John, l'artista
Pràctica de Programació Funcional

Paradigmes i Llenguatges de Programació

CURS 22/23

Ester Casado Piqueras
Paula Reixach Bonnin

Taula de Continguts

Descripció del problema resolt

Problema:

En aquesta pràctica ens hem hagut de posar en la posició d'en “John”, i aprendre a crear imatges a partir de normes gramaticals, utilitzant un tipus de gramàtica formal que s’utilitza per crear patrons i formes.

Aquesta gramàtica consisteix en un conjunt de regles que descriuen com substituir els caràcters d’una cadena inicial, per altres caràcters (Avança, Gira, Para, Branca…).

A través d’aquesta pràctica, podem arribar a implementar diferents figures de fractals, les quals són unes regles que s’apliquen recursivament, i cada vegada són més complexes, fins i tot poden arribar a formar patrons que s’assemblen a formes naturals, com arbres, plantes,…

Limitacions:

Com a limitacions de l’algorisme, podem dir que, hem hagut de crear una funció auxiliar a *ajunta*, anomenada: *ajunta2.*

Aquesta realitza el mateix que l’original, però sense contemplar els dos casos base, on quan no hi havia cap element a la llista, mostrava un “Para” final o quan era l’últim element de la llista mostrava aquest element, més el Para.

Aquesta implementació extra, l’hem implementada perquè a la funció ***polígon,*** volem ajuntar les diferents comandes replicades com ho fa l’ajunta base, però sense el Para final.

Un altre detall a tenir en compte, en la funció ***pentagon***, l’hem realitzat tal com ho diu a la pràctica:

- “Fent ús de la funció *copia,* implementa la funció pentagon”

Però realitzant la pràctica ens hem adonat compte que també hi ha altres maneres de realitzar-ho, una manera és cridant a la funció ***poligon*****,** amb els paràmetres corresponents.

El resultat és el mateix.

Proposta de millora:

Com a extensió de la pràctica i millora d'aquesta, hem realitzat una nova figura, que es basa en una gramàtica que tingui branques.

A continuació es mostra la seva gramàtica:

extra :: Int -> Comanda

– angle: 35

– inici: f

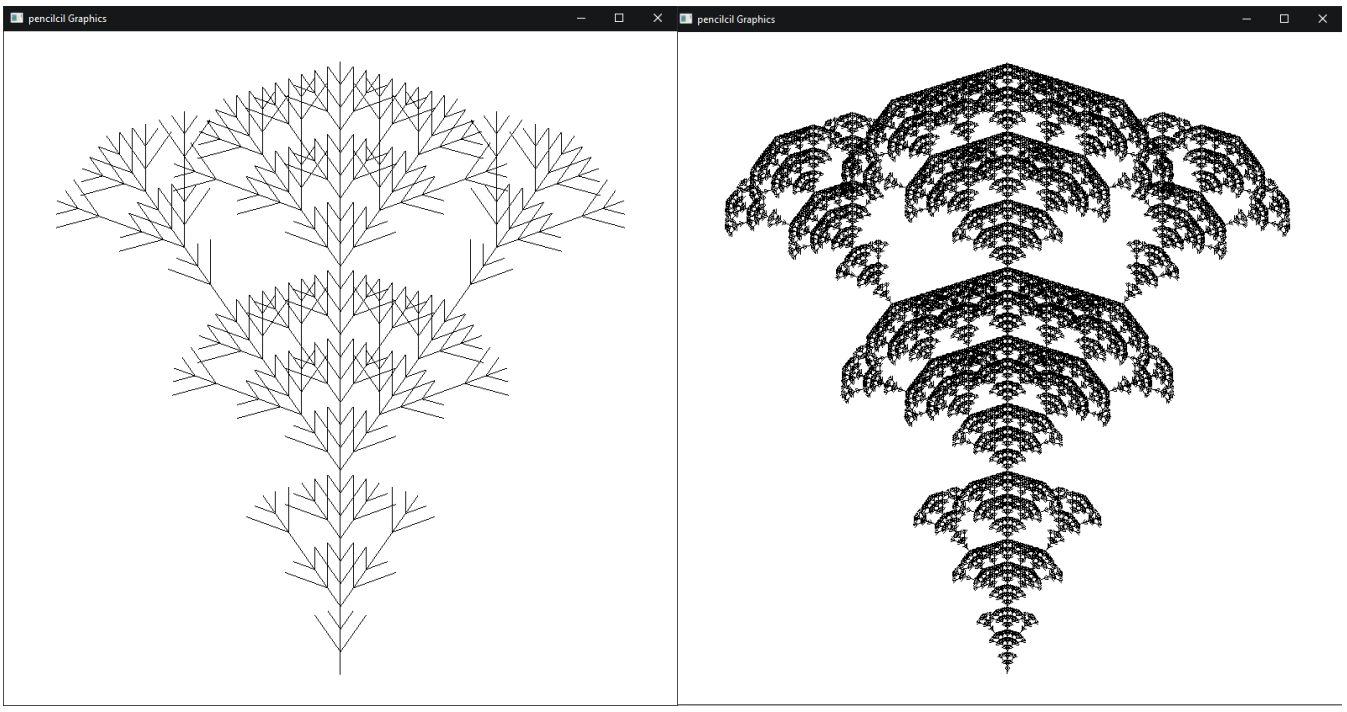
– reescriptura: f[+ff][-ff]f[-f][+f]f

Fragment de codi:

```
extra :: Int -> Comanda
extra n = f n                                   – Defineix la implementació, crida la funció f amb l'argument n
  where giraPos = Gira 35                       – Definició dels dos Gira, amb 35 graus positius i 35 graus negatius
        giraNeg = Gira $ -35
        f 0 = Avança 10                         – El cas base, quan n sigui 0, es realitzarà un Avança 10
        f n = f (n-1) :#: Branca(giraPos :#: f (n-1) :#: f (n-1)) :#: Branca(giraNeg :#: f (n-1) :#:  f (n-1))
                    :#: f (n-1) :#: Branca(giraNeg :#: f (n-1)) :#: Branca(giraPos :#: f(n-1)) :#: f (n-1)
                                                – La regla recursiva de la funció f, sempre que n sigui > 0.
                                                – Aquesta regla, utilitza tant la recursivitat, Branca, i tant GiraPos com
                                                GiraNeg, segons la reescriptura establerta
```

Amb les dues entrades següents obtenim aquestes previsualitzacions:

- **display $ extra 2**
- **display $ extra 5**



Particularitats del codi:

Com a predicats més importants hem considerat els següents, tant pel volum de feina, com per importància en la pràctica: (En cas de voler veure veure qualsevol dels codis en detall, es podran trobar a l'apartat de Codi Comentat)

- **Optimitza:** La finalitat de l'optimitza, és que donada una comanda p, retorni una comanda q, que dibuixi la mateixa imatge, però amb unes propietats en concret.

