert (utilitzant display(hilbert 5)):



Problema 13 (Fletxa)

El següent codi mostra l'implementació de la funció **fletxa** (fractal), on donat un enter **n** genera una codificació d'una corba de fletxa de Sierpińskien forma de Comanda a partir de la següent gramàtica:

angle: 60
inici: f
reescritura: f → g+f+g
g → f-g-f

Aquesta funció és una funció recursiva on cada caràcter de la gramàtica que no sigui un + o un - (que són equivalents a les Comandes Gira 60 i Gira -60 respectivament), té una funció equivalent amb el mateix nom però amb el caràcter al davant (fFletxa i gFletxa en aquest cas), on en cada una d'aquestes tenen un cas base:

- Quan l'enter **n** és 0 retorna la Comanda CanviaColor color concatenat amb un Avança 10.

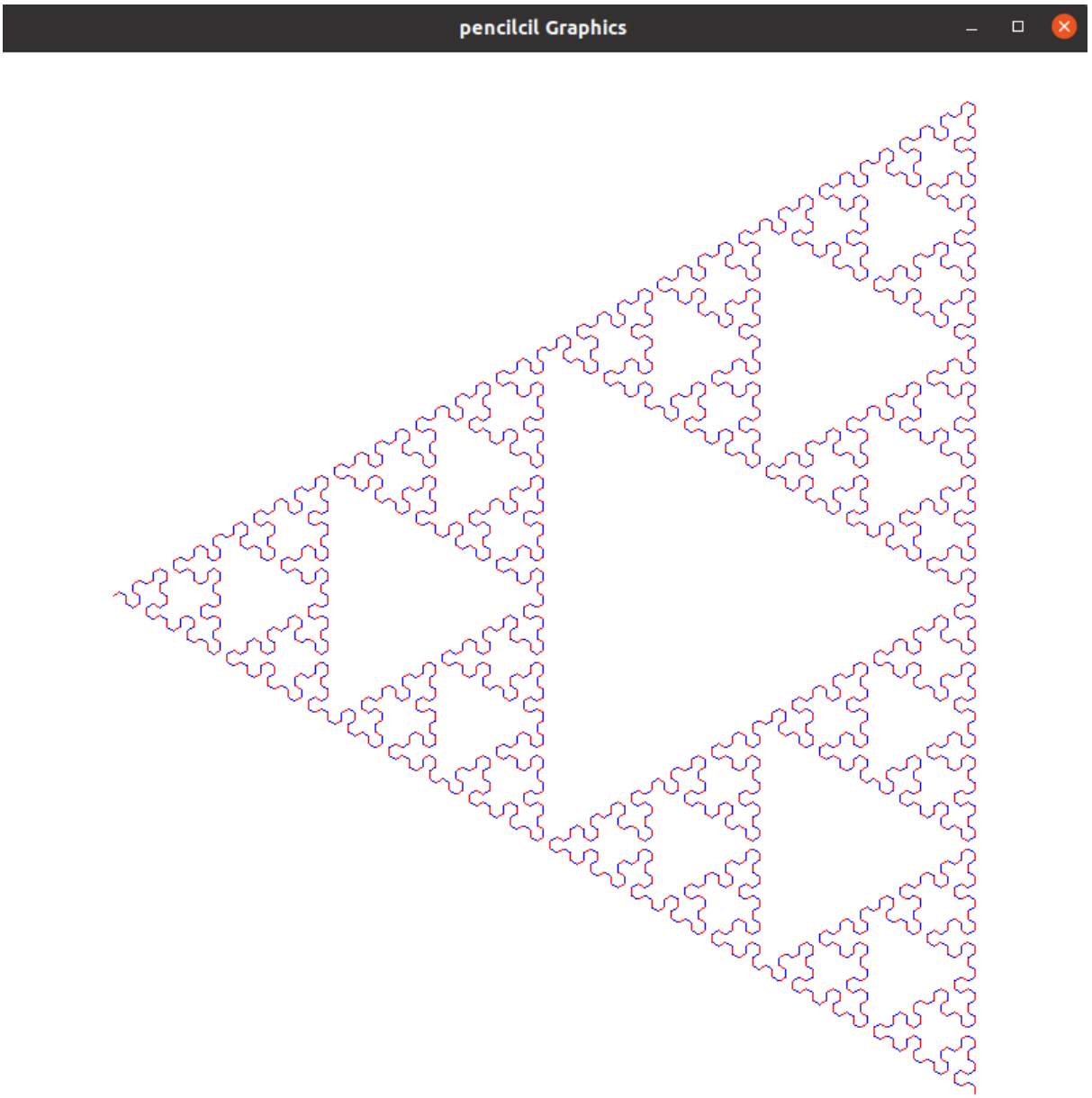
El cas recursiu és dóna quan l'enter **n** no és 0 (1 o més gran), en aquest cas s'expandeix recursivament segons la gramàtica (reescriptura) fins a arribar a n==0 en cada cas, per tant l'expansió (reescriptura) depèn de l'enter **n**.

