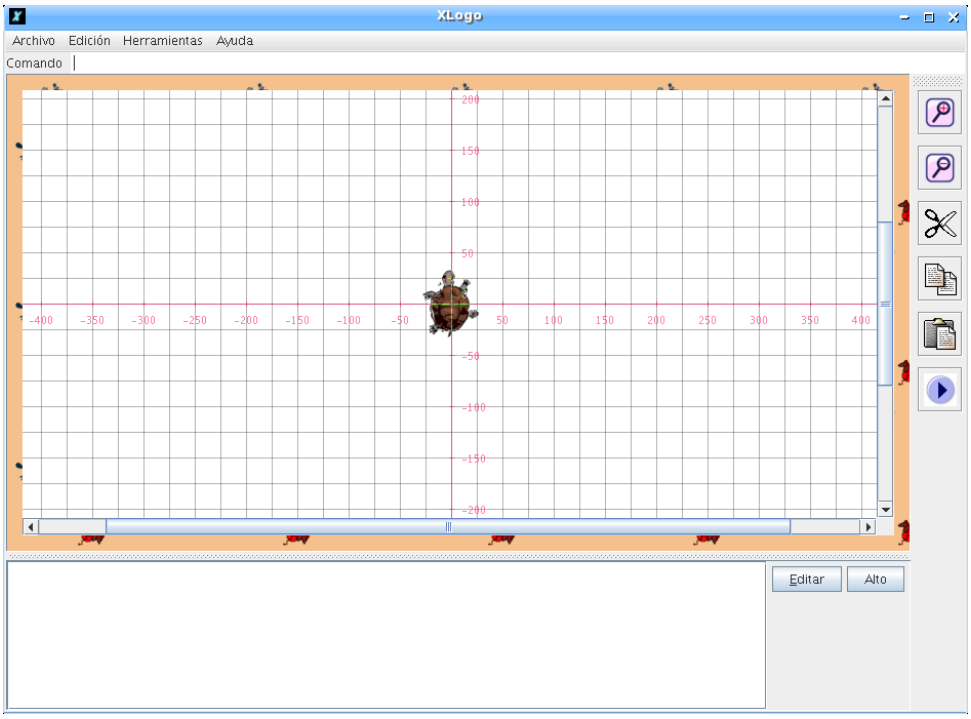

# Curso de Iniciación

Álvaro Valdés Menenéndez
Loïc Le Coq
Marcelo Duschkin

http://xlogo.tuxfamily.org



# Introducción

Logo es un lenguaje desarrollado a finales de los años 60 por Seymour Papert. Papert trabajó con Piaget en la Universidad de Ginebra desde 1959 hasta 1963 y, basándose en su Teoría del Constructivismo, desarrolló el aprendizaje Construccionista:

**El aprendizaje mejora si se aplica activamente a la vida cotidiana**

resumido en la expresión *learning–by–doing*

Logo es una potente herramienta para desarrollar los procesos de pensamiento lógico-matemáticos y un lenguaje excelente para comenzar a estudiar programación, que enseña lo básico acerca de temas como bucles, condicionales, procedimientos, etc. El usuario puede mover un objeto llamado "tortuga" dentro de la pantalla, usando instrucciones (comandos) simples como "`avanza`", "`retrocede`", "`giraderecha`" y similares.

**Poder usar órdenes en el idioma natural favorece su aprendizaje y asimilación**

Con cada movimiento, la tortuga deja un "rastro" (dibuja una línea) tras de sí, y de esta manera se crean gráficos. Logo en general, y xLogo en particular, no sirven solo para hacer dibujos; también es posible realizar operaciones complejas, manipular palabras y listas, evaluar condicionales, programar robots, interactuar con el usuario, tocar música,...

Logo es un *lenguaje interpretado*. Esto quiere decir que las órdenes introducidas por el usuario son interpretadas por el ordenador y ejecutadas inmediatamente en el orden en que son escritas.

xLogo es un intérprete Logo escrito en Java. Actualmente (versión 0.9.96) soporta trece idiomas (Francés, Inglés, Español, Portugués, Alemán, Árabe, Esperanto, Gallego, Asturiano, Griego, Italiano, Catalán y Húngaro) y se distribuye bajo licencia GPL. Por lo tanto, este programa es libre en cuanto a libertad y gratuidad y puede descargarse desde:

`http://xlogo.tuxfamily.org`

En nuestra *web* también puedes descargar la documentación del programa y copiar o ejecutar en línea varios ejemplos con los que comprobar la capacidad de xLogo.

Java es un lenguaje que tiene la ventaja de ser multi-plataforma; esto es, xLogo podrá ejecutarse en cualquier sistema operativo que soporte Java; tanto usando Linux como

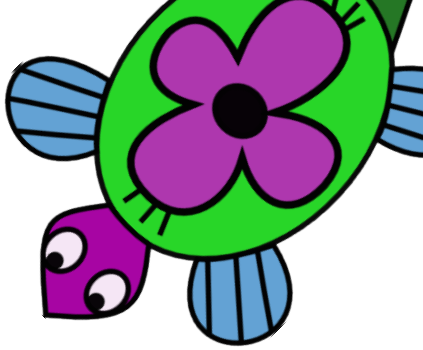
Windows o MacOS, xLogo funcionará sin problemas. Recientemente Java ha sido liberada bajo licencia GPL, lo que también garantiza su disponibilidad y gratuidad.

A lo largo del manual se irán planteando ejercicios y resolviendo ejemplos. Hemos elegido **no utilizar** tildes ni *eñes* en los mismos debido a que en la mayoría de los lenguajes de programación no las aceptan, pero xLogo las admite y trabaja perfectamente con ellas, permitiendo el uso acentuado tanto de primitivas, como de variables y procedimientos.

Las soluciones se mostrarán en el **Tutorial**, también disponible para descarga en la sección de **Documentación** de nuestra *web*:

`http://xlogo.tuxfamily.org/sp/documentacion.html`

# Índice general



# Capítulo 1

# Características de la Interfaz

## 1.1. Instalación de Java

Antes que nada, debes tener instalado el JRE (*Java Runtime Environment*) en tu ordenador. Si no lo tienes instalado, lo puedes descargar de la página *web* de Sun (es libre y gratuito):

`http://java.com/es/download/manual.jsp`

Debes descargar la versión que corresponda a tu sistema operativo (Linux, Windows, ...). Si usas Linux es posible que JRE ya esté instalado. Para confirmarlo, escribe en una consola:

```
java -version
```

Si no responde con un error, JAVA está listo.

Lo siguiente entonces es descargar XLOGO desde

`http://xlogo.tuxfamily.org/`

seleccionando el idioma español y haciendo *clic* en "Descargas", en el menú de la izquierda. Más directamente, el enlace es:

`http://downloads.tuxfamily.org/xlogo/common/xlogo.jar`

## 1.2. Ejecutar y configurar xLogo

### 1.2.1. En Windows

En teoría, al hacer doble *clic* en el archivo `xlogo.jar`, el programa debería iniciarse. Si es así, pasa a la siguiente sección.

Si no es así, y se ejecuta otra aplicación (WinZIP, WinRAR, ...), eso se debe a que en realidad los archivos `.jar` (como se presenta xLogo) son archivos comprimidos equivalentes a `.zip`. A veces, puedes darte cuenta de ello antes de hacer *clic* según el icono que tenga el archivo:

Se abre con Java

Se abre con WinRAR

Se abre con WinZIP

La forma más fácil de solucionarlo consiste en hacer *clic* con el botón derecho sobre el archivo, seleccionar

**Abrir con ...**

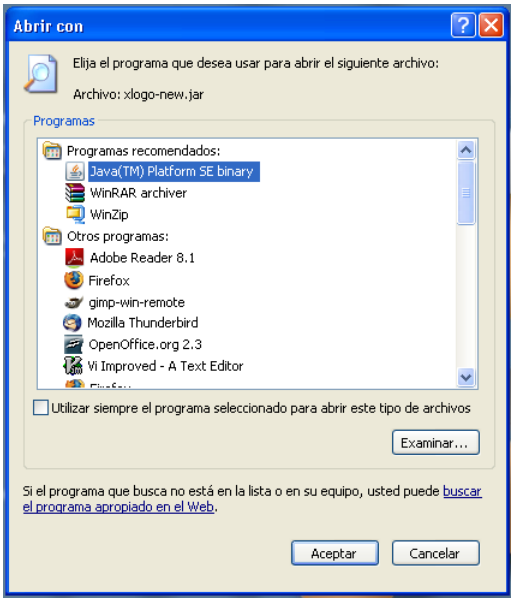
en el menú elegir Java y activar

`Utilizar siempre ...`

siendo esta la forma recomendada.

Si no funciona, las alternativas son:

- eliminar las extensiones `.jar` en la configuración del compresor
- reinstalar Java




Finalmente, conviene incorporar las librerías asociadas a la música. La versión Windows de `jre` no incorpora los *bancos de sonido* que contienen los instrumentos, y hay que descargarlos desde:
`http://java.sun.com/products/java-media/sound/soundbank-min.gm.zip`
la versión mínima (unos 350 kb),
`http://java.sun.com/products/java-media/sound/soundbank-mid.gm.zip`
la versión intermedia (algo más de 1 Mb) y
`http://java.sun.com/products/java-media/sound/soundbank-deluxe.gm.zip`
la versión *de luxe* (casi 5 Mb).

Una vez descargados, debemos descomprimirlos en el directorio `audio` de la instalación Java que, dependiendo de la versión, puede ser:

```
C:\Archivos de programa\Java\jre1.6.0\lib\audio
```

creando el directorio `audio` si este no existe.

Hecho esto, la lista de instrumentos estará disponible.

### 1.2.2. En Mac y Solaris

Las cosas en un Mac son aún más fáciles. Desde hace tiempo Apple incorpora JAVA en sus sistemas operativos, así que simplemente haciendo *clic* sobre el archivo `xlogo.jar` debería ejecutarse XLOGO.


Solaris, por su parte, utiliza el escritorio *Java Desktop System*, en el que JAVA está perfectamente configurado.

### 1.2.3. En Linux

Las cosas en Linux pueden ser muy fáciles si la distribución que utilizas incorpora JAVA (algo bastante probable desde que Sun la liberó bajo licencia GPL). Si es así, simplemente haciendo doble *clic* sobre el archivo `xlogo.jar` debería iniciarse XLOGO.

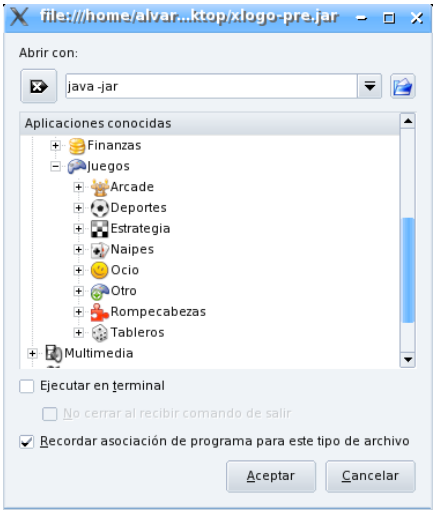
Si no, tras instalar JAVA, debes conocer en qué directorio se encuentra el ejecutable, habitualmente `/usr/bin/` o `/usr/java/` seguido de la versión instalada, por ejemplo:

```
/usr/java/jre1.6.0/bin/java
```

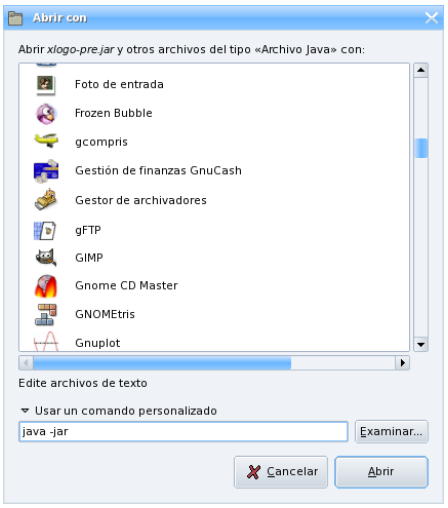
A continuación, descargamos XLOGO (por ejemplo en el escritorio), hacemos *clic* con el botón derecho sobre el archivo `xlogo.jar` → **Abrir con** → **Otros ...** y en el cuadro de diálogo `Abrir con:` escribir

```
/usr/java/jre1.6.0/bin/java -jar
```

y en KDE activar `Recordar asociación de programa para este tipo de archivo`.




En KDE




En GNOME

A diferencia de lo que ocurre en Windows, Linux permite cambiar el icono únicamente al archivo `xlogo.jar` sin afectar a otros archivos `jar`, así que puedes elegir tu tortuga favorita, por ejemplo entre las cinco que puedes usar en XLOGO o incluso el dragón galés.

Este procedimiento puede variar ligeramente según uses **KDE**, **GNOME**, ... pero como ves, configurar Linux no es tan difícil como algunos creen.

## 1.3. Actualizaciones

Para actualizar XLOGO basta reemplazar el archivo `xlogo.jar` por la nueva versión. Es posible tener una alerta en tu correo suscribiéndose a la *fuente* `RSS` en
`http://xlogo.tuxfamily.org/rss.xml`

## 1.4. Desinstalar

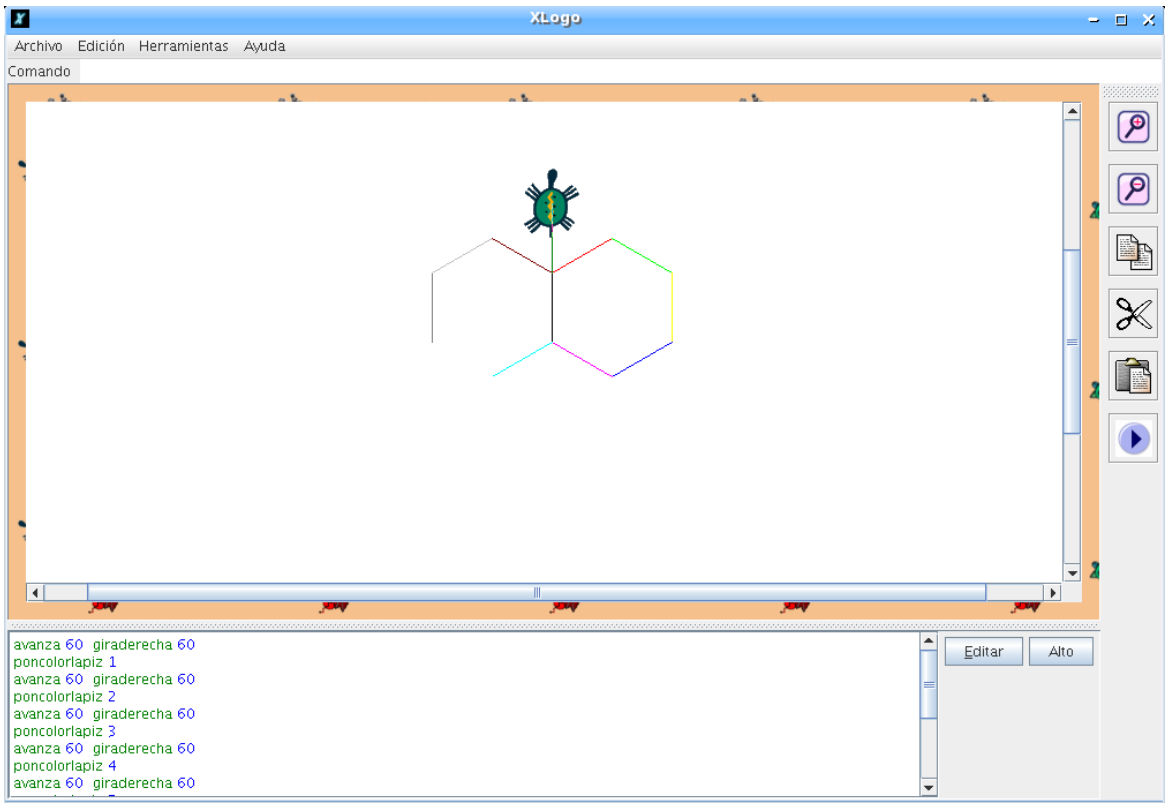
Para desinstalar XLOGO, todo lo que hace falta es borrar el archivo `xlogo.jar` y el archivo de configuración `.xlogo` que se encuentra en `/home/tu_nombre` en Linux, o `c:\windows\.xlogo` en Windows.

## 1.5. Primera Ejecución

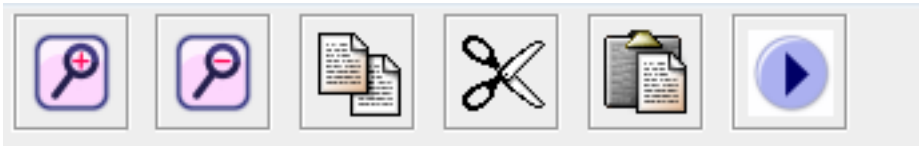
La primera vez que ejecutes XLOGO (o si has borrado el fichero `.xlogo` – ver sección 1.4) deberás elegir el idioma con que quieres trabajar, seleccionando la bandera correspondiente y haciendo *clic* en el botón `OK`.

Esta elección no es definitiva; puedes elegir otro idioma en cualquier momento desde las opciones de menú (sección 2.3)

## 1.6. La ventana principal



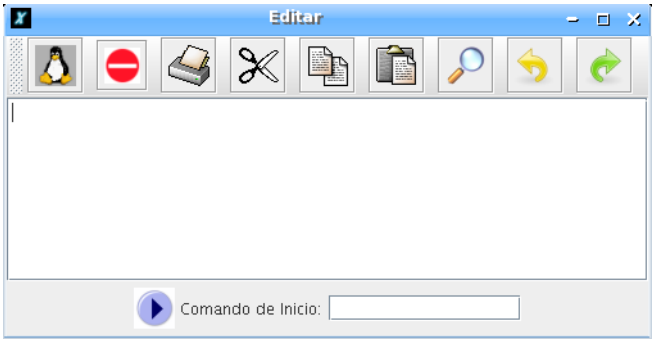

- En la fila superior, están las entradas típicas de menú: **Archivo**, **Edición**, **Herramientas**, **Ayuda**
- Justo debajo está la **Línea de Comando**, donde se escriben las instrucciones Logo.
- En el medio de la pantalla, está el **Área de Dibujo** (donde se mueve la tortuga).
- A la derecha del área de dibujo se encuentra una barra de herramientas vertical con las funciones:



*zoom* (acercar y alejar), *copiar*, *cortar*, *pegar* y *Comando de Inicio*.

- Al pie, está la ventana del **Histórico de Comandos**, que muestra todo lo ingresado y sus respuestas asociadas. Para reutilizar un comando previamente ingresado, hay dos opciones: Hacer un *clic* en un comando del histórico, o usar las teclas de flecha arriba y flecha abajo del teclado (lo que es más práctico).
- A la derecha de la ventana del histórico hay dos botones: **Editar** y **Alto**.
  - **Editar** permite abrir la ventana del editor de procedimientos.
  - **Alto** interrumpe la ejecución del programa ingresado.

## 1.7. El editor de procedimientos




Hay cuatro maneras de abrir el editor:

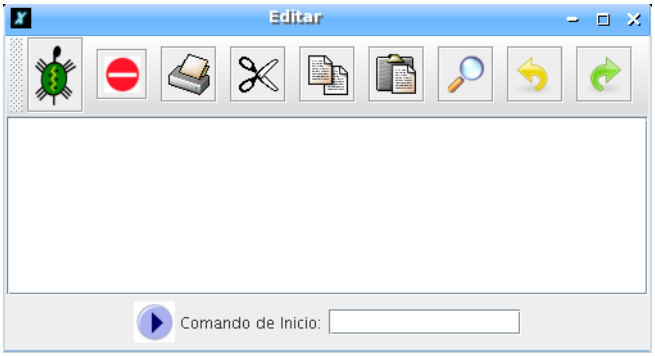
- Escribir `editatodo` o `edtodo` en la **Línea de Comando**. La ventana del editor se abrirá mostrando todos los procedimientos definidos hasta ese momento.
- Si deseas editar un procedimiento en especial (o algunos), debes usar `ed` o `edita` en la línea de comandos seguido del nombre de procedimiento, o la lista con los nombres de procedimientos que deseas editar:
  `edita "nombre_procedimiento`
  o:
  `edita [proc_1 proc_2]`
- Hacer *clic* en el botón **Editar**.
- Usar el atajo de teclado `Alt+E`

Estos son los diferentes botones que encontrarás en la ventana del Editor:

| | |
|---|---|
|  | Guarda en memoria los cambios hechos en el editor y cierra la ventana. Es este botón el que se debe usar cada vez que quieras aplicar los procedimientos recientemente incorporados.<br>Atajo de teclado: `ALT+Q`. |

| | |
|---|---|
|  | Cierra la ventana del editor sin guardar los últimos cambios. Ten presente que NO aparece ningún mensaje de confirmación. Asegúrate bien de que realmente no hay nada que guardar. Atajo de teclado: `ALT+C`. |
|  | Imprime el contenido del editor. |
|  | Copia el texto seleccionado al portapapeles. Atajo de teclado: `Control+C`. |
|  | Corta el texto seleccionado y lo copia al portapapeles. Atajo de teclado: `Control+X`. |
|  | Pega el contenido del portapapeles. Atajo de teclado: `Control+V`. |
|  | Permite realizar búsquedas y reemplazos en los procedimientos. |
|  | Permite deshacer los últimos cambios realizados en los procedimientos. |
|  | "Rehace" lo deshecho con el botón anterior. |

**Nota:** Aunque aquí se representa la imagen de Tux (mascota de Linux) en el botón "*Guardar*", en realidad se muestra la tortuga activa para dar la idea de "*enviar la información a la tortuga*"; por ejemplo, si la tortuga activa es la número 3 (sección 2.3):




En la parte inferior se encuentra línea donde definir el `Comando de Inicio`, que se activa con el botón situado a la derecha del Área de Dibujo.
Al pulsar el botón, se ejecuta inmediatamente el `Comando de Inicio` sin necesidad de escribirlo en la Línea de Comandos, lo que es útil para:

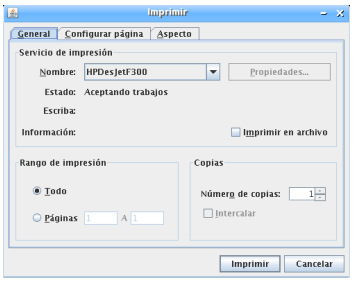
- Ahorrar tiempo mientras se desarrolla un programa

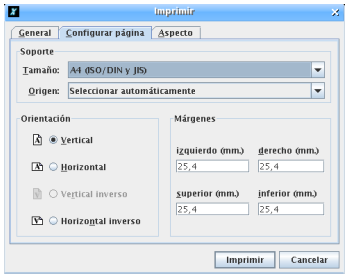
- Al enviar un programa a alguien que se inicia en LOGO, simplemente tiene que hacer *clic* en ese botón para ejecutarlo
- ...