  En aquesta funció, fem ús de la funció *separa*, i una funció auxiliar per tractar la comanda.

```
optimitza :: Comanda -> Comanda
optimitza comanda = optimitza1 $ separa $ comanda
```

- **Execute:** L'execute, ens permet crear les línies que es passen al display per així poder mostrar-les per pantalla amb els càlculs del dibuix adients.

  Com a particularitat, es podria dir que a part de crear les línies per poder dibuixar correctament la figura, també s'ha implementat que no sempre tingui el mateix color del llapis, és a dir, que es pugui canviar el color amb el qual pintaràs (mostrarà per pantalla), afegint nous casos per a poder fer-ho.
  També s'han hagut de crear nous casos, un cop es va implementar el Branca, ja que s'havia de comportar de manera diferent que la resta de comandes, entre altres problemes que van sorgir.
  Per acabar, una particularitat més de l'execute, és que hem hagut de crear una nova funció auxiliar (execute2) per a poder inicialitzar els valors en què comença el llapis a dibuixar com el punt inicial, el punt final i l'angle.

```
execute :: Comanda -> [Ln]
execute c = execute2 c 0.0 0.0 0
```

- **Ajunta2:** És una funció que hem utilitzat en una part de la pràctica, i ens ha servit per ajuntar una llista de comandes, en una sola comanda i sense que acabi obligatòriament amb la comanda Para.

  Com hem esmentat a l'inici de la pràctica, hem hagut de crear la funció auxiliar Ajunta2, per poder utilitzar-lo en el cas del **poligon.**

  Nou ajunta:

```
ajunta2 :: [Comanda] -> Comanda
ajunta2 [x] = x
ajunta2 (x:xs) = x :#: ajunta2 xs
```

- **Canvi de color:** Considerem que el canvi de color és una particularitat important del codi, vam afegir-lo a la definició de la Comanda com una comanda més del llapis, per així poder pintar les nostres línies de diferents colors.

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

També hem modificat el mòdul de UdGraphics.hs, per tal d'afegir els colors corresponents, declarats en les següents línies (també s'ha afegit el display i l'execute per poder usar-los en l'Artist, com la resta dels que es troben aquí):

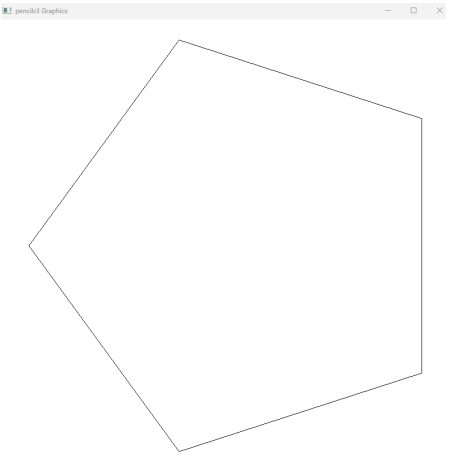
```
module UdGraphic (
    Comanda(..),
    Distancia,
    Angle,
    execute,
    display,
    blau, vermell,
    negre, verd
    )
    where
```

```
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
```

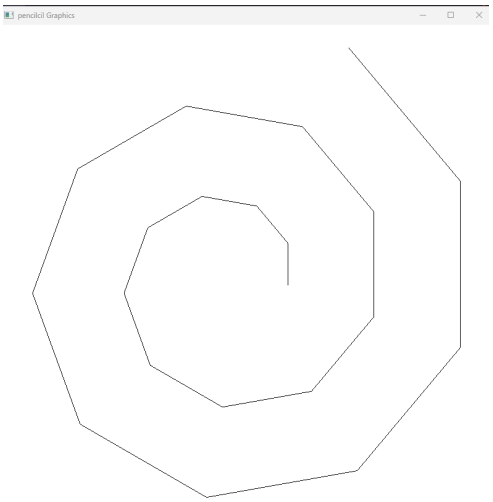
Implementacions:

Hem realitzat totes les proves pertinents, a continuació, en cada execució es mostrarà una breu descripció amb l'entrada i el resultat d'aquesta, amb una captura de pantalla.

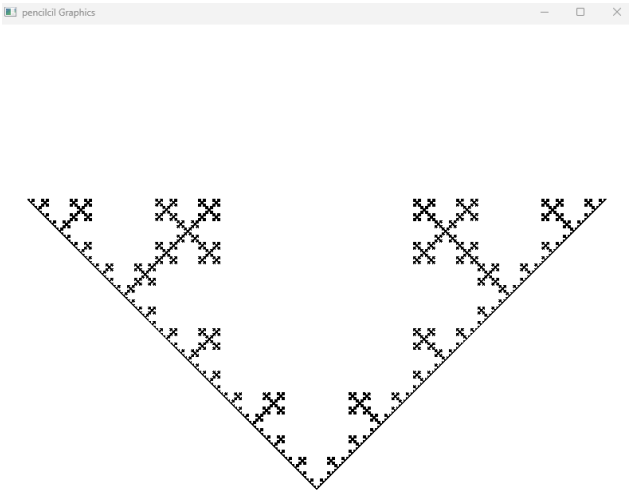
- *Pentagon*: Tots els costats tenen la mateixa mida, en aquest cas, 50.
    - Entrada: **pentagon 50**

