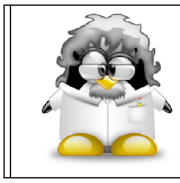
Capítulo 11

Bucles y recursividad

xLogo dispone de siete primitivas que permiten la construcción de bucles: `repite`, `repitepara`, `mientras`, `paracada`.y `repitesiempre`, `repitemientras` y `repitehasta`.



Los bucles son otro de los pilares de la **Programación Estructurada**, es decir, una de las piezas clave en el desarrollo de programas complejos pero fáciles de leer.

11.1. Bucles

11.1.1. Bucle con `repite`

La sintaxis para `repite` es:

```
repite n [ lista_de_comandos ]
```

`n` es un número entero y `lista_de_commandos` es una lista que contiene los comandos a ejecutarse. El intérprete xLogo ejecutará la secuencia de comandos de la lista `n` veces. Esto evita copiar los mismos comandos repetidas veces.
Ya vimos varios ejemplos:

```
repite 4 [avanza 100 giraderecha 90]     # un cuadrado de lado 100
repite 6 [avanza 100 giraderecha 60]     # un hexagono de lado 100
repite 360 [avanza 2 giraderecha 1]      # abreviando, casi un circulo
```

aunque podemos añadir otro con un poco de humor:

Con el bucle `repite`, se define una variable interna `contador` o `cuentarepite`, que determina el número de la iteración en curso (la primera iteración se numera con el 1)

```
repite 3 [escribe contador]
```

proporciona

```
1
2
3
```

11.1.2. Bucle con `repitepara`

`repitepara` hace el papel de los bucles `for` en otros lenguajes de programación. Consiste en asignar a una variable un número determinado de valores comprendidos en un intervalo y con un incremento (paso) dados. Su sintaxis es:

```
repitepara [ lista1 ] [ lista2 ]
```

La `lista1` contiene tres parámetros: el nombre de la variable y los límites inferior y superior del intervalo asignado a la variable. Puede añadirse un cuarto argumento, que determinaría el incremento (el paso que tendría la variable); si se omite, se usará 1 por defecto.

Ejemplo 1:

```
repitepara [i 1 4] [escribe :i*2]
```

proporciona

```
2
4
6
8
```

Ejemplo 2:

```
# Este procedimiento hace variar i entre 7 y 2, bajando de 1.5 en 1.5
# nota el incremento negativo
repitepara [i 7 2 -1.5]
  [escribe lista :i potencia :i 2]
```

proporciona

```
7 49
5.5 30.25
4 16
2.5 6.25
```

El mismo chiste de antes quedaría ahora:

11.1.3. Bucle con `mientras`

Esta es la sintaxis para `mientras`:

```
mientras [lista_a_evaluar] [ lista_de_comandos ]
```

`lista_a_evaluar` es la lista que contiene un conjunto de instrucciones que se evalúan como `cierto` o `falso`. `lista_de_comandos` es una lista que contiene los comandos a ser ejecutados. El intérprete XLOGO continuará repitiendo la ejecución de `lista_de_comandos` todo el tiempo que `lista_a_evaluar` devuelva cierto.

Ejemplos:

```
mientras [cierto] [giraderecha 1] # La tortuga gira sobre si misma eternamente.
```

```
# Este ejemplo deletrea el alfabeto en orden inverso:
haz "lista1 "abcdefghijklmnñopqrstuvwxyz
mientras [no vacio? :lista1]
[es ultimo :lista1 haz "lista1 menosultimo :lista1]
```

Para usar `mientras`, ahora nuestro "alumno castigado" debe añadir una variable:

`mientras` es muy útil para trabajar con listas:

```
mientras no vacio? :nombre.lista
  [ escribe primero :nombre.lista
    haz "nombre.lista menosprimero :nombre.lista]
```

irá mostrando todos los elementos de una lista, eliminando en cada iteración el primero de ellos.