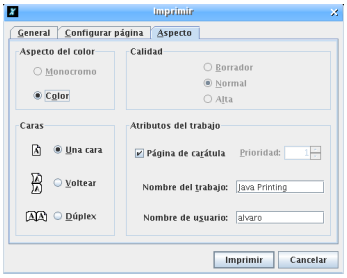
**IMPORTANTE**:

- Nota que hacer *clic* en el icono de cierre ( ) de la barra de título de la ventana del Editor, no hace nada. Solamente funcionan los dos botones principales.
- Para borrar los procedimientos que no se necesitan, usa los comandos `borra` y `borratodo` o en la barra de menús: Herramientas → Borra procedimientos.

Al hacer *clic* para imprimir, aparecerá una ventana de diálogo donde podremos configurar distintas opciones de impresión:




- **General**: Impresora a utilizar, Imprimir a un archivo, Rango de Impresión y Número de copias.
- **Configurar Página**: Tipo de papel, Origen del papel, Orientación de la Hoja y Márgenes
- **Aspecto**: Color (cuando disponible), Calidad, Caras y otros Atributos

## 1.8. Salir

Para salir simplemente seleccionamos: **Archivo** → **Salir**, o hacemos *clic* en en el icono de cierre ( ). XLOGO presentará una ventana de confirmación:


Pulsamos `Sí` y termina la ejecución.

## 1.9. Reiniciar xLogo

Si en algún momento deseamos devolver al intérprete xLogo a sus valores por defecto:

| | | | |
|---|---|---|---|
| Color del lápiz: | negro | Color del papel: | blanco |
| Modo animación: | falso | Tamaño de fuente: | 12 puntos |
| Forma del lápiz: | cuadrado | Calidad del dibujo: | normal |
| Número tortugas: | 16 | Tipo tortuga: | 0 |

disponemos de las primitivas `inicializa` y `reponetodo` que, tecleadas en la Línea de comandos hacen que la tortuga "olvide" los ajustes realizados hasta ese momento.

## 1.10. Convenciones adoptadas para xLogo

Esta sección define aspectos "especiales" acerca del lenguaje xLogo.

### 1.10.1. El caracter especial \

El caracter \ (barra invertida o *backslash*) permite que las "palabras" (sección 6.1) contengan espacios o saltos de línea.

- `\n` produce un salto de línea
- `\␣` produce un espacio entre palabras (␣ representa un espacio en blanco)

**Ejemplos**:

```
escribe "xlogo\ xlogo     produce    xlogo xlogo
escribe "xlogo\nxlogo     produce    xlogo
                                     xlogo
```

Esto tiene implicaciones a la hora de obtener el caracter \ en la **Línea de Comandos**: se debe teclear `\\`. Todo caracter \ es ignorado. Este aviso es importante en particular para la gestión de archivos (capítulo 15). Para establecer como directorio de trabajo `c:\Mis Documentos` se debe escribir:

```
pondirectorio "c:\\Mis\ Documentos
```

Nota el uso de `\␣` para indicar el espacio entre `Mis` y `Documentos`. Si se omite el doble *backslash*, la ruta definida sería interpretada como:

```
c:Mis Documentos
```

y el intérprete mostrará un mensaje de error.

Del mismo modo, permite obtener símbolos con un significado especial en xLogo:
`\(` ⟶ `(` `\)` ⟶ `)` `\[` ⟶ `[` `\]` ⟶ `]` `\#` ⟶ `#` `\"` ⟶ `"`

### 1.10.2. Mayúsculas y minúsculas

xLogo no distingue entre mayúsculas y minúsculas en el caso de nombres de procedimientos y primitivas. Así, la primitiva `pondirectorio` utilizada antes, si escribes `PONDIRECTORIO` o `PoNDiReCToRio`, el intérprete de comandos interpretará y ejecutará correctamente `pondirectorio`. Por otro lado, xLogo sí respeta mayúsculas y minúsculas en listas y palabras:

```
escribe "Hola
```

proporciona

```
Hola
```

(la H inicial se mantuvo)

### 1.10.3. Las tildes

Desde la versión 0.9.92 las primitivas de xLogo en español admiten tildes. Tratándose de un *software* para uso educativo, es importante que la ortografía sea la adecuada.

Para la acentuación de las primitivas se siguen las normas ortográficas habituales, especialmente en aquellas primitivas compuestas por varias palabras. Por ejemplo:

- `poncolorlápiz`. La palabra `lápiz` lleva tilde y la mantiene al formar parte de la primitiva, ya que la acentuación de esta recae sobre la "`a`"
- `leelineaflujo`. Aunque `línea` lleva tilde al ser esdrújula, al pronunciar la primitiva completa, observamos que es una palabra llana (el acento se encuentra en la "`u`" de `flujo`), así que no se le asigna tilde.

  Sí que lleva tilde en `definelínea` y `finlínea`, por el mismo motivo explicado antes para `lápiz`
- Se procede del mismo modo en las formas cortas de las primitivas. Las formas cortas de `definepolígono` y `finpolígono` son, respectivamente, `defpoli` y `finpoli`. Escuchando a los alumnos pronunciarlas, se optó por considerarlas llanas y sin tilde.

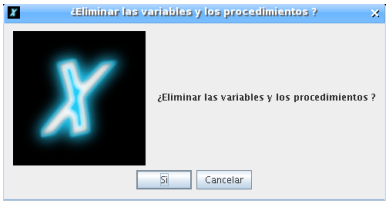
Dicho lo anterior, debemos avisar de que por compatibilidad con otras versiones se mantiene la posibilidad de usar primitivas sin tilde, si bien recomendamos el uso acentuado de las mismas.

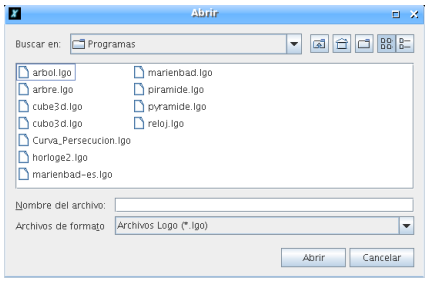
# Capítulo 2

# Opciones del Menú

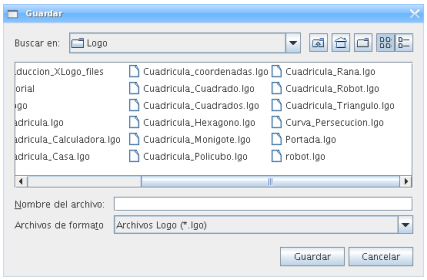
## 2.1. Menú "*Archivo*"

- **Archivo** → **Nuevo**: Elimina todos los procedimientos y variables definidos hasta el momento para comenzar un nuevo espacio de trabajo. Se abrirá una ventana para confirmar la eliminación de todos los procedimientos y variables:




- **Archivo** → **Abrir**: Carga un archivo LOGO previamente guardado en disco.
- **Archivo** → **Guardar como...**: Graba un archivo (`.lgo`) de procedimientos definidos hasta ese momento en el disco, permitiendo asignarle un nombre.

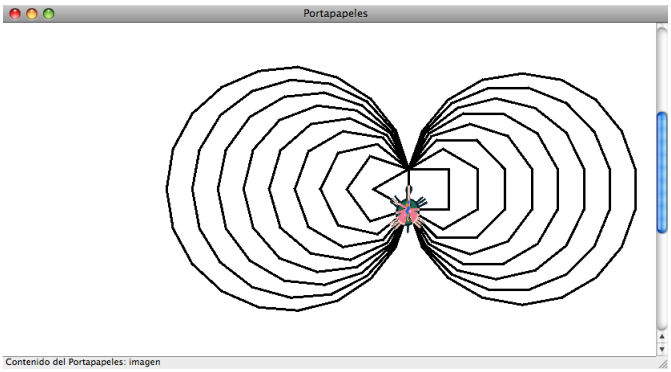



- **Archivo** → **Guardar**: Graba un archivo (`.lgo`) con los procedimientos definidos hasta ese momento en el disco. Esta opción estará deshabilitada mientras no se le haya asignado un nombre como se acaba de explicar en el punto anterior.

- **Archivo → Capturar imagen → Guardar imagen como...**: Permite guardar la imagen del área de dibujo en formato jpg o png. Para ello, puedes seleccionar previamente una parte de la imagen pulsando el botón izquierdo del ratón y arrastrando.

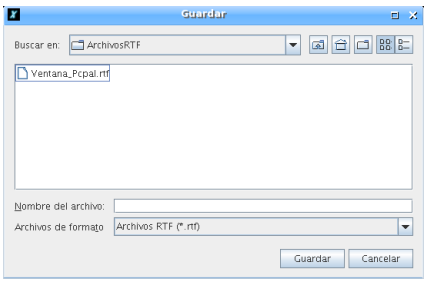
- **Archivo → Capturar imagen → Imprimir imagen**: Permite imprimir la imagen del área de dibujo. Se puede seleccionar una zona a imprimir tal como se explicó para **Guardar**.

- **Archivo → Capturar imagen → Copiar al portapapeles:** Permite enviar la imagen al portapapeles del sistema. Del mismo modo que para **Imprimir** y **Guardar**, se puede seleccionar una zona. Esta opción funciona perfectamente en entornos Windows y Mac:




  pero no así en entornos Linux (el portapapeles no tiene el mismo funcionamiento).

- **Archivo → Zona de texto → Guardar en formato RTF**: Guarda el contenido del **Histórico de Comandos** en un fichero con formato RTF (*Rich Text Format*), manteniendo los colores de los mensajes. Si no se escribe la extensión `.rtf`, se añade automáticamente.



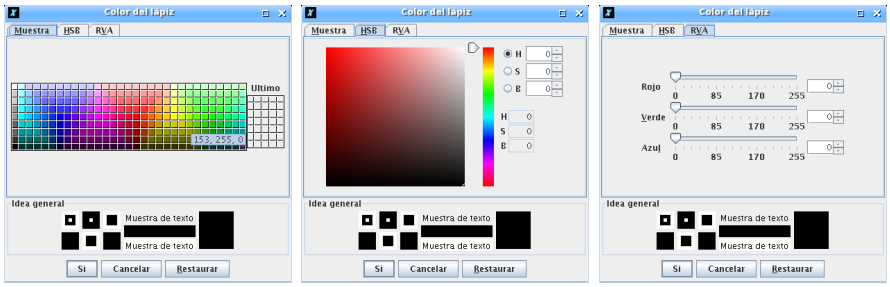

- **Archivo → Salir**: Finaliza la ejecución de xLogo. También puede terminarse la ejecución desde la Línea de comandos con la primitiva `adios`.

## 2.2. Menú "*Edición*"

- **Edición → Copiar**: copia el texto seleccionado en el portapapeles. Atajo de teclado: `Control+C`
- **Edición → Cortar**: corta el texto seleccionado y lo copia en el portapapeles. Atajo de teclado: `Control+X`
- **Edición → Pegar**: pega el texto desde el portapapeles a la línea de comandos. Atajo de teclado: `Control+V`
- **Edición → Seleccionar todo**: Selecciona todo lo que se encuentra escrito en la **Línea de Comandos**.

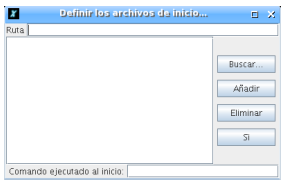
## 2.3. Menú "*Herramientas*"

- **Herramientas → Elegir el color del lápiz**: Permite elegir el color con que la tortuga dibujará, desde la paleta de colores, mediante una definición `HSB` (*Hue*, *Saturation*, *Brightness* - Tonalidad, Saturación, Brillo) o desde una codificación `RVA` (Rojo, Verde y Azul).



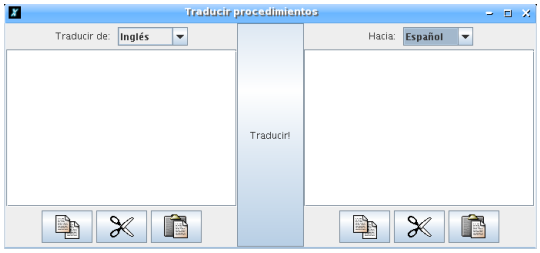

También accesible con el comando `poncolorlapiz`. (Sección 13.2.1)

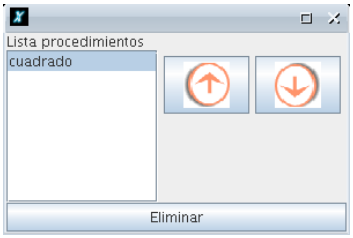
- **Herramientas → Elegir el color de fondo (papel)**: Pone un color como fondo de pantalla, con las mismas características que **Elegir Color del Lápiz**. También accesible con el comando `poncolorpapel`. (Sección 13.2.1)
- **Herramientas → Definir archivos de inicio**: Permite establecer la ruta a los archivos de inicio.




Cualquier procedimiento contenido en estos archivos `.lgo` se convertirán en "*seudo-primitivas*" del lenguaje xLogo. Pero no pueden ser modificadas por el usuario. Así se pueden definir primitivas personalizadas.

- **Herramientas → Traducir procedimientos**: Abre una caja de diálogo que permite traducir los comandos xLogo entre dos idiomas. Es muy útil, en particular, cuando se obtienen códigos Logo en inglés (de *internet*, por ejemplo) para expresarlos en el idioma deseado (Español, Francés, ...)

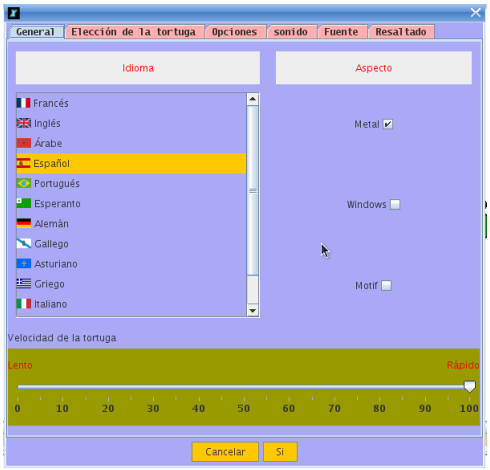


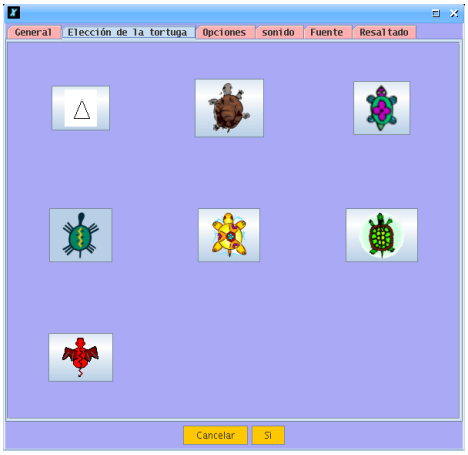




- **Herramientas → Borra procedimientos**: Abre una caja de diálogo que permite selccionar el procedimiento que se desea borrar, de una forma más sencilla que con la primitiva `borra`.

- **Herramientas → Preferencias**: Abre una caja de diálogo donde se pueden configurar varios aspectos:

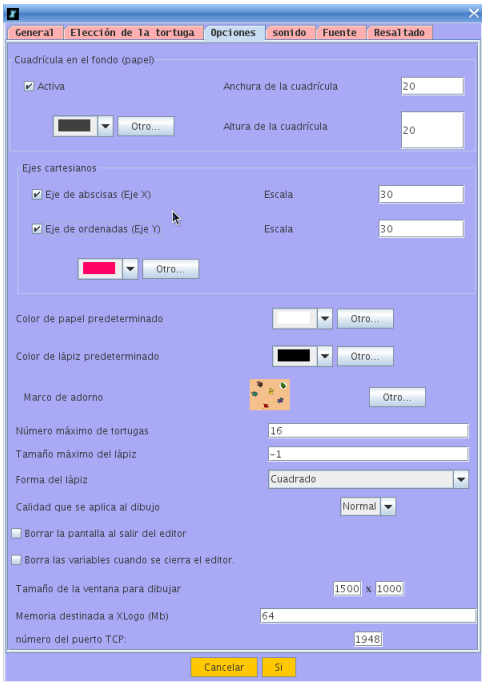
  - **General:**
    - **Idioma**: Permite elegir entre Francés, Inglés, Español, Portugués, Alemán, Esperanto, Árabe, Gallego, Asturiano, Griego, Italiano, Catalán y Húngaro. Nota que las primitivas se adecúan a cada lenguaje.
    - **Aspecto**: Permite definir el aspecto de la ventana xLogo. Están disponibles los estilos "Windows", "Metal" y "Motif".
    - **Velocidad de la tortuga**: Si prefieres ver todos los movimientos de la tortuga, puedes reducir la velocidad con la barra deslizante.



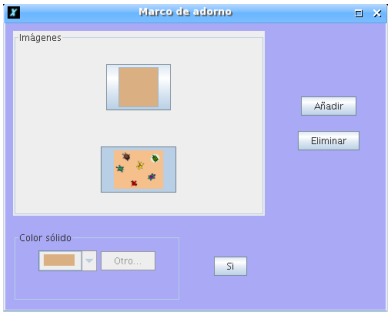




  - **Elección de la tortuga**: Elige entre siete tortugas distintas. También accesible con el comando `ponforma` (Sección 14.1)

- **Opciones:**

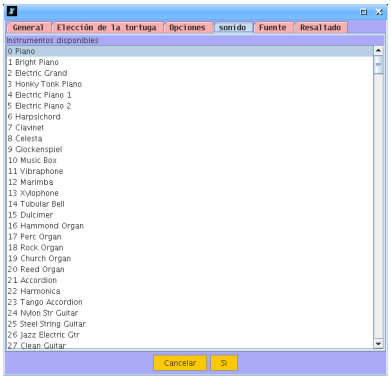


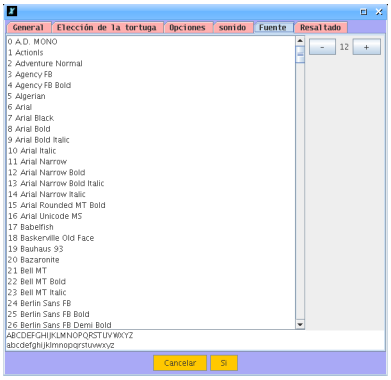

  - **Cuadrícula en el fondo**: Establece (o elimina) una cuadrícula en el fondo del Área de dibujo, así como las medidas de la misma. También accesible con las primitivas `cuadricula` y `borracuadricula`. (Sección 8)
  - **Ejes cartesianos**: Muestra (o retira) los ejes cartesianos (X e Y) del Área de dibujo, establece su escala (separación entre marcas) y su color. También accesible con las primitivas `ejes`, `borraejes`, `ejex`, `ejey` y `poncolorejes` (Sección 8).
  - **Color de papel preasignado**: Permite elegir el color por defecto del papel, es decir, el mostrado al iniciar XLogo.
  - **Color de lápiz preasignado**: Permite elegir el color por defecto del lápiz, es decir, el utilizado al iniciar XLogo.
  - **Marco de adorno**: Permite elegir qué marco de adorno se muestra alrededor del **Área de Dibujo**, una imagen o un color sólido. Para no superar la memoria asignada, la imagen no puede ocupar más de 100 kb.



- **Número máximo de tortugas**: Para el modo multitortuga (sección 14.1). Por defecto 16.
- **Tamaño máximo del lápiz**: Puedes fijar un tamaño límite para el lápiz. Si no se va a utilizar esta limitación, introduce el número `-1` en el recuadro asociado.
- **Forma del lápiz**: Cuadrado o Redondo
- **Tamaño de la ventana para dibujar**: Puedes establecer un tamaño personalizado para el **Área de Dibujo**. Por defecto XLOGO establece un área de 1000 por 1000 *pixels*.
  **Atención**: según aumenta el tamaño de la imagen, puede ser necesario aumentar la memoria destinada a XLOGO. Un mensaje de error advertirá si ocurre esto.
- **Calidad que se aplica al dibujo**: Normal, Alto o Bajo. En calidad "Alta", no se aprecia ningún efecto en particular, pero puede producirse una ralentización de la ejecución.
- **Borrar pantalla al salir del editor**: Sí o No
- **Tamaño de la ventana para dibujar**: Determina las dimensiones del Área de dibujo.
- **Memoria destinada a xLogo**: (en Mb) Por defecto, el valor asignado es 64 Mb. Puede ser necesario aumentarlo cuando el tamaño del **Área de Dibujo** sea demasiado grande. La modificación de este parámetro sólo se hará efectiva tras cerrar y reiniciar XLOGO.
  **Atención**: no aumentar en exceso y/o sin razón este valor, ya que puede ralentizar considerablemente el sistema.
- **Número del puerto TCP**: Puedes cambiar el puerto por defecto para el uso de XLOGO en red y robótica (capítulo 19)

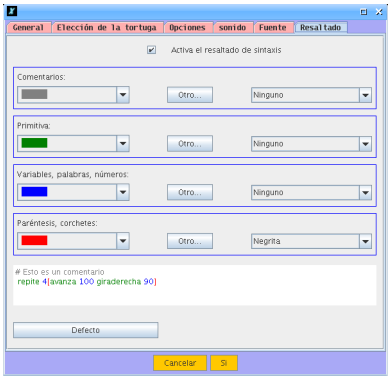
- **Sonido**: La lista de instrumentos que puede imitar la tarjeta de sonido a través de la interfaz MIDI. Puedes seleccionar un instrumento concreto en cualquier momento, mediante la primitiva `poninstrumento n`.




Si usas Windows, recuerda instalar las librerías `soundbank.gm` como se explicó en la sección 1.2.1.