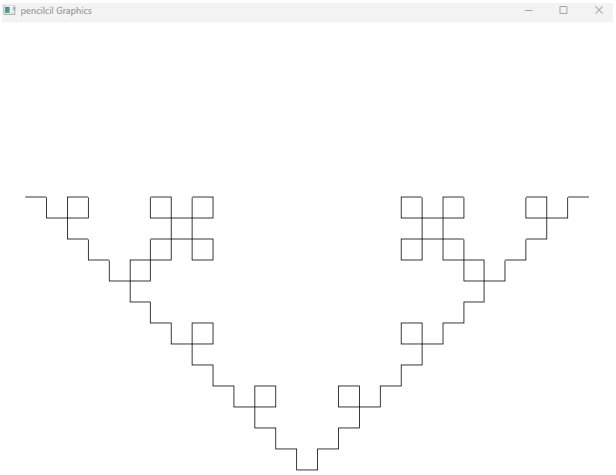


- *Espiral*: Hem provat d'introduir "espiral 30 4 5 30" però el resultat no ens agradava, per tant, hem modificat els números per tal que s'assembli més a un espiral.
    - Entrada: **espiral 30 20 5 40**

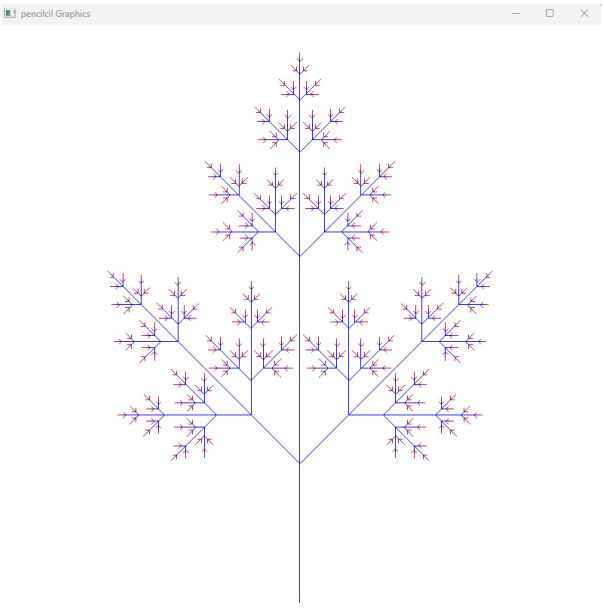


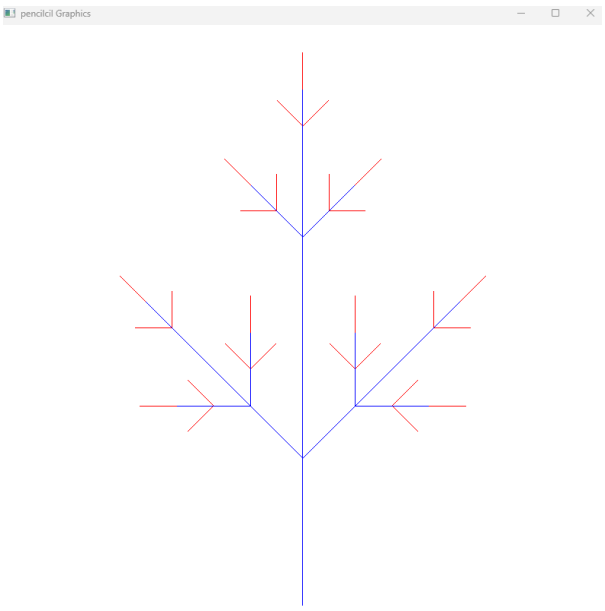
- *Triangle*: Hem realitzat dues execucions, per tal de veure l'avanç en el triangle i com canvia en la seva forma (com es va repetint, la seva seqüència).
    - Entrada primera: **display $ triangle 6**
    - Entrada segona imatge: **display $ triangle 3**





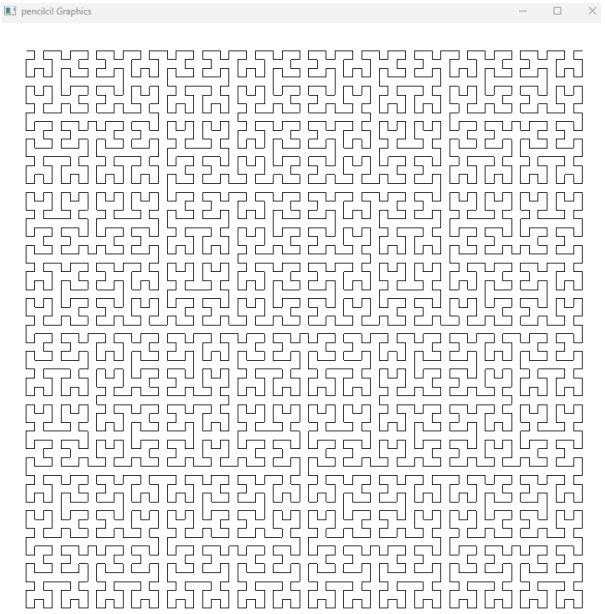
- *Fulla*: Aquesta és la primera execució on implementem el canvi de color, i on s'utilitza també els nous constructors a Comanda (Branca x) i modificació de la funció execute.
    - Entrada primera imatge: **display $ fulla 6**
    - Entrada segona imatge: **display $ fulla 3**

