Explorando la Geometría en Educación Secundaria con los Gráficos de la Tortuga

Rosario Vera Ruiz

I.E.S. Jarifa

Cártama, Málaga

Spain

jarifacharovera@hotmail.com

RESUMEN

Desde el punto de vista de la programación funcional, un proceso de cómputo se describe como una función matemática que a partir de unos argumentos (correspondientes a los datos del problema) produce un resultado. Los gráficos de la tortuga permiten describir gráficos dirigiendo los movimientos de una tortuga virtual, la cual deja un rastro correspondiente a su movimiento sobre un plano. En este trabajo presentaremos una librería funcional para los gráficos de la tortuga que hemos desarrollado para el lenguaje de programación funcional Haskell. Mostramos, a través de ejemplos, cómo se puede usar esta librería para ilustrar gráficamente distintos conceptos matemáticos, fundamentalmente del área de geometría, que forman parte del currículo de ESO y bachillerato.

Palabras Clave
Competencia digital, Geometría, gráficos por ordenador, programación funcional.

1. Introducción

La programación funcional es un estilo de programación alternativo al paradigma de programación más habitual, conocido como estilo imperativo. Desde el punto de vista funcional, un proceso de cómputo se describe como una función matemática que a partir de unos argumentos (correspondientes a los datos del problema) produce un resultado. En este trabajo usaremos Haskell [1], un lenguaje de programación funcional puro. Este lenguaje surge originalmente en la década de los 90 con objeto de recoger de modo coherente los distintos avances que se habían producido en el campo de la programación funcional. El lenguaje se ha convertido en un estándar internacional para la docencia e investigación en este campo y ha pasado por distintas revisiones, siendo la versión más actual Haskell 2010, la cuál ha sido presentada recientemente. Existe abundante bibliografía sobre programación funcional con Haskell. El lector interesado puede consultar las referencias [2,3,4,5].

Comparados con los programas imperativos, los funcionales resultan más concisos y fáciles de entender y se expresan con una notación muy cercana a la notación matemática habitual. Es por ello que consideramos estos lenguajes más adecuados para presentar conceptos propios de la programación de ordenadores a alumnos y alumnas de

secundaria, contribuyendo además a que entiendan la importancia del uso de la notación formal matemática.

Con objeto de ilustrar la sintaxis de Haskell, mostramos a continuación una función para el cálculo del máximo común divisor de dos números usando el algoritmo de Euclides:

```
mcd :: Int -> Int -> Int
mcd x 0        = x
mcd 0 y        = y
mcd x y
  | x >= y     = mcd (x-y) y
  | otherwise  = mcd x (y-x)
```

Se trata de una definición recursiva en las que las dos primeras ecuaciones capturan, mediante el uso de patrones, los casos en los que uno de los números es cero. La tercera ecuación es una definición condicional, en la que se distinguen dos casos según se verifique o no la condición $x \geq y$. Nótese el uso de la notación parcializada (*currying* en inglés) según la cual los distintos argumentos van separados por espacios, la aplicación de funciones tiene máxima prioridad y se reserva el uso de paréntesis para agrupar expresiones compuestas.

Los gráficos de la tortuga[6] fueron introducidos originalmente por Seymour Papert en el lenguaje de programación LOGO[7]. Básicamente, permiten describir gráficos dirigiendo los movimientos de una tortuga virtual, la cual deja un rastro correspondiente a su movimiento sobre un plano. El modelo geométrico en el que se basan estos gráficos es euclídeo, ya que se usan traslaciones y rotaciones relativas a la posición actual de la tortuga, lo que permite al estudiante describir un gráfico como si estuviese inmerso en él, facilitando de este modo la descripción del mismo.

En este trabajo presentaremos una librería funcional para los gráficos de la tortuga que hemos desarrollado para el lenguaje de programación funcional Haskell. En primer lugar, describiremos las distintas funciones que constituyen el interfaz de nuestra librería. Posteriormente, mostraremos a través de ejemplos cómo se puede usar ésta para ilustrar gráficamente distintos conceptos matemáticos, fundamentalmente del área de geometría, que forman parte del currículo de ESO y bachillerato.