- **Fuente**: Elige el tipo de letra y su tamaño

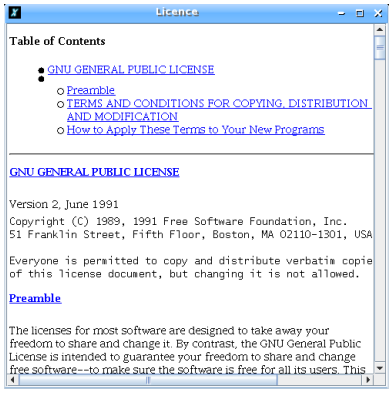


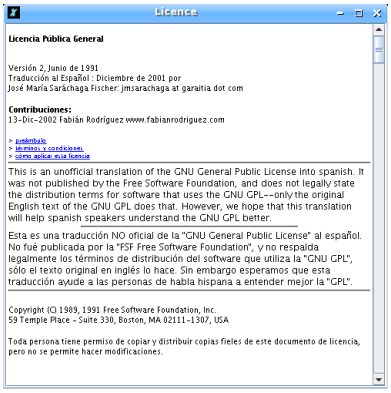

- **Resaltado**: Elige los colores que se utilizarán en el resaltado de primitivas, palabras, variables números, paréntesis y corchetes.

## 2.4. Menú "*Ayuda*"

- **Ayuda → Manual en línea**: Muestra el manual de referencia de XLOGO, para lo que es necesario estar conectado a *internet*.

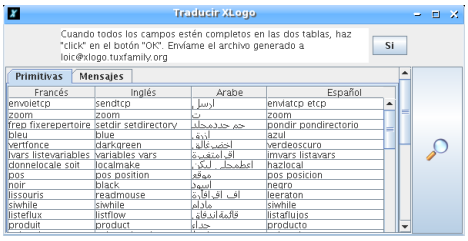
- **Ayuda → Licencia**: Muestra la licencia original GPL (en Inglés) bajo la cual se distribuye este programa.



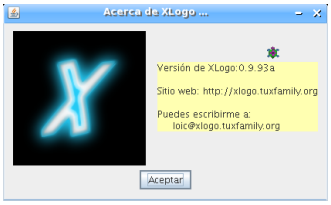

- **Ayuda → Traducción de la Licencia**: Refiere a una traducción al español de la licencia GPL, sin efectos legales, sólo como referencia para entender la versión en Inglés.

- **Ayuda → Traducir xLogo**: Abre una ventana para añadir y/o corregir traducciones.


Desde ella pueden crearse y/o modificarse tanto las primitivas como los mensajes. Una vez creados/modificados, haz *clic* **fuera** de la celda que acabas de escribir y pulsa el botón `Si`. Se abrirá una ventana donde debes elegir el fichero de texto donde guardar los cambios y que debes enviar a `loic@xlogo.tuxfamily.org`


- **Ayuda → Acerca de...**: Lo de siempre ... y ¡¡`xlogo.tuxfamily.org` para guardar en favoritos!! o:)



# Capítulo 3

# Presentación de la tortuga

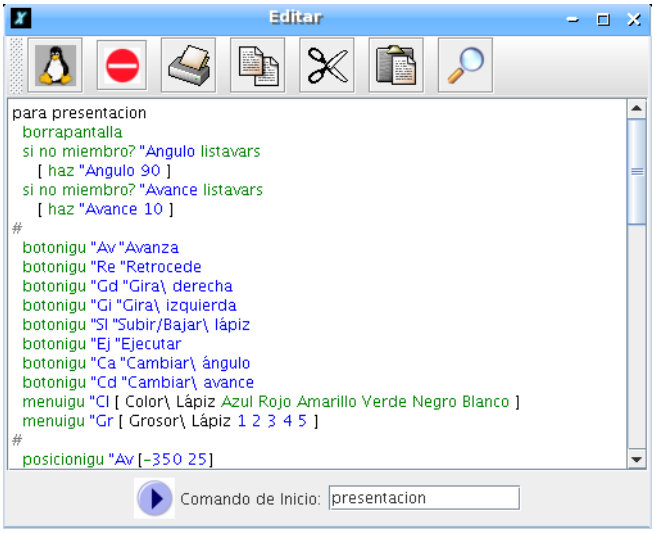
¿Cómo presentamos a nuestra tortuga?

## 3.1. Un programa de ejemplo

Empecemos por el siguiente procedimiento:

`http://downloads.tuxfamily.org/xlogo/downloads-sp/curso/presentacion.lgo`

Descarguemos el fichero a nuestro disco duro, abrámoslo (**Archivo** → **Abrir** → Elegir la ubicación donde lo hayamos descargado → *clic* en `Abrir`) y en la ventana que nos aparece (la del Editor de Procedimientos):




comprueba que la línea de `Comando de inicio` aparece la palabra:

`presentacion`

(si no, escríbela tú). No te preocupes de momento por todo lo que hay escrito, en breve serás capaz de diseñarlo tú mismo/a). Finalmente, hacemos *clic* en el pingüino (o tortuga)

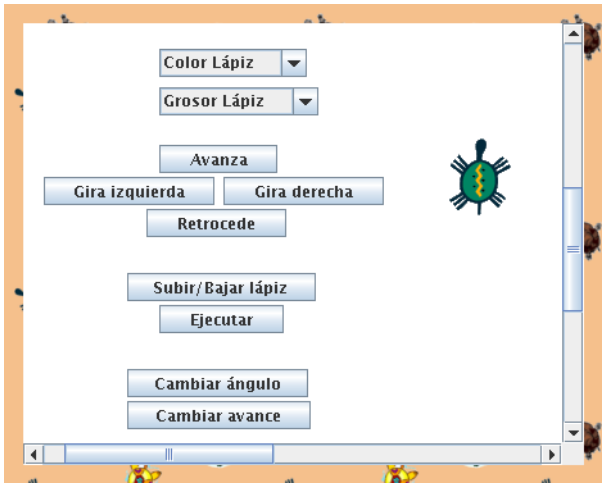
y ... ¿no pasa nada?

Observa el Histórico de Comandos. xLogo te informa de que:

```
Acaba de definir presentacion.
```

pero además, has definido la orden que se ejecutará al pulsar el botón de `Comando de inicio`

No esperes más. Pulsa el botón y ... deberías obtener una pantalla como esta:



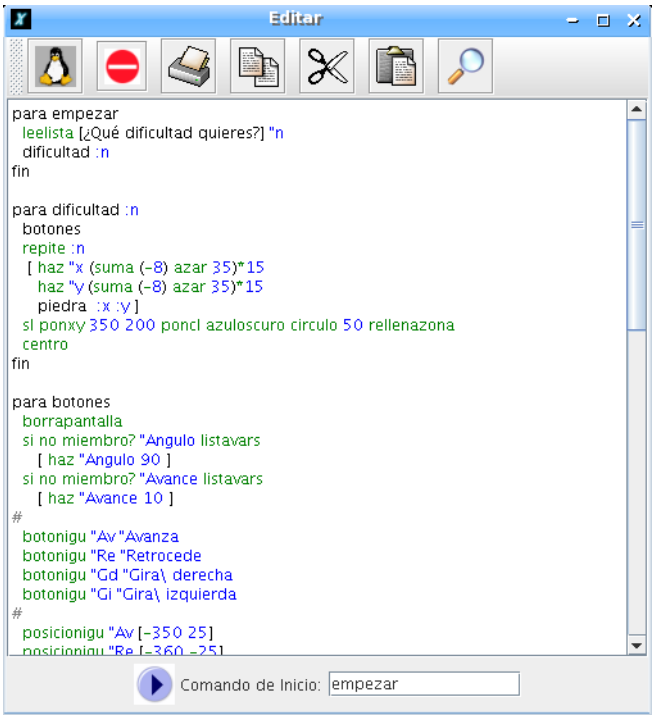

**Haz todas las pruebas que necesites para "entender" cómo se mueve la tortuga, cómo mide los ángulos, las distancias, ... Fíjate, especialmente, en cómo realiza los giros y procura entender el punto de vista de la tortuga.**

¿Qué podemos hacer con una pantalla como la de arriba? Podemos, por ejemplo, enseñar la lateralidad en los últimos cursos de Infantil y reforzarla en los primeros cursos de Primaria:

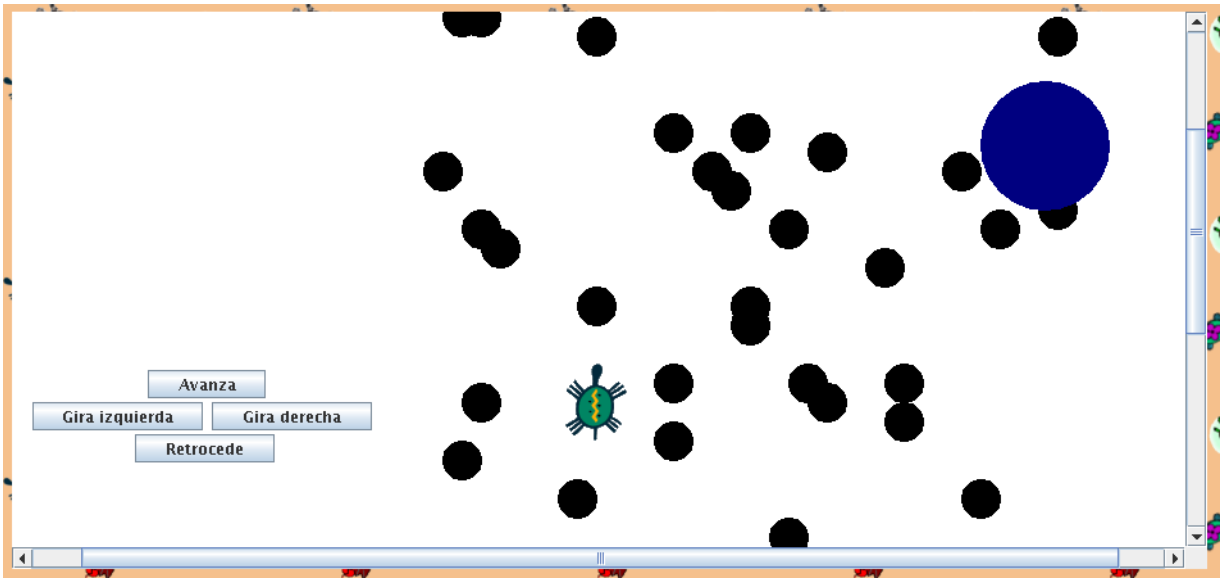
Descarguemos el fichero

```
http://downloads.tuxfamily.org/xlogo/downloads-sp/curso/juego.lgo
```

abrámoslo y comprueba que el Comando de Inicio ahora es `empezar`:




Guardamos (hacemos *clic* en el pingüino o tortuga) y después en el de Comando de inicio. Después de contestar a la pregunta (un valor de 20 ó 25 está bien) el resultado debe ser una pantalla similar a esta:

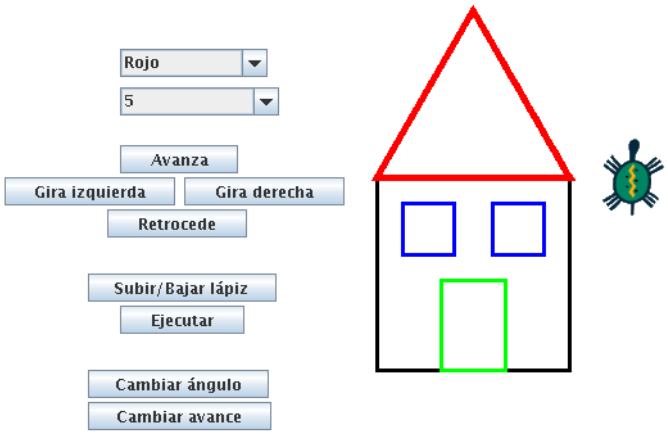



(Usa las barras de desplazamiento si no ves alguna de las figuras de esta captura).

Acabas de cargar un sencillo juego que consiste en llevar a la tortuga hasta el "lago" situado en la parte superior derecha usando sólamente los botones de desplazamiento. Los círculos negros representan piedras y están colocadas aleatoriamente:

- Si "choca" con una piedra, nos aparece un aviso y vuelve al punto de partida
- Cuando llegue al lago, nos felicitará y sonará una cancioncilla (concretamente, *supercalifragilisticoexpialidoso* de la película **Mary Poppins**).

Sin embargo, la creación de este tipo de juegos no es la principal característica que xLogo aporta al aula. El verdadero método de trabajo con Logo se basa en proponer actividades (especialmente gráficas) para que el alumno se comporte como el **maestro de la tortuga** y le indique, razonadamente, qué pasos debe seguir para conseguirlas. Por ejemplo, para dibujar esta casa:




El alumno debe:

1. Entender él mismo aquello que va a enseñar.
2. Planear una forma de impartirlo.
3. "Trocear" el problema en *mini*problemas más fácilmente abordables.
4. Saber cómo comunicarlo claramente.
5. Establecer este nuevo *conocimiento* como las bases de uno futuro.
6. Ser consciente del conocimiento que *su* "alumno" (la tortuga) ya tiene, y construir basándose en él.
7. Ser receptivo para explorar nuevas ideas según aparezcan.
8. Responder a los malentendidos de *su* alumno.

Además, lo más importante en el proceso de aprendizaje con Logo **no** es el resultado final, sino **cómo haces lo que haces**; es decir, ver cómo se crea el diseño es más interesante y

educativo que el diseño en sí.

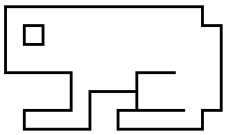
Logo no está limitado a ningún área o tema en particular (a lo largo de este curso se irán planteando ejercicios de Lengua, Física, ...). Sin embargo es muy útil para matemáticas, ya que las gráficas generadas por la tortuga, medir sus movimientos en distancias y grados, ... permiten estudiar geometría mediante la construcción de polígonos y figuras.

**Echa un vistazo a las órdenes contenidas en los ficheros con los que hemos trabajado, e intenta "adivinar" qué hacen, qué parámetros les acompañan, por qué algunas están coloreadas y otras no, ...**

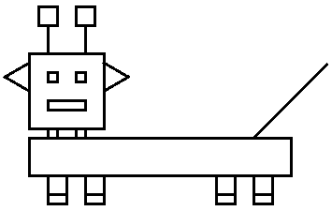
## 3.2. Ejercicios

Con la pantalla del programa `presentacion`, intenta dibujar:

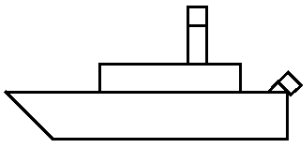
*a*) Una rana:



*b*) Un robot:

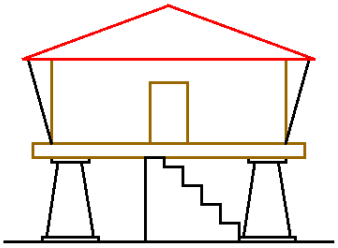


*c*) Un barco:

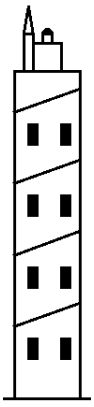


*d*) Tres pirámides:

*e*) Un hórreo:



*f*) Este faro:

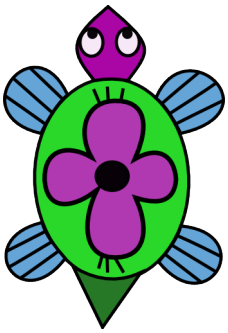


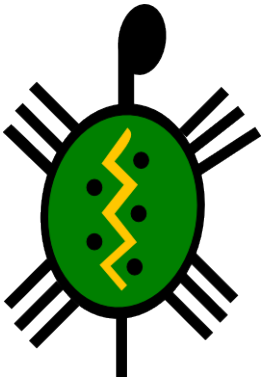
## 3.3. Una ayuda al dibujo

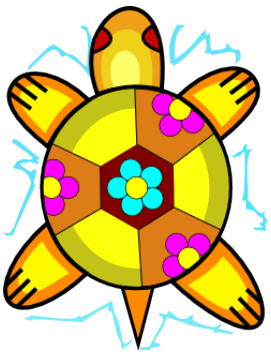
Antes de empezar a dar órdenes a nuestra tortuga, deberíamos acostumbrarnos a dibujar nosotros mismos las figuras en un papel (lo mismo vale cuando los procedimientos no dibujen, sino que hagan operaciones numéricas o con listas). Ver los dibujos en pantalla no es fácil y, en muchos casos, no obtendremos buenos resultados sin haber dibujado antes sobre el papel.

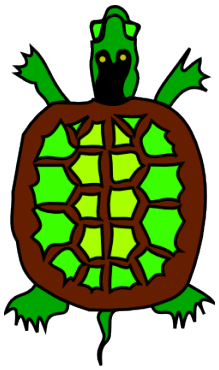
Una pequeña ayuda: en la página siguiente, recorta la imagen de la tortuga que prefieras, o dibuja tú otra que te guste más, y pónsela al lápiz. Así harás dibujar a la tortuga tanto en tu papel como en el ordenador.

Te aconsejamos que uses un lápiz con una goma al otro lado y que tengas cerca lápices de colores, ya que encontraremos órdenes que nos ayudarán a borrar lo que hicimos mal y ya has visto que podemos cambiar el color del lápiz.









# Capítulo 4

# Iniciación a la Geometría de la tortuga

Comenzamos la descripción de las *primitivas* de xLogo.

## 4.1. Descripción de las primitivas o comandos

Las primitivas son órdenes básicas que el programa ya tiene incorporadas; al escribirlas en la *línea de comandos* ordenan a la tortuga realizar una acción determinada. La ventaja de Logo frente a otros lenguajes radica en que las órdenes pueden escribirse en el idioma *natural* del usuario/programador, de modo que resultan muy fáciles de entender.

Cada primitiva puede tener un cierto número de parámetros que son llamados *argumentos*. Por ejemplo, la primitiva `pi`, que devuelve el valor del número $\pi$ (`3.141592653589793`), no lleva argumentos: la primitiva `escribe` espera uno (`escribe` muestra en el Histórico de Comandos ese argumento), mientras que la primitiva `suma` (sección 7.1) tiene dos argumentos.

`escribe suma 2 3` devuelve `5`.

Los argumentos en Logo son de tres tipos: **Número**, **Palabra** o **Lista**

- **Números**: Algunas primitivas esperan números como su argumento: `avanza 100` (sección 4.2) es un ejemplo.
- **Palabras**: Las palabras se escriben precedidas por `"`. Un ejemplo de una primitiva que admite una palabra como argumento es `escribe`.

  `escribe "hola` devuelve `hola`

  Nota que si olvidas las comillas (`"`), el intérprete devuelve un mensaje de error. En efecto, `escribe` esperaba ese argumento, pero para el intérprete, `hola` no representa nada, ya que no fue definido como número, ni palabra, ni lista, ni procedimiento.

- **Listas**: Se definen encerrándolas entre corchetes.

  `escribe [El gato es gris]` devuelve `El gato es gris`

Los números son tratados a veces como un valor (por ej: `avanza 100`), o bien como una palabra (por ejemplo: `escribe vacio? 12` devuelve `falso` – sección 9.4).

Algunas primitivas tienen una forma general, esto es, pueden ser utilizadas con números o *argumentos opcionales*. Estas primitivas son:

- `escribe`
- `suma`, `producto` (sección 7.1)
- `o`, `y` (sección 9.2)
- `lista`, `frase`, `palabra` (sección 10.1)

Para que el intérprete las considere en su forma general, tenemos que escribir las órdenes entre paréntesis. Observa los ejemplos:

```
escribe (suma 1 2 3 4 5)
```

devuelve:

```
15
```

También:

```
escribe (lista [a b] 1 [c d])
```

devuelve:

```
[a b] 1 [c d]
```

y

```
si (y 1=1 2=2 8=5+3) [avanza 100 giraderecha 90]
```

## 4.2. Movimientos de la tortuga

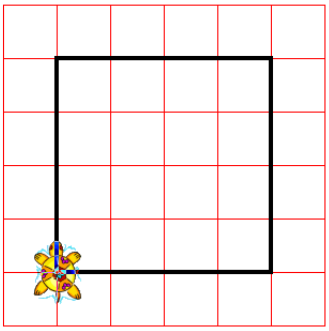
Empecemos por las primitivas que controlan el movimiento de la tortuga:

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| `AV`anzar `n` pasos | `avanza n` | `av n` |
| `RE`troceder `n` pasos | `retrocede n` | `re n` |
| Gira `D`erecha `n` grados | `giraderecha n` | `gd n` |
| Gira `I`zquierda `n` grados | `giraizquierda n` | `gi n` |
| Llevar la tortuga al `centro` de la pantalla | `centro` | |

Observa que no todas las primitivas son necesarias. Por ejemplo:

- `giraizquierda 90` equivale a `giraderecha -90`
- `retrocede 100` equivale a `avanza -100`

Por ejemplo, para dibujar un cuadrado,



podemos teclear:

```
avanza 200 giraderecha 90 avanza 200 giraderecha 90 av 200 gd 90 av 200
```

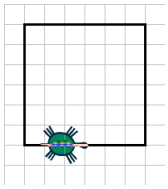
donde puedes ver que podemos utilizar indistintamente las primitivas en su forma larga o en la forma abreviada sin problemas.

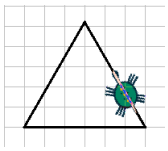
**Haz todas las pruebas que necesites para entender perfectamente estas primitivas. Comprueba que la predicción del tema anterior de cómo realiza los giros es correcta, y si has entendido bien el punto de vista de la tortuga**

## 4.3. Ejercicios

En los dibujos, el lado de cada cuadrado de la cuadrícula mide 25 "pasos de tortuga".

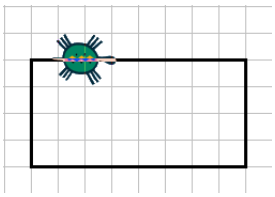
1. Dibuja el borde de un cuadrado en sentido *antihorario*
2. Dibuja el borde de un cuadrado en sentido *horario*

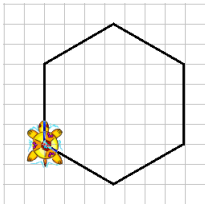




3. Dibuja el borde de un triángulo equilátero en sentido *antihorario*

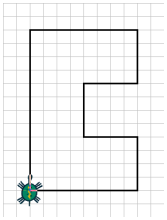
4. Dibuja el borde de un rectángulo

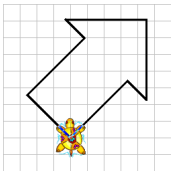


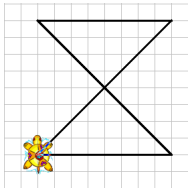


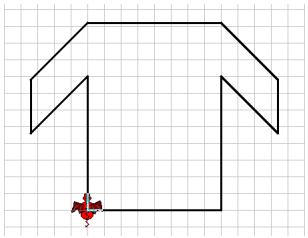
5. Dibuja el borde de un hexágono regular

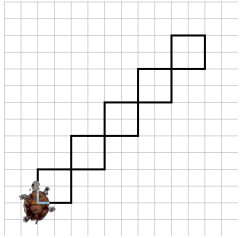
6. Dibuja:

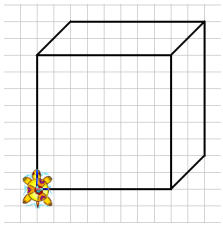
*a*) 

*b*) 

*c*) 

*d*) 

*e*) 

*f*) 

La tortuga tiene muy "mala memoria", así que todas las órdenes que has dado para dibujar los ejemplos **se perderán**. Puedes hacer dos cosas: copiarlas en un papel, o usar la Opción del Menú: Archivo ⟶ Zona de Texto ⟶ Guardar en formato RTF.