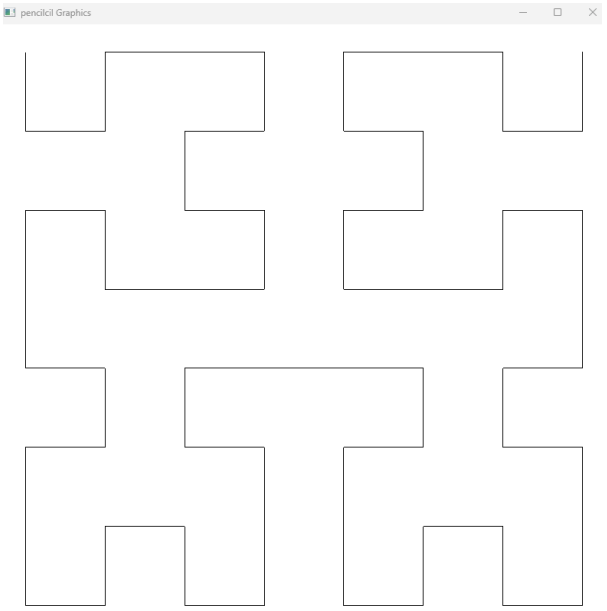




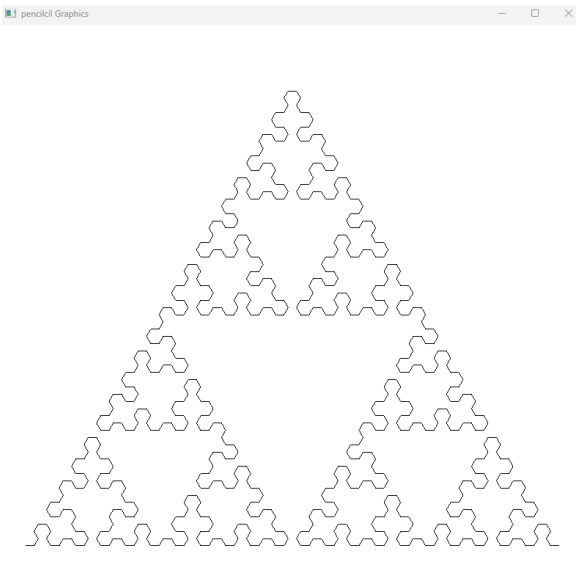
-

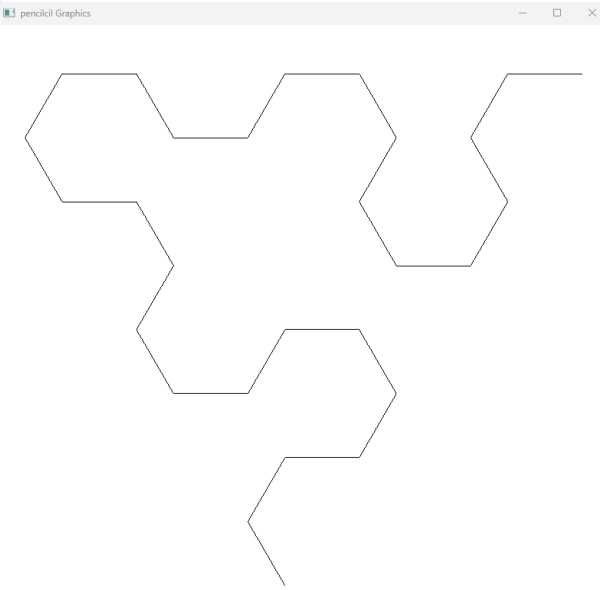
- *Hilbert:* Es pot veure la gran diferència entre posar el número 3 o 6. La corba de Hilbert és autosimilar; cada secció d'un determinat ordre correspon a la corba en l'ordre anterior.
    - Entrada primera imatge: **display $ hilbert 6**
    - Entrada segona imatge: **display $ hilbert 3**



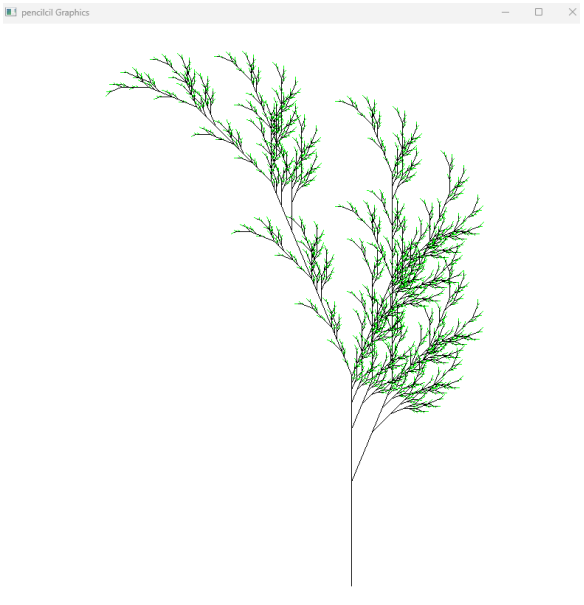


- *Fletxa:* Com en els anteriors, hem representat dos exemples, podem observar com la primera imatge acaba formant un triangle equilàter i en el segon costa una mica més de veure.
    - Entrada primera imatge: **display $ fletxa 6**
    - Entrada segona imatge: **display $ fletxa 3**

