# John, l'artista

### Pràctica de Programació Funcional

Paradigmes i Llenguatges de Programació

Curs 22/23

## Introducció

John, un jove aspirant a artista, fascinat per la feina de Scott Draves i el seu ús dels fractals per crear sorprenents patrons, sempre ha estat apassionat per l'art i ha experimentat amb diferents mitjans. Però sent que hi ha alguna cosa especial en la manera que els fractals poden crear patrons intricats i fascinants, així doncs, decideix aprendre més sobre aquests sistemes.
Inspirat per la feina de Draves, John passa innumerables hores estudiant els principis dels fractals i experimentant amb diferents configuracions i paràmetres per generar patrons únics. Ell vol desenvolupar el seu propi estil per crear alguna cosa realment original i única.
Durant aquesta pràctica, vosaltres tindreu el rol d'en John en els seus inicis, i aprendreu a com generar imatges a partir de normes gramaticals. Fareu ús d'un tipus de gramàtica formal que es pot utilitzar per generar patrons i estructures complexes. Això consisteix en un conjunt de regles que descriuen com substituir els símbols d'una cadena inicial per altres símbols, basant-se en certes condicions. A mesura que les regles s'apliquen recursivament, les cadenes resultants es tornen cada vegada més complexes i poden formar patrons intricats que es semblen a formes naturals com ara arbres, plantes i fractals.
Els fractals s'utilitzen sovint en gràfics per ordinador i en modelització per crear imatges i animacions realistes i visualment interessants.

### Què i com ho hem de fer

La pràctica busca que implementeu diferents figures basades en fractals i fer comprovació de codi amb el mòdul QuickCheck.

- La plantilla de la pràctica la trovareu al repositori `https://github.com/wilberquito/JohnTheArtist`
- El projecte necesita dependencies de gràfics i de testeig que s'han d'instal·lar, al README del repositori s'expliquen els passos per poder posar el projecte en marxa, així com les llibreries de C que potser necessiteu.
- Farem servir el codi facilitat en el repositori per poder mostrar per pantalla les figures generades per les comandes. El pas a gràfic està pràcticament implementat a excepció de la funció `execute`, que haurem d'implementar.

## Funcionament del llapis

El llapis, és l'eina amb la qual John fa dibuixos a línies. El llapis té una ubicació determinada al llenç i es manté en una orientació determinada. Una comanda descriu una seqüència d'accions que ha de realitzar el llapis, incloent-hi moure's endavant una distància determinada o girar un angle determinat.
Les comandes per al llapis de John es poden representar en Haskell mitjançant un tipus de dades algebraic.

```
type Angle     = Float
type Distancia = Float
data Comanda   = Avança Distancia
               | Gira Angle
               | Comanda :#: Comanda
               . . .
```

L'última línia declara un constructor de dades infix. Ja hem vist constructors d'aquest tipus en les pràctiques (constructor per les llistes). Mentre que els constructors ordinaris han de començar per una lletra majúscula, els constructors infixos han de començar per dos punts (:). Aquí, hem utilitzat el constructor infix `:#:` per unir dues comandes.
Les comandes que el llapis ha de suportar tenen la següent forma:

- `Avança d`, on `d` representa una distància: mou el llapis a la distància indicada en la direcció que està mirant. (Nota: no s'espera que les distàncies siguin negatives.)
- `Gira a`, on `a` és un angle: gira el llapis en sentit antihorari a través de l'angle donat.
- `p:#:q`, on `p` i `q` són comandes - les dues comandes s'han d'executar en seqüència.
- `Para` - implica fer res. El llapis es queda en la posició i direcció amb la que estava.
- `CanviaColor p`, on `p` és un llapis: canvia a un llapis del color donat. Els següents llapis estan predefinits:

  ```
  blanc, negre, vermell, verd, blau :: Llapis
  ```

  Podeu crear llapisos amb altres colors utilitzant el constructor `Color'`, que pren un valor entre 0 i 1 per a cada un dels components vermell, verd i blau del color. El llapis especial Transparent no fa cap sortida.

- `Branca p`, aquest constructor de comanda ens permetrà fer bifurcacions de qualsevol manera. Sense aquest constructor només podrem dibuixar camins lineals.

  `p` és una comanda: dibuixa el camí donat i després retorna la mà del pintor a la direcció i la posició que tenia al principi del camí (en lloc de deixar-lo al final). Els canvis de llapis dins d'una branca no tenen efecte fora de la branca.