11.1.4. Bucle con `paracada`

La sintaxis de `paracada` es:

```
paracada nombre_variable lista_o_palabra [ lista_de_comandos ]
```

La variable va tomando como valores los elementos de la lista o los caracteres de la palabra, y las órdenes se repiten para cada calor adquirido.
Ejemplos:

```
paracada "i "xLogo
   [escribe :i]
```

muestra:

```
x
L
o
g
o
```

```
paracada "i [a b c]
   [escribe :i]
```

muestra:

```
a
b
c
```

```
haz "suma 0
paracada "i 12345
   [haz "suma :suma+:i]
```

muestra:

```
15
```

(la suma de los dígitos de 12345)

11.1.5. Bucle con `repitesiempre`

Aunque un bucle como este es muy peligroso en programación, es muy fácil crear un bucle infinito, por ejemplo con `mientras`:

```
mientras ["cierto]
   [giraderecha 1] # La tortuga gira sobre si misma eternamente.
```

La sintaxis de `repitesiempre` es:

```
repitesiempre [ lista_de_comandos ]
```

El ejemplo anterior sería:

```
repitesiempre [giraderecha 1]
```

¿Cuándo se hace necesario un bucle infinito? De nuevo en la *web* de Guy Walker podemos encontrar respuestas: Simulaciones en Física, Química, Biología, ... como los movimientos planetario y browniano, la división celular, ... pueden hacer interesante que la animación (Sección 14.4) se mantenga activa durante el tiempo que dure una explicación.

Este bucle puede ser una alternativa a la recursividad cuando el uso de memoria se prevé muy importante. De todos modos, repetimos: **Mucho cuidado al usar bucles infinitos**

11.1.6. Bucle con `repitemientras`

Este bucle se ejecuta con dos listas:

```
repitemientras [lista_de_comandos] [lista_a_evaluar]
```

La primera lista contiene las órdenes a ejecutar mientras la condición de la segunda, `lista_a_evaluar`, sea cierta.

La principal diferencia con el bucle `mientras` es que el bloque de instrucciones se ejecuta al menos una vez, incluso si `lista_a_evaluar` es falsa.

Ejemplo:

```
haz "i 0
repitemientras [escribe :i haz "i :i+1] [:i<4]
```

devuelve:

```
0
1
2
3
4
```

11.1.7. Bucle con `repitehasta`

La estructura es similar al anterior:

```
repitehasta [ lista_de_comandos ] [lista_a_evaluar]
```

Repite el conjunto de órdenes contenidas en `lista_de_comandos` hasta que la condición establecida en `lista_a_evaluar` sea cierta.

Ejemplo:

```
haz "i 0
repitehasta [escribe :i haz "i :i+1] [:i>4]
```

devuelve:

```
0
1
2
3
4
```

Evidentemente, la traducción literal de `do while` y `do until` sería `hazhasta` y `hazmientras`. Hemos elegido una sintaxis distinta para XLOGO por dos motivos: **(1)** Un bucle **repite** operaciones y **(2)** `haz` es la primitiva utilizada para asignar valores a una variable. Nos parece una contradicción mezclar ambos verbos.

Intenta reproducir el "castigo" de nuestro alumno con los bucles que faltan por usar. ¿Qué dificultades encuentras?

11.2. Ejemplo

Observa el siguiente problema:

Halla el número n tal que, la suma de sus dígitos más él mismo, dé 2002:

$$n + S(n) = 2002$$

Podemos abordarlo de muchas maneras. Empecemos con una bastante intuitiva:

Es obvio que el número tiene que ser menor que 2002, ya que se suma a todos sus dígitos, así que:

```
para resuelve
  repitepara [i 1 2002]
    [ haz "digitos cuenta :i   # Contamos el numero de digitos
      haz "contador :i         # Vamos a controlar la iteracion en curso
      haz "suma 0              # Guardaremos aqui la suma de los digitos
#                              # Debemos ponerla a cero en cada iteracion
      repite :digitos
        [ haz "suma :suma + primero :contador     # Voy sumando los digitos
          haz "contador menosprimero :contador ]  # y quitando el primero
      si (:suma + :i) = 2002          # Comprobamos si cumple la condicion
       [ escribe :i ] ]               # En caso afirmativo, lo muestra
fin
```

Por supuesto, podríamos haber pensado un poco más el problema, y reducido el número de iteraciones, pero intentemos que sea general, y no dependa de que sea 2002 u otro número.