## 4.4. Avanzando un poco

Continuemos con primitivas que controlan otros aspectos:

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| Borrar Pantalla y tortuga al centro | `borrapantalla` | `bp` |
| Subir Lápiz (no deja trazo al moverse) | `subelapiz` | `sl` |
| Bajar Lápiz (sí deja trazo al moverse) | `bajalapiz` | `bl` |
| Ocultar Tortuga | `ocultatortuga` | `ot` |
| Mostrar Tortuga | `muestratortuga` | `mt` |

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| Cambiar el color del trazo con que dibuja | `poncolorlapiz n` | `poncl n` |
| Cambiar el grosor del trazo con que dibuja | `pongrosor n` | |
| Borrar por donde pasa, en vez de escribir | `goma` | `go` |
| Dejar de borrar y volver a escribir | `bajalapiz` | `bl` |
| Rellenar con el color activo una región cerrada | `rellena` | |
| Rellenar una región limitada por el color activo | `rellenazona` | |
| `Limpiar` la pantalla dejando la tortuga en el sitio | `limpia` | |
| Repetir `n` veces lo indicado entre corchetes | `repite n` | |

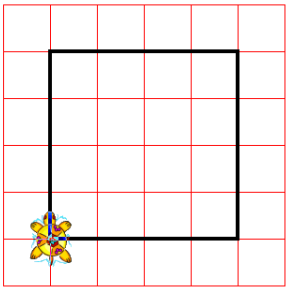
La primitiva `poncolorlapiz` debe ir acompañada de un número (ver seción 13.2.2), y las opciones con las que trabajaremos de momento serán:

| | | | |
|---|---|---|---|
| 0: negro | 1: rojo | 2: verde | 3: amarillo |
| 4: azul | 5 magenta | 6: cyan | 7 blanco |

Respecto a `repite`, las órdenes a repetir deben ir entre corchetes, por ejemplo:

```
repite 4 [escribe "Hola ]
```

escribe 4 veces la palabra `Hola`
Por ejemplo, para dibujar el mismo cuadrado del ejemplo anterior:



habíamos escrito:

```
av 200 gd 90 av 200 gd 90 av 200 gd 90 av 200
```

Es fácil observar que hay órdenes que se repiten cuatro veces. Si pensamos un momento, podemos ver que añadir un `giraderecha` más no va a modificar el dibujo, así que podremos escribir:

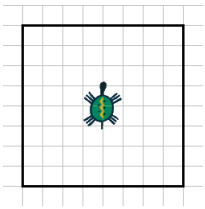
```
repite 4 [ avanza 200 giraderecha 90 ]
```

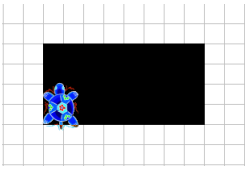
con el mismo resultado.

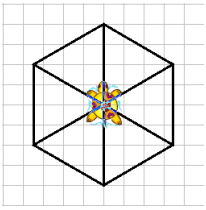
**Analiza los programas con los que dibujaste antes los cuadrados, el triángulo y el hexágono.** ¿Ves cómo se repiten varias veces las mismas órdenes? Aplica lo que acabamos de ver con la primitiva `repite` para hacer más sencillos tus programas.

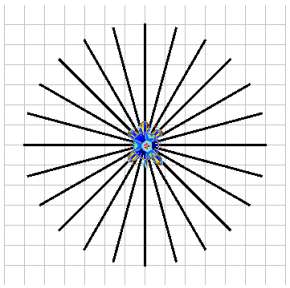
## 4.5. Ejercicios

1. Dibuja el borde de un cuadrado, pero ahora usa la primitiva `repite`
2. Dibuja el borde de un triángulo equilátero usando la primitiva `repite`
3. Dibuja el borde de un hexágono regular usando la primitiva `repite`
4. Dibuja el borde de un cuadrado, cuyo centro esté en el centro de la pantalla
5. Dibuja un rectángulo, rellenando el interior


Problema 4


Problema 5


Problema 6

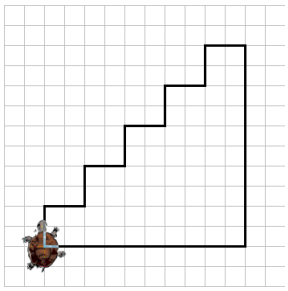

Problema 7

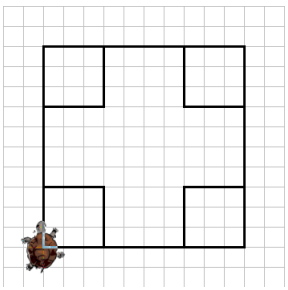
De nuevo, el lado de cada cuadrado de la cuadrícula mide 25 "pasos de tortuga"

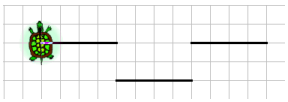
6. Dibuja el borde de un hexágono regular y las diagonales cuyos extremos son dos vértices opuestos del mismo

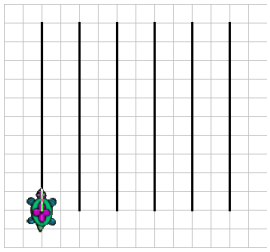
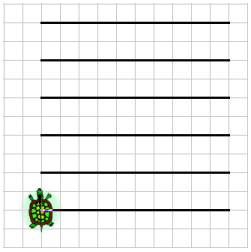
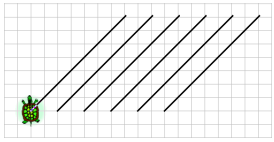
**Acabas de dibujar polígonos de 3, 4 y 6 lados. ¿Serías capaz de determinar una regla para hallar el ángulo de giro en cualquier polígono?**
Observa que la tortuga da una vuelta completa alrededor del polígono, que al final del proceso vuelve a estar mirando hacia arriba y fíjate cuántas veces tiene que girar.

7. Dibuja los radios de una rueda. En total tienen que salirte 24
8. Dibuja:

*a*) 

*b*) 

*c*) 

*d*)  *e*)  *f*) 

| | |
|---|---|
|  | Recuerda copiar las respuestas a los problemas en un papel, o guardarlas en el disco duro:<br>Archivo ⟶ Zona de Texto ⟶ Guardar en formato RTF. |

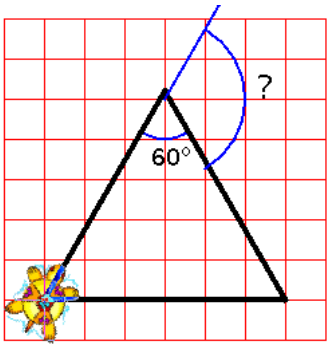
| | |
|---|---|
|  | **¿Qué te parece este método para dibujar? ¿Crees que puede mejorarse? ¿Cómo?**<br>¿Qué pasa si quiero que los polígonos tengan lados más largos o más cortos? ¿Y si quiero cambiar el tamaño de las figuras que acabas de conseguir? |

## 4.6. Aplicación didáctica de xLogo

Ya hemos visto cómo dibujar el cuadrado, pero podemos analizar un poco mejor el proceso que hemos seguido para hacerlo.

### 4.6.1. El triángulo equilátero

Vamos a ver cómo trazar este triángulo equilátero de 180 pasos de tortuga:




Aquí, un cuadrado representa 30 pasos de tortuga

Las órdenes serán algo del estilo:

```
repite 3
  [ avanza 180 giraderecha ... ]
```

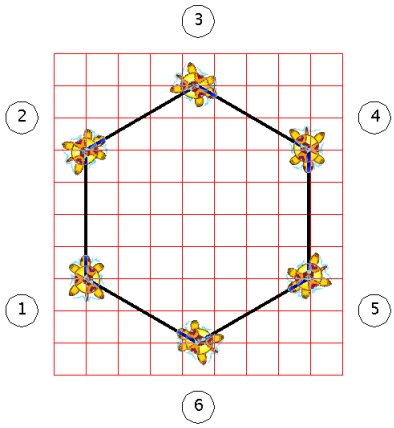
Queda por determinar el ángulo correcto. En un triángulo equilátero, los ángulos valen todos 60 grados, y como la tortuga debe volver por el exterior del triángulo, el ángulo valdrá:

```
180 - 60 = 120 grados
```

Las órdenes son, pues:

```
repite 3
  [ avanza 180 giraderecha 120 ]
```

### 4.6.2. El hexágono




Un cuadrado = 20 pasos de tortuga.

Para un cuadrado repetíamos 4 veces, para el triángulo 3 veces, parece claro que para el hexágono será:

```
repite 6
  [ avanza 80 giraderecha ... ]
```

Date cuenta que en su desplazamiento, la tortuga realmente da una vuelta completa sobre ella misma. (Inicialmente está orientada hacia arriba y termina en esta misma posición). Esta rotación de 360 grados se efectúa en 6 etapas. Por tanto, cada vez, gira

```
360/6 = 60 grados
```

Las órdenes son, entonces:

```
repite 6
  [ avanza 80 giraderecha 60 ]
```

### 4.6.3. Trazar un polígono regular en general

En realidad, reiterando el razonamiento anterior, puedes darte cuenta de que para trazar un polígono de `n` lados, el ángulo se obtendrá dividiendo `360` por `n`. Por ejemplo:

- Para trazar un pentágono regular de lado `100`: `(360:5=72)`

  ```
  repite 5 [ avanza 100 giraderecha 72 ]
  ```

- Para trazar un eneágono regular de lado `20`: `(360:9=40)`

  ```
  repite 9 [ avanza 20 giraderecha 40 ]
  ```

- Para trazar un eh ... `360`-gono[1] regular de lado `2` (que se parece mucho a un círculo):

  ```
  repite 360 [ avanza 2 giraderecha 1 ]
  ```

- Para trazar un heptágono regular de lado `120`:

  ```
  repite 7 [ avanza 120 giraderecha 360/7 ]
  ```

En el siguiente capítulo vamos a aprender cómo evitar que *se pierdan* las órdenes que hemos dado a la tortuga, de un modo más simple que guardando en RTF y copiando.

## 4.7. Función avanzada de relleno

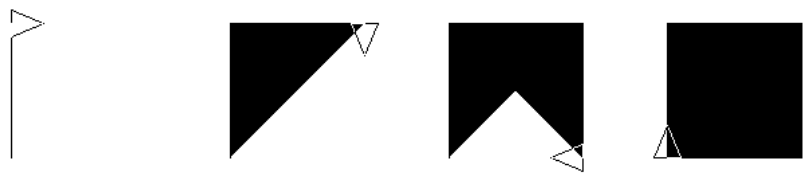
xLogo posee una tercera primitiva de relleno, `rellenapoligono`. Su único argumento es una lista que debe contener las instrucciones para dibujar una figura poligonal cerrada:

```
rellenapoligono [ lista_de_instrucciones]
```

Esta primitiva rellena la forma creada por triangulación (utilizando una serie de triángulos), de modo que cada vez que la tortuga dibuja una línea, el triángulo que se genera es rellenado con el color activo.

Por ejemplo:

```
rellenapoligono [ repite 4 [avanza 100 giraderecha 90]]
```

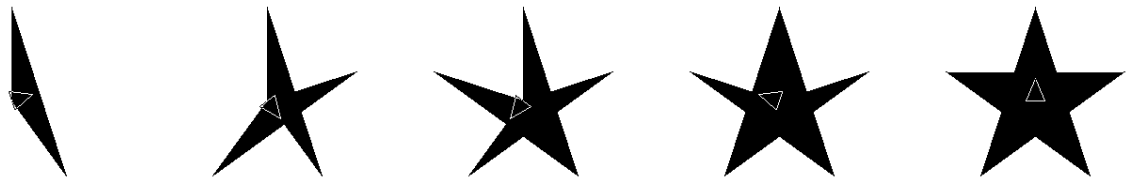


Paso 1 Paso 2 Paso 3 Paso 4

[1] Trihectahexacontágono: tri = 3, hecta = 100, hexaconta = 60, gono = ángulo

Utilizado adecuadamente, puede proporcionar resultados llamativos:

```
repite 5
  [ avanza 100 rellenapoligono [ retrocede 100 giraderecha 144 avanza 100 ]
    avanza 100 giraizquierda 72]
```



Paso 1 Paso 2 Paso 3 Paso 4 Paso 5

# Capítulo 5

# Procedimientos y subprocedimientos

## 5.1. Procedimientos

El capítulo anterior nos mostró varias cosas que deberían "preocuparnos": En primer lugar, la tortuga "olvida" cómo dibujamos las figuras, es decir, las órdenes que le damos desaparecen de una vez para la siguiente.

En segundo, cada figura obligaba a teclear las órdenes cambiando las medidas una a una. Por ejemplo, para dibujar un cuadrado de lado `100`, debíamos escribir:

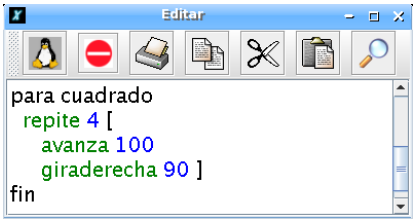
```
repite 4 [ avanza 100 giraderecha 90 ]
```

cada vez que queríamos que apareciera en pantalla.

¿Y si es un rectángulo? La secuencia es más larga, y si quiero otro tamaño debería modificar **dos** medidas.

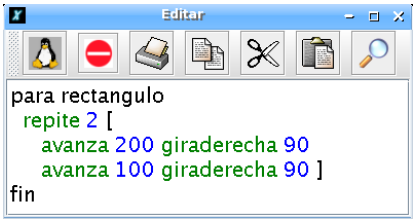
**Podemos hacer que xLogo "aprenda" nuevas primitivas, definiendo lo que se denomina `procedimientos`.**

Haz *clic* en el botón `Editar`, en la ventana emergente que acaba de aparecer escribe:




y haz *clic* en el botón del pingüino/tortuga. Acabas de definir el procedimiento `cuadrado`, y la tortuga dibujará un cuadrado de lado `100` cada vez que le digas `cuadrado`.

Prueba ahora con el siguiente procedimiento:




¿Qué aparece al escribir `rectangulo` en la línea de comandos?

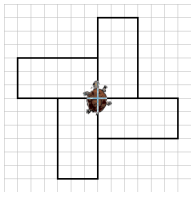
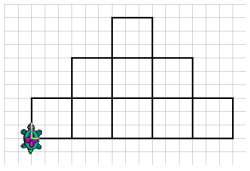
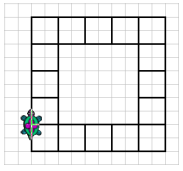
**Intenta crear procedimientos llamados `avanza`, `cosa`, `cuadricula` y `circulo`. ¿Qué observas al escribir el nombre? ¿Por qué crees que es? ¿Qué ocurre al guardar el procedimiento?**

Un procedimiento debe contener **obligatoriamente**, las "primitivas" `para`, seguida de una `palabra` que indicará el **nombre** que deseamos darle al procedimiento, y `fin`, que indica el final de un procedimiento.

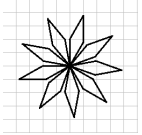
Más adelante veremos otras opciones para los procedimientos.

## 5.2. Ejercicios

1. Plantea un procedimiento `triangulo`, que dibuje el borde de un triángulo equilátero
2. Plantea un procedimiento `hexagono`, que dibuje el borde de un hexágono regular
3. Plantea un procedimiento que dibuje un cuadrado en el centro de la pantalla
4. Plantea un procedimiento que dibuje los 24 radios de la rueda de una bicicleta
5. Usando los procedimientos `rectangulo` y `cuadrado` definidos antes, dibuja:

*a*)  *b*)  *c*) 

6. Plantea un procedimiento que dibuje un rombo, y con él dibuja la hélice:

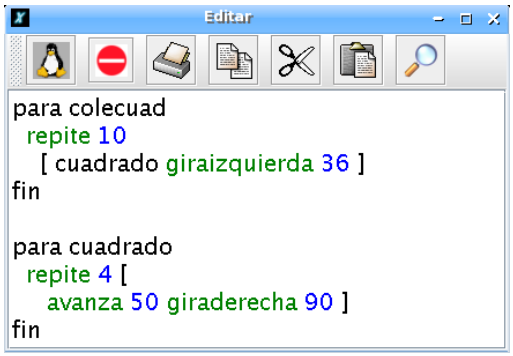


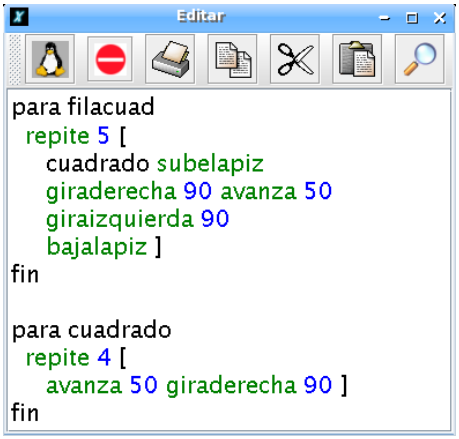
| | |
|---|---|
|  | **Observa que los nombres de los procedimientos están relacionados con el dibujo (u objetivo) que deseamos. Esta es una norma de *buena educación* a la hora de programar, y hace más inteligibles los programas.** |

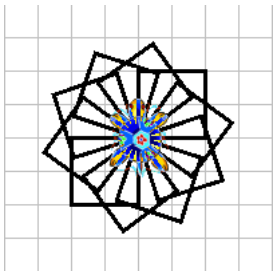
## 5.3. Sub-procedimientos

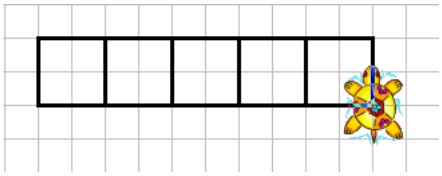
Podemos conseguir efectos interesantes combinando procedimientos, es decir, haciendo que un procedimiento llame a otro.

Los programas siguientes dibujan una colección de cuadrados con un vértice o un lado común, aprovechando el procedimiento `cuadrado` definido antes:



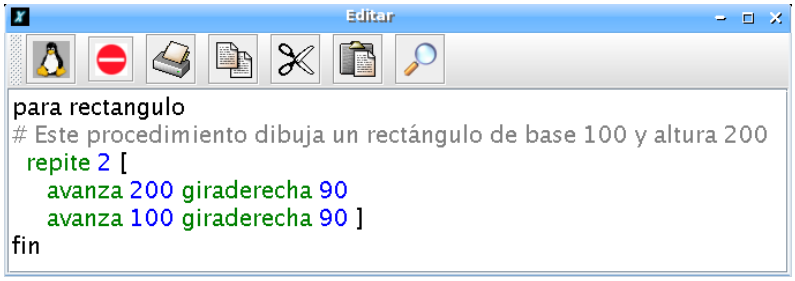





Antes citábamos una norma de *buena educación*, a partir de ahora la llamaremos *estilo*. También se aconseja *indentar* las líneas para reconocer más fácilmente dónde empiezan y dónde acaban determinadas secuencias de órdenes.

Finalmente, hablaremos de los *comentarios*. En xLogo pueden añadirse líneas que NO serán interpretadas por la tortuga. Estas líneas se llaman `comentarios`; sirven para explicar qué hace un programa. Teclea el siguiente procedimiento:




Puedes ver que la segunda línea empieza por `#`, y aparece en color gris. La "*almohadilla*" indica a la tortuga que es un comentario, así que ignora la línea y sigue leyendo las siguientes.

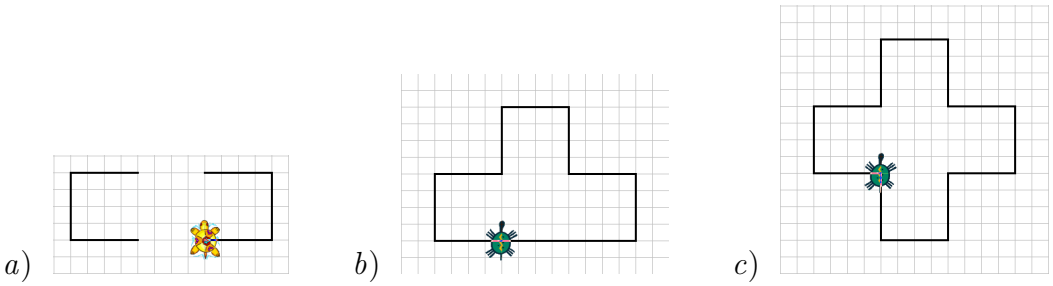
**Usar la forma larga de las primitivas, buscar nombres para las variables relacionados con aquello con lo que estamos trabajando, indentar y comentar los programas ayuda a hacerlos más fácilmente legibles.**

Para parar la ejecución de un procedimiento antes de llegar al final (es decir, hacerle saltar directamente hasta `fin`) se puede usar la primitiva `alto`, pero es mejor intentar no usarla. Igualmente, si vemos que un procedimiento no se termina por él mismo, podemos hacer *clic* en el botón `Alto`, situado al lado del botón de `Editar`.