  Per veure l'efecte de la bifurcació, dibuixa el següent camí (ho podràs veure quan tinguis implementat la funció *ajunta*):

  ```
  let inDirection angle = Branca (Gira angle :#: Avança 100)
      in ajunta (map inDirection [20,40..360])
  ```

- `CanviaPunta p` (opcional), on `p` és la punta del llapis i representa el gruix d'un segment dibuixat.

Les comandes:

- `Para`
- `CanviaColor p`
- `Branca p`
- `CanviaPunta p`

No estan en l'esquelet de la pràctica, s'hauran d'implementar al llarg d'aquesta.

Per exemple, per dibuixar un triangle equilàter amb costats de trenta unitats, hem de crear les comandes que fan que el llapis es mogui endavant tres vegades, girant $120^{\underline{o}}$ entre moviments:

```
Avança 30 :#: Gira 120 :#: Avança 30 :#: Gira 120 :#: Avança 30
```

## Mostrar la pintada

Pots veure la pintada del llapis escrivint:

```
Main> display cami
```

on cami és una expressió de tipus Comanda. Això obrirà una nova finestra gràfica i dibuixarà el gràfic fet pel llapis. Per exemple:

```
Main> display (Avança 30 :#: Gira 120 :#: Avança 30 :#: Gira 120 :#: Avança 30)
```

dibuixa el triangle descrit anteriorment. La funció `display` funcionarà un cop implementem la funció `execute`.
Quan tanquis la finestra gràfica, GHCi també sortirà, així que hauràs de tornar a carregar el teu codi per dibuixar una altra imatge.

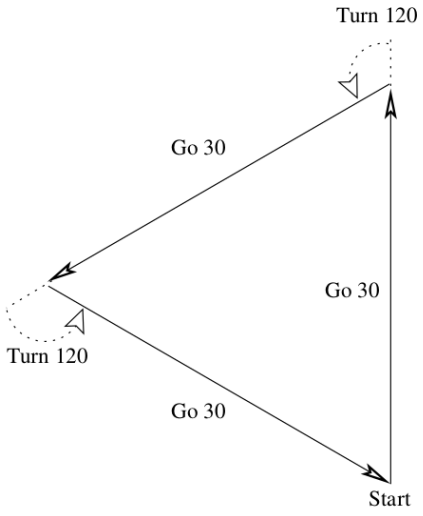


Figure 1: Pintant un triangle amb el llapis

## Equivalencies

Fixeu-vos que el operador `:#:` es asociatiu a la dreta i l'identitat d'una comanda es Para. Per tant:

```
p :#: Para      = p
Para :#: p      = p
p :#: (q :#: r) = (p :#: q) :#: r
```

Podem ometre els parèntesis en expressions amb `:#:` perquè, sigui on siguin posats, el significat permaneix el mateix. Quan digüem que dues comandes són equivalents volem dir que són iguals segons les igualtats llistades anteriorment.
Tanmateix, per avaluar una expressió, Haskell ha de posar parèntesis; si li demanes que et mostri la comanda, et mostrarà on posa els parèntesis.

```
Main> Para :#: Para :#: Para
Para :#: (Para :#: Para)
```

## Problema 1

Implementa la funció

```
separa :: Comanda -> [Comanda]
```

que transforma una comanda a una llista de comandas que no compté `:#:` ni el constructor `Para`. Per exemple,

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

## Problema 2

Implementa la funció

```
ajunta :: [Comanda] -> Comanda
```

que transforma una llista de comandes a una sola comanda. Per exemple,

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

Per tots els exemples, el resultat pot ser qualsevol comanda equivalent.

## Problema 3

Cal tenir en compte que dos comandes són equivalents, en el sentit de les lleis d'equivalència explicades previament, si la funció `separa` retorna el mateix resultat per a ambdues.

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

Implementa la funció

```
prop_equivalent = undefined
```

que testeja que dues comandes són equivalents. Dona el tipus de la funció i la implementació.
Per poder comprovar una propietat, farem us del modul QuickCheck. Per comprovar les propietats d'un tipus de dades es necesari definir aquest tipus com a part de la categoria Arbitrary. La classe Arbitrary proporciona una manera de generar valors aleatoris d'un tipus determinat, la qual cosa resulta útil per als tests basats en propietats amb QuickCheck.
Per poder comparar les propietats feu us de la funció `quickCheck`, per exemple,

```
Test.QuickCheck> quickCheck (prop_equivalent Para Para)
+++ OK, passed 1 test.
```

Implementa dos funcions per comprovar les propietats de les funcions `separa` i `ajunta`. Dona el tipus i l'implementació de les dues funcions. La primera funció `prop_separa_ajunta` ha de mirar que `ajunta (separa c)` es equivalent a `c`, on `c` es una comanda qualsevol. La segona funció `prop_separa` ha de mirar que la llista retornada per la funció `separa` no contingui cap `Para` ni comanda `(:#:)` composta.

