Paradigmas de la Programación
Obligatorio 1
Fractales y L-Systems

Setiembre 2012

1. Introducción

Un problema desafiante en la industria del juego y el cine es la generación computarizada de escenas realistas. Un problema particularmente difícil, es la generación de estructuras orgánicas, como plantas y árboles, más aún cuando estas escenas tienen que ser generadas en tiempo real, o cuando las escenas requieren un alto grado de detalle y realismo.

Una solución es tener una base de datos desde donde se puedan consultar estas estructuras orgánicas, y ser así colocadas en una escena. Existen diversas desventajas en esta solución, por ejemplo, el tamaño del almacenamiento requerido y el costo de obtener esta información. Además de que si las mismas estructuras son usadas una y otra vez, la escena generada tiende a parecer demasiado artificial.

Una alternativa es generar dinámicamente estas estructuras al momento de crear las escenas, mediante un pequeño número de "reglas" que las generan. De esta forma se evita el almacenamiento explícito de estas. Más aún, aplicando reglas probabilísticas pueden generarse distintas estructuras a través de las mismas reglas.

Un método popular de definir estas reglas son los Sistemas de Lindenmayer (*L-Systems*). Este nombre proviene del biólogo que desarrolló este método interesado en describir la aparente estructura fractal del crecimiento natural de plantas. El método se basa en reglas de reescritura muy similares a las gramáticas libres de contexto, y ha sido usado extensivamente en aplicaciones gráficas. Por ejemplo, la película "El señor de los Anillos: Las dos torres" contiene escenas generadas por computadoras con vegetación creada mediante variantes probabilísticas de los L-Systems.

Este trabajo está basado en [ICoSM11] y utilizamos de forma simple los L-Systems para generar imágenes de fractales en dos dimensiones. Es posible generalizar la solución a tres dimensiones, y aplicar números randómicos de forma de parametrizar las reglas de reescritura.

2. L-Systems

Los L-Systems trabajan sustituyendo repetidamente símbolos en una secuencia de acuerdo a un conjunto fijo de reglas de reescritura, de forma similar a como trabajan las gramáticas libres de contexto. A diferencia de estas últimas, en vez de comenzar por un símbolo inicial, estos sistemas comienzan por una secuencia inicial de símbolos llamada *semilla*. Otra diferencia es que en las gramáticas libres de contexto se cambia un símbolo por vez en cada etapa de la derivación, mientras que en estos sistemas se aplican las reglas de reescritura a todos los símbolos de la secuencia a la vez en cada etapa del proceso. Así, esta secuencia inicial comienza a crecer exponencialmente en tamaño a medida que las reglas son aplicadas.

Un ejemplo de L-System es el definido por la semilla "M−−M−−M" y una única regla de reescritura "M → M+M−−M+M". Podemos ver a continuación una etapa de reescritura a partir de la semilla, donde vemos como todos los símbolos "M" que aparecen en la semilla son reescritos mediante la única regla de reescritura de este L-System.

$$M--M--M \implies \overbrace{M+M--M+M}^{M}--\overbrace{M+M--M+M}^{M}--\overbrace{M+M--M+M}^{M}$$

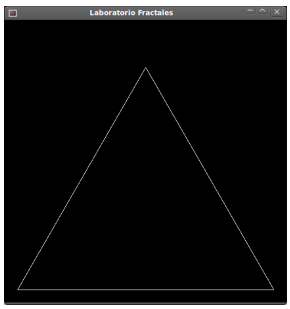
3. Imágenes dibujadas por una tortuga

Para dibujar los fractales se utiliza una tortuga. Una tortuga puede ser vista como un robot que se mueve sobre una hoja de papel de acuerdo a un conjunto de comandos y tiene un lápiz que va dejando una línea al moverse de acuerdo a los comandos recibidos. Los comandos básicos son:

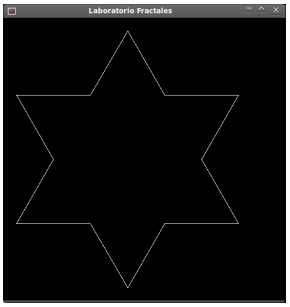
- "F" (*Forward*): Mueve la tortuga hacia adelante una unidad de distancia
- "L" (*Left*): Gira la tortuga en sentido antihorario un determinado ángulo predeterminado
- "R" (*Right*): Gira la tortuga en sentido horario un determinado ángulo predeterminado

Si ahora hacemos corresponder cada símbolo de un L-System con una orden a una tortuga, podemos generar una imagen.