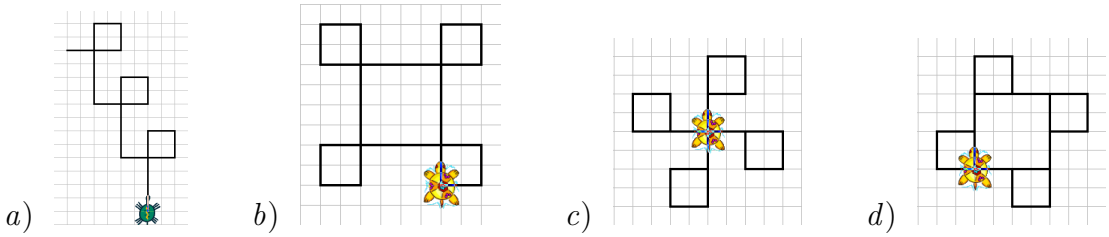

## 5.4. Ejercicios

1. Plantea un procedimiento que dibuje una colección de hexágonos regulares con un vértice en común, cada uno girado 60 grados respecto del anterior

2. Observa que en las siguientes figuras puedes ver un cuadrado al que le falta un lado. Modifica el procedimiento `cuadrado` para que sólo dibuje tres lados y dibuja:




3. Observa las siguientes figuras y busca un patrón que se repite. Úsalo para dibujar:

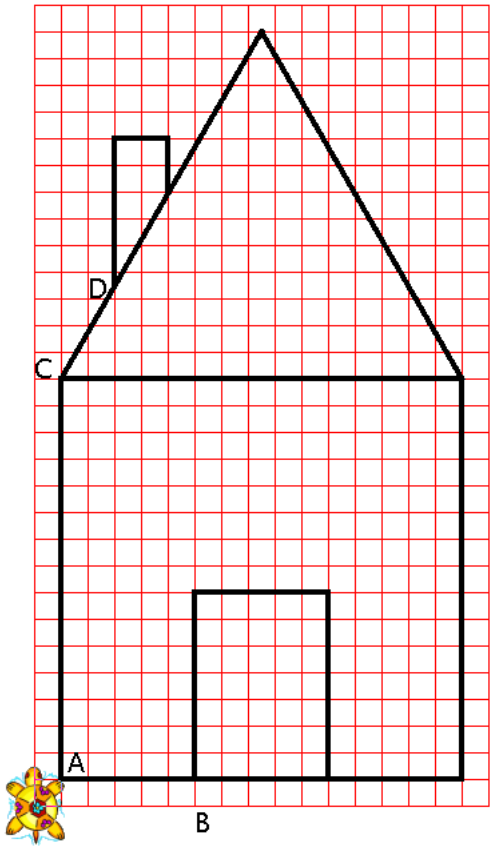



4. Plantea un procedimiento que dibuje una fila horizontal de cinco triángulos equiláteros, cuyas bases estén contenidas en la misma recta

**IMPORTANTE**: Los procedimientos pueden guardarse en el disco duro de tu ordenador de forma que luego puedes recuperarlos sin re-escribirlos como se explicó en 2.1

## 5.5. Actividad avanzada

Debes conseguir el dibujo que se muestra a continuación.




Cada cuadrado vale 10 pasos de tortuga.

Para ello, deberás definir ocho procedimientos:

- Un procedimiento "`cuadrado`" que trazará el cuadrado básico de la casa
- Un procedimiento "`tri`" que trazará el triángulo equilátero que representa el tejado
- Un procedimiento "`puerta`" que trazará el rectángulo que representa la puerta
- Un procedimiento "`chi`" que trazará la chimenea
- Un procedimiento "`desp1`" que desplazará la tortuga de la posición A a la B
- Un procedimiento "`desp2`" que llevará a la tortuga desde la posición B a la C
- Un procedimiento "`desp3`" que hará a la tortuga ir de la posición C a la D
- Un procedimiento "`casa`" que trazará la casa en su totalidad ayudándose de todos los demás procedimientos

# Capítulo 6

# Variables. Procedimientos con argumentos

Muchas veces se necesita dibujar una misma figura varias veces, pero con distintas dimensiones. Por ejemplo, si queremos dibujar un cuadrado de lado 100, otro de lado 200 y un tercero de lado 50, con lo que sabemos hasta ahora necesitaríamos tres procedimientos distintos:

```
para cuadrado 1
  repite 4 [avanza 100 giraderecha 90]
fin
para cuadrado 2
  repite 4 [avanza 200 giraderecha 90]
fin
para cuadrado 3
  repite 4 [avanza 50 giraderecha 90]
fin
```

Es evidente que necesitamos una forma más simple de hacerlo, y que debería ser posible definir un único procedimiento que, de algún modo, permitiera cambiar el **argumento** de la primitiva `avanza`, es decir, el lado del cuadrado.

Ese es el papel de las **variables**.

## 6.1. Primitivas asociadas

Definimos ahora seis nuevas primitivas:

| Descripción | Primitiva | Ejemplo |
|---|---|---|
| Guardar un valor en una variable | `haz` | `haz "lado 115` |
| Utilizar el valor de `a` | `:` | `escribe :a` |
| | `cosa` | `escribe cosa "a` |
| | `objeto` | `escribe objeto "a` |
| Enumerar todas las variables definidas. | `listavars` | `listavars` |
| Eliminar la variable `var`. | `borravariable`, `bov` | `borravariable "lado` |

Por compatibilidad con otros intérpretes Logo, se admite `imvars` (imprime todas las variables) con la misma función que `listavars`.

Fíjate en la diferencia:

- Para definir la variable, se antepone `"`
- Para leer la variable, se precede de `:`, la forma más cómoda de las tres posibles: `cosa "a`, `objeto "a` y `:a` son notaciones equivalentes.

Aunque lo detallaremos más adelante, debemos comentar que xLogo trata de distinta forma los números, las palabras y las frases. Para distinguir cuándo una variable almacena un tipo distinto, debemos usar un *vocabulario* específico:

**Número:** Para guardar en la variable `lado` el valor `100`:

```
haz "lado 100
```

**Palabra:** Para guardar en la variable `animal` la palabra `GATO`:

```
haz "animal "GATO
```

**Frase:** Para guardar en la variable `descripcion` la frase `El gato es gris`:

```
haz "descripcion [El gato es gris]
```

En xLogo (y en otros lenguajes de programación) se utiliza el término *Lista* para referirse a aquellas variables que constan de varios elementos, por ejemplo:

```
haz "primitiva [ 5 9 23 26 45 48 ]
```

que contiene una posible combinación del sorteo de la *Lotería primitiva* NO es una frase, ya que no consta de palabras. Es una **Lista**.

Una lista puede constar de varias *sublistas*, por ejemplo:

```
haz "primitiva [ [5 9 23 26 45 48] [5 8 18 26 40 46] [20 24 28 31 36 39] ]
```

consta de tres sublistas, y se pueden combinar variables de cualquier tipo:

```
haz "listado [ [[Pepe Perez] 15 CuartoA] [[Lola Lopez] 16 CuartoB] ]
```

contiene dos sublistas, cada una con una lista (nombre), un número (edad) y una palabra (el grupo de clase)

Si el valor que guarda la variable es un número, puede operarse con ella igual que con un número:

```
haz "lado 100
avanza :lado
```

e incluso pueden usarse para definir otras:

```
haz "alto 100
haz "ancho 2*:alto
repite 2 [ avanza :alto giraderecha 90
    avanza :ancho giraderecha 90 ]
```

que dibuja un rectángulo de base doble que la altura.

**¿Qué otras utilidades le ves al uso de variables?** ¿Cómo las usarías para responder a las preguntas con las que cerrábamos el tema 4? ¿Puedes imaginar algún uso de las listas?

## 6.2. Procedimientos con variables

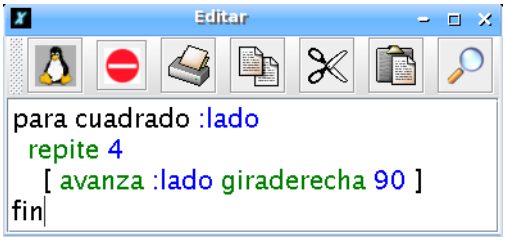
Recuperando nuestro procedimiento `cuadrado`:

```
para cuadrado
  repite 4 [
    avanza 100 giraderecha 90 ]
fin
```

introducir variables es muy simple:

- Indicamos cuál va a ser la variable, de nuevo, con un nombre adecuado: `lado`
- Sustituimos el valor numérico que nos interesa por la variable

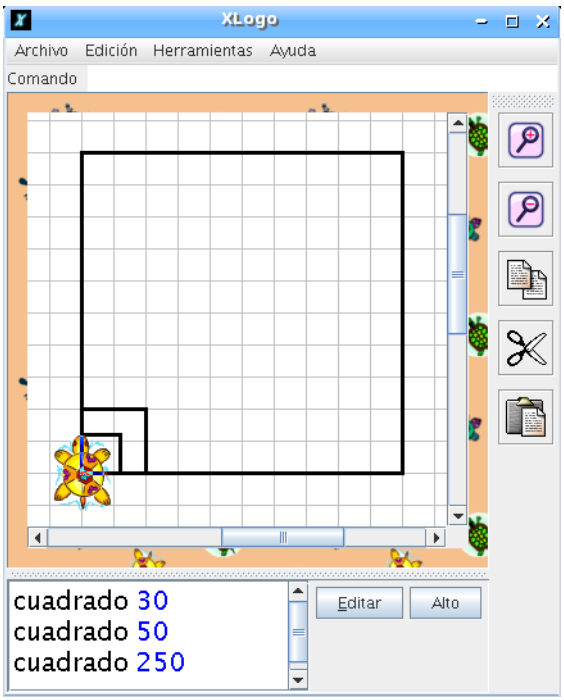
El resultado es:



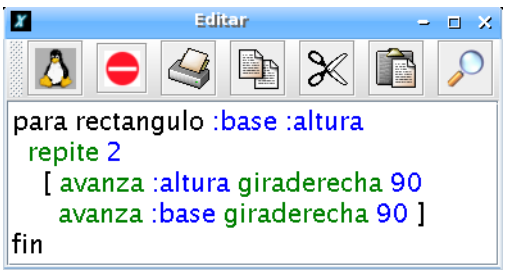

que dibuja, como ya habrás adivinado, un cuadrado. La diferencia está en que ahora el `lado` es desconocido, y debemos indicarle a la tortuga cuánto debe medir:

```
cuadrado 30
cuadrado 50
cuadrado 250
```

dibujarán cuadrados de lados 30, 50 y 250, respectivamente:



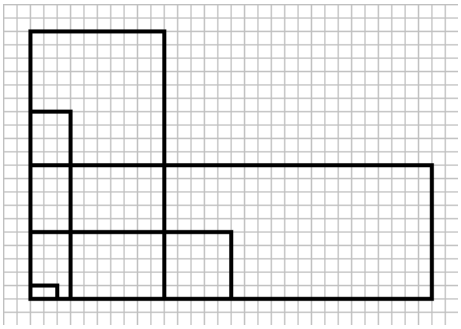

Podemos prever varios *argumentos*:



donde vemos que depende de dos variables. Por ejemplo, `rectangulo 200 100` trazará un rectángulo de altura `200` y anchura `100`.

```
borrapantalla ocultatortuga
rectangulo 200 100 rectangulo 100 300
rectangulo  50 150 rectangulo  10  20
rectangulo 140  30
```

genera:



Ahora bien, si no se proporciona alguno de los argumentos al procedimiento `rectangulo`, el intérprete nos indicará con un mensaje de error que el procedimiento necesita otro argumento:

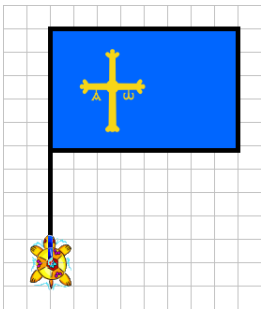
```
No hay suficientes datos para rectangulo
```

## 6.3. Ejercicios

1. Plantea un procedimiento `triangulo` que necesite una variable `lado` y que dibuje un triángulo equilátero cuyo lado sea ese valor

2. Plantea un procedimiento `rueda` que dibuje los 36 radios de longitud `largo` de una rueda

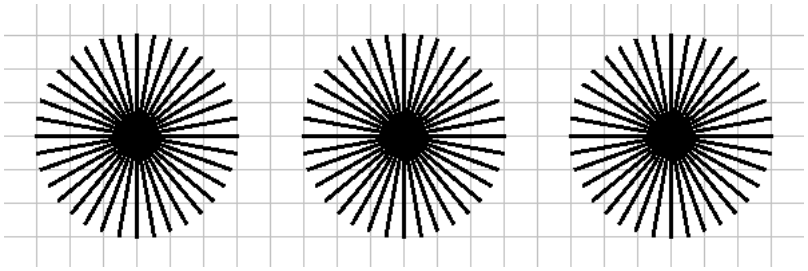
3. Plantea un procedimiento `bandera` que dibuje una bandera consistente en un mástil de longitud `mastil` y cuya tela sea un rectángulo de lados `ancho` y `alto`



4. Plantea un procedimiento `poligono`, que reciba dos entradas: `n` y `largo`, y dibuje un polígono regular de `n` lados de longitud `largo`

   **Pista:** Para hacer divisiones, xLogo utiliza la primitiva `/`.
   Por ejemplo: `escribe 256/5` devuelve `51.2`

5. Plantea un programa que dibuje una fila de `n` ruedas, cada una con 36 radios de longitud `largo`, de modo que la distancia entre los centros de dos ruedas contiguas sea `distancia`



**Prueba con distintos valores de `largo` y `distancia` y observa qué ocurre.** ¿Cuál es la relación entre `distancia` y `largo` cuando se superponen las ruedas? ¿Y cuando están separadas? ¿Qué podríamos hacer para que nunca se superpusieran?

## 6.4. Trazar una forma con distintos tamaños

Vimos como trazar un cuadrado y un rectángulo con dos tamaños distintos. Ahora volvamos al ejemplo de la casa de la página 43 y cómo modificar el código para dibujar la casa sin que importen las dimensiones. Estamos introduciendo, de este modo, el concepto de **proporcionalidad** y **semejanza**.

El objetivo es pasar un argumento al procedimiento `casa` para que, según el parámetro, la casa sea más o menos grande. Es decir:

- `casa 10` dibujará la casa de la sección 5.5.
- `casa 5` dibujará la casa a escala 0,5.
- `casa 20` dibujará una casa con las dimensiones dos veces más grandes

Según el dibujo de la sección 5.5, un cuadrado representa 10 pasos. El procedimiento `cuadrado` era el siguiente:

```
para cuadrado
  repite 4
    [ avanza 150 giraderecha 90 ]
fin
```

que ahora se va a convertir en:

```
para cuadrado :c
  repite 4
    [ avanza :c giraderecha 90 ]
fin
```

Así, cuando se escriba `cuadrado 10`, el cuadrado tendá un lado igual a `15 * 10 = 150`. ¡Las proporciones se mantienen correctamente! De hecho, hay que darse cuenta de que va a ser necesario reescribir todos los procedimientos y cambiar las longitudes de desplazamiento de la siguiente manera.

- `70` se convertirá en `7 * :c`
- `av 45` se convertirá en `av 4.5 * :c`
- etc.

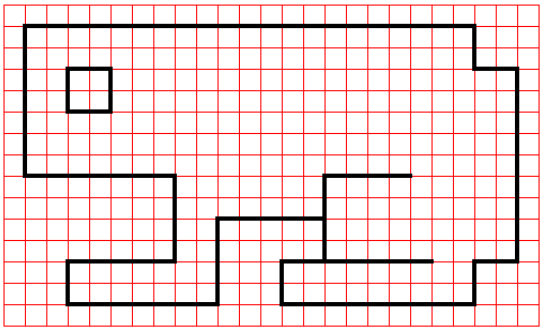
Eso hace que, en realidad, ¡sólamente haya que contar el número de cuadrados para cada longitud! Se obtiene:

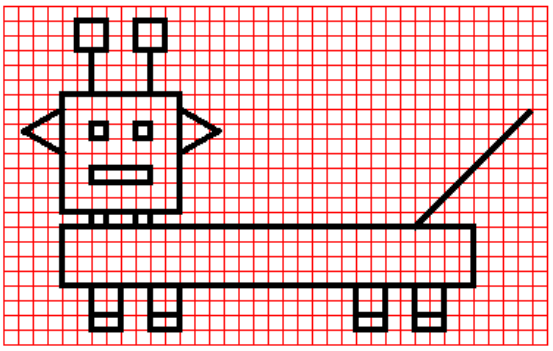
```
para cuadrado :c
  repite 4
    [ avanza 15*:c giraderecha 90 ]
fin

para tri :c
  repite 3
    [ avanza 15*:c giraderecha 120 ]
fin

para puerta :c
  repite 2
    [ avanza 7*:c giraderecha 90
      avanza 5*:c giraderecha 90 ]
fin

para chi :c
  avanza 5.5*:c giraderecha 90
  avanza 2*:c giraderecha 90
  avanza 2*:c
fin

para desp1 :c
  subelapiz
```

```
  giraderecha 90 avanza 5*:c
  giraizquierda 90
  bajalapiz
fin

para desp2 :c
  subelapiz
  giraizquierda 90 avanza 5*:c
  giraderecha 90 avanza 15*:c
  giraderecha 30
  bajalapiz
fin

para desp3 :c
  subelapiz
  giraderecha 60 avanza 2*:c
  giraizquierda 90 avanza 3.5*:c
  bajalapiz
fin

para casa :c
  cuadrado :c desp1 :c puerta :c desp2 :c tri :c desp3 :c chi :c
fin
```

## 6.5. Actividad avanzada

Realiza los siguientes dibujos con dos variables de modo que sea posible obtenerlos a distintos tamaños:





## 6.6. Conceptos acerca de variables

Hay dos tipos de variables:

- **Variables globales**: están siempre accesibles desde cualquier parte del programa.
- **Variables locales**: sólo son accesibles dentro del procedimiento donde fueron definidas.

En esta implementación del lenguaje Logo, las variables locales no son accesibles desde otro sub–procedimiento. Al finalizar el procedimiento, las variables locales son eliminadas.

Las primitivas asociadas son:

| Primitivas | Argumentos | Uso |
|---|---|---|
| `haz` | `palabra, b` | Si la variable local `palabra` existe, se le asigna el valor `b` (de cualquier tipo). Si no, será la variable global `palabra` la asignada con el valor `b`. |
| `local` | `palabra` | Crea una variable llamada `a`. Atención: la variable no es inicializada. Para asignarle un valor, hay que usar `haz`. |
| `hazlocal` | `palabra b` | Crea una nueva variable llamada `palabra` y le asigna el valor `b`. |

Supongamos que en el último ejercicio de la sección anterior quisiéramos controlar la separación entre ruedas para evitar que se superpongan unas con otras. Podríamos hacer que `distancia` fuera siempre algo más del doble que `largo`, para lo que planteamos dos *sub*procedimientos distintos:

```
para ruedas :n :largo
  repite :n
   [ rueda :largo
     separa :largo ]
fin

para rueda :radio
  repite 36
    [ avanza :radio retrocede :radio giraderecha 10 ]
fin

para separa :largo
  hazlocal "distancia 2.5 * :largo
  subelapiz
    giraizquierda 90 avanza :distancia giraderecha 90
  bajalapiz
fin
```

Observa que se usan tres variables relacionadas con la longitud: `largo`, `radio` y `distancia`. Al ejecutar el programa tecleando:

```
borrapantalla ruedas 3 100
```

la tortuga lee `largo`, y le asigna el valor `100`. Sin embargo, `radio` sólo "existe" mientras se está ejecutando el procedimiento `rueda` y "desaparece" al finalizar este. Puedes comprobarlo modificando el procedimiento:

```
para ruedas :n :largo
  repite :n
   [ rueda :largo separa :largo ]
  escribe :largo
  escribe :radio
fin
```

que devolverá `100` (el valor de `largo`) y un mensaje de error:

```
En ruedas, línea 4:
radio no tiene valor.
```

**Estudia qué ocurre con `distancia` en las dos definiciones que hemos hecho de `separa`.** Cambia `escribe :radio` por `escribe :distancia` y observa qué responde xLogo según hayas usado `haz` o `hazlocal`.

```
para ruedas :n :largo
  repite :n
   [ rueda :largo separa :largo ]
  escribe :largo
  escribe :distancia
fin
```

**Las variables locales son muy útiles en programas largos, con varios procedimientos.** Si cada uno usa sus propias variables, no es probable que haya errores debidos a que alguna de ellas sea modificada en el procedimiento equivocado.

## 6.7. Desde la Línea de Comandos

Los procedimientos pueden ser creados y borrados desde la Línea de Comandos. Igualmente, podemos determinar cuáles han sido ya definidos y cuáles no o ejecutar una serie de órdenes sin necesidad de crear un procedimiento asociado.

### 6.7.1. La primitiva `define`

La primitiva `define` crea un nuevo procedimiento sin usar el Editor. Para ello debemos proporcionar el nombre, las variables y las instrucciones a ejecutar:

```
define nombre [variables] [instrucciones]
```

Por ejemplo:

```
define "cuadrado [lado] [repite 4 [ avanza :lado giraderecha 90]]
```

crea el procedimiento `cuadrado` con el que ya hemos trabajado antes.

### 6.7.2. Las primitivas `borra` y `borratodo`

La primitiva `borra` elimina el procedimiento indicado. La sintaxis es:

```
borra nombre
```

Por ejemplo:

```
borra "cuadrado
```

elimina el procedimiento `cuadrado` definido antes.

Por su parte, `borratodo`, sin argumentos, elimina todas las variables y procedimientos actuales.

**Mucho cuidado al usar esta primitiva.** Teniendo en cuenta que es muy improbable que sea escrita por error, NO presenta ninguna ventana de confirmación, y realmente elimina TODO lo que hayas definido hasta ese momento.

### 6.7.3. La primitiva `texto`

Si deseamos conocer la información asocidada a un procedimiento, tecleamos:

```
escribe texto nombre_proc
```

La primitiva `texto` devuelve una lista que contiene toda la información asociada al procedimiento indicado. Concretamente, devuelve una lista que contiene sub–listas:

- La primera lista contiene todas las variables fijas y opcionales del procedimiento.
- Las demás sub-listas son las líneas del procedimiento.

### 6.7.4. La primitiva `listaprocs`

Esta primitiva no necesita argumentos, y enumera todos los procedimientos definidos hasta ese momento en el Histórico de Comandos. Por compatibilidad con otros intérpretes Logo, se admite `imts` (imprime todos) con la misma función.

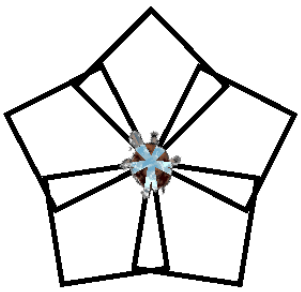
### 6.7.5. La primitiva `ejecuta`

Tecleando `ejecuta [lista]`, las órdenes contenidas en `lista` son ejecutadas consecucutivamente.