Hemos dicho varias veces que el problema se simplifica si lo dividimos en partes. Creemos un procedimiento para sumar los dígitos de un número dado:

```
para suma.digitos :numero
  hazlocal "suma 0               # Inicializo a cero la variable por si acaso
  mientras [numero? :numero]     # Cuando quite todos los digitos, no sera numero
    [ haz "suma :suma + primero :numero   # Voy sumando los digitos y quitando
      haz "numero menosprimero :numero ]  # siempre el primero. La variable es
  devuelve :suma                          # local, y no cambia el valor de "numero
fin                                       # en el procedimiento principal
```

de modo que el procedimiento principal quedaría:

```
para resuelve
  repite 2002
    [ si 2002 = contador + suma.digitos contador
       [ escribe contador ] ]
fin
```

En este caso, no hemos necesitado usar variables auxiliares, ya que en ningún momento hemos modificado el valor de `contador` en el bucle, y la suma se hacía "fuera" del procedimiento principal.

11.3. Comandos de ruptura de secuencia

xLogo tiene tres comandos de ruptura de secuencia: `alto`, `detienetodo` y `devuelve`.

- `alto` puede tener dos resultados:

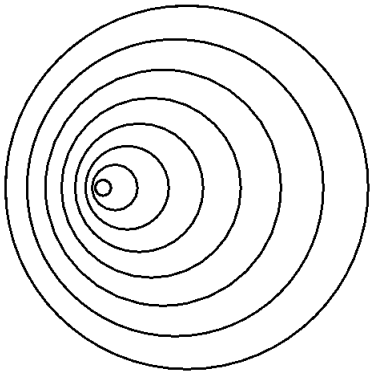
  - Si está incluído en un bucle `repite` o `mientras`, el programa sale del bucle inmediatamente.
  - Si está en un procedimiento, este es terminado.

- `detienetodo` interrumpe total y definitivamente todos los procedimientos en ejecución
- `devuelve` (`dev`) permite salir de un procedimiento "llevándose" un resultado. (ver sección 11.7.1)

11.4. Ejercicios

Intenta hacer los ejercicios con todos los tipos de bucle. Si no puedes con alguno de ellos no desesperes, en programación se trata de encontrar la forma más fácil de programar, no de obtener la más rara.

1. ¿Cómo dibujarías esta serie de círculos NO concéntricos?



   Puedes incluso dejar el número de círculos como variable

2. Escribe procedimientos que muestren las siguientes salidas en el Histórico de Comandos:

```
a)    1    2    3    4    5    6    7    8    9   10
b)    2    4    6    8   10   12   14   16   18   20
c)   20   22   24   26   28   30   32   34   36   38
d)   10   14   18   22   26   30   34   38   42   46
e)   45   40   35   30   25   20   15   10    5    0
f)    1    4    9   16   25   36   49   64   81  100
g)    2    5   10   17   26   37   50   65   82  101
h)    8   27   64  125  216  343  512  729 1000 1331
i)  1.0  0.5  0.33.. 0.25  0.2  0.166.. 0.1428 0.125
```

```
j)     2    6   12   20   30   42   56   72   90  110
k)     1   10  100 1000 10000 100000 1000000
l)   1.0  0.1 0.01 0.001 0.0001 0.00001 0.000001
m)     1   -1    1   -1    1   -1    1   -1    1   -1
```