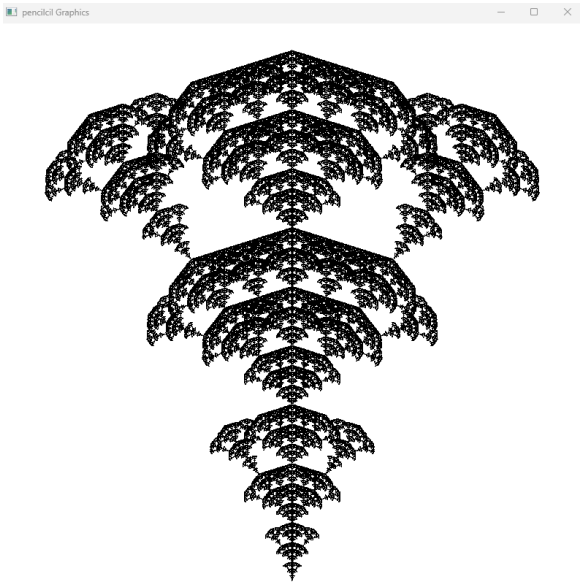


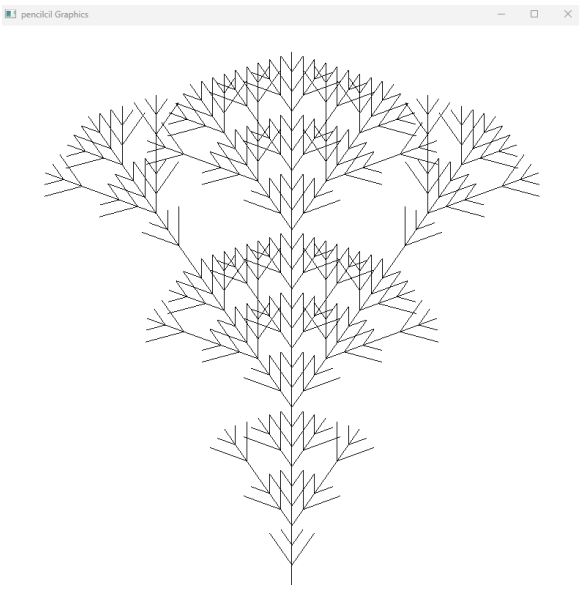


- *Branca:* Es pot observar com a diferència entre les dues imatges de sortida, com la base de la branca és igual, però el que més destaca són les “fulles” i les petites branques del final d’aquesta.
    - Entrada primera imatge: **display $ branca 6**
    - Entrada segona imatge: **display $ branca 3**



- *Extra:* Hem realitzat una gramàtica extra, com hem explicat anteriorment, fent ús de Branca i aquest ha estat el nostre resultat.
    - Entrada primera imatge: **display $ extra 6**
    - Entrada segona imatge: **display $ extra 3**





Codi Comentat:

Hem considerat que tot el fitxer “Artist.hs” és important, ja que l’hem implementat nosaltres en tota la seva totalitat.
Del fitxer “UdGraphics.hs” t’afegim les funcions més rellevants, com “execute”, per no posar-te tot el codi.
(La identació que podem oferir en el document, no és la mateixa que en el fitxer original on es pot veure més clarament el contingut i els comentaris aplicats correctament)