Por ejemplo:

```
giraizquierda 27
ejecuta [ repite 5
            [ repite 4
                [ avanza 100 giraizquierda 90 ]
              giraderecha 72 ] ]
```

proporciona:

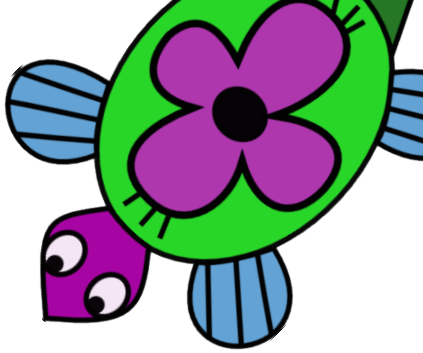


Un ejemplo más curioso de esta primitiva se muestra en la página de nuestro compañero Guy Walker:

`http://www.logoarts.ko.uk`

donde para dibujar un "arco iris" utiliza una lista que contiene las primitivas asociadas a seis colores (Sección 13.2.1) y con un bucle (Capítulo 11) cambia el color del lápiz (Sección 4.4) "ejecutando" su nombre:

```
...
haz "color [ rojo naranja amarillo verde azul violeta ]
repitepara [colores 1 6]
 [ poncolorlapiz ejecuta elemento :colores :color rellena
   subelapiz giraderecha 90 avanza 20 giraizquierda 90 bajalapiz ]
...
```



# Capítulo 7

# Operaciones

¿Qué ocurre si necesitamos realizar operaciones en xLogo? Disponemos de las siguientes primitivas:

## 7.1. Operaciones binarias

Son aquellas que implican a dos elementos.

### 7.1.1. Con números

| Descripción | Primitiva/s | Ejemplo |
|---|---|---|
| Sumar dos números | `suma` ó `+` | `suma 15 20` ó `15 + 20` devuelven `35` |
| Multiplicar dos números | `producto` ó `*` | `producto (-5) 6` ó `(-5) * 6` devuelven `-30` |
| Restar dos números | `diferencia` ó `-` | `diferencia 35 (-10)` ó `35 - (-10)` devuelven `45` |
| Dividir dos números reales | `division` ó `/` | `division 35 3` ó `35 / 3` devuelven `11.666666666666666` |
| Cociente entero | `cociente` | `cociente 35 3` devuelve `11` |
| Resto de una división entera | `resto` | `resto 35 3` devuelve `2` |
| Calcular una potencia | `potencia` | `potencia 3 1.5` devuelve `5.196152422706632` |

Si se trata de sumas o productos de varios números, podemos usar la forma general de `suma` y `producto`:

```
13 + 7 + 2.5                (suma 13 7 2.5)
13 * 7 * 2.5                (producto 13 7 2.5)
```

Fíjate en los paréntesis de la segunda forma; son obligatorios. El resultado puede ser simplemente mostrado por pantalla:

```
escribe 13 + 7 + 2.5          escribe (suma 13 7 2.5)
escribe 13 * 7 * 2.5          escribe (producto 13 7 2.5)
```

o puede ser usado para dibujar:

```
avanza 13 + 7 + 2.5       giraderecha (suma 13 7 2.5)
retrocede 13 * 7 * 2.5    giraizquierda (producto 13 7 2.5)
```

Si, por ejemplo, queremos calcular cuántas formas distintas tenemos de rellenar una quiniela, le pediremos a la *tortuga*:

```
escribe potencia 3 15
```

devuelve

```
1.4348907E7
```

$(3^{15}$ `= 14 348 907)`, el resultado se muestra en notación científica.

**Analiza el siguiente procedimiento.** Recuerda el uso de la barra invertida (sección 1.10.1) para generar espacios y saltos de línea. Te presentamos además la primitiva `tipea`, que escribe en el Histórico de comandos pero, a diferencia de `escribe`, no produce un salto de línea

```
para tabla
  haz "contador1 1
  haz "espacio [ \  ]
  haz "saltolinea [ \n ]
  repite 9
    [ haz "contador2 1
      repite 9
        [ tipea (:contador1 * :contador2)
          tipea :espacio
          haz "contador2 :contador2 + 1 ]
      tipea :saltolinea
      haz "contador1 :contador1 + 1 ]
fin
```

¿Qué crees que hacen las variables `contador1` y `contador2`? ¿Por qué crees que las hemos usado? ¿Se te ocurre una forma mejor de conseguir lo mismo? ¿Cómo mejorarías el aspecto con el que salen los los resultados?

### 7.1.2. Con listas

No sólo pueden efectuarse operaciones con números. Las palabras y las frases (listas de palabras) pueden *concatenarse* (ponerse una a continuación de la otra). Disponemos de las primitivas `frase` y `lista` (haremos un estudio más profundo de las listas en el capítulo 10). Por ejemplo:

```
escribe frase [El gato es ] [gris]
```

muestra en pantalla:

```
El gato es gris
```

pero:

```
escribe lista [El gato es ] [gris]
```

muestra en pantalla:

```
[El gato es] [gris]
```

es decir, crea una lista cuyos elementos son los argumentos, lo que en este caso lleva a obtener una *lista de listas*.

También es posible concatenar listas con números o variables. Por ejemplo, si la variable `area` contiene el valor `250` podemos pedirle a XLOGO:

```
escribe frase [La superficie es ] :area
```

que proporciona:

```
La superficie es 250
```

ya que `frase` ha concatenado la *lista* `La superficie es` y el valor de `:area`

**Concatenar listas con variables es una forma de que los mensajes de XLOGO a la hora de presentar los resultados se entiendan mejor.**

**Ejemplo**: Vamos a crear un procedimiento que calcule el área de un triángulo dándole la base y la altura

```
Recuerda que el área de un triángulo es: A = b * h / 2
```

```
para area_triangulo :base :altura
  haz "area (:base * :altura) / 2
  escribe :area
fin
```

o bien:

```
para area_triangulo :base :altura
  haz "area (division (producto :base :altura) 2)
  escribe :area
fin
```

Para ejecutarlo, escribimos:

```
area_triangulo 3 5
```

Es posible ser un poco más elegante. ¿Tú sabrías lo que hace este programa sólo viendo los resultados o, incluso, leyéndolo? Cambiemos la penúltima línea:

```
para area_triangulo :base :altura
  haz "area (division (producto :base :altura) 2)
  escribe frase [El área del triángulo es ] :area
fin
```

o bien:

```
para area_triangulo :base :altura
# Este procedimiento calcula el área de un triángulo
# pidiendo su base y altura
  haz "area (division (producto :base :altura) 2)
  haz "texto frase [El área del triángulo de base] :base
  haz "texto frase :texto [y altura]
  haz "texto frase :texto :altura
  haz "texto frase :texto [es]
  haz "texto frase :texto :area
  escribe :texto
fin
```

que al ejecutarlo:

```
area_triangulo 3 5
```

proporciona:

```
El area del triangulo de base 3 y altura 5 es 7.5
```

¿No se entiende mejor?

Observa otra capacidad del lenguaje xLogo. Hemos *reutilizado* la variable `texto` varias veces, incluyendo en su definición **a ella misma**. Esto es útil cuando no quieres definir varias variables para un proceso que se refiere a una misma cosa (en este caso ir aumentando palabra a palabra el texto a mostrar en pantalla).

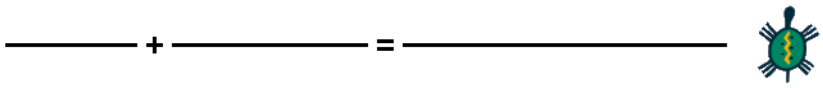
Podríamos haber aprovechado la *forma general* de la primitiva `frase`:

```
para area_triangulo :base :altura
# Este procedimiento calcula el área de un triángulo
# pidiendo su base y altura
  haz "area (division (producto :base :altura) 2)
  escribe (frase [El área del triángulo de base ] :base [y altura ] :altura [es ] :area)
fin
```

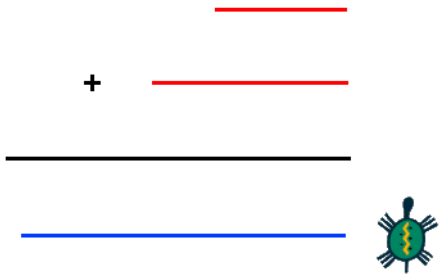
## 7.2. Ejercicios

1. Modifica el procedimiento `tabla` presentado como ejemplo para que muestre la *tabla de restar* y la *tabla de multiplicar*. ¿Qué observas?

2. Crea un procedimiento `acumula` que reciba dos números (`n1` y `n2`) como argumentos, dibuje dos segmentos cuyas longitudes sean precisamente `n1` y `n2` separados por un signo "más" y a continuación dibuje el segmento cuya longitud sea la suma de `n1` y `n2` precedido del signo "igual".



3. Modifica el procedimiento anterior para que los segmentos aparezcan colocados como en las sumas tradicionales: los sumandos unos encima de otros, el signo "más" a la izquierda, una línea de "operación" y el resultado en la parte inferior. Usa colores para diferenciar los sumandos del signo y la línea y éstos del resultado



(Nota que en este caso los segmentos empiezan a dibujarse desde la derecha)

4. A partir de los procedimientos anteriores, construye los procedimientos `sustrae` que en lugar de sumas hagan "restas de segmentos".

Prueba distintas posibilidades, observando qué ocurre cuando `n1` es mayor que `n2` y viceversa.

5. Crea un procedimiento `superficie` con dos argumentos, `n1` y `n2`, dibuje un rectángulo cuyos lados midan `n1` y `n2` y calcule su área.

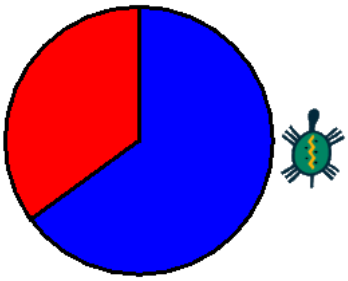
6. Echa un vistazo a tu clase, y cuenta el número de chicos y de chicas que hay. Con esa información, crea el procedimiento `quesitos` que:

   a) Calcule cuántos/as estudiantes hay en tu clase

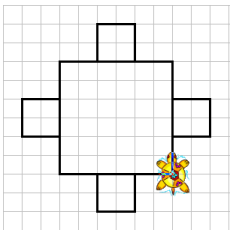
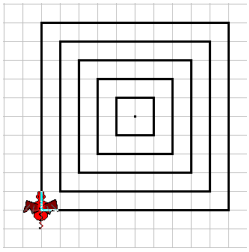
   b) Dibuje una circunferencia. Para ello usa la primitiva `circulo`, que necesita un argumento (`número`), y dibuja una circunferencia de radio `número` centrada en la posición actual de la tortuga

c) Divida el círculo en dos partes proporcionales al número de chicas y chicos y los coloree con distintos colores

Por ejemplo, si en una clase hay un 40 % de chicos y un 60 % de chicas, el dibujo debería ser algo así:

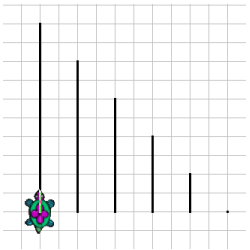
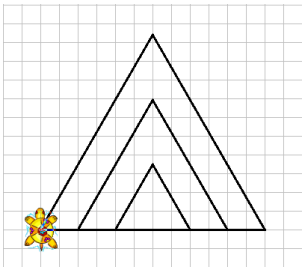


7. Con el procedimiento `cuadrado` definido en el capítulo anterior, dibuja:

*a*)  *b*) 

8. Programa un procedimiento que dibuje una fila horizontal de `n` baldosas cuadradas cuyo lado sea la variable `lado` y que aparezcan centradas en pantalla

9. Dibuja:

*a*)  *b*) 

10. Escribe un procedimiento `rayos`, que dibuje los `n` radios de longitud `largo` de una rueda (sólo los radios)

## 7.3. Operaciones unitarias

Son aquellas que sólo necesitan un elemento:

| Descripción | Primitiva/s | Ejemplo |
|---|---|---|
| Hallar la raíz cuadrada de un número | `raizcuadrada` ó `rc` | `raizcuadrada 64` ó `rc 64` devuelven `8` |
| Cambiar el signo | `cambiasigno` ó `-` | `cs 5` devuelve `-5` |
| Valor absoluto de un número | `absoluto` ó `abs` | `abs (-5)` devuelve `5` |
| Truncamiento de un número decimal | `trunca` | `trunca 3.565` devuelve `3` |
| Redondear un número al entero más cercano | `redondea` | `redondea 3.565` devuelve `4` |
| Generar un número aleatorio comprendido entre `0` y `n-1` | `azar n` | `azar 6` devuelve un entero entre `0` y `5` |
| Generar un número aleatorio comprendido entre `0` y `1` | `aleatorio` | `aleatorio` devuelve un número entre `0` y `1` |

La primitiva `truncar` puede usarse para conseguir un número entero cuando nos aparece en notación científica. En el apartado 7.1.1 calculábamos cuántas formas distintas tenemos de rellenar una quiniela, y escribimos:

```
escribe potencia 3 15
```

Si usamos

```
escribe truncar potencia 3 15
```

nos devuelve:

```
14348907
```

**Ejemplo**: Queremos un procedimiento que calcule el cociente y el resto de la división entera entre `A` y `B`, es decir, conseguir lo que proporcionan las primitivas `cociente` y `resto`, pero sin usarlas.

```
para division_entera :A :B
  haz "C truncar (:A / :B)
  escribe :C
  escribe :A - :B * :C
fin
```

**La primitiva `azar` devuelve números aleatorios comprendidos entre `0` y `n - 1`.** ¿Cómo harías para simular el lanzamiento de un dado? Lee con cuidado la sección 7.8 para evitar conclusiones erróneas.

## 7.4. Ejercicios

1. Modifica el procedimiento `division_entera` de modo que devuelva los resultados indicando con mensajes qué es cada número.
2. Plantea un procedimiento `Pitagoras` que refiriéndose a un triángulo rectángulo, acepte los valores de los catetos y calcule el valor de la hipotenusa.
3. Intenta escribir un procedimiento `suerte`, que genere un número al azar del conjunto `{20 , 25 , 30 , 35 , 40 , 45 , 50}`.[1]
4. Plantea un procedimento `dados`, que simule el lanzamiento de **dos** dados y cuya salida sea la suma de ambos
5. El procedimiento `juego` con el que presentamos a la tortuga en el capítulo 3 ubica las "piedras" de forma aleatoria. Piensa cómo harías para ubicar `n` "piedras" circulares y coloreadas en posiciones aleatorias
6. Plantea un procedimiento que simule un sorteo de Lotería Nacional, esto es, generar un número de 5 cifras por extracción consecutiva de 5 "bolas" con valores comprendidos entre 0 y 9 y colocarlas una a continuación de otras (recuerda la prmitiva `tipea`)

En un sorteo de la "Lotería Primitiva" hay 49 bolas numeradas en un único bombo, y de él se extraen consecutivamente 6 bolas. ¿Podríamos usar la primitiva `azar` para simular un sorteo de la Primitiva?

## 7.5. Cálculo superior

xLogo puede evaluar funciones trigonométricas y logarítmicas.

| Descripción | Primitiva/s | Ejemplo |
|---|---|---|
| Calcular el seno de un ángulo | `seno` o `sen` | `seno 45` |
| Calcular el coseno de un ángulo | `coseno` o `cos` | `coseno 60` |
| Calcular la tangente de un ángulo | `tangente`, `tg` o `tan` | `tangente 135` |
| Obtener el ángulo cuyo seno es `n` | `arcoseno`, `arcsen` o `asen` | `arcsen 0.5` |
| Obtener el ángulo cuyo coseno es `n` | `arcocoseno`, `arccos` o `acos` | `arccos 1` |
| Obtener el ángulo cuya tangente es `n` | `arcotangente`, `arctg` o `atan` | `arctg 1` |
| Evaluar el logaritmo decimal de un número | `log` ó `log10` | `log 100` |
| Evaluar el logaritmo neperiano de un número | `logneperiano` ó `ln` | `ln 100` |
| Exponencial de un número ($e^x$) | `exp` | `exp 2` |

[1] Recuerda que `azar 5 + 10` es lo mismo que `azar (5 + 10)`, es decir, `azar 15`. Igualmente `10 + azar 5` es equivalente a `(azar 5) + 10`

## 7.6. Precisión en los cálculos

En ocasiones, necesitaremos trabajar con más precisión de la que el programa permite en sus ajustes iniciales. Si queremos que xLogo realice los cálculos utilizando un mayor número de dígitos, debemos utilizar la primitiva `pondigitos` o `pondecimales`.

Las dos formas de esta primitiva se deben a las diferencias de nivel académico a las que podemos enfrentarnos, siendo más fácil entender para un niño pequeño que la precisión aumenta con *los decimales*. Por lo demás, su sintaxis es muy simple:
`pondecimales n` o `pondigitos n`
fija la precisión del cálculo:

1. Por defecto, xLogo usa 16 dígitos.
2. Si `n` es negativa, se vuelve al valor inicial (el valor inicial es $-1$).
3. Si `n` es cero, todos los números se redondean a la unidad.

De modo equivalente, las primitivas `decimales` y `digitos` devuelven el número de dígitos utilizados en el cálculo.

En el apartado 7.1 calculábamos el número de combinaciones distintas en una quiniela ($3^{15}$) usando la primitiva `potencia`: `escribe potencia 3 15` –>`1.4348907E7`.
Si tecleamos antes: `pondecimales 0`, la misma orden nos devuelve ahora `14348907`
Un ejemplo más avanzado se encuentra en la página 102.

## 7.7. Ejercicios

**Pista para los tres problemas siguientes**: Puedes usar las primitivas `repite` y `contador`[2] y una variable en la que ir guardando los resultados parciales.

1. ¿Sabrías crear un procedimiento `poten :x :n` que calcule $x^n$ (supuesto que `n` es un número natural) sin usar la primitiva `potencia`?
2. Crea un procedimiento `factorial :n`, que calcule el factorial del número `n`.

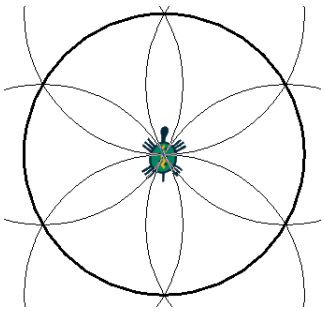
   Recuerda: `n! = n * (n - 1) * ...* 2 * 1`
3. Intenta ahora conseguir un procedimiento `suma_potencias :n` que calcule la suma `2 + 4 + 8 + ...+` $2^n$

   **Para los problemas siguientes necesitarás hacer algunos cálculos trigonométricos antes de ponerte a programar.**

[2]Échale un vistazo a la sección 11.1.1 para entenderlas bien

4. Plantea un procedimiento `poligono_regular`, que dibuje un polígono regular de `n` lados de longitud `lado` inscrito en una circunferencia

5. La flor *hexapétala* (conocida como flor de agua o flor galana en Asturias) es muy sencilla de dibujar únicamente con un compás:



   Plantea un procedimiento `flor` que haga uso de la primitiva `arco` (`arco` necesita tres argumentos: el radio, el ángulo donde empieza y el ángulo donde termina, medido siempre en forma absoluta, es decir, 0° es **siempre** hacia arriba, independientemente de hacia dónde "mire" la tortuga) y que reciba un argumento `radio` para dibujar la flor galana.

6. Plantea un procedimiento con dos argumentos: `radio` y `n`, que dibuje un círculo con ese `radio`, y que inscriba en él un polígono de `n` lados.

7. (**Este problema requiere importantes conocimientos de geometría y/o dibujo técnico**) Plantea un procedimiento que dibuje una flor, pidiendo el `radio` y el número `n` de pétalos que tiene.

## 7.8. Prioridad de las operaciones

Utilizando la forma corta, `+`, `-`, `*` y `/`, xLogo realiza las operaciones (como no podía ser de otra manera) obedeciendo a la prioridad de las mismas. Así si escribimos:

```
escribe 3 + 2 * 4
```

xLogo efectúa primero el producto y luego la suma, siendo el resultado `11`.

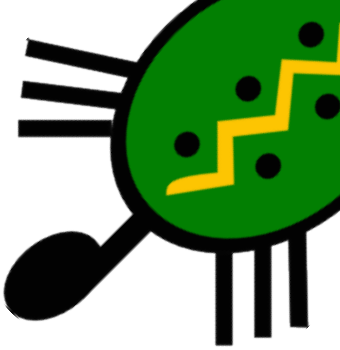
Como sabemos, la presencia de paréntesis modifica el orden en que se deben realizar las operaciones. Por ello, si escribimos:

```
escribe (3 + 2) * 4
```

xLogo efectuará la suma antes que el producto, y el resultado será `20`.

Hay que tener cuidado, y esto es **muy importante**, si se usan las primitivas `suma`, `diferencia`, `producto`, `division`, `potencia`, ..., ya que **internamente** tienen una prioridad inferior a las anteriores. Este es un comportamiento que estamos intentando solucionar, ya que incumple las reglas por todos conocidas. Hasta entonces, aconsejamos utilizar los paréntesis para evitar sorpresas desagradables:

```
escribe (seno 20) + 10          --> 10.3420201433256687
escribe (raizcuadrada 3) + 1    --> 2.7320508075688772
escribe (azar 6) + 1            --> 1, 2, 3, 4, 5 o 6
```



# Capítulo 8

# Coordenadas y Rumbo

¿Cómo podemos saber dónde se encuentra la tortuga? ¿Puedo enviarla a un punto concreto de la pantalla con una sola orden? ¿Hacia dónde está *mirando*? Estas preguntas se responden en este capítulo.

## 8.1. Cuadrícula y ejes

Comencemos por las primitivas que pueden ayudarnos enormemente en nuestro trabajo con las coordenadas:

| Primitivas | Argumentos | Uso |
|---|---|---|
| `cuadricula` | `a b` (números) | Dibuja una cuadrícula en el **Área de dibujo** de dimensiones `a` x `b` y borra la pantalla |
| `detienecuadricula` | `no` | Quita la cuadrícula del **Área de dibujo** y borra la pantalla |
| `poncolorcuadricula`, `pcc` | `primitiva`, `lista` o `numero` | Establece el color de la cuadrícula del **Área de dibujo** |
| `ejes` | `número` | Dibuja los ejes cartesianos (X e Y) de escala (separación entre marcas) `n`, con las etiquetas correspondientes. |
| `ejex` | `número` | Dibuja el eje de abscisas (eje X) de escala (separación entre marcas) `n`, con las etiquetas correspondientes. |
| `ejey` | `número` | Dibuja el eje de ordenadas (eje Y) de escala (separación entre marcas) `n`, con las etiquetas correspondientes. |
| `detieneejes` | `ninguno` | Quita los ejes del **Área de dibujo** y borra la pantalla |
| `poncolorejes`, `pce` | `primitiva`, `lista` o `numero` | Establece el color de los ejes en el **Área de dibujo** |

**El sistema cartesiano con coordenadas ortogonales no es el único que existe.** Investiga otros sistemas de representación e intenta crear procedimientos que los muestren en pantalla.