Los espacios son para ver mejor la serie para obtener. Tú usa `escribe`

3. Escribe un procedimiento que admita dos números y escriba la suma de enteros desde el primer número hasta el segundo.

```
suma.entre 30 32
La suma desde 30 hasta 32 es: 93
```

porque $30 + 31 + 32 = 93$

4. Escribe un procedimiento que pida un número y calcule su factorial:

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

```
factorial 5
El factorial de 5 es 120
```

porque $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

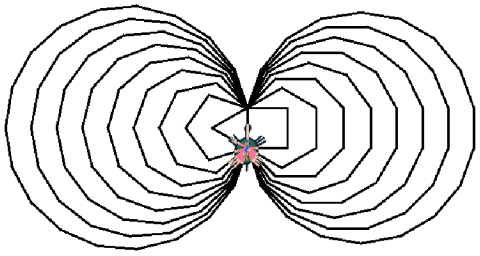
5. Escribe un procedimento con un número como argumento y escriba sus divisores.

```
divisores 200
Los divisores de 200 son 1 2 4 5 8 10 20 25 40 50 100 200
```

6. Escribe un procedimento con un número como argumento y determine si es primo o no.

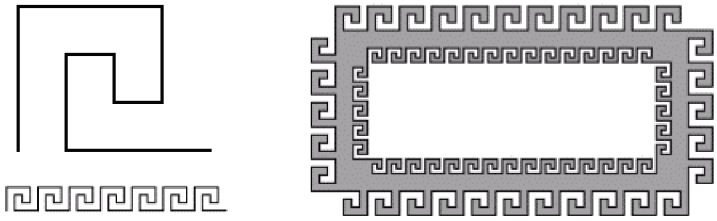
```
primo? 123
  no es primo
primo? 127
  127 es primo
```

7. Dibuja esta serie de polígonos, en la que los que tienen un número impar de vértices están hacia la izquierda, pero los que tienen un número impar de vértices están hacia la derecha:



Puedes, como en el primer ejercicio, dejar el número de polígonos como variable

8. Intenta dibujar la figura de la derecha, basándote en el motivo mostrado a la izquierda:



9. Escribe un programa para jugar a adivinar un número. El procedimento:

   a) admite dos argumentos, los valores entre los que está el número a adivinar

   b) se "inventa" un número al azar entre esos dos

   c) el usuario va probando valores y el programa va diciendo si son demasiado grandes o pequeños.

```
juego 0 100
   -> A ver si adivinas un numero entero entre 0 y 100.
   -> Escribe un numero: 20
   -> Es mas grande: Intentalo de nuevo: 30
   -> Es demasiado grande: Intentalo de nuevo: 30
   -> Es demasiado grande: Intentalo de nuevo: 27
   -> Acertaste. Te ha costado 3 intentos
```

10. Escribe un programa que permita crear una lista de palabras. Para ello, el procedimiento:

    a) Tiene un número como argumento

    b) Solicita ese número de palabras con un mensaje del tipo:

```
Dame el elemento numero ...
```

    c) Según se van introduciendo, se va aumentando la lista

    d) Por último, el programa tiene que escribir la lista.

```
crea.lista 3
  -> Dame el elemento numero 1: Alberto
  -> Dame el elemento numero 2: Benito
  -> Dame el elemento numero 3: Carmen
La lista creada es: [Alberto  Benito Carmen]
```

Modifica el procedimiento para que los elementos se vayan agregando a la lista en orden inverso al que se introducen

11. Escribe un procedimiento cuyo argumento sea una lista y que la ordene alfabéticamente (considera sólo la primera letra).

```
orden.alf [ Carmen Alberto Daniel Benito]
La lista ordenada es: [Alberto Benito Carmen Daniel]
```

11.5. Recursividad

Un procedimiento se llama *recursivo* cuando se llama a sí mismo (es un *subprocedimiento* de sí mismo). Un ejemplo muy simple es el siguiente:

```
para ejemplo1
 giraderecha 1
 ejemplo1
fin
```

Este procedimiento es recursivo porque se llama a sí mismo. Al ejecutarlo, vemos que la tortuga gira sobre sí misma sin parar. Si queremos detenerla, debemos hacer *clic* en el botón `Alto`.