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

fFletxa :: Int -> Comanda
fFletxa 0 = CanviaColor blau :#: Avança 10
fFletxa n = gFletxa (n - 1) :#: Gira 60 :#: fFletxa (n - 1) :#: Gira 60
:#: gFletxa (n - 1)

gFletxa :: Int -> Comanda
gFletxa 0 = CanviaColor vermell :#: Avança 10
gFletxa n = fFletxa (n - 1) :#: Gira (-60) :#: gFletxa (n - 1) :#: Gira
(-60) :#: fFletxa (n - 1)
```

Exemple d'execució:

```
*Main UdGraphic Artist> fletxa 1
Color Color' 1.0 0.0 0.0 :#: Avança 10.0 :#: Gira 60.0 :#: Color Color' 0.0 0.
0 1.0 :#: Avança 10.0 :#: Gira 60.0 :#: Color Color' 1.0 0.0 0.0 :#: Avança 10
.0
```

Representació gràfica de la fletxa (utilitzant display(fletxa 7)):



Problema 14 (Branca)

El següent codi mostra l'implementació de la funció **branca** (fractal), on donat un enter **n** genera una codificació d'una corba de fletxa de Sierpińskien forma de Comanda a partir de la següent gramàtica:

angle: 22.5
inici: g
reescritura: g → f-[[g]+g]+f[+fg]-g
f → ff

Aquesta funció és una funció recursiva on cada caràcter de la gramàtica que no sigui un + o un - (que són equivalents a les Comandes Gira 22.5 i Gira -22.5 respectivament), té una funció equivalent amb el mateix nom però amb el caràcter al davant (fBranca i gBranca en aquest cas), on en cada una d'aquestes tenen un cas base:

- Quan l'enter **n** és 0 retorna la Comanda CanviaColor color concatenat amb un Avança 10.

El cas recursiu és dóna quan l'enter **n** no és 0 (1 o més gran), en aquest cas s'expandeix recursivament segons la gramàtica (reescriptura) fins a arribar a n==0 en cada cas, per tant l'expansió (reescriptura) depèn de l'enter **n**.

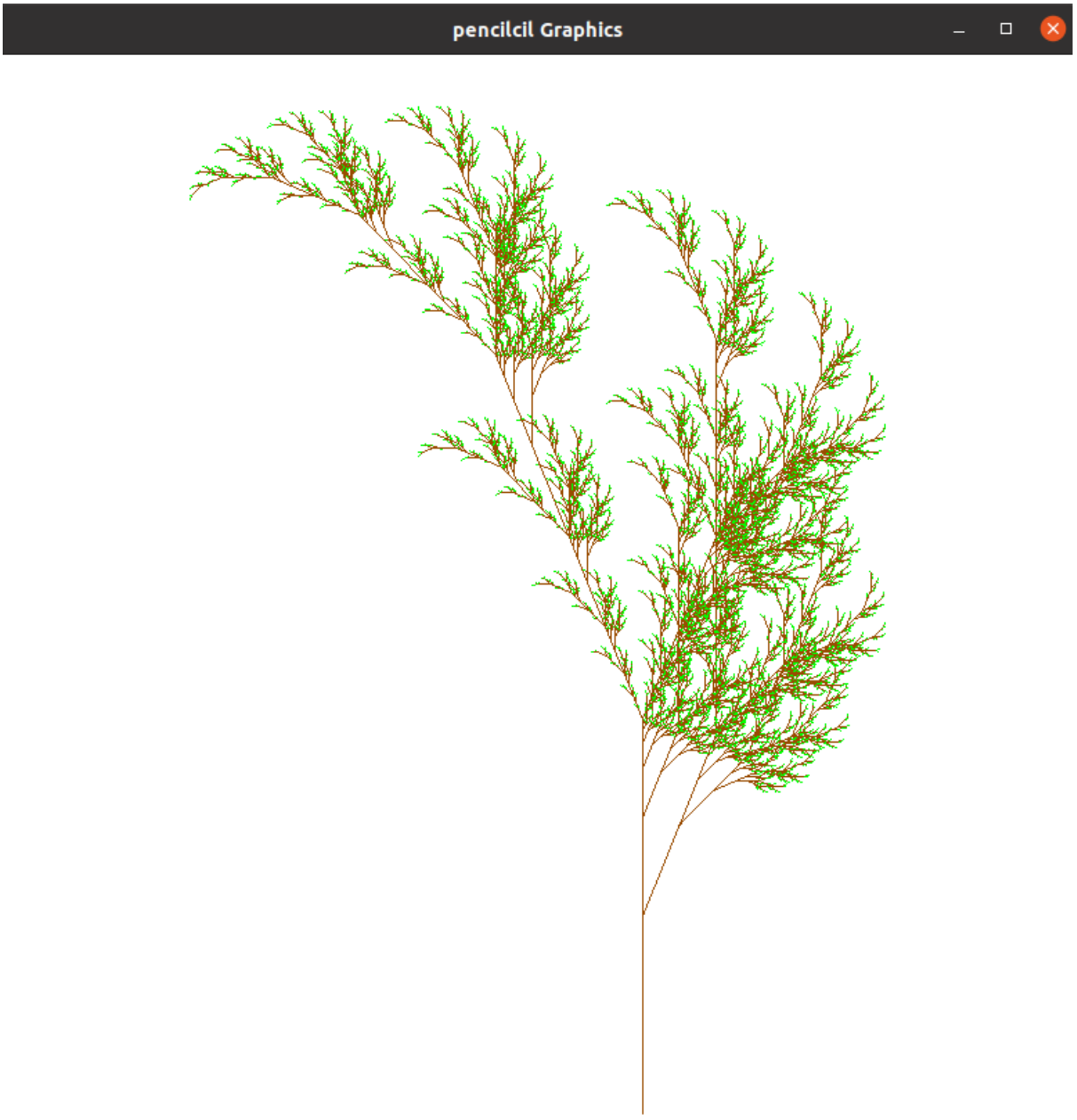
Aquesta funció també utilitza branques, però a més a més conté una branca aniuada, és a dir una branca dins un altre branca, el comportament del codi no cambia gaire respecte el de fulla, ja que només hem d'escriure Branca dins de una Branca i el codi d'execute ja s'encarregar de gestionar-ho tot.