Artist.hs

```haskell
module Artist where

import UdGraphic
import Test.QuickCheck
import Debug.Trace

-- Problema 1
-- Transforma una comanda a una llista de comandas que no compté :#: ni el constructor Para.
separa :: Comanda -> [Comanda]
separa (x :#: xs) = separa x ++ separa xs
-- Aplica el separa a una comanda i la concatena amb el separa de la següent/s comanda/es
separa Para = []
-- Si la comanda es Para retorna una llista buida
separa comanda = [comanda]
-- Si només hi hi ha una comanda ho passa a una llista amb una sola comanda


-- Problema 2
-- Transforma una llista de comandes a una sola comanda
ajunta :: [Comanda] -> Comanda
ajunta [] = Para
-- Ajunta si troba la lista buida ens crea la comanda Para
ajunta [x] = x :#: Para
-- Si només hi ha un element a la llista crea la comanda composta x :#: Para
ajunta (x:xs) = x :#: ajunta xs
-- Agafa el primer element de la llista, el posa com a element de la primera comanda i recursivament fa el
mateix concatenent amb el l'operador d'unió de les comandes, la següent comanda


-- Problema 3
-- Han de retornar la mateixa llista donada dues comandes siguin equivalents (escrites de diferent manera)
prop_equivalent :: Comanda -> Comanda -> Bool
prop_equivalent com1 com2 = separa com1 == separa com2
-- Es separa la primera comanda i es fa el mateix amb la segona comanda passada, i retorna true si les dues
llistes són iguals

-- Ha de mirar que ajunta (separa c) es equivalent a c, on c es una comanda qualsevol
prop_split_join :: Comanda -> Bool
prop_split_join c = ajunta (separa c) == c
-- Separem i ajuntem la comanda passada i comprovem que sigui igual a la que ens han donat

-- Aquesta comanda mira que després de fer el separa no hi hagi cap comanda composta ni el Para, si és així
retorna true
-- All és una funció d'ordre superior que agafa un predicat i una llista i retorna true si el predicat es
cert per a tots els elements de la llista i fals en cas contrari,
```

```haskell
-- És a dir en aquest cas s'utilitza per verificar que totes les subcomandes son simples i no estan unides per :#: no són la comanda Para
prop_split :: Comanda -> Bool
prop_split c = all isSimple $ separa c
-- Separa divideix la comanda, en subcomandes (llista) i verifica mitjançant la funció isSimple que totes les comandes siguin simples i no n'hi hagi cap de composta ni cap que sigui Para
    where isSimple Para = False
-- Comprova que la comanda que s'està comprovant no sigui Para
          isSimple (com1 :#: com2) = False
-- Comprova que la comanda que s'està comprovant no sigui una comanda composta
          isSimple _ = True
-- En cas que no sigui qualsevol de les anteriors, vol dir que és simple, retorna true


-- Problema 4
-- Donat un número n i una comanda, generi una nova comanda amb el nombre n de copies de la comanda
copia :: Int -> Comanda -> Comanda
copia 1 x = x
-- Quan el número de còpies es 1 mostrem, la comanda
copia n x = x :#: copia (n-1) x
 -- Quan el número de còpies és > 1 mostrem la comanda, la concatenem amb l'operador de comandes i apliquem recursivament còpia a la comanda


-- Problema 5
-- Retorna la comanda que genera un pentàgon, tots els costats del pentagon tenen la mateixa mida
pentagon :: Distancia -> Comanda
pentagon d = copia 5 (Avança d :#: Gira 72.0)
-- Fent ús de la comanda copia, aquesta replica 5 vegades el la comanda (Avança d :#: Gira 72.0) amb la distància desitjada


-- Problema 6
-- Aquest problema el que fa és generar la comanda que fa que el llapis traci una ruta amb el nombre de costats especificat, la longitud especificada i l'angle especificat
-- La funció concat uneix varies llistes en una sola llista.
-- En aquest cas, concat unirà les subllistes de comandes retornades pel map en una sola llista
poligon :: Distancia -> Int -> Angle -> Comanda
poligon d x a = ajunta2 $ concat $ map (replicate x) [Avança d :#: Gira a]
-- Amb el map el que fem és usar la funció replicate per aplicar-ho a la llista on hi tenim la comanda a

-- Replicar i aquesta és replicada i concatenada amb la resta de comandes replicades i al final les ajuntem
-- A una sola comanda fent ús de la comanda ajunta

-- S'encarrega de comprovar que la funció anterior funciona correctament, és a dir, ens crea l'estructura de manera correcte i això ho comprovem (amb pentagon) mirant si les dues comandes són iguals
prop_poligon_pentagon :: Distancia -> Int -> Angle -> Distancia -> Bool
prop_poligon_pentagon distPo num angle distPe  = poligon distPo num angle == pentagon distPe
-- Generem les dues comandes, una amb poligon i l'altre amb pentagon i comprovem l'igualtat



-- FUNCIÓ AUXILIAR EXTRA SENSE PARA
-- transforma una llista de comandes a una sola comanda
ajunta2 :: [Comanda] -> Comanda
ajunta2 [x] = x
-- Si només hi ha un element a la llista crea la comanda composta per x
ajunta2 (x:xs) = x :#: ajunta2 xs
```

```
-- Agafa el primer element de la llista, el posa com a element de la primera comanda i recusivament fa el
mateix concatenent amb el l'operador d'unió de les comandes, la següent comanda


-- Problema 7
-- Crea un espiral, fent que el llapis viatgi a distàncies cada vegada més llargues (o més curtes) i girant
una mica entre elles
espiral :: Distancia -> Int -> Distancia -> Angle -> Comanda
espiral dist 1 sum angle = Avança dist :#: Gira angle
-- Quan el número de vegades recursives és 1 mostrem, la comanda composta
espiral dist n sum angle = Avança dist :#: Gira angle :#: espiral (dist+sum) (n-1) sum angle
-- Quan el número de recursions és > 1 mostrem la comanda composta  la concatenem amb l'operador de comandes
i

-- Apliquem recursivament espiral a la comanda però modificant els paràmetres, és a dir,

-- Incrementem la distància i decrementem el número de vegades de la recursió


-- Problema 9
-- Donada una comanda, retorna una comanda que dibuixa la mateixa imatge, però de manera optimitzada
optimitza :: Comanda -> Comanda
optimitza comanda = optimitza1 $ separa $ comanda
-- crida a la funció secundària, fent ús del separa per poder passar-li una llista de comandes com a
paràmetre

optimitza1 :: [Comanda] -> Comanda
-- Retorna la Comanda final de la simplificació que es va fent amb l'acomulació del foldr
optimitza1 comanda = foldr funcio Para comanda
-- usem el foldr amb una funcio creada per nosaltres que es diu funcio (definieix varies regles per combinar
i simplifpicar el resultat), l'inicialitzem amb el Para
  where
    funcio x Para = x
-- Si el segon element es un Para, retorna el primer element
    funcio (Gira 0) (Gira 0) = Para
-- Si els dos elements són Gira 0, ho tranforma amb un Para
    funcio x (Gira 0) = x
-- Si tenim dos elements i el segon es un Para, retorna el primer element
    funcio x (Gira 0 :#: comandes) = (x :#: comandes)
-- Si tenim el primer element i el segon element es un Gira 0 seguit de més comanda/es retorna l'element amb
el seguit de comandes
    funcio (Gira 0) x = x
-- Si el primer element es un Gira 0, retornem el segon element
    funcio (Avança 0) (Avança 0) = Para
-- Si els dos elements són Avança 0, ho tranforma amb un Para
    funcio x (Avança 0) = x
-- Si tenim dos elements i el segon es un Para, retorna el primer element
    funcio x (Avança 0 :#: comandes) = (x :#: comandes)
-- Si tenim el primer element i el segon element es un Avança 0 seguit de més comanda/es retorna l'element
amb el seguit de comandes
    funcio (Avança 0) x = x
-- Si el primer element es un Avança 0, retornem el segon element
    funcio (Avança d) (Avança d1)
-- Els dos elements són Avança amb diferents números
      | (d+d1) == 0 = Para