El objetivo de este trabajo es doble: por un lado, contribuir al desarrollo de la competencia digital, en tanto y cuanto estamos trabajando con el ordenador, utilizándolo en este caso, como una herramienta gráfica que ayuda a que nuestro alumnado visualice mucho mejor el significado de los movimientos en el plano y en el espacio. Por otro lado pretendemos con esta actividad intentar desterrar de la cabeza de nuestros alumnos la visión de que las matemáticas son algo meramente académico. Les mostramos que conceptos matemáticos, en este caso de geometría, son la base para la síntesis de imágenes por ordenador, y por tanto resultan esenciales para el desarrollo de materiales audiovisuales cotidianos, como pueden ser los vídeo-juegos, películas de animación, etc.

2. Una Librería para los Gráficos de la Tortuga en Haskell

En esta sección describimos las distintas funciones y tipos que constituyen la interfaz de nuestra librería para gráficos de la tortuga en Haskell. Primero describimos las primitivas básicas para cada una de las acciones que puede realizar la tortuga y luego describimos funciones para poder combinarlas.

2.1 Interfaz Básico para los Gráficos de la Tortuga

La tortuga queda en todo momento caracterizada por una serie de valores que constituyen lo que denominaremos su *estado*. Este estado incluye, entre otras características, la posición actual de la tortuga en la pantalla y el ángulo en que se orienta. Inicialmente, la tortuga se encuentra situada en el centro de la pantalla y se encuentra orientada hacia los valores positivos del eje de abscisas. La tortuga puede realizar distintas acciones, las cuales se realizan mediante distintas funciones de la librería. La función

```
derecha :: Grados -> Acción
```

cambia la orientación de la tortuga, haciéndola girar un número de grados en el sentido de giro de las agujas del reloj. Para girar la tortuga en el sentido contrario se dispone la función

```
izquierda :: Grados -> Acción.
```

En ambas funciones, el tipo `Grados` es un número real que se corresponde con grados sexagesimales. Ambas funciones están relacionadas por la siguiente propiedad:

```
∀ α :: Grados . derecha α = izquierda (-α)
```

La tortuga está equipada con un lápiz. Este lápiz puede levantarse o bajarse usando las siguientes funciones:

```
levanta :: Acción
baja    :: Acción
```

Las funciones

```
color   :: Color -> Acción
ancho   :: Ancho -> Acción
```

permiten modificar el color y el grueso de la mina del lápiz.

La función

```
avanza :: Unidades -> Acción
```

hace que la tortuga avance un número de unidades en el plano desde su posición actual en la dirección de su orientación. Si el lápiz esta bajado, la tortuga dejará un rastro con el color y grueso actual del lápiz al desplazarse por el plano. Si el lápiz está levantado, la tortuga no dejará rastro al desplazarse; en cualquier caso, la nueva posición de la tortuga será la posición alcanzada tras el desplazamiento. Inicialmente, el lápiz de la tortuga es de color negro, está bajado y tiene un ancho de una unidad.

Para hacer que la tortuga retroceda (es decir, avance en dirección contraria a su orientación) es posible usar la función

```
atrás :: Unidades -> Acción
```

La relación de esta función con avanza es:

```
∀ n :: Unidades . atrás n = avanza (-n)
```

2.2 Composición secuencial y paralela de acciones

Con objeto de poder describir gráficos de cierta complejidad, es necesario poder combinar las acciones primitivas anteriores para definir acciones compuestas. Para ello,

la librería proporciona distintos combinadores. En primer lugar, disponemos del operador

```
(|>) :: Acción -> Acción -> Acción
```

que permite construir una acción compuesta secuenciando dos acciones. Así, la tortuga, a partir de su estado actual, realizará la primera acción y alcanzará un estado intermedio a partir del cual realizará la segunda acción para llegar a su estado final.

La acción

```
nada :: Acción
```

no realiza ninguna acción y es el elemento neutro del operador `(|>)`:

```
∀ ac :: Acción . ac |> nada = nada |> ac = ac
```

por lo que es útil al definir comportamientos recursivos de la tortuga.