## Problema 4

Implementa la funció

```
copia :: Int -> Comanda -> Comanda
```

que donat un número n i una comanda, generi una nova comanda amb el nombre n de copies de la comanda. Les dues comandes següents harien de ser equivalent:

```
Main> copia 3 (Avança 10 :#: Gira 120)
Avança 10.0 :#: Gira 120.0 :#: Avança 10.0 :#: Gira 120.0 :#: Avança 10.0 :#: Gira 120.0
```

## Problema 5

Fent us de la funció copia implementa la funció

```
pentagon :: Distancia -> Comanda
```

que retorna la comanda que genera un pentagon, tots els costats del pentagon tenen la mateixa mida. Les següents comandes són equivalents:

```
Main> pentagon 50
Avança 50.0 :#: Gira 72.0 :#:
Avança 50.0 :#: Gira 72.0 :#:
Avança 50.0 :#: Gira 72.0 :#:
Avança 50.0 :#: Gira 72.0 :#:
Avança 50.0 :#: Gira 72.0
```

## Problema 6

Implementa la funció

```
poligon :: Distancia -> Int -> Angle -> Comanda
```

que ha de generar la comanda que fa que el llapis traci una ruta amb el nombre de costats especificat, la longitud especificada i l'angle especificat. Així, les següents dues comandes haurien de ser equivalents:

```
poligon 50 5 72
pentagon 50
```

Implementa la funció

```
prop_poligon_pentagon = undefined
```

encarregada de testejar la propietat anterior.

## Problema 7

Aproximarem una espiral, fent que el llapis viatgi a distàncies cada vegada més llargues (o més curtes) i girant una mica entre elles. El tipus de la funció es la següent:

```
espiral :: Distancia -> Int -> Distancia -> Angle -> Comanda
```

Els seus paràmetres són:

- `costat`, la longitud del primer segment
- `n`, el nombre de segments de línia a dibuixar
- `pas`, la quantitat per la qual la longitud dels segments successius canvia, i
- `angle`, l'angle a girar després de cada segment.

Per dibuixar una espiral, dibuixem n segments de línia, cadascun dels quals fa un angle amb el segment anterior; la primera línia hauria de ser tan llarg com el paràmetre costat i després cada segment hauria de ser més llarg per pas (o més curt, si el paràmetre pas és negatiu). Cap línia pot tenir distància de segment negatiu...
Per lo tant, les següents comandes són equivalents:

```
espiral 30 4 5 30
Avança 30.0 :#: Gira 30.0 :#:
Avança 35.0 :#: Gira 30.0 :#:
Avança 40.0 :#: Gira 30.0 :#:
Avança 45.0 :#: Gira 30.0
```

## Problema 8

És el moment de veure per pantalla les comandes. Per poder-ho fer, necesitem implementar la funció

```
execute :: Comanda -> [Ln]
```

la qual ha de generar les linies a pintar. El tipus `Ln`, representa una linea amb un color de llapis, un origen i un final.

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

Per ara, la funció `execute` ha de suportar les comandes,

```
Avança dst, Gira angle, Para, (d :#: p) :: Comanda
```

Tots els dibuixos generats pel sistema han de començar a l'origen del canvas. Fixem-nos que a la plantilla també està definit el tipus `Pnt` el qual representa un punt 2d, heu de notar també que el tipus Pnt és instancia dels Num i dels Fractional, per tant, feu servir els operadors definits d'aquestes classes per definir els comportaments de les comandes quan ho veieu necesari.
Exemple de comportament de la funció `execute`.

```
Main> 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))]
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 49.696156 (-3.4729638))
(Pnt 59.658104 (-2.6014063))]
```

## Problema 9

A més de les igualtats que vam veure abans, també podríem considerar les següents:

```
Avança 0                = Para
Avança d :#: Avança e   = Avança (d+e)
Gira 0                  = Para
Gira a :#: Gira b       = Gira (a+b)
```

Per tant, la comanda `Para` és equivalent a `Avança 0` i a `Gira 0`, i qualsevol seqüència d'avançar o girar consecutius es pot reduir a un sol d'avançar o girar (sempre i quan els moviments tinguin una distància no negativa!).
Implementeu la funció:

```
optimitza :: Comanda -> Comanda
```

que, donada una comanda `p`, retorna una comanda `q` que dibuixa la mateixa imatge, però té les següents propietats:

- `q` no conté les comandes `Para`, `Avança 0` o `Gira 0`, llevat que la comanda sigui equivalent a `Para`.
- `q` no conté comandes `Avança` consecutives.
- `q` no conté comandes `Gira` consecutives.