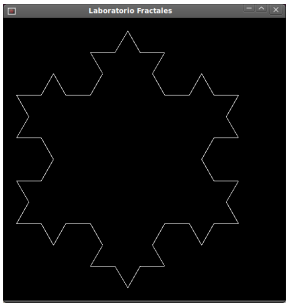
En el ejemplo anterior, si hacemos corresponder el símbolo "M" con la orden "F", el símbolo "+" con la orden "L" y el símbolo "-" con la orden "R", y si tomamos el ángulo de rotación de 60° , obtenemos entonces las imágenes mostradas en la Figura 1, de acuerdo al número de etapas de reescritura hechas a la semilla.



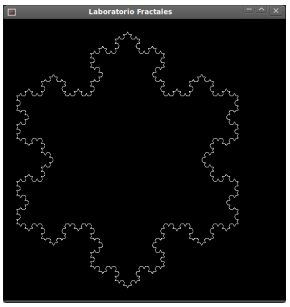
(a) 0 etapas de reescritura



(b) 1 etapa de reescritura



(c) 2 etapas de reescritura



(d) 3 etapas de reescritura

Figura 1: Imágenes formadas por una tortuga a partir de un L-System

Para generar imágenes más elaboradas se pueden agregar otros comandos : “Bo” y “Bc” (*Branch open* y *Branch close*) que se disponen de forma balanceada (por cada “Bo” tiene que haber un “Bc” que lo balancee). Estos comandos permiten introducir bifurcaciones en las líneas generadas por la tortuga: cuando un comando “Bo” es encontrado la tortuga se bifurca en dos: por un lado evoluciona de la forma descrita por la secuencia que se encuentra entre el “Bo” encontrado y el correspondiente “Bc” que lo cierra, y por otro lado evoluciona en paralelo según la secuencia que sigue al “Bc”. Por ejemplo, en el siguiente L-System se muestra el uso de comandos de bifurcación:

Ángulo inicial de la tortuga: 90°	
Ángulo de giro: 45°	
Semilla: “M”	
Reglas:	M → N[-M][+M][NM]
	N → NN
Mapeos:	M → F, N → F, + → L, - → R, [→ Bo,] → Bc

Si aplicamos una etapa de reescritura a partir de la semilla “M” en este L-System, obtenemos la secuencia “N[-M][+M][NM]”.
Haciendo corresponder ahora esta secuencia a los comandos de una tortuga, obtenemos la lista:

$$[F, Bo, R, F, Bc, Bo, L, F, Bc, Bo, F, F, Bc]$$

Tomando como ángulo inicial y de giro de la tortuga 90° y 45° respectivamente, esta secuencia de comandos se corresponden con la figura 2.

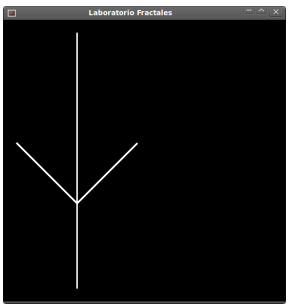


Figura 2: L-System con bifurcaciones

Otros comandos que se utilizan también son:

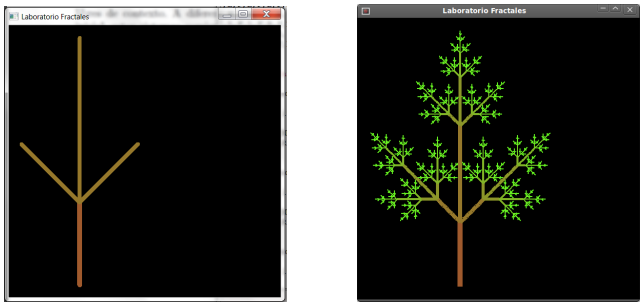
- “Pi” Incrementa el grosor del lápiz de la tortuga en cierta cantidad predeterminada
- “Pd” Decrementa el grosor del lápiz en cierta cantidad predeterminada, en caso que el grosor resultante sea positivo (si no lo es, deja el grosor que estaba)
- “Ci” Incrementa el color del lápiz en ciertas cantidades predeterminadas cada una de sus componentes RGB[1]
- “Cd” Decrementa el color del lápiz en ciertas cantidades predeterminadas cada una de sus componentes RGB
- “N” No realiza movimiento alguno, manteniéndose inalterado el estado de la tortuga

[1]La descripción RGB (Red, Green, Blue) de un color hace referencia a la composición del color en términos de la intensidad de los colores rojo, verde y azul

Modificamos ahora el L-System presentado anteriormente, incluyendo algunos de estos comandos:

Ángulo inicial de la tortuga: 90°	
Ángulo de giro: 45°	
Grosor inicial del lápiz:10	
Variación del grosor del lápiz: 2	
Color inicial del lápiz: (160, 90, 45)	
Variación del color del lápiz: (-10, 30, -2)	
Semilla: “M”	
Reglas:	M → NPC[-M][+M][NM]
	N → NN
Mapeos:	M → F, N → F, + → L, - → R, [→ Bo,] → Bc P → Pd, C → Cd

En la figura 3 podemos ver la imagen correspondiente a una y a seis etapas de reescritura de este último L-System presentado.



(a) 1 etapa de reescritura (b) 6 etapas de reescritura

Figura 3: L-System con bifurcaciones, colores y graduación del grosor del lápiz

Aumentando la complejidad de las reglas se pueden lograr imágenes más naturales como en la figura 4

Figura 4: Bush con 7 etapas de reescritura

4. Descripción del Obligatorio

En este trabajo se quiere desarrollar un ambiente para dibujar fractales a partir de L-Systems. Para ello, se deberán definir funciones que implementen la reescritura en estos sistemas, otras que permitan representar los comandos de dibujo de la tortuga y finalmente la lectura de los datos de un L-System. Las funciones definidas serán utilizadas por una función principal que usa la biblioteca gráfica `Graphics.HGL` de Hugs para dibujar los fractales.

4.1. Definición de los L-Systems

En la página web del curso se encuentra publicado el archivo `LSys.hs`, en el cual se define el módulo `LSys`.

Este módulo importa la siguiente definición del tipo `Color` basada en la descomposición RGB de la biblioteca gráfica `Graphics.HGL` (`Word8` es el tipo que Haskell usa para representar bytes).

```
data Color = RGB Word8 Word8 Word8
```

En el módulo `LSys` se definen los siguientes tipos básicos:

El tipo de datos `Move` representa los comandos de una tortuga ya descritos en la sección 3.

```
data Move = F | L | R | Pi | Pd | Ci | Cd | Bo | Bc | N
  deriving (Show,Read,Eq)
```

Cada regla de reescritura es representada mediante el tipo `Rule`, y a su vez se define el tipo de las reglas `Rules` como una lista de reglas.

```
type Rule = (Char, String)
type Rules = [Rule]
```

Análogamente se define el tipo `Maps` como la lista de pares correspondiente al mapeo entre símbolos y comandos de una tortuga.

```
type Map = (Char, Move)
type Maps = [Map]
```

Enumeramos una serie de alias para definir el tipo que representa un L-System:

```
type Angle = Float  -- angulo
type DeltaAngle = Float  -- variacion del angulo

type PenWidth = Int  -- grosor del lapiz
type DeltaPenWidth = Int  -- variacion del grosor del lapiz
type DeltaRGB = (Int,Int,Int)  -- variacion del color

type Seed = String  -- semilla
```

Ahora se define el tipo `LSys` utilizado para representar L-Systems:

```
data LSys = LSys Angle DeltaAngle PenWidth DeltaPenWidth
                 RGB DeltaRGB Seed Rules Maps
  deriving Show
```