Una versión general del operador `(|>)` que permite secuenciar tantas acciones como sea necesario la proporciona la función

```
sec :: [Acción] -> Acción
```

que toma una lista con las acciones a realizar como parámetro, y cuyo comportamiento es:

$$\texttt{sec [}ac_1\texttt{, }ac_2\texttt{, …, }ac_n\texttt{] = }ac_1\texttt{ |> }ac_2\texttt{ |> … |> }ac_n$$

Un concepto fundamental a la hora de describir algoritmos es el de repetición. En un lenguaje funcional como Haskell, la repetición de un proceso puede realizarse mediante el uso de la recursión. Dado que esto es bastante habitual, nuestra librería proporciona la función

```
repite :: Int -> [Acción] -> Acción
```

para poder repetir una secuencia de acciones tantas veces como sea necesario. Partiendo de un estado inicial, se realizan secuencialmente las distintas acciones de la lista, alcanzando un estado intermedio. A partir de este nuevo estado, se vuelven a realizar todas las acciones de la lista, y este proceso se repite hasta alcanzar el número de repeticiones especificado.

La librería también proporciona la función

```
para :: [a] -> (a -> [Acción]) -> Acción
```

que toma una lista de valores y una función, que dado un valor realiza una acción. La función será invocada con cada uno de los valores de la lista, y las acciones así obtenidas se realizarán de modo secuencial. Por ejemplo,

```
para [30, 60, 90] (\alfa -> [izquierda alfa, avanza 100])
```

es equivalente a

```
sec [ izquierda 30, avanza 100, izquierda 60, avanza 100
    , izquierda 90, avanza 100 ].
```

A veces es necesario realizar distintas acciones desde un mismo estado de la tortuga. Para poder especificar esto usando la secuenciación de acciones sería necesario volver al estado inicial tras realizar cada una de las acciones. Esto complicaría excesivamente la descripción de gráficos, ya que gran parte del programa no estaría relacionado con el gráfico a dibujar, sino con restablecer el estado inicial de la tortuga después de realizar

una serie de acciones. Con objeto de poder describir este tipo de tareas de un modo más conveniente, la librería introduce el siguiente operador

```
(<|>) :: Acción -> Acción -> Acción
```

que permite realizar dos acciones *en paralelo*. Para ello, partiendo del estado inicial, se crean dos tortugas independientes y cada una de ellas lleva a cabo cada una de las acciones. Cada tortuga así creada tiene su propio estado, de modo que las acciones que realiza una tortuga no afectan a la otra. También se proporciona una generalización de este operador mediante la función

```
par :: [Acción] -> Acción
```

que toma una lista de acciones, y realiza cada una de ellas con una tortuga independiente, es decir

$$\texttt{par [ac}_1\texttt{, ac}_2\texttt{, …, ac}_n\texttt{] = ac}_1 \texttt{ <|> ac}_2 \texttt{ <|> … <|> ac}_n \texttt{ .}$$

Nótese que la función `nada` también es elemento neutro del operador `(<|>)`:

```
∀ ac :: Acción . ac <|> nada = nada <|> ac = ac .
```

3. Ejemplos de uso de la librería

En esta sección presentamos distintos ejemplos de gráficos que se pueden crear usando la librería de la tortuga.

3.1 Polígonos regulares

Como primer ejemplo del uso de las funciones de nuestra librería, veamos cómo pintar un cuadrado de lado `l`, repitiendo cuatro veces el proceso consistente en avanzar `l` unidades y girar 90 grados:

```
cuadrado :: Unidades -> Acción
cuadrado l = repite 4 [ avanza l, derecha 90 ]
```

Con objeto de establecer algunas propiedades para el gráfico, definimos la siguiente función:

```
conMiEstilo :: Acción -> Acción
conMiEstilo ac = sec [ color negro, ancho 10, relleno rojo, ac ]
```

que selecciona un lápiz negro de 10 unidades de ancho y establece como color de relleno el rojo antes de realizar la acción que toma como parámetro. El gráfico obtenido para la expresión `conMiestilo (cuadrado 100)` se muestra en la figura 1.a.