Piensa qué hace este segundo ejemplo (la primitiva `espera` pertence al capítulo 18: Hace una pausa igual a la 60.ª parte del número indicado):

```
para ejemplo2
  avanza 200 goma espera 60
  retrocede 200 ponlapiz giraderecha 6
  ejemplo2
fin
```

Inícialo con la orden: `bp ejemplo2`. Obtenemos ... ¡La aguja segundera de un reloj!

Para no depender de que un usuario externo detenga el programa, debemos incluir **siempre** un condicional de parada.

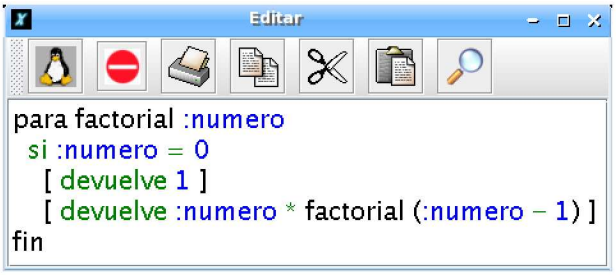
El caso más típico es el cálculo del **factorial**. En lugar de definir

$$\texttt{n! = n * (n - 1) * ...* 2 * 1},$$

podemos hacer:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n \neq 0 \end{cases} \qquad \forall\, n \in \mathbb{N}$$

En xLogo:

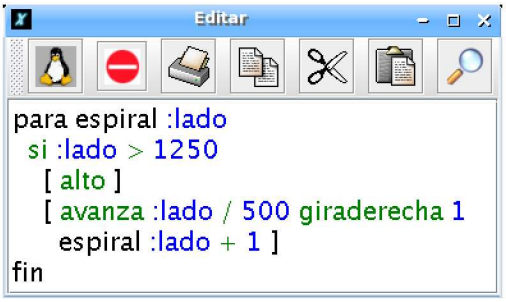


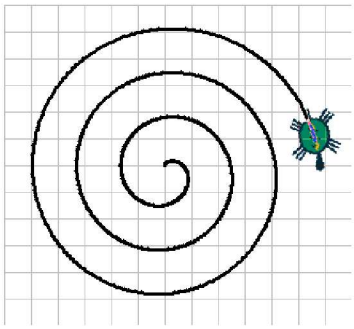
que va llamándose a sí mismo en cada ejecución, pero cada vez con el número reducido en una unidad. Cuando dicho número es cero, devuelve "1", y se termina la recursión. Es decir:

```
factorial 5
-> 5 * factorial 4
       -> 4 * factorial 3
              -> 3 * factorial 2
                     -> 2 * factorial 1
                            -> 1 * factorial 0
                                   -> 1
```

siendo el resultado 120.

Un segundo ejemplo recursivo es la **espiral**:





que en cada llamada aumenta en una unidad el avance de la tortuga, siendo el límite 1250.

La recursividad o recurrencia es muy importante en robótica, (sección 19.3) cuando un sistema permanece "en espera" (ver ejemplo `verbos.lgo`). Sin embargo, debe ser utilizada con cuidado, ya que cada llamada recursiva va ocupando un espacio de memoria en el ordenador, y no la libera hasta que se cumple la condición final.

Muchas de las cosas que pueden conseguirse con la recursividad pueden hacerse con bucles, pero eso conlleva en algunos casos complicar el diseño y dificultar la lectura del código y/o ralentizar su ejecución.