Per exemple:

```
Main> optimitza (Avança 10 :#: Para :#: Avança 20 :#:
Gira 35 :#: Avança 0 :#: Gira 15 :#: Gira (-50))
Avança 30.0
```

Podeu utilitzar les funcions separa i ajunta per facilitar la vostra tasca. (Si la vostra versió de ajunta afegeix una comanda Para, caldrà definir una nova versió que no ho faci.)

## La gramàtica de fractals

Un fractal es construeix a partir d'un patró inicial i un conjunt de regles de reescriptura que s'apliquen de forma recursiva al patró per produir patrons cada vegada més complexos. Per exemple, la Figura 2 es va produir a partir de la gramàtica "triangle":

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

Cada símbol en la cadena generada per la gramàtica representa una comanda de camí: aquí, `+` i `-` representen rotació en sentit horari i antihorari i `f` representa un moviment cap endavant. Quins símbols representen quines comandes és una qüestió de convenció. En aquest gramàtica, només el símbol `f` és reescrit, mentre que els símbols `+` i `-` no és reescriuen. La reescriptura reemplaça les línies rectes per figures més complexes.

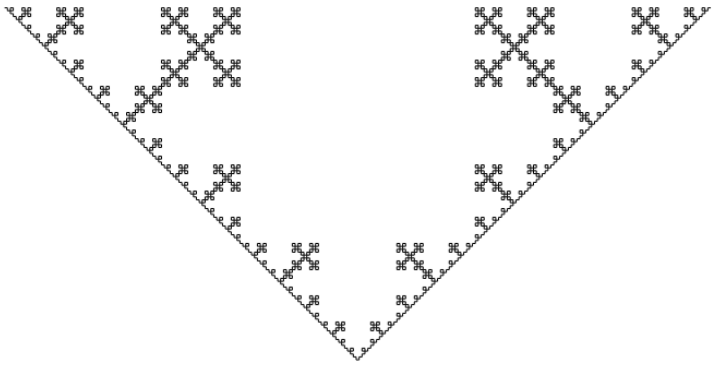

Figure 2: Sortida de la gramàtica triangle

Per exemple, aplicant la regla anterior tres vegades dona les següents cadenes:

```
Pas Patro
0   +f

1   +f+f-f-f+f

2   +f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f

    +f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f
    +f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f
3   -f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f
    -f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f
    +f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f
```

## Problema 10

Implementa la funció

```
triangle :: Int -> Comanda
```

i mostra el resultat per pantalla amb la següent gramàtica:

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

## Branques i colors

Fins ara només hem pogut dibuixar camins lineals; no hem pogut bifurcar el camí de cap manera, i hem estat pintant sempre d'un mateix color. A partir d'ara, farem servir dos constructors de comandes addicionals.

```
data Comanda = . . .
             | CanviaColor Llapis
             | Branca Comanda
```

on Llapis és defineix com:

```
data Llapis = Color' Float Float Float
            | Inkless
```

## Problema 11

Implementa la funció fulla

```
fulla :: Int -> Comanda
```

que te com a gramàtica:

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

el significat dels símbols de les gràmatiques es basen en l'explicació de la secció *La gràmatica de fractals*. En aquesta gramàtica s'introdueixen simbols nous `[` i `]`, aquests representen inici de branca i final de branca respectivament.
El resultat de l'implementació de la gramàtica ha de ser semblant a la Figura 3. Noteu que per poder implementar aquesta funció necessiteu ampliar el tipus `Comanda` amb nous constructor i donar comportament a aquests constructors en la funció `execute`.