Figura 1. Polígonos regulares. **a)** Cuadrado. **b)** Triángulo. **c)** Octógono.

De modo similar, podemos definir la siguiente función para pintar triángulos equiláteros de lado `l`:

```
triángulo :: Unidades -> Acción
triángulo l = repite 3 [ avanza l, derecha 120 ]
```

cuyo resultado se muestra en la figura 1.b. Generalizando ambos ejemplos, obtenemos la siguiente función:

```
polígono :: Int -> Unidades -> Acción
polígono n l = repite n [ avanza l, derecha alfa ]
  where alfa = 360 / fromIntegral n
```

para pintar polígonos regulares con `n` lados de longitud `l`. En el ejemplo, `fromIntegral` es una función que convierte valores enteros en reales y `alfa` es una variable que depende del número de lados del polígono. (Obsérvese que el ángulo ente cada dos lados del polígono es 180 - `alfa`). La figura 1.c. muestra el octógono resultado de evaluar la expresión `conMiEstilo (polígono 8 50)`.

Aunque la librería de la tortuga no proporciona ninguna primitiva para pintar curvas, es posible simular un círculo pintando un polígono con 360 lados de longitud 1:

```
círculo :: Acción
círculo = repite 360 [ avanza 1, derecha 1 ]
```

Con esta expresión obtenemos un círculo con unas dimensiones fijas, pero siguiendo el mismo razonamiento, y teniendo en cuenta que la longitud de la circunferencia es $2\pi \cdot r$, siendo `r` el radio del círculo, podemos pintar un círculo con el radio que queramos:

```
círculoRadio :: Unidades -> Acción
círculoRadio r = polígono 360 l
 where l = 2*pi*r/360
```

También podemos dibujar un pétalo enlazando dos arcos:

```
pétalo :: Acción
pétalo = repite 2 [ arco, derecha 90 ]
 where arco = repite 90 [ avanza 2 , derecha 1 ]
```

La Figura 2 muestra los gráficos obtenidos con estos ejemplos.

Figura 2. **a)** Círculo. **b)** Pétalo.

3.2 Estrellas

Mostramos ahora una función para dibujar una estrella regular con `n` puntas, dadas la longitud del lado de una punta (`l`) y el ángulo (α) que forman los vértices que constituyen una punta:

```
estrella :: Int -> Unidades -> Grados -> Acción
estrella n l alfa = repite n  [ avanza l, derecha beta
                              , avanza l, izquierda gamma
                              ]
    where  beta = 180 - alfa
           gamma = beta - 360/fromIntegral n
```

Los distintos ángulos utilizados están detallados en la Figura 3.

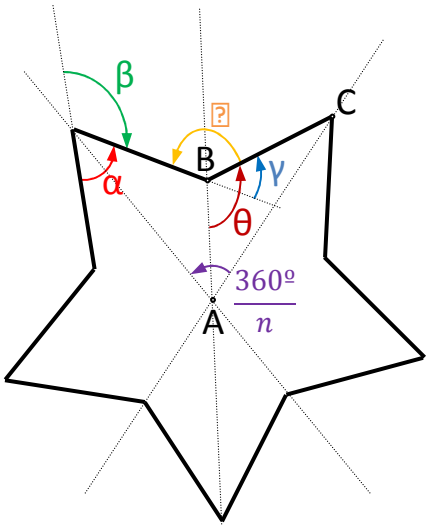

Figura 3. Ángulos para dibujar una estrella regular.

Por el triángulo ABC, tenemos:

$$\frac{360^{\circ}}{2n} + \frac{\alpha}{2} + \theta = 180^{\circ}$$

Mientras que los ángulos alrededor del punto B verifican:

$$\varphi + 2\theta = 360^{\circ}$$

y teniendo en cuenta que:

$$\gamma = 180^{\circ} - \varphi \qquad\qquad \beta = 180^{\circ} - \alpha$$

Obtenemos:

$$\gamma = 180^{\circ} - \alpha - \frac{360^{\circ}}{n} = \beta - \frac{360^{\circ}}{n}$$

que es la fórmula finalmente utilizada en la función `estrella`. La Figura 4 muestra distintas estrellas de cinco puntas obtenidas con dicha función para distintos valores del ángulo α.