**Nota:** `detienecuadricula` y `detieneejes` retiran la cuadrícula y los ejes de pantalla, y también la borra del mismo modo que `borrapantalla`

## 8.2. Coordenadas

El **Área de Dibujo** es un sistema cartesiano cuyo origen está situado en el centro de la pantalla. De este modo, podemos alcanzar cualquiera de los puntos de dicho área ayudados por sus coordenadas. Las primitivas asociadas son:

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| Mostrar la posición (devuelve una lista) | `posicion` | `pos` |
| Mostrar sólo la abscisa (coordenada `X`) | `primero posicion`<br>`coordenadax` | `pr pos`<br>`coordx` |
| Mostrar sólo la ordenada (coordenada `Y`) | `ultimo posicion`<br>`coordenaday` | `ultimo pos`<br>`coordy` |
| Mover al punto `[X,Y]` (`X,Y` números) | `ponposicion [X Y]` | `ponpos [X Y]` |
| Pintar el `punto` de coordenadas `[X,Y]` | `punto [X Y]` | |
| Mover hacia `[X,Y]` (`X,Y` números o variables) | | `ponxy :X :Y` |
| Llevar hacia el punto de abscisa `X` (número o variable) | | `ponx :X` |
| Llevar hacia el punto de ordenada `Y` (número o variable) | | `pony :Y` |

De nuevo debemos tener cuidado con la prioridad de las primitivas. Si alguna coordenada es negativa, debemos usar paréntesis:

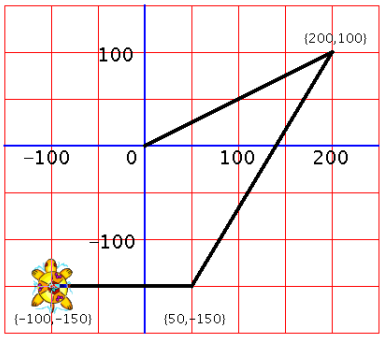
```
ponxy 100 (-60)
```

Para usar `ponposicion` con variables, debemos usar la primitiva `frase`:

```
ponposicion frase :abscisa :ordenada
```

Un ejemplo sencillo de su utilización:

```
borrapantalla
ponpos [200 100]
ponpos [50 -150]
ponpos [-100 -150]
```



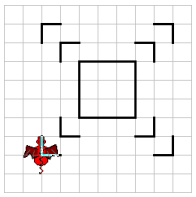
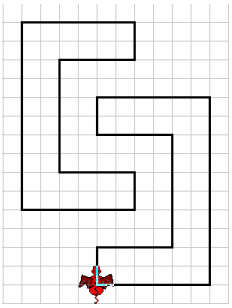
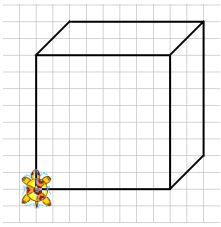

## 8.3. Ejercicios

1. Dibuja un rectángulo usando solamente las primitivas `ponx` y `pony`
2. Dibuja un triángulo rectángulo isósceles usando únicamente `ponposicion`
3. Construye el procedimiento `segmento`, cuyas 4 entradas sean las coordenadas de dos puntos y que dibuje el segmento cuyos extremos son esos dos puntos.
4. Define el procedimiento `cuadrilatero`, cuyas OCHO entradas sean las coordenadas de cuatro puntos y que dibuje el cuadrilátero cuyos vértices son esos ocho puntos.
5. Plantea un procedimiento `dist_ptos`, que a partir del procedimiento `segmento` calcule la distancia entre los dos puntos (o lo que es lo mismo, la longitud del segmento)

   **Dato**: La distancia entre dos puntos `(x0,y0)` y `(x1,y1)` se calcula mediante la fórmula:

   $$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

6. Define el procedimiento `triangulo`, cuyas SEIS entradas sean los vértices de un triángulo, lo dibuje y lo rellene.
7. Dibuja:

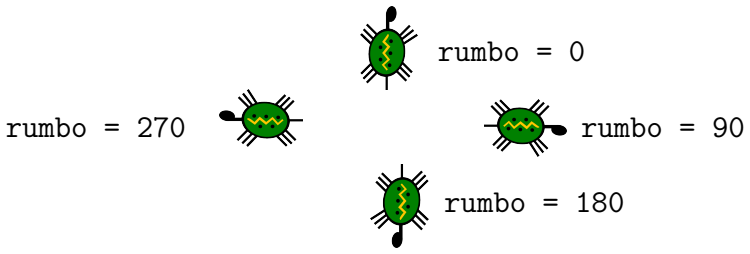
*a*)  *b*)  *c*) 

## 8.4. Rumbo

Las primitivas asociadas al rumbo son:

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| Orientar la tortuga hacia un rumbo `n` | `ponrumbo :n` | `ponr :n` |
| Pedir el `rumbo` (en grados respecto de la vertical y sentido horario) | | `rumbo` |
| Para volver al origen con rumbo 0 (mirando hacia arriba) | | `centro` |
| Pedir el rumbo que la tortuga debe seguir `hacia` el punto `[X Y]` | | `hacia [X Y]` |
| Pedir la `distancia` (en pasos) hasta el punto `[X Y]` | | `distancia [X Y]` |

Una imagen vale más que mil palabras:

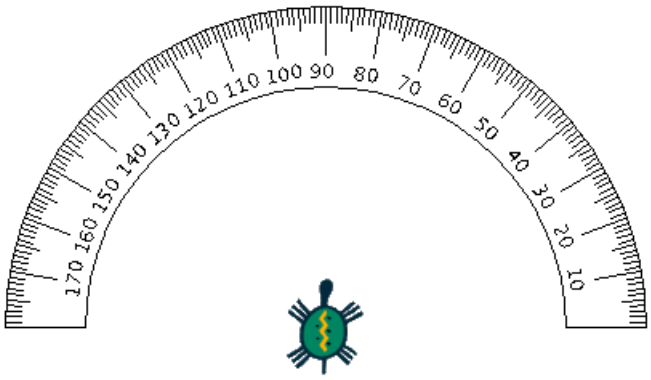



## 8.5. Ejercicios

1. Define procedimientos que pongan a la tortuga rumbo a los puntos cardinales: `norte`, `sur`, `este`, `oeste`, `noroeste`, `nordeste`, `sudeste` y `suroeste`.

2. En la sección anterior te pedíamos que buscaras otros sistemas de coordenadas. Te pedimos aquí que crees un procedimiento que represente en pantalla un sistema de coordenadas `polares`

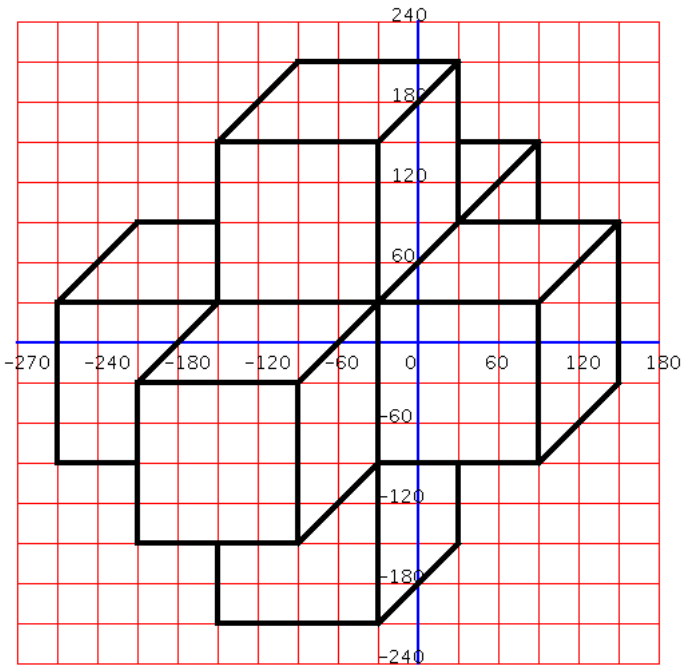
3. Para manejar bien los rumbos es conveniente tener a la vista un transportador. Crea un procedimiento `transportador` que dibuje uno en pantalla:




Para indicar la escala, puedes usar la primitiva `rotula` (sección 12.1)

## 8.6. Actividad avanzada

En esta actividad debes realizar la siguiente figura.




Sólo puedes utilizar las primitivas

- `ponposicion` (`ponpos`)
- `borrapantalla` (`bp`)
- `subelapiz` (`sl`)
- `bajalapiz` (`bl`)

# Capítulo 9

# Condicionales y Operaciones lógicas

En ocasiones será necesario decidir qué acción realizar en función de una determinada condición, lo que en programación recibe el nombre de *flujo de control de un algoritmo o programa*.

Por ejemplo, si quiero calcular una raíz cuadrada, antes debo mirar si el número es positivo o no. Si es positivo, no tendré ningún problema, pero si es negativo xLogo dará un mensaje de error, ya que la raíz cuadrada de un negativo no es un número real.

Las estructuras condicionales son uno de los pilares de la **Programación Estructurada**, es decir, una de las piezas clave en el desarrollo de programas complejos pero fáciles de leer.

## 9.1. Condicional

xLogo define para ello la primitiva `si`, cuya sintaxis es:

```
si condicion
  [ Acciones a realizar si la condicion es cierta ]
```

Por ejemplo, el siguiente programa dice si un número es negativo:

```
para signo :numero
  si :numero < 0
    [ escribe [El  numero es negativo] ]
fin
```

Ahora bien, no dice nada si el número es positivo o nulo. Disponemos de otra opción:

```
si condicion
  [ Acciones a realizar si la condicion es cierta ]
  [ Acciones a realizar si la condicion es falsa ]
```

de modo que podemos mejorar nuestro programa `signo`:

```
para signo :numero
  si :numero < 0
    [ escribe [El numero es negativo] ]
    [ escribe [El numero es positivo] ]
fin
```

Si queremos que los argumentos para `cierto` y `falso` estén guardados en sendas variables, no podemos usar `si`. En este caso, la primitiva correcta es:

`sisino`

En este ejemplo, xLogo mostrará un error:

```
haz "Opcion_1 [escribe "cierto]
haz "Opcion_2 [escribe "falso]
si 1 = 2 :Opcion_1 :Opcion_2
```

ya que la segunda lista nunca será evaluada:
`¿Qué hacer con [escribe "falso]?`

La sintaxis correcta es:

```
haz "Opcion_1 [escribe "cierto]
haz "Opcion_2 [escribe "falso]
sisino 1 = 2 :Opcion_1 :Opcion_2
```

que devolverá:

```
"falso
```

## 9.2. Operaciones Lógicas

Con los condicionales es muy interesante conocer las *operaciones lógicas*:

| **Primitiva** y **Argumentos** | **Uso** |
|---|---|
| `y condicion_1 condicion_2` ó `condicion_1 & condicion_2` | Devuelve `cierto` si ambas condiciones son ciertas. Si una (o las dos) son falsas, devuelve `falso` |
| `o condicion_1 condicion_2` ó `condicion_1 \| condicion_2` | Devuelve `cierto` si al menos una de las condiciones es cierta. Si las dos son falsas, devuelve `falso` |
| `no condicion` | Devuelve la negación de `condicion`, es decir, `cierto` si `condicion` es falsa y `falso` si `condicion` es cierta. |

Para comparaciones numéricas, disponemos de cuatro operadores que pueden expresarse de dos formas:

| Operador "menor" | Operador "mayor" | Operador "menor o igual" | Operador "mayor o igual" |
|---|---|---|---|
| `<` | `>` | `<=` | `>=` |
| `menor?` | `mayor?` | `menoroigual?` | `mayoroigual?` |

si bien es evidente que no serían estrictamente necesarios:

- `a <= b` es equivalente a `no (a > b)`
- `a >= b` puede sustituirse por `no (a < b)`

Por ejemplo (los paréntesis están para entender mejor el ejemplo):

```
para raiz_con_prueba :numero
  si o (:numero > 0) (:numero = 0)
    [ escribe raizcuadrada :numero ]
fin
```

```
para raiz_con_prueba :numero
  si (:numero > 0) | (:numero = 0)
    [ escribe raizcuadrada :numero ]
fin
```

que comprueba si el número es positivo **ó** cero antes de intentar calcular la raíz. Este procedimiento podría hacerse con `no`:

```
para raiz_con_prueba :numero
  si no (:numero < 0)
    [ escribe raizcuadrada :numero ]
fin
```

y ahora comprueba que **no** sea negativo.

Imaginemos ahora un procedimiento que diga si la temperatura exterior es agradable o no:

```
para agradable :temperatura
  si y (:temperatura < 25) (:temperatura > 15)
    [ escribe [La temperatura es agradable] ]
    [ si no (:temperatura > 15)
      [ escribe [Hace frío] ]
      [ escribe [Hace demasiado calor] ] ]
fin
```

```
para agradable :temperatura
  si (:temperatura < 25) & (:temperatura > 15)
    [ escribe [La temperatura es agradable] ]
    [ si no (:temperatura > 15)
      [ escribe [Hace frio] ]
      [ escribe [Hace demasiado calor] ] ]
fin
```

que estudia primero si la temperatura está en el intervalo `(15 , 25)` – o sea `15 < T < 25` – y si lo está dice `La temperatura es agradable`. Si no pertenece a ese intervalo, analiza si está por debajo de él (y dice `Hace frio`) o no (y devuelve `Hace demasiado calor`). Hemos *encadenado* condicionales.

Recuerda que tanto `y` como `o` admiten una forma general y, por tanto, pueden efectar más de una comparación:

```
si (y condicion1 condicion2 condicion3 ...) [ ... ]
si (o condicion1 condicion2 condicion3 ...) [ ... ]
```

son equivalentes a:

```
si (condicion1 & condicion2 & condicion3 & ...) [ ... ]
si (condicion1 | condicion2 | condicion3 | ...) [ ... ]
```

## 9.3. Ejercicios

1. Modifica el procedimiento `raiz_con_prueba` para **no** usar ninguna operación lógica

2. Plantea un procedimiento `no_menor` que decida si, dados dos números, el primero es mayor o igual que el segundo y responda `sí` en caso afirmativo
   `no_menor 8 5` ⟶ `sí`
   `no_menor 3 5` ⟶ (nada)

3. Plantea el procedimiento `edad_laboral`, que compruebe si la `edad` de una persona verifica la condición `17 < edad < 65`, respondiendo `Está en edad laboral` en caso afirmativo

4. Escribe el procedimiento `múltiplo?`, que verifique si un número `dato` es múltiplo de otro `divisor`, respondiendo `Es múltiplo` o `No es múltiplo`, en cada caso.

   **Pista**: puedes usar `resto`, `cociente` y/o `division` (`/`)
   En caso de que sea múltiplo, la tortuga debe dibujar un rectángulo cuya base sea el divisor y su área el múltiplo
   `múltiplo? 18 5` ⟶ `No es múltiplo`
   `múltiplo? 320 40` ⟶ `Es múltiplo`

5. Diseña un procedimiento que diga si un año es bisiesto no. Recuerda que un año es bisiesto si es múltiplo de 4, pero no es múltiplo de 100 aunque sí de 400.
   `bisiesto? 1941` ⟶ `1941 no es bisiesto`
   `bisiesto? 1900` ⟶ `1900 no es bisiesto`
   `bisiesto? 2000` ⟶ `2000 sí es bisiesto`

6. Diseña el procedimiento `calificaciones` que, dada una `nota` la califique de acuerdo con el baremo usual:

| Nota | $n < 5$ | $5 \leq n < 6$ | $6 \leq n < 7$ | $7 \leq n < 9$ | $9 \leq n < 10$ |
|---|---|---|---|---|---|
| Calificación | Suspenso | Aprobado | Bien | Notable | Sobresaliente |

7. Diseña un programa que calcule la hipotenusa de un triángulo rectángulo dados sus catetos, pero que llame a un subprocedimiento que devuelva el cuadrado de un número dado, y que además dibuje el triángulo en pantalla

   **Pista**: Coloca los catetos paralelos a los ejes cartesianos, y y utiliza las primitivas asociadas a las coordenadas

8. Diseña un programa que dibuje un triángulo dados sus tres lados.

**Cuidado:** Dados tres lados, no siempre es posible construir un triángulo. Piensa qué condición debe cumplirse para que sea posible dibujarlo y después realiza los cálculos trigonométricos necesarios.

9. Plantea el procedimiento `mismo_signo`, que decida si dos números no nulos tienen el mismo signo. **Pista**: Comprueba el signo de su producto

## 9.4. Booleanos

Una variable o primitiva es *booleana* si sus únicos valores posibles son `cierto` o `falso`. En xLogo un booleano es la respuesta a las primitivas terminadas con `?`

| Primitivas | Argumentos | Uso |
|---|---|---|
| `cierto` | `ninguno` | Devuelve `"cierto` |
| `falso` | `ninguno` | Devuelve `"falso` |
| `palabra?` | `variable` | Devuelve `cierto` si `a` es una palabra, `falso` si no. |
| `numero?` | `variable` | Devuelve `cierto` si `a` es un número, `falso` si no. |
| `entero?` | `un número` | Devuelve `cierto` si `a` es un número entero, `falso` si no. |
| `lista?` | `variable` | Devuelve `cierto` si `a` es una lista, `falso` si no. |
| `vacio?` | `variable` | Devuelve `cierto` si `a` es una lista vacía o una palabra vacía, `falso` si no. |
| `iguales?`, `=` | `a b` | Devuelve `cierto` si `a` y `b` son iguales, `falso` si no. |
| `antes?`, `anterior?` | `a b` (dos palabras) | Devuelve `cierto` si `a` está antes que `b` siguiendo el orden alfabético, `falso` si no. |
| `miembro?` | `a b` | Si `b` es una lista, determina si `a` es un elemento de `b`. Si `b` es una palabra, determina si `a` es un caracter de `b`. |
| `cuadricula?` | `no` | Devuelve `cierto` si la cuadrícula está activa, `falso` si no. |
| `ejex?`, | `no` | Devuelve `cierto` si está activo el eje de abscisas (eje X), `falso` si no. |
| `ejey?`, | `no` | Devuelve `cierto` si está activo el eje de ordenadas (eje Y), `falso` si no. |
| `bajalapiz?`, `bl?` | `ninguno` | Devuelve la palabra `cierto` si el lápiz está abajo, `falso` si no. |
| `visible?` | `ninguno` | Devuelve la palabra `cierto` si la tortuga está visible, `falso` si no. |
| `primitiva?`, `prim?` | `una palabra` | Devuelve `cierto` si la palabra es una primitiva de xLogo, `falso` si no. |

| Primitivas | Argumentos | Uso |
|---|---|---|
| `procedimiento?`, `proc?` | `una palabra` | Devuelve `cierto` si la palabra es un procedimiento definido por el usuario, `falso` si no. |
| `variable?`, `var?` | `una palabra` | Devuelve `cierto` si la palabra es una variable definida por el usuario, `falso` si no. |

En la sección 5.1 te pedimos que hicieras pruebas con determinados nombres de procedimientos. Si queremos estar seguros de que un `nombre` está "libre" antes de usarlo, podemos preguntar:

```
variable? "nombre
```

y en caso que que la variable `nombre` no haya sido definida, asignarle un valor:

```
si no (variable? "nombre) & no (procedimiento? "nombre)
  [ haz "nombre pi ]
```

Las primitivas `cuadricula?`, `ejes?`, ... permiten controlar aspectos que serán importantes, por ejemplo, al colorear regiones (sección 13.2.3).

Analiza ahora el siguiente procedimiento:

```
para cuadrado_cortado
  haz "paso cambiasigno 50
  repite 4
    [ repite 3
        [ si bajalapiz? [subelapiz] [bajalapiz]
          avanza 75 ]
      giraderecha 90 ]
fin
```

¿Eres capaz de reproducir su resultado?

## 9.5. Ejercicios

1. Utilizando las primitivas `listavars` y `listaprocs`, intenta crear un procedimiento que investigue si una determinada palabra es una variable o un procedimiento sin usar las primitivas `variable?` o `procedimiento?` (ni sus formas cortas)

2. Diseña un procedimiento que compruebe si la cuadrícula está activa o no.

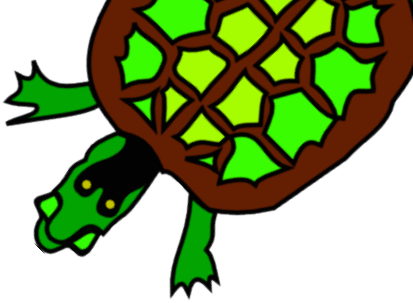
   - Si está activa, que la borre y dibuje otra de dimensiones `[100 * 100]`
   - Si no lo está, que dibuje una de dimensiones `[50 * 50]`

   A continuación, que sitúe a la tortuga en una posición generada por dos números aleatorios de la serie:

   ```
   { -505 , -455 , -405 , ... , 405 , 455 , 505 }
   ```

y rellene o no el cuadrado en el que se encuentra siguiendo la secuencia de un tablero de ajedrez.

Finalmente, repetir la secuencia varias veces hasta ver si se consigue dibujar un tablero de ajedrez



# Capítulo 10

# Listas

Ya hablamos antes de listas. `[ 53 gato [7 28] 4.9 ]` es una lista en xLogo; su primer elemento es `53`, el segundo es `gato`, el tercero es `[7 28]` y el último es `4.9`. Para almacenarlo en la variable `ejemplo` hacemos:

```
haz "ejemplo [ 53 gato [7 28] 4.9 ]
```

Debemos entender que las listas permiten guardar la información ordenada, y que son un elemento muy importante para simplificar muchos programas.