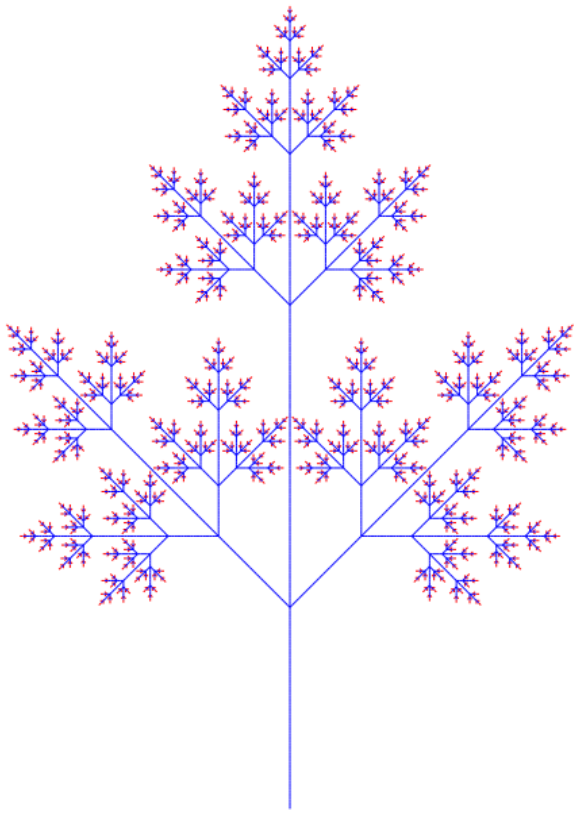

Figure 3: Sortida de la gramàtica fulla

## Problema 12

Escriu la funció

```
hilbert :: Int -> Comanda
```

amb la següent gràmatica:

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

**Nota**: No tots els símbols aquí fan moure el llapis. Comproveu el vostre resultat amb les imatges a `http://en.wikipedia.org/wiki/Hilbert_curve` i ajusteu els valors finals (p. ex. `r 0 = ...`) fins que s'assembli.

## Problema 13

Escriu la funció

```
fletxa :: Int -> Comanda
```

amb la següent gràmatica:

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

## Problema 14

Escriu la funció

```
branca :: Int -> Comanda
```

amb la següent gràmatica:

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

## Consideracions sobre l'avaluació i propostes de millores

Es valorarà que feu les implementacions que es demanen, que el codi sigui clar i lo més funcional possible. No cal que utilitzeu llibreries de tercers, podeu fer ús però el codi dels mòduls propis del llenguatge.
Les següents millores permeten arrodonir al 10 o recuperar nota d'algun problema no resol o no resol correctament.

1. Podeu fer que el llapis pinti una nova figura basada en una gramàtica que tingui branques?
2. Podeu crear el tipus `Gramàtica`, amb la qual podem representar de forma simbòlica una gramàtica de fractals? Recordeu que una gramàtica de fractals està composta de:

   - `n`: nombre de repeticions
   - `angle`
   - `inici`
   - `reescriptura`

   Utilizeu el tipus `Gramàtica` per mostrar una figura.
3. Podeu fer que la traçada del llapis tingui diferents grossors `(CanviaPunta p)`, similar a la manera en què fem que el llapis pinti amb diferents colors. Implementa una nova o modifica alguna gramàtica existent i mostra-ho.

   Per poder resoldre aquest problema, primer heu d'entendre que és una Mónada. El terme Mónada ve de les matemàtiques, la idea que hi ha darrera de les mónades es encapsular els *side effects* o representar computació basades en estats (recordeu que les funcions en Haskell són pures i tampoc tenim canvis d'estats). La Mónada amb la que traballareu és una Mónada basada en estats, la podeu trobar al codi font del repositori d'OpenGL `https://github.com/haskell-opengl/OpenGL/blob/f7af8fe04b0f19c260a85c9ebcad612737cd7c8c/src/Graphics/Rendering/OpenGL/GL/LineSegments.hs#L81`.

## Lliurament

- Les pràctiques S'HAN de fer en equips de dos.
- S'haurà de lliurar un document pdf amb el codi comentat i les execucions que es demanen mostrant els resultats obtinguts (captures de pantalla). En concret el document caldrà que tingui els següents apartats:

  1. Descripció del problema resolt i abast de la vostra solució: breument definir el problema que heu resolt, incloent-hi les possibles extensions o limitacions de la vostra solució.
  2. Codi comentat: codi breument comentat, ben indentat i llegible.
  3. Particularitats del codi: descripció detallada dels predicats més rellevants.
  4. Implementacions: descripció de les implementacions i "pantallassos" d'us.
  5. Referències de bibliografia de les funcions o moduls que no hagin vist a classe.

- Entrega del projecte GitHub juntament amb la documentació.
- Les entregues molt probablement seran presencials per poder avaluar la implicació en la pràctica per part dels membres de l'equip. També es possible que per acabar d'avaluar se us faci fer alguna modificació de la pràctica in situ.

- Com a indicacions de puntuació: El problema 9 val 2 punts, El problema 8 i 11 valen 1 punt. La resta d'exercicis tenen el mateix pes.
- Data de lliurament (pel moodle): 04 de juny.