Figura 4. Estrellas de cinco puntas. **a)** α=30. **b)** α=45. **c)** α=60.

3.3 Dibujos artísticos

Utilizando las primitivas que proporciona nuestra librería, es fácil definir funciones para obtener dibujos artísticos. Como primer ejemplo, veamos una función para dibujar los aros olímpicos:

```
olímpicosRadio :: Unidades -> Acción
olímpicosRadio r = sec [ transparencia 0.90, ancho 10
                       , para [ azul, negro, rojo ] aro
                         <|>
                         sec [ derecha 90, adelanta r
                             , izquierda 90, adelanta r
                             , para [ amarillo, verde ] aro
                             ]
                       ]
      where  aro c = [ color c, círculoRadio r, adelanta d ]
                 where d = 2*r + r/3


adelanta :: Unidades -> Acción
adelanta n = sec [ levanta, avanza n, baja ]

retro :: Unidades -> Acción
retro n  = sec [ levanta, atrás n, baja ]
```

son dos funciones que permiten trasladar la tortuga a otra posición sin que ésta deje rastro, y `transparencia` hace que el pincel de la tortuga no sea totalmente opaco. Obsérvese cómo se crean dos tortugas para pintar cada una de las filas de aros. El resultado de este ejemplo para un radio 60 aparece en la Figura 5.a.

El siguiente ejemplo muestra también la potencia de poder crear tantas tortugas como sea necesario:

```
banderas :: Acción
banderas =
 par [ bandera alfa c
     | (alfa,c) <- zip [0,30..360]
                   (cycle [rojo,verde,azul,amarillo])
     ]
  where
    bandera alfa c = sec [ izquierda alfa, ancho 5, relleno c
                         , color negro, avanza 150, cuadrado 60
                         ]
```

Se crean distintas tortugas para pintar banderas de distintos colores giradas cierto ángulo. El resultado puede verse en la Figura 5.b. Nótese el uso de una secuencia

aritmética de Haskell (la lista `[0,30..360]`) para generar ángulos que se diferencien en 30º, de la función `cycle` para obtener tantos colores como sean necesarios siguiendo el patrón `[rojo,verde,azul,amarillo]` y la función `zip` para emparejar ambas listas.

Por último, la función `flor`, genera el gráfico que aparece en la Figura 5.c.

```
flor :: Acción

      flor = sec [ izquierda 90, ancho 10, avanza 200       -- tallo
                 , relleno verde, derecha alfa, pétalo
                 , derecha 180, pétalo                      -- hojas
                 , derecha (alfa+90), avanza 200            -- tallo
                 , relleno amarillo, izquierda 90, círculo -- corona
                 , derecha 90, adelanta 60, relleno rojo, pétalos
                 ]
       where
        alfa = 45
        pétalos = repite n [ izquierda beta, adelanta 30
                           , pétalo, retro 30
                           ]
             where n = 9
                   beta = 360 / fromIntegral n
```

Simplemente se trata de dibujar las distintas partes de la figura. Obsérvese que la variable `n` define el número de pétalos en la flor.