11.5.1. Retomando el ejemplo

Recuperemos el problema anterior:

Halla el número n tal que, la suma de sus dígitos más él mismo, dé 2002:

$$n + S(n) = 2002$$

¿Cómo haríamos para resolverlo recursivamente? Sencillamente modificando el procedimiento `suma.digitos`:

```
para resuelve
  borratexto
  repite 2002
    [ si  2002 = suma suma.digitos contador contador
        [ escribe contador ] ]
fin

para suma.digitos :numero
  si numero? :numero
    [ devuelve suma (primero :numero) (suma.digitos menosprimero :numero) ]
    [ devuelve 0 ]
fin
```

que devuelve las mismas (obviamente) soluciones que los programas anteriores:

$$n = 1982 \qquad \text{y} \qquad n = 2000$$

Observa que el procedimiento `suma.digitos` se llama a sí mismo, pero quitando el primer dígito de `"numero`.

	Intenta ponerte en el lugar de la tortuga, coge un papel y realiza los mismo cálculos que ella. Por supuesto, elige solamente dos o tres números como muestra, siendo uno de ellos una de las soluciones. De esa manera podrás entender mejor la recursividad.

11.5.2. Ejercicios

1. Plantea un programa recursivo que calcule potencias de exponente natural
2. Plantea un programa recursivo que calcule el término `n`–simo de la sucesión de Fibonacci. Esta sucesión se obtiene partiendo de `1 , 1` , y cada término es la suma de los dos anteriores:

   `1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , 34 , 55 , ...`

3. Diseña un procedimiento recursivo que devuelva la suma de los `n` primeros números pares (excluido el cero):

   ```
   2 + 4 + 6 + 8 + ... + 2 n
   ```

4. Diseña el procedimiento `cuadrados` que tenga de entrada `lado` y dibuje, recursivamente, cuadrados de lados 15, 25, 35, 45, 55, 65 y 75. Es decir, debe ir incrementando el lado de 10 en 10, y debe tener un condicional para parar.

5. Diseña un programa `cuadrados_1000` que tenga una entrada `numero` y escriba los números naturales que sean menores que 1000 y cuadrados de otro natural.

   Pista: No se trata de ir comprobando qué números son cuadrados perfectos, sino generar con un programa recursivo los cuadrados de los sucesivos naturales y que no pare mientras estos cuadrados sean menores que 1000

6. En la sección 7.5 obtuviste un procedimiento que dibujaba un polígono inscrito en una circunferencia. Modifica ese procedimiento para que:

   a) Dibuje también un polígono circunscrito
   b) Calcule el perímetro de ambos (inscrito y circunscrito)
   c) Divida ambos resultados entre el doble del radio y lo muestre

   Haz que el dibujo vaya aumentando progresivamente el número de lados, y piensa si el valor que obtienes te recuerda a algún número.

11.6. Recursividad avanzada

Un tipo de curvas muy especiales de las Matemáticas son las denominadas **fractales**. xLogo puede generar fractales mediante procedimientos recursivos de forma muy sencilla.

11.6.1. Copo de nieve

La curva de Koch se construye a partir de un segmento inicial con tres sencillos pasos:

1. dividimos el segmento en tres partes iguales.
2. dibujamos un triángulo equilátero sobre el segmento central.
3. borramos el segmento central.

Los primeros pasos en la construcción de la curva, son:

Qué es importante: Si observamos el paso número 2, vemos que las líneas quebradas contienen cuatro motivos idénticos que corresponden a las etapas previas, pero tres veces más pequeños. Esa es la estructura recursiva del fractal.