Así el último L-System presentado en la sección anterior queda representado mediante la siguiente declaración (que está incluida junto con otras en el archivo `LSys.hs`):

```
tree:: LSys
tree = LSys 90 45 10 2 (RGB 160 90 45) (-10, 30, -2)
            "M" [('M',"NPC[-M][+M][NM]"),('N',"NN")]
            [('M',F),('N',F),('-',R),('+',L),('[',Bo),
             (']',Bc),('P',Pd),('C',Ci)]
```

Finalmente en el archivo se pueden encontrar varias funciones selectoras de las componentes del tipo `LSys`.

Funciones a definir en el módulo `LSys`

Se pide definir las siguientes funciones en el módulo `LSys`:

1. `lookupRule::Rules->Char->String`

   Esta función recibe una lista de reglas y un símbolo, y devuelve la palabra por la cual se reescribe el símbolo dado según la lista de reglas dada. En caso de que no exista ninguna regla para el símbolo ingresado esta función devuelve la palabra formada únicamente por el símbolo recibido.

2. `lookupMaps::Maps->Char->Move`

   Esta función recibe una lista de mapeos y un símbolo, y devuelve el movimiento de la tortuga que se corresponde al símbolo ingresado. En caso de que no exista ningún mapeo definido para el símbolo dado, esta función devuelve el movimiento nulo `N`.

3. `expandOne::Rules->String->String`

   Esta función recibe una lista de reglas y una palabra, y aplica un paso de reescritura, esto es, aplica una reescritura a cada símbolo de la palabra.

4. `expandN::Rules->String->Int->String`

   Esta función recibe una lista de reglas, una palabra y un natural `n`, y devuelve la palabra resultante de aplicar `n` pasos de reescritura a la palabra dada.

5. `rewriteMaps::Maps->String->[Move]`

   Esta función recibe una lista de mapeos y una palabra, y devuelve la lista de movimientos resultante de aplicar a cada símbolo de la palabra el mapeo correspondiente en la lista dada.

4.2. Manejo de la tortuga

En el archivo `Turtle.hs` se define el módulo `Turtle`. En este módulo se definen primeramente los tipos que representan un punto en el plano cartesiano y el estado de una tortuga (posición, ángulo, color y grosor del lápiz):

```
type Vertex = (Float, Float)

type TurtleState = (Vertex,Angle,RGB,PenWidth)
```

Para implementar los comandos de bifurcación ("Bo" y "Bc") es necesario tener una pila de estados, donde, cada vez que se encuentra un "Bo" se guarda el estado actual, al cual se deberá volver al encontrar el correspondiente "Bc". Esta es la forma clásica de implementar las llamadas usando una pila.

```
type StackTurtleStates = [TurtleState]
```

Luego se define el tipo `ColouredLine` que representa una línea. Este tipo es usado para dibujar las líneas generadas por un L-System mediante la siguiente función ya definida en el archivo `ICGraphics.hs`

```
type ColouredLine = (Vertex, Vertex, RGB, PenWidth)

drawLines::[ColouredLine]->IO()
```

Finalmente se definen las funciones `sumColor` y `subtractColor:: RGB ->DeltaRGB ->RGB` que permiten sumar y restar colores.

Funciones a definir en el módulo `Turtle`

Se pide definir las siguientes funciones en el módulo `Turtle`:

6. ```
   move::Move->TurtleState->StackTurtleStates->LSys->
           (TurtleState,StackTurtleStates)
   ```

   Esta función recibe un movimiento, un estado de la tortuga, una pila de estados de la tortuga y un L-System, y devuelve el estado resultante de ejecutar el movimiento dado a partir del estado dado y la nueva pila de estados de la tortuga. Esta pila debe ser usada para ir guardando y restaurando los estados de la tortuga a medida que se encuentran comienzos y fines de bifurcaciones.

7. ```
   lsyslMoves::LSys->Int->[Move]
   ```