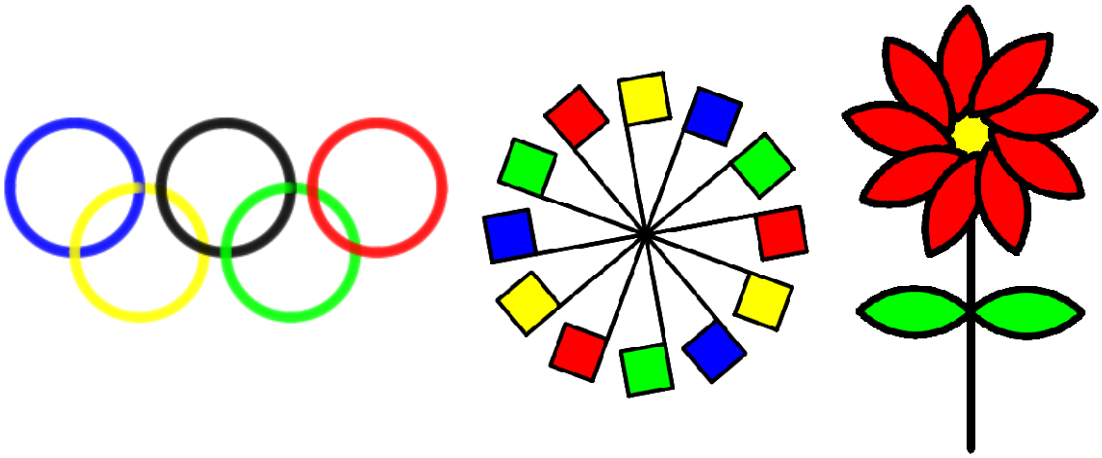

Figura 5. **a)** Aros olímpicos. **b)** Banderas. **c)** Una flor

3.4 Fractales y figuras autosemejantes

El término fractal fue propuesto por el matemático polaco Benoît Mandelbrot para denotar un objeto semigeométrico cuya estructura básica se repite a diferentes escalas [8]. Aunque matemáticamente se trata de objetos infinitos, podemos obtener un gráfico que represente un número finito de niveles. Para ello, usaremos funciones recursivas sobre un número natural que represente el nivel.

Como primer ejemplo, veamos cómo podemos obtener la curva de Hilbert[9], un ejemplo de curva de Peano que rellena por completo un cuadrado. Los tres primeros niveles de la curva se muestran en la Figura 6.

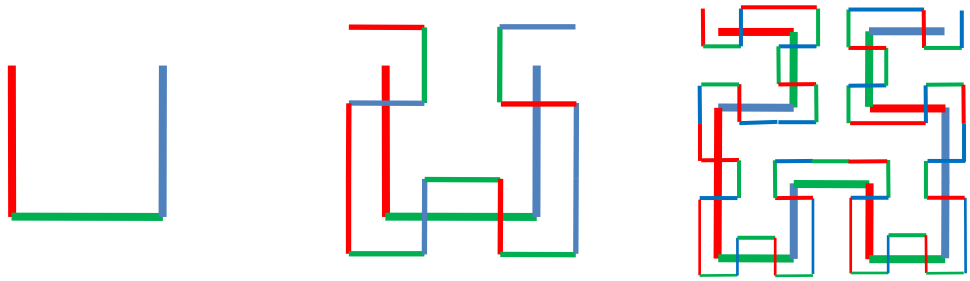

Figura 6. Curva de Hilbert. **a)** Nivel 1. **b)** Niveles 1 y 2. **c)** Niveles 2 y 3.

La siguiente función podría invocarse con un ángulo de 90º para pintar la curva de Hilbert de nivel 1:

```
hilbert1 :: Grados -> Acción
hilbert1 ang = sec [ derecha ang
                   , color rojo, av
                   , izquierda ang
                   , color verde, av
                   , izquierda ang
                   , color azul, av
                   , derecha ang
                   ]
```

donde

```
av :: Acción
av = avanza 10
```

pinta un segmento de la curva. De modo similar, la curva de nivel 2 quedaría descrita así:

```
hilbert2 :: Acción
hilbert2 = sec  [ derecha 90
                , hilbert1 (-90)
                , color rojo, av
                , izquierda 90
                , hilbert1 90
                , color verde, av
                , hilbert1 90
                , izquierda 90
                , color azul, av
                , hilbert1 (-90)
                , derecha 90
                ]
```

Nótese que al finalizar un nivel de la curva, la orientación de la tortuga coincide con la que tenía antes de comenzar el nivel. Generalizando las funciones anteriores, obtenemos las función recursiva `hilbert`, que dibuja la curva de nivel `n`:

```
hilbert :: Int -> Acción
hilbert n = sec [ ancho 2, nivel n 90 ]
 where  nivel 0     ang = nada
        nivel (n+1) ang = sec [ derecha ang
                              , nivel n (-ang)
                              , color rojo, av
```

```
                                    , izquierda ang
                                    , nivel n ang
                                    , color verde, av
                                    , nivel n ang
                                    , izquierda ang
                                    , color azul, av
                                    , nivel n (-ang)
                                    , derecha ang
                                    ]
```