```
branca :: Int -> Comanda
branca n = gBranca n

gBranca :: Int -> Comanda
gBranca 0 = CanviaColor verd :#: Avança 10
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 = CanviaColor marro :#: Avança 10
fBranca n = fBranca (n-1) :#: fBranca (n-1)
```

Exemple d'execució:

```
*Main UdGraphic Artist> branca 1
Color Color' 0.6 0.3 0.0 :#: Avança 10.0 :#: Gira -22.5 :#: Branca (Branca (Co
lor Color' 0.0 1.0 0.0 :#: Avança 10.0) :#: Gira 22.5 :#: Color Color' 0.0 1.0
 0.0 :#: Avança 10.0) :#: Gira 22.5 :#: Color Color' 0.6 0.3 0.0 :#: Avança 10
.0 :#: Branca (Gira 22.5 :#: Color Color' 0.6 0.3 0.0 :#: Avança 10.0 :#: Colo
r Color' 0.0 1.0 0.0 :#: Avança 10.0) :#: Gira -22.5 :#: Color Color' 0.0 1.0
0.0 :#: Avança 10.0
```

Representació gràfica de la branca(utilitzant display(branca 7)):



Jocs de prova

En el fitxer Main.hs hi han s'executa a l'escriure "cabal run" i aquest executa totes les proves que s'han fet fins ara (els pantallasos d'ús que estan a cada problema) i també un programa que et demana escollir quin tipus de fractal es vol representar i depenent del triat et demana els paràmetres pertinents i et mostra el resultat per pantalla fent ús de la funció display, les opcions són les següents:

5. Pentagon
6. Poligon
7. Espiral
10. Triangle
11. Fulla
12. Hilbert
13. Fletxa
14. Branca

Per exemple posem l'execució Opció 14 amb nivell 7 (display(Branca 7)):

Bibliografia

Sessions Haskell Github
Tema 1-8 Haskell(Moodle)
API Haskell