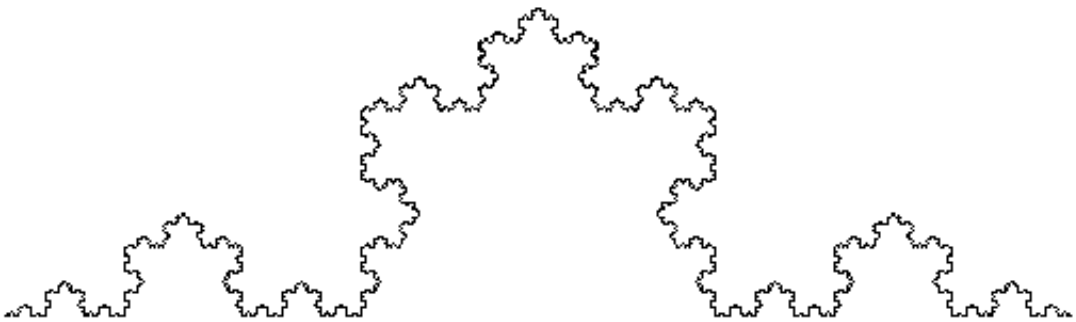
Llamemos $L_{n,\ell}$ al elemento de longitud ℓ, correspondiente al paso n. Para dibujar este elemento:

1. Dibujamos $L_{n-1,\ell/3}$
2. Giramos izquierda 60 grados
3. Dibujamos $L_{n-1,\ell/3}$
4. Giramos derecha 60 grados
5. Dibujamos $L_{n-1,\ell/3}$
6. Giramos izquierda 60 grados
7. Dibujamos $L_{n-1,\ell/3}$

Con xLogo, es muy fácil de escribir:

```
# :l longitud del elemento
# :p paso
para linea :l :p
si :p = 0
 [ avanza :l ]
 [ linea :l/3 :p-1 giraizquierda 60
   linea :l/3 :p-1 giraderecha 120
   linea :l/3 :p-1 giraizquierda 60
   linea :l/3 :p-1 ]
fin
```

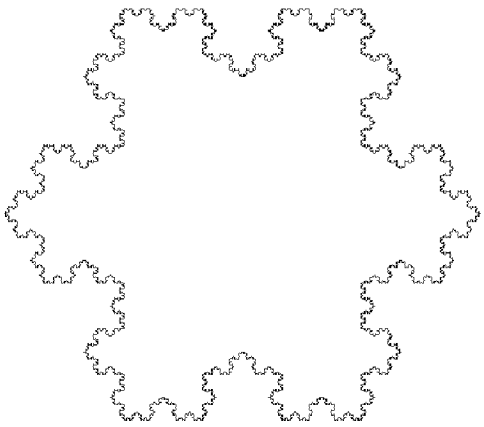
Ejecutando `linea 500 8` obtenemos:



Si dibujamos un triángulo equilátero cuyos lados sean segmentos de Koch, obtendremos un hermoso *copo de nieve de Koch.*

```
para coponieve :l :p
  repite 3 [
    linea :l :p
    giraderecha 120 ]
fin
```

Ej: `coponieve 200 6`



11.6.2. Aproximando π (1)

Podemos aproximar el valor del número π usando la fórmula:

$$\pi \approx 2^k \sqrt{2 - \sqrt{2 + \sqrt{2 + \ldots \sqrt{2 + \sqrt{2}}}}}$$

siendo k el número de raíces cuadradas. Cuanto mayor sea k, mejor es la aproximación de π .

La fórmula contiene la expresión recursiva $2 + \sqrt{2 + \ldots \sqrt{2 + \sqrt{2}}}$, por tanto:

```
# k es el numero de raices cuadradas
para aproxpi :k
  mecanografia "aproximacion:\  escribe (potencia 2 :k) * raizcuadrada
           (2- raizcuadrada (calc :k-2))
  escribe "------------------------
  mecanografia "pi:\  escribe pi
fin
```

```
para calc :pi
  si :pi = 0
    [devuelve 2]
    [devuelve 2 + raizcuadrada calc :pi - 1]
end
```

De modo que, ejecutando:

```
approxpi 10
Aproximacion: 3.141591421568446
-------------------------
Pi: 3.141592653589793
```

Hemos conseguido 5 dígitos correctos. Si queremos más dígitos de π, deberíamos permitir una mayor precisión usando más dígitos en el cálculo. Para ello, vamos a usar la primitiva `pondecimales`, que estudiamos en la sección 7.6. Haciendo:

```
pondecimales 100
aproxpi 100
Aproximacion: 3.14159265358979323846264338327950288419733930696701609758076843138804
-------------------------
Pi: 3.14159265358979323846264338327950288419716939937510582097494459230781640 6....
```

Logramos 39 dígitos exactos.