Otro ejemplo similar es la curva de punta de flecha de Sierpinski:

```
sierpinski :: Int -> Acción
sierpinski n = sec [ ancho 4, nivel n 60, av ]
 where nivel 0     ang = nada
       nivel (n+1) ang = sec  [ nivel n (-ang)
                              , color rojo, av
                              , izquierda ang
                              , nivel n ang
                              , color azul, av
                              , izquierda ang
                              , nivel n (-ang)
                              ]
```

La Figura 7 muestra los gráficos obtenidos con las funciones anteriores para niveles de las curvas 5 y 6.

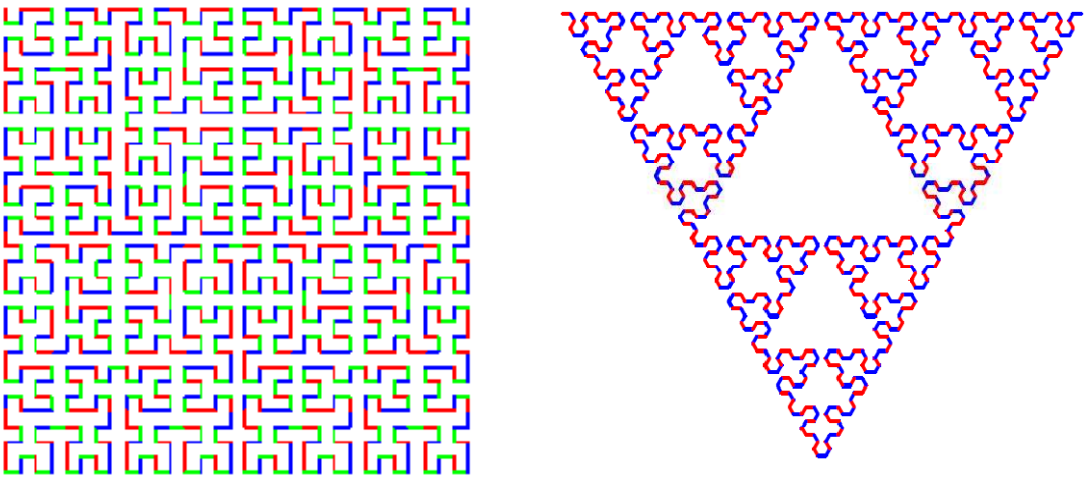

Figura 7. Fractales **a)** De Hilbert a 5 niveles. **b)** De Sierpinski a 6 nieveles.

Por último, mostramos un ejemplo en el que un modelo recursivo matemático simple permite obtener un gráfico similar a formas presentes en la naturaleza. Se trata de dibujar un árbol autosemejante.

```
árbol :: Acción
árbol = sec [ izquierda 90, arb 300 colores 8 ]
 where
  arb n (c:cs) anch
    | n < 3     = nada
    | otherwise = par [ sec [ color c, ancho anch, avanza n ]
                      , rama 0.60 0.76 (derecha 35)
                      , rama 0.55 0.33 (derecha 45)
                      , rama 0.40 0.50 (izquierda 60)
                      ]
    where
      rama dist esc rot = sec [ adelanta (dist*n), rot
                              , arb (esc*n) cs (max 1 (anch-1))
                              ]
```

En este ejemplo, el proceso recursivo no está controlado por un número de niveles fijado a priori, sino que éste termina cuando longitud del tronco del nivel actual es menor a 3 unidades. Si no estamos en el caso base, se establece el color y ancho del pincel, se dibuja el tronco correspondiente a nivel y se realizan en paralelo los tres súbarboles autosimilares, cada uno con su correspondiente desplazamiento (`dist`), escala (`esc`) y rotación (`rot`). Nótese que `colores` es una lista con los colores de los distintos niveles del árbol. El gráfico resultado se muestra en la Figura 8.