-- Si la suma d'aquests dona 0, ho convertim amb un Para
      | otherwise = Avança (d+d1)                          -
- En qualsevola altre cas, retornem Avança amb la suma dels dos elements
```

```
    funcio (Avança d) (Avança d1 :#: x)
-- Si el primer element és un Avança i en el segon element tenim un altre Avança i la resta de la comanda
      | (d+d1) == 0 = x
-- Si la suma d'aquests dona 0, retornem la resta de les comandes
      | otherwise = (Avança (d+d1) :#: x)
-- En qualsevola altre cas, retornem Avança amb la suma dels dos elements + la resta de les comandes
    funcio (Gira g) (Gira g1)
-- Els dos elements són Gira amb diferents números
      | (g+g1) == 0 = Para
-- Si la suma d'aquests dona 0, ho convertim amb un Para
      |otherwise = Gira (g+g1)
-- En qualsevola altre cas, retornem Gira amb la suma dels dos elements
    funcio (Gira g) (Gira g1 :#: x)
-- Si el primer element és un Gira i en el segon element tenim un altre Gira i la resta de la comanda
      | (g+g1) == 0 = x
-- Si la suma d'aquests dona 0, retornem la resta de les comandes
      | otherwise = (Gira (g+g1) :#: x )
-- En qualsevola altre cas, retornem Gira amb la suma dels dos elements + la resta de les comandes
    funcio x (Gira g) = (Gira g :#: x)
-- Si el primer element es un element qualsevol, i el segon element es un Gira retorna Gira + l'element
    funcio x (Avança d) = (Avança d :#: x)
-- Si el primer element es un element qualsevol, i el segon element es un Avança retorna Gira + l'element


-- Problema 10
-- Retorna la comanda que representa un triangle, a través de la gramàtica establerta
triangle :: Int -> Comanda
triangle n = giraPos :#: f n
-- Defineix la implementació, comença amb un gira positiu, i crida la funció f amb l'argument n
  where giraPos = Gira 90
-- Definició dels dos Gira, amb 90 graus positius, i 90 graus negatius
        giraNeg = Gira $ -90
        f 0 = Avança 10
-- El cas base de f, quan n sigui 0, es realitzarà un avança 10
        f n = f (n-1) :#: giraPos :#: f (n-1) :#: giraNeg :#: f (n-1) :#: giraNeg :#: f (n-1) :#: giraPos :#:
f(n-1)  -- La regla recursiva de la funció f, sempre que n sigui > 0.

-- Aquesta regla, només utilitza la recursivitat, i els gira negatius i positius


-- Problema 11
-- Retorna la comanda que representa una fulla a través de la gramàtica establerta
fulla :: Int -> Comanda
fulla n = f n
-- Defineix la implementació, crida la funció f amb l'argument n
  where giraPos = Gira 45
-- Definició dels dos Gira, amb 45 graus positius, i 45 graus negatius
        giraNeg = Gira $ -45
        f 0 = CanviaColor vermell :#: Avança 5
-- El cas base de f, quan n sigui 0, es farà un canvi de color a vermell, i un avança 5
        f n = g (n-1) :#: Branca (giraNeg :#: f (n-1))
-- La regla recursiva de la funció f, sempre que n sigui > 0.
                    :#: Branca (giraPos :#: f (n-1))
                    :#: Branca (g (n-1) :#: f (n-1))
-- Aquesta regla, només utilitza la recursivitat, el branca, els gira negatius i positius, l'ús de la funció
g
        g 0 = CanviaColor blau :#: Avança 5
-- El cas base de g, quan n sigui 0, es farà un canvi de color a blau, i un avança 5
```

```
        g n = g (n-1) :#: g (n-1)
-- La regla recursiva de la funció g, sempre que n sigui > 0.

-- Aquesta regla, només utilitza la recursivitat, i la concatenació

-- Problema 12
-- Retorna la comanda que representa un tipus de quadrats a través de la gramàtica establerta
hilbert :: Int -> Comanda
hilbert n = Gira 90 :#: l n
-- Defineix la implementació, comença amb un Gira 90, i crida la funció l amb l'argument n
  where giraPos = Gira 90
-- Definició dels dos Gira, amb 90 graus positius, i 90 graus negatius
        giraNeg = Gira $ -90
        l 0 = Para
-- El cas base de l, quan n sigui 0, es realitzarà un Para
        l n = giraPos :#: r (n-1) :#: f :#: giraNeg :#: l (n-1) :#: f :#: l (n-1) :#: giraNeg :#: f :#: r
(n-1) :#: giraPos       -- La regla recursiva de la funció l, sempre que n sigui > 0

-- Aquesta regla, només utilitza la recursivitat, els gira positius, l'ús de la funció r i l'ús de f, que
representa un avança 10
        r 0 = Para
-- El cas base de r, quan n sigui 0, es realitzarà un Para
        r n = giraNeg :#: l (n-1) :#: f :#: giraPos :#: r (n-1) :#: f :#: r (n-1) :#: giraPos :#: f :#: l
(n-1) :#: giraNeg       -- La regla recursiva de la funció r, sempre que n sigui > 0

-- Aquesta regla, només utilitza la recursivitat, els gira positius, l'ús de la funció l i l'ús de f, que
representa un avança 10
        f = Avança 10
-- Defineix f com Avança 10

-- Problema 13
-- Retorna la comanda que representa una fletxa a través de la gramàtica establerta
fletxa :: Int -> Comanda
fletxa n = Gira 90 :#: f n
-- Defineix la implementació, comença amb un Gira 90, i crida la funció f amb l'argument n
  where giraPos = Gira 60
-- Definició dels dos Gira, amb 60 graus positius, i 60 graus negatius
        giraNeg = Gira $ -60
        f 0 = Avança 5
-- El cas base de f, quan n sigui 0, es realitzarà un Avança 5
        f n = g (n-1) :#: giraPos :#: f(n-1) :#: giraPos :#: g (n-1)
-- La regla recursiva de la funció f, sempre que n sigui > 0

-- Aquesta regla, només utilitza la recursivitat, els gira positius i l'ús de la funció g
        g 0 = Avança 5
-- El cas base de g, quan n sigui 0, es realitzarà un Avança 5
        g n = f (n-1) :#: giraNeg :#: g (n-1) :#: giraNeg :#: f (n-1)
-- La regla recursiva de la funció g, sempre que n sigui > 0

-- Aquesta regla, només utilitza la recursivitat, els gira negatius i l'ús de la funció f

-- Problema 14
--Retorna la comanda que representa una branca a través de la gramàtica establerta
branca :: Int -> Comanda
branca n = g n
-- Defineix la implementació, crida la funció g amb l'argument n
  where giraPos = Gira 22.5
-- Definició dels dos Gira, amb 22.5 graus positius i 22.5 graus negatius
```

```
        giraNeg = Gira $ -22.5
        g 0 = CanviaColor verd :#: Avança 10
-- El cas base de g, quan n sigui 0, es canviarà de color a verd, i es realitzarà un Avança 10
        g n = f (n-1) :#: giraNeg :#: Branca (Branca (g (n-1)) :#: giraPos :#: g (n-1)) :#: giraPos :#: f
(n-1)
                                  :#: Branca (giraPos :#: f (n-1) :#: g (n-1)) :#: giraNeg :#: g (n-1)
-- La regla recursiva de la funció g, sempre que n sigui > 0.
-- Aquesta regla, només utilitza la recursivitat, els gira tant positius com negatius, el branca i l'ús de la
funció f
        f 0 = Avança 10
-- El cas base de f, quan n sigui 0, es realitzarà un Avança 10
        f n = f (n-1) :#: f(n-1)
-- La regla recursiva de la funció g, sempre que n sigui > 0.

-- Aquesta regla, només utilitza la recursivitat i la concatenació

-- angle: 35
-- inici: f
-- reescriptura: f[+ff][-ff]f[-f][+f]f

-- Retorna la comanda que representa una arbre a través de la gramàtica establerta
extra :: Int -> Comanda
extra n = f n
-- Defineix la implementació, crida la funció f amb l'argument n
  where giraPos = Gira 35
-- Definició dels dos Gira, amb 35 graus positius i 35 graus negatius
        giraNeg = Gira $ -35
        f 0 = Avança 10
-- El cas base, quan n sigui 0, es realitzarà un Avança 10
        f n = f (n-1) :#: Branca(giraPos :#: f (n-1) :#: f (n-1)) :#: Branca(giraNeg :#: f (n-1) :#:  f
(n-1))
                    :#: f (n-1) :#: Branca(giraNeg :#: f (n-1)) :#: Branca(giraPos :#: f(n-1)) :#: f (n-1)

-- La regla recursiva de la funció f, sempre que n sigui > 0.

-- Aquesta regla, utilitza tant la recursivitat, Branca, i tant GiraPos com GiraNeg, segons la reescriptura
establerta
```