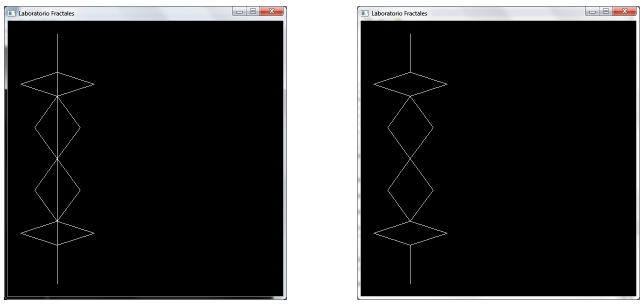
   Esta función recibe un L-System y un natural `n`, y devuelve la liista de movimientos resultante de aplicar `n` reescrituras a partir de la semilla del L-System dado.

8. ```
   turtleLines::LSys->TurtleState->[Move]->[ColouredLine]
   ```

   Esta función recibe un L-System, un estado de una tortuga, y una lista de movimientos, y devuelve la lista de líneas resultante de aplicar los movimientos al estado dado para el L-System dado. Tener en cuenta que cuando se cierra una bifurcación ("Bo") no se debe dibujar la línea entre el punto donde terminó la rama que se cierra y el estado al que se vuelve en la pila (ver figura 5, donde se muestra el dibujo generado por una etapa de reescritura para el L-System `azulejo.ls`).

9. ```
   lsys::LSys->Int->IO ()
   ```

   Esta función recibe un L-System y un natural `n`, y muestra la imagen resultante de aplicar `n` etapas de reescritura al L-System dado, con la tortuga inicializada en el punto `(0,0)`. Para definir esta función utilizar la función `drawLines` definida en el archivo `ICGraphics.hs`. En el módulo `Turtle`



(a) Azulejo con líneas de más (b) Azulejo correcto

Figura 5: Ejemplo de dibujo erróneo de L-System con bifurcaciones

4.3. Análisis léxico y sintáctico

La información de los L-Systems se levantará desde un archivo de texto. El formato del archivo que define un L-System es el mostrado en la figura 6, donde se muestra la secuencia de unidades léxicas o *tokens* (encerradas entre `<>`, salvo las constantes "−>" y "∼") que definen el L-System. Estas se encuentran separadas por espacios en blanco o fines de línea.

```
<Número> <Número>
<Número> <Número>
<Número> <Número> <Número>
<Número> <Número> <Número>
<String>
<Char> -> <String>
...
<Char> -> <String>
<Char> ~ <Move>
...
<Char> ~ <Move>
```

Figura 6: Formato de archivo que define un L-System

El significado de esta sintaxis está dado en la figura 7:

```
<Ángulo inicial> <Ángulo de giro>
<Grosor inicial del lápiz> <Cambio de grosor>
<Componente R del color> <Componente G del color> <Componente B del color>
<Cambio de color R>  <Cambio de color G>  <Cambio de color B>
<Semilla inical>
<Reglas>
<Mapeos>
```

Figura 7: Significadode la sintaxis

Por ejemplo, el último L-System presentado se representa en un archivo con el siguiente contenido:

```
90 45
10 2
160 90 45 -10 30 -2
"M"
'M'  -> "NPC[-M][+M][NM]"
'N'  -> "NN"
'M'  ~ F
'N'  ~ F
'+'  ~ L
'-'  ~ R
'P'  ~ Pd
'C'  ~ Ci
'['  ~ Bo
']'  ~ Bc
```

En el archivo `ParseLSys.hs` que está publicado en la página del curso se define el módulo `ParseLSys` donde se implementará el parser que reconoce esta sintaxis de L-Systems. El reconocimiento sintáctico se hará en tres etapas; primero se separan los tokens según los separadores (espacios en blanco y fines de línea), luego se verifica que cada uno de los tokens pertenezca a una de las categorías de tokens existentes, y finalmente se verifica que la secuencia de tokens siga el patrón mostado en la figura 6.