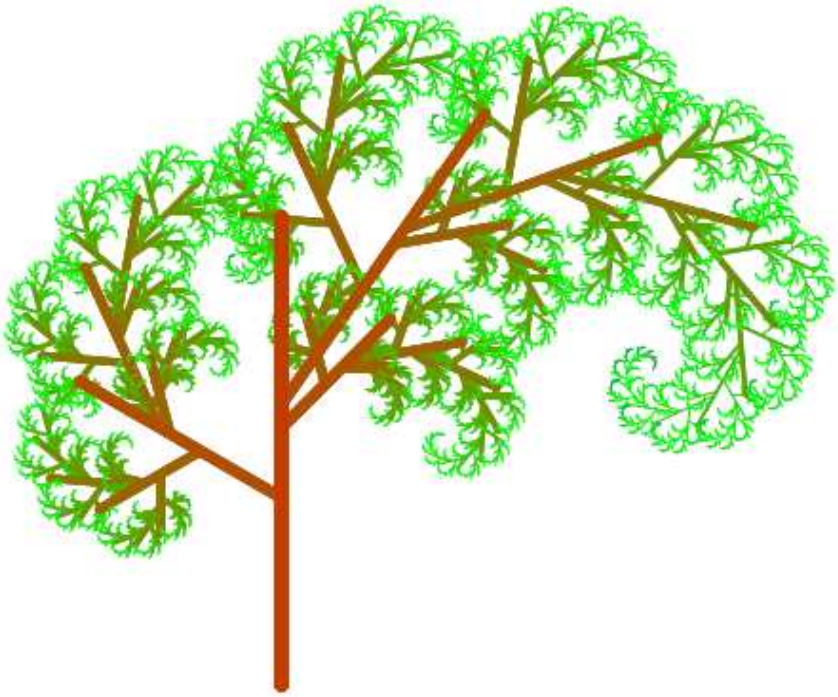

Figura 8. Un árbol autosemejante.

4. Conclusiones

Hemos presentado en este artículo la programación funcional, aplicada a los gráficos de la tortuga como una herramienta que nos permite trabajar conceptos matemáticos con nuestro alumnado de un modo más lúdico.

No se trata de que los alumnos aprendan a programar en la clase de matemáticas, lo que se propone es que el profesor vaya guiando la actuación en el aula de manera que la

elaboración de los programas sea tarea principalmente suya, y los estudiantes se encarguen de hacer los razonamientos matemáticos implicados en los distintas funciones que definimos, y según el nivel del alumnado con el que trabajemos abarcaremos unos u otros conceptos matemáticos.

Lo que intentamos con este artículo es, sobre todo, proponer un modo de hacer ver a los alumnos que las matemáticas, que con frecuencia ellos piensan que fuera del aula no tienen mucha utilidad, están detrás de materiales audiovisuales presentes en su entorno como vídeo-juegos y películas de animación. Aunque a un nivel muy básico, con los gráficos de la tortuga les estamos mostrando que para que en nuestro ordenador aparezca un gráfico o una animación, antes alguien tiene que elaborar un programa que lo haga, y que para ello son imprescindibles las matemáticas.

Bibliografía

[1] S. Peyton Jones. Haskell 98 language and libraries. The revised report. Cambridge University Press. 2003.

[2] R. Bird. Introduction to functional programming using Haskell. Prentice Hall, 1998.

[3] G. Hutton. Programming in Haskell. Cambridge University Press, 2007.

[4] B. O'Sullivan, D. Stewart y J. Goerzen. Real World Haskell. O'Really, 2008.

[5] B. Ruiz, F. Gutiérrez, P. Guerrero y J. Gallardo. Razonando con Haskell. Un curso sobre programación funcional. Thompson, 2004.

[6] Abelson, H y diSessa, A. Turtle Geometry. The Computer as a medium for exploring Mathematics. MIT Press, 1986.

[7] LOGO Foundation. http://el.media.mit.edu/logo-foundation/index.html

[8] Fractal. http://es.wikipedia.org/wiki/Fractal

[9] Hilbert, D. Über die stetige Abbildung einer Linie auf ein Flächenstück. Math. Ann. 38 (1891), 459–460.