UdGraphics.hs

```
-- Problema 8
-- Pas de comandes a lines a pintar per GL graphics (veure per pantalla les comandes)
-- genera les linies a pintar per UdGraphic
execute :: Comanda -> [Ln]
execute c = execute2 c 0.0 0.0 0
-- crida a una funció secundaria execute amb la comanda i les inicialitzacions dels punts i l'alngle inicial

-- Funció secundaria
-- Aquesta funció crea les línies  amb els càlculs dels punts interpolats, que mostrarà el display
-- En els següents comenteris quan posem tenim, volem dir com a head de la Comanda
execute2 :: Comanda -> Float -> Float -> Angle-> [Ln]
execute2 (Gira angle :#: xs) x y angl = execute2 xs x y (angl+angle)
-- Si tenim un Gira + la resta de la comanda, incrementa l'angle
execute2 (CanviaColor color :#: Avança dist :#: xs) x y angl =  let xNou =(x+(dist*cos(degToRad (-angl))))
-- Si tenim un Canvi de color i un avança (i la resta de la comanda) això ens indica que hem de canviar el
color del llapis i avançar la línia segons l'angle que teniem
                                                                yNou = (y+(dist*sin(degToRad ((-angl)))))
-- Calculem la x i la y nova dels punts interpolats
                                                            in  [Ln (color) (Pnt x y) (Pnt xNou yNou)] ++
execute2 xs xNou yNou angl
-- Creació de la línia amb el color, el punt anterior i el següent punt (novament calculat) i apliquem
execute a la resta de la comanda amb els nou càlculs

execute2 (Avança dist :#: xs) x y angl =  let xNou =(x+(dist*cos(degToRad (-angl))))
-- Si tenim un Avança, una distància i la resta de la comanda
                                    yNou = (y+(dist*sin(degToRad ((-angl)))))
-- Calculem la x i la y nova dels punts interpolats
                                    in  [Ln (negre) (Pnt x y) (Pnt xNou yNou)] ++ execute2 xs xNou yNou angl
-- Creació de la línia amb el color (negre/default), el punt anterior i el següent punt (novament calculat) i
apliquem execute a la resta de la comanda amb els nou càlculs

execute2 (CanviaColor color :#: Avança dist) x y angl = let xNou =(x+(dist*cos(degToRad (-angl))))
-- Si tenim un Canvi de color i un avança això ens indica que hem de canviar el color del llapis i avançar la
línia segons l'angle que teniem
                                                    yNou = (y+(dist*sin(degToRad ((-angl)))))
-- Calculem la x i la y nova dels punts interpolats
                                                    in  [Ln (color) (Pnt x y) (Pnt xNou yNou)]
-- Creació de la línia amb el color, el punt anterior i el següent punt (novament calculat) i apliquem
execute a la resta de la comanda amb els nou càlculs

execute2 (Avança dist) x y angl = let xNou =(x+(dist*cos(degToRad (-angl))))
-- Si tenim un Avança només
                                  yNou = (y+(dist*sin(degToRad ((-angl)))))
-- Calculem la x i la y nova dels punts interpolats
                                 in  [Ln (negre) (Pnt x y) (Pnt xNou yNou)]
-- Creació de la línia amb el color (negre/default), el punt anterior i el següent punt (novament calculat) i
apliquem execute a la resta de la comanda amb els nou càlculs

execute2 (Branca c :#: xs) x y angl = (execute2 c x y angl) ++ (execute2 xs x y angl)
-- Si tenim una Branca com a comanda significa que a dins tindrem mes comandes per lo tant fem una crida
recursiva de l'execute2 amb la comanda de l'interior de la branca (on s'aniran modificant els valors) i amb
els punts originals també els passem a la resta de la comanda
execute2 (Branca c) x y angl = execute2 c x y angl
-- Si només tenim Branca sola, farem la crida recursiva a la comanda interna de la Branca amb els mateixos
valors
```

```haskell
execute2 (Para :#: xs) x y angl = execute2 xs x y angl
-- Si tenim un Para i la resta de la comanda, ignorarem el Para i aplicarem la recursió a la resta de la comanda "com si no hi fos"
execute2 ((c1 :#: c2) :#: c3) x y angl =  execute2 (c1 :#: c2 :#: c3) x y angl
-- Si tenim més parèntesis perquè tenim una comanda dins una altre comanda (varies comandes (Branca)), aplicarem l'execute2 recursivament a les x comandes obviant els parèntesis
execute2 (_) x y angl = []
-- En cas de tenir qualsevol altre cosa, l'execute2 retornarà una llista buida

-- Pels punts d'interpolació hem fet servir la següent expressions aritmètiques:
-- x' = x1 + r * cos(α + θ)
-- y' = y1 + r * sin(α + θ)

-- Funció que ens permet passar els radiants en format graus
degToRad :: Float -> Float
degToRad degrees = degrees * pi / 180.0
-- Un cop ens han passat els radiants apliquem la fórmula per passar a graus i retornem els graus en tipus Float
```

Bibliografia:

Sessió de lab.1
https://github.com/wilberquito/Paradigms-and-Programming-Languages/blob/master/Haskell/Session1.short.md

Sessió de lab.2
https://github.com/wilberquito/Paradigms-and-Programming-Languages/blob/master/Haskell/Session2.md

Sessió de lab.3
https://github.com/wilberquito/Paradigms-and-Programming-Languages/blob/master/Haskell/Session3.md

Hilbert curve
https://en.wikipedia.org/wiki/Hilbert_curve