11.6.3. Con palabras y listas

En los ejercicios de la sección 10.3 se pide construir un procedimiento que compruebe si una frase o palabra es un palíndromo. La recursividad nos lo pone muy fácil. Empecemos por crear el procedimiento:

```
para inviertepalabra :m
  si vacio? :m [devuelve "]
  devuelve palabra ultimo :m inviertepalabra menosultimo :m
fin
```

que proporciona:

```
inviertepalabra abcdefghijkl
lkjihgfedcba
```

Dado que un palíndromo es una palabra que se lee igual de derecha a izquierda que de izquierda a derecha:

```
para palindromo? :m
  si :m = inviertepalabra :m [devuelve cierto] [devuelve falso]
fin
```

Olivier SC nos dejó este programa:

```
para palin :n
  si palindromo? :n [escribe :n alto]
  escribe (lista :n "mas inviertepalabra :n "igual suma :n inviertepalabra :n)
  palin :n + inviertepalabra :n
fin
```

que proporciona:

```
palin 78
78 mas 87 igual 165
165 mas 561 igual 726
726 mas 627 igual 1353
1353 mas 3531 igual 4884
4884
```

11.7. Uso avanzado de procedimientos

11.7.1. La primitiva `devuelve`

Es posible conseguir que un procedimiento se comporte como una función similar a las antes definidas en xLogo. Por ejemplo:

```
para con_IVA :precio :IVA
# Este procedimiento aumenta el precio con el IVA
  devuelve :precio * (1 + :IVA / 100)
fin
```

permite escribir:

```
escribe (con_IVA 134 7 + con_IVA 230 16)
```

algo que no sería posible si usáramos `escribe` en vez de `devuelve`.

11.7.2. Variables opcionales

En un procedimiento pueden usarse variables opcionales, es decir, variables cuyo valor puede ser dado por el usuario y, si no lo hace, disponer de un valor *por defecto*.

```
para poligono :vertices [ :lado 100  ]
  repite :vertices
    [ avanza :lado giraderecha 360/:vertices ]
fin
```

El procedimiento se llama `poligono`, lee una variable *forzosa* `vertices` que debe ser introducida por el usuario, y otra variable opcional `lado`, cuyo valor es `100` si el usuario no introduce ningún valor. De este modo que ejecutando

```
poligono 8
```

Durante la ejecución, la variable `:lado` se sutituye por su valor por defecto, esto es, `100`, y xLogo dibuja un octógono de lado `100`. Sin embargo, ejecuando

```
(poligono 8 300)
```

xLogo dibuja un octógono de lado `300`. Es importante fijarse en que ahora la ejecución se realiza encerrando las órdenes entre paréntesis. Esto indica al intérprete que se van a usar variables opcionales.

11.7.3. La primitiva `trazado`

Para seguir el desarrollo de un programa, es posible conocer los procedimientos que se están ejecutando en cada momento. Igualmente, también se puede determinar si los procedimientos están recibiendo correctamente los argumentos usando la primitiva `devuelve`.

La primitiva `trazado` activa el modo **trazado**:

```
trazado
```

mientras que para desactivarla:

```
detienetrazado
```

Un ejemplo puede verse en el cálculo del factorial definido antes.

```
trazado
escribe factorial 4
factorial 4
factorial 4
 factorial 3
  factorial 2
   factorial 1
   factorial devuelve 1
  factorial devuelve 2
 factorial devuelve 6
factorial devuelve 24
24
```

Veremos ejemplos avanzados de recursividad en los Apéndices D y E, donde generaremos distintos fractales tanto en 2-D como en 3-D (veremos en el capítulo 16 cómo dibujar en tres dimensiones)