Funciones a definir en el módulo `ParseLSys`

Para la primera etapa del reconocimiento se debe definir la siguiente función:

10. `tokenizeStr::[Char]->String->[String]`

    Esta función recibe una lista de caracteres que serán tomados como separadores léxicos y una palabra con el contenido del archivo, y devuelve los posibles tokens como una lista de palabras. Esta función no verifica que cada cadena en la lista que se devuelve sea un token válido, ya que esto se hará en la próxima etapa.
    Se puede asumir que el string de entrada tiene al menos una palabra y que el mismo no empieza ni termina con caracteres que son separadores.

Para la segunda etapa del reconocimiento definimos el tipo de `Token` de las categorías de tokens existentes:

```
data Token = Num Int|Str String|Mv Move|Chr Char|Arrow|Equiv
  deriving (Eq,Show)
```

Luego definimos el parser `parseToken` que reconoce cuando una palabra es un token. Este parser se define como la unión de todas las posibilidades existentes:

```
parseToken :: Parse Char Token
parseToken = parseNum `alt` parseStr `alt` parseMove `alt`
             parseChr `alt` parseArrow `alt` parseEquiv
```

Para completar esta definicón se pide definir los parsers para cada una de las categorías de tokens:

11. `parseNum,parseStr,parseMove,parseChr,parseArrow,parseEquiv`

    Estas funciones son todas todas de tipo `Parse Char Token`, y reconocen cada una de las categorías de tokens definidas. Estos parsers se deben definir utilizando las primitivas de parsing vistas en clase.

Ahora utilizando el parser `parseToken` definido anteriormente se puede definir la función que resuelve la segunda etapa, esto es, validar que cada una de las palabras de una lista sea un token, y lo devuelva.

12. `recognizeTokens :: [String] -> [Token]`

    Esta función recibe una lista de palabras y devuelve la correspondiente lista de tokens.

Resolvemos ahora la última etapa del reconocimiento sintáctico, o sea, reconocer que la lista de tokens respeta el patrón definido en la figura 6. Para esto utilizamos un parser también, pero esta vez orientado a tokens (o sea, que recibe una lista de tokens, en vez de una lista de caracteres).

13. `parseLSys::Parse Token LSys`

    Este parser recibe una lista de tokens, reconoce si se cumple el patrón dado en la figura 6, y devuelve el L-System correspondiente siguiendo el significado descrito en 7.

Finalmente haciendo uso de las funciones anteriores, se pide implementar la siguiente función:

14. `readLSys::String -> LSys`

    Esta función recibe una palabra y reconoce un L-System, si es que esto es posible.

4.4. Módulo Prinicpal

Finalmente en el archivo `Main.hs` se define la función `main`, que es la encargada de implementar la lectura y el dibujo de los fractales representados por los L-System. Esta función utiliza todas las funciones definidas en los módulos anteriormente descriptos, por lo que **es importante utilizar los nombres y tipos de las funciones tal cual están pedidos en la letra de este obligatorio**.

En la página del curso se encuentran distintos archivos `.ls` de ejemplos de L-Systems que pueden ser utilizados para probar el trabajo completo desde el módulo `Main`.

5. Entregables

Se deben entregar los archivos `.hs` listos para ser compilados en Hugs. No se piden reportes extra, pero los programas deben ser **legibles** y estar **comentados** apropiadamente (descripción de las funciones auxiliares utilizadas cuando sea necesario). Se tendrá en cuenta el estilo de programación y el nivel de abstracción utilizado en la definición de las funciones. Recuerde que una buena definición de una función no debería ocupar más de 2 o 3 líneas. Se debe entregar todo el código en forma impresa y en un CD, en un sobre transparente. La entrega se realizará en Bedelía con boleta de entrega de obligatorio.

6. Fechas

- Lectura: 10/9/12
- Puntaje (máx): 17 puntos
- Fecha de Entrega: 11/10/12
- Defensa: A fijar con los docentes

7. Extra

Están invitados a participar de un concurso en el cual seleccionaremos las mejores imágenes generadas por un L-Systemusando las funciones de este obligatorio. Las mejoesr imágenes serán publicada en la página del curso dando los créditos correspondientes a los autores.

Referencias

[ICoSM11] Technology Imperial College of Science and Department of Computing Medicine, *L-systems in haskell*, 2011.