## 10.1. Primitivas

Disponemos de las siguiente primitivas para trabajar con listas y con palabras:

### Para manejarlas

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| Devolver el primer elemento | `primero :ejemplo` | `pr :ejemplo` |
| Devolver el último elemento | `ultimo :ejemplo` | |
| Devolver el elemento `n`–simo | `elemento n :ejemplo` | |
| Investigar `algo` en `variable` | `miembro "algo "lista` | |
| Contar el número de elementos | `cuenta :ejemplo` | |
| Devolver un elemento al azar | `elige :lista` | |

Ejemplo con una lista:

```
    haz "lista1 [Esto es una lista en xLogo]
#
    escribe primero :lista1            ---> Esto
```

```
escribe ultimo :lista1              ---> xLogo
escribe elemento 3 :lista1          ---> una
escribe miembro "es :lista1         ---> es una lista en xLogo
escribe cuenta :lista1              ---> 6
escribe elige :lista1               ---> en
```

Ejemplo con una palabra:

```
escribe primero "Calidociclos       ---> C
escribe ultimo "Calidociclos        ---> s
escribe elemento 3 "Calidociclos    ---> l
escribe cuenta "Calidociclos        ---> 12
escribe miembro "es "Calidociclos   ---> falso
escribe elige "Calidociclos         ---> i
```

obviamente, el resultado de `elige` variará de una ejecución a otra.
Para la primitiva `miembro`:

- Si `variable` es una lista, investiga dentro de esta lista; hay dos casos posibles:
  - Si `algo` está incluido en `variable`, devuelve la sub—lista generada a partir de la primera aparición de `algo` en `variable`.
  - Si `algo` no está incluido en `variable`, devuelve la palabra `falso`.
- Si `variable` es una palabra, investiga los caracteres `algo` dentro de `variable`. Dos casos son posibles:
  - Si `algo` está incluido en `variable`, devuelve el resto de la palabra a partir de `algo`.
  - Si no, devuelve la palabra `falso`.

### Para modificarlas

En este caso, sólo las dos primeras pueden usarse con palabras:

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| Quitar el primer elemento/letra | `menosprimero :ejemplo` | `mp` |
| Quitar el último elemento/letra | `menosultimo :ejemplo` | `mu` |
| Quitar el elemento `gato` | `quita "gato :ejemplo` | |
| Añadir `algo` el primero | `ponprimero "algo :ejemplo` | `pp` |
| Añadir `algo` el último | `ponultimo "algo :ejemplo` | `pu` |
| Intercalar `algo` en el lugar `n` | `agrega Lista n "algo` | |
| `Reemplaza`r el elemento `n` | `reemplaza Lista n "algo` | |
| Invertir la lista | `invierte :lista` | |

Ejemplo con una lista (`lista1` es la misma de antes):

```
escribe menosprimero :lista1      ---> es una lista en xLogo
escribe menosultimo :lista1       ---> Esto es una lista en
escribe quita "es :lista1         ---> Esto una lista en xLogo
escribe ponprimero "Super :lista1 ---> Super Esto es una lista en xLogo
escribe ponultimo "2007 :lista1   ---> Esto es una lista en xLogo 2007
escribe agrega :lista1 4 "super   ---> Esto es una super lista en xLogo
escribe reemplaza :lista1 3 "ye   ---> Esto ye una lista en xLogo
escribe invierte :lista1          ---> xLogo en lista una es Esto
```

Ejemplo con una palabra:

```
escribe menosprimero "Calidociclos  ---> alidociclos
escribe menosultimo "Calidociclos   ---> Calidociclo
```

### Para combinar

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| Combinar en una sola lista | `frase :ejemplo :algo` | `fr` |
| Concatenar en una sola palabra | `palabra :ejemplo :algo` | |
| Combinar en una lista de sublistas | `lista :ejemplo :algo` | |

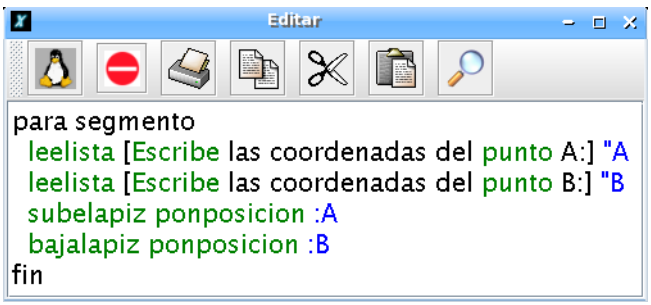
Ejemplos con una lista:

```
escribe frase :lista1 [y es genial]  ---> Esto es una lista en xLogo y es genial
escribe lista :lista1 [y es genial]
       '---> [Esto es una lista en xLogo] [y es genial]
```

Ejemplo con una palabra:

```
escribe frase "Calidociclos "de\ Escher  ---> Calidociclos de Escher
escribe lista "Calidociclos "de\ Escher  ---> Calidociclos de Escher
escribe palabra "Super "Calidociclos     ---> SuperCalidociclos
```

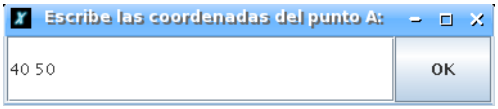
Existe una primitiva que permite que el *usuario* introduzca valores en xLogo, `leelista`:

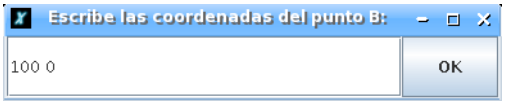


que se ejecuta tecleando:

```
segmento
```

xLogo abrirá una ventana pidiendo las coordenadas de `A`. Contestamos, por ejemplo, `40 50` y pulsamos `Intro`; nos pide luego las coordenadas de `B`, por ejemplo `100 0` e `Intro`.




La tortuga dibujará el segmento cuyos extremos son los puntos cuyas coordenadas hemos introducido.

**Observación:** `A` y `B` son **listas**, no números. En ejemplo anterior `A` contiene `[40 50]`, y podemos *convertir* sus elementos en números usando `primero`, `ultimo` ó `ultimo`, y usarlos con el resto de primitivas haciendo:

```
avanza primero :A
```

A la inversa, para convertir dos números en una lista tenemos la primitiva `lista`:

```
lista :lado :altura
```

Si se tratara de más de tres números disponemos de `frase` y `lista`:

```
haz "a 50 haz "b 60 haz "c 70
escribe lista :a frase :b :c
escribe lista :a lista :b :c
```

proporcionan: `50 [60 70]` pero sus *formas generales*:

```
escribe (lista :a :b :c)          escribe (frase :a :b :c)
```

proporcionan: `50 60 70` esto es, una única frase. La diferencia con `palabra` es que esta concatena las variables. En el ejemplo anterior:

```
escribe palabra :a :b
escribe (palabra :a :b :c)
```

proporcionan:

```
5060
506070
```

esto es, una única palabra (en este caso, un número).

## 10.2. Ejemplo: Conjugación

Vamos a construir un programa que conjugue el futuro simple de un verbo (sólo para verbos regulares) de distintas formas, y veremos cómo el uso de listas simplifica el manejo de la información.

### 10.2.1. Primera versión

La primera posibilidad que se nos puede ocurrir es usar una línea para cada persona:

```
para futuro :verbo
  es frase "yo palabra :verbo "é
  es frase "tú palabra :verbo "ás
  es frase "él palabra :verbo "á
  es frase "nosotros palabra :verbo "emos
  es frase "vosotros palabra :verbo "éis
  es frase "ellos palabra :verbo "án
fin
```

lo que consigue fácilmente nuestro objetivo:

**Ejemplo**: `futuro "amar`

```
yo amaré
tú amarás
él amará
nosotros amaremos
vosotros amaréis
ellos amarán
```

### 10.2.2. Segunda versión

Podemos ser un poco más elegantes, con la primitiva `repite` o `repitepara` (sección 11.1.2) combinada con listas:

```
para futuro :verbo
  haz "pronombres [yo tú él nosotros vosotros ellos]
  haz "terminaciones [é ás á emos éis án]
  repitepara [i 1 6]
   [ escribe frase elemento :i :pronombres
          palabra :verbo elemento :i :terminaciones ]
fin
```

ya que, realmente, estamos repitiendo seis veces la misma estrategia: combinar el pronombre con el verbo y la terminación. El resultado, el mismo de antes.

### 10.2.3. Tercera versión

En esta versión usaremos la recursividad, una característica muy útil en el uso de listas (ver sección 11.5)

Analiza qué hace este programa y cómo lo hace. Usa un papel para seguir la secuencia de pasos que da, y razona qué puede ser eso que llamamos *recursividad* (o *recurrencia*)

```
para futuro :verbo
  haz "pronombres [yo tú él nosotros vosotros ellos]
  haz "terminaciones [é ás á emos éis án]
  conjugar :verbo :pronombres :terminaciones
fin


para conjugar :verbo :pronombres :terminaciones
  si vacio? :pronombres [alto]
  escribe frase primero :pronombres palabra :verbo primero :terminaciones
  conjugar :verbo menosprimero :pronombres menosprimero :terminaciones
fin
```

## 10.3. Ejercicios

1. Plantea un programa sobre conjugación de verbos que:
   a) Pida un verbo con una ventana emergente
   b) Determine a qué conjugación pertenece
   c) Contenga dos listas:
      - `"pronombres` con los pronombres personales
      - `"morfemas` con los morfemas para conjugar el presente, siendo distinta en función de la conjugación del verbo
   d) Combine la raiz del verbo con las terminaciones del presente en una única palabra
   e) Combine en una frase los pronombres con la palabra generada, y las muestre en el Histórico de Comandos con `escribe`
2. ¿Cómo puede extraerse el `22` de la lista de listas `[[22 3] [4 5] [8 35]]`?
3. Plantea el procedimiento `prime`, con una entrada, `listado`, que devuelva su primer elemento, sin usar `primero`
4. Plantea el procedimiento `ulti`, con una entrada, `listado`, que devuelva su último elemento, sin usar `ultimo`

5. Diseña el procedimiento `triangulo`, que **pida** las coordenadas de los vértices de un triángulo uno a uno y lo dibuje

6. Escribe un procedimiento que pida la medida del lado de un cuadrado y devuelva la medida de su diagonal

7. Plantea el procedimiento `mengua` que pida escribir una serie de números y escriba las listas que se obtienen al ir eliminando un elemento de cada vez (por ejemplo el último) hasta quedar vacía.

8. Diseña un procedimiento `inversa` que reciba una lista y la devuelva con los elementos dispuestos en orden inverso al inicial

9. Plantea el procedimiento `maximo`, que pida una serie de números y devuelva el mayor de todos ellos

10. Diseña un procedimiento `suprime`, con dos entradas: `n` y `listado`, que devuelva la lista que se obtiene al suprimir el elemento `n`–simo, sin usar `quita`

11. Plantea el procedimiento `adjunta`, con tres entradas; `n`, `listado_1` y `listado_2`, que añada `listado_1` en la posición `n` de `listado_2` (sin usar `agrega`)

12. Escribe un procedimiento que determine si una palabra es un palíndromo, es decir, que se lee igual en la forma habitual que de derecha a izquierda

    `palíndromo? reconocer ---> cierto`

    `palíndromo? anilina ---> cierto`

    `palíndromo? animal ---> falso`

13. ¿Cómo ampliarías el procedimiento anterior para que trabajara también con frases?

    `palíndromo? Sé verlas al revés ---> cierto`

    `palíndromo? Átale o me delata ---> falso`

    Puedes encontrar palíndromos en:

    `http://es.wikipedia.org/wiki/Palíndromo`

    `http://es.wikiquote.org/wiki/Palíndromos`

    `http://www.juegosdepalabras.com/palindromo.htm`

    `http://www.carbajo.net/varios/pal.html`

## 10.4. Listas de Propiedades

Desde la versión 0.9.92, pueden definirse Listas de Propiedades con xLogo. Cada lista tiene un nombre específico y contiene una pareja de "valores–clave".

Para manejar estas listas, podemos utilizar las siguientes primitivas, para las que, por ejemplo, podemos considerar una lista llamada "coche", que debe contener la clave "`color`" asociado al valor "`rojo`", otra clave "`tipo`" con el valor "`4x4`" y una tercera clave "`vendedor`" asociada al valor "`Citröen`",

| Descripción | Primitiva | Ejemplo (con forma corta) |
|---|---|---|
| Añadir una propiedad a la lista | `ponpropiedad` | `ponprop "Coche "Color "Rojo` |
| Devolver el valor asociado a una `clave` de la lista | `leepropiedad` | `leeprop "Coche "Color` |
| Borrar el par `clave-valor` de una lista | `borrapropiedad` | `boprop "Coche "Color` |
| Mostrar todos los elementos de una lista | `listapropiedades` | `listaprop "Coche` |
| Mostrar todas las listas creadas | `listaspropiedades` | `listasprop "Coche` |

Juguemos con los elementos de la lista de propiedades llamada "coche".

```
# Llenado de la Lista de Propiedades
ponpropiedad "Coche "Color "Rojo
ponpropiedad "Coche "Tipo "4x4
ponpropiedad "Coche "Vendedor "Citroen

# Mostrar un valor
escribe leepropiedad "Coche "Color    ---> Rojo

# Mostrar todos los elementos
escribe listapropiedades "Coche       ---> Color Roja Tipo 4x4 Vendedor Citroen
```

# 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:

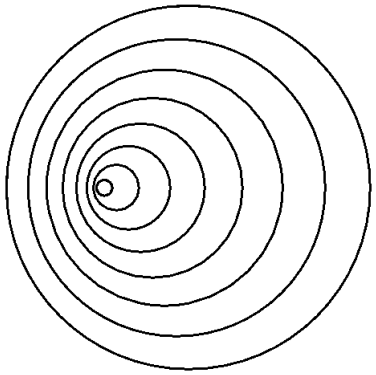
  - Si está incluído en un bucle `repite` o `mientras`, el programa sale del bucle inmediatamente.
  - Si está en un procedimiento, este es terminado.

- `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 $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

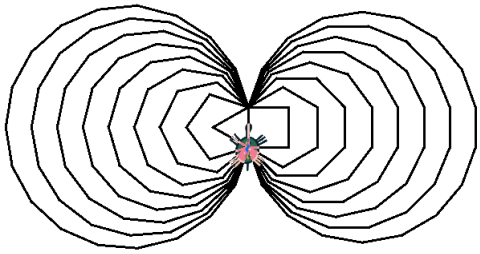
5. Escribe un procedimento con un número como argumento y escriba sus divisores.

```
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.

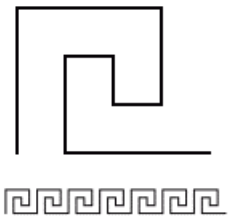
```
primo? 123
  no es primo
primo? 127
  127 es primo
```

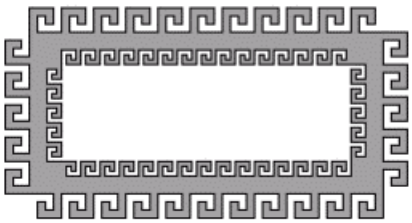
7. Dibuja esta serie de polígonos, en la que los que tienen un número impar de vértices están hacia la izquierda, pero los que tienen un número impar de vértices están hacia la derecha:



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.

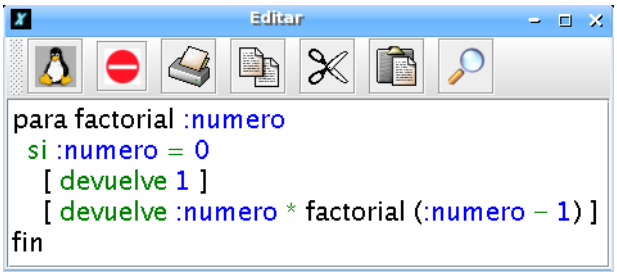
El caso más típico es el cálculo del **factorial**. En lugar de definir

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

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:

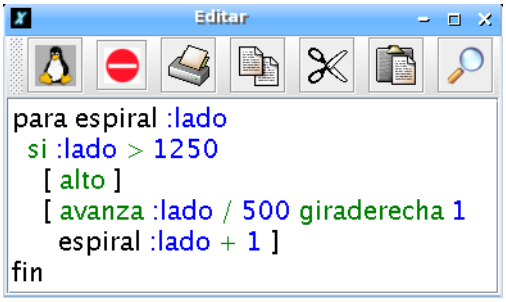


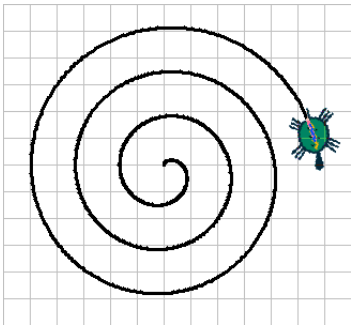

que va llamándose a sí mismo en cada ejecución, pero cada vez con el número reducido en una unidad. Cuando dicho número es cero, devuelve `"1"`, y se termina la recursión. Es decir:

```
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.

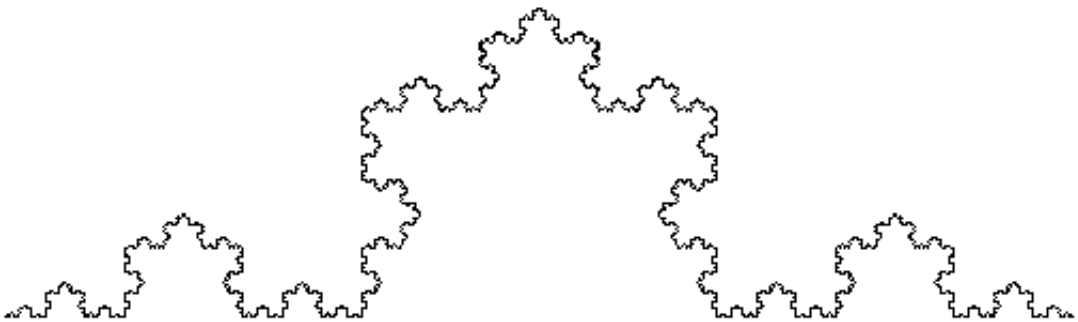
Llamemos $L_{n,\ell}$ al elemento de longitud $\ell$, correspondiente al paso $n$. Para dibujar este elemento:

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
```

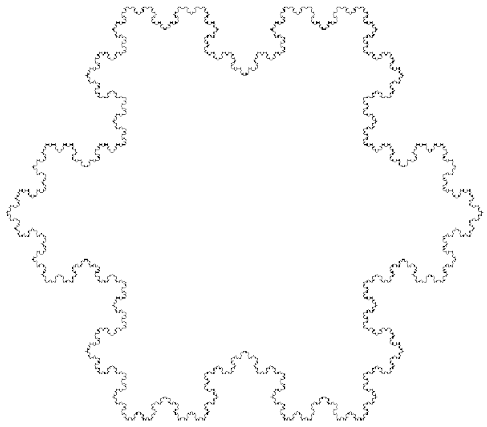
Ejecutando `linea 500 8` obtenemos:



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 $\pi$ (1)

Podemos aproximar el valor del número $\pi$ 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 $\pi$ .

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 $\pi$, 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....
```

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)

# Capítulo 12

# Recibir entrada del usuario

Si consideramos *Interacción* como *comunicación*, es evidente que debe ir en las dos direcciones: mensajes de xLogo hacia el ususario y respuesta de este, bien por teclado o con el ratón.

## 12.1. Comunicación con el usuario

Ya conocemos varias primitivas que permiten escribir mensajes, bien en el Histórico de comandos, bien en pantalla o incluso con una ventana emergente:

| Primitivas | Argumentos | Uso |
|---|---|---|
| `escribe` | `cualquiera` | Escribe en el Histórico de comandos el argumento. |
| `tipea` | `cualquiera` | Idéntico a `escribe`, pero el cursor queda en la linea donde se mostró el contenido del argumento. |
| `rotula` | `palabra o lista` | Dibuja la palabra o lista especificada, en la posición actual, y en la dirección que está mirando. |
| `largoetiqueta` | `lista` | Devuelve, en píxels, la longitud en pantalla de la lista. |
| `mensaje`, `msj` | `lista` | Muestra una caja de diálogo con el mensaje que está en la lista. El programa se detiene hasta que el usuario hace un *clic* en el botón "Aceptar" |

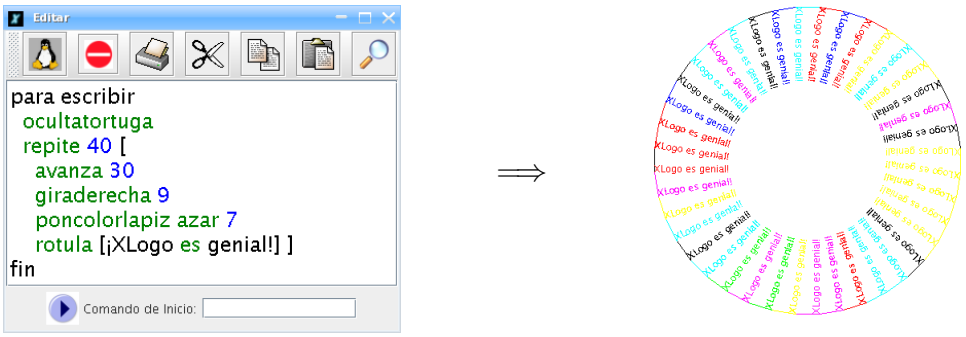
Llevamos ya tiempo trabajando con `escribe`, y utilizamos `tipea` al hablar de las operaciones (sección 7.1). Por su parte, `mensaje` apareció en el capítulo 3, en el programa `juego.lgo` para indicarnos si habíamos chocado con una "piedra" o llegado al "lago".

La primitiva `largoetiqueta` permite saber, entre otras muchas posibilidades, si al escribir en pantalla con `rotula` tienes suficiente espacio.

**Ejemplo**:

`largoetiqueta [Hola, ¿cómo estás?]` devuelve, en píxels la longitud en pantalla de la frase *Hola, ¿cómo estás?*

**Ejemplo**:




## 12.1.1. Ejercicios

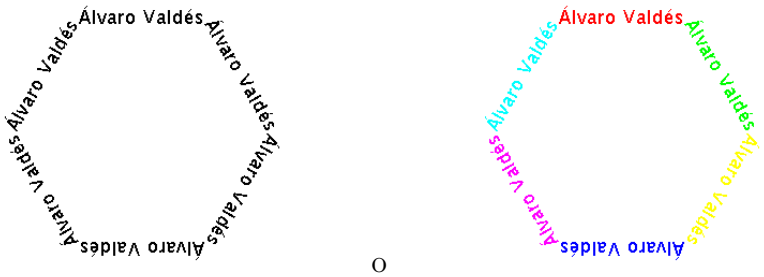
1. Escribe tu nombre centrado en la pantalla
2. Plantea un procedimiento que recibe una lista como argumento, siendo esa lista el nombre y primer apellido de una persona. Con ello:

   a) Determina el número de letras del nombre, $n$

   b) Dibuja un polígono de $n$ lados, siendo el lado el nombre y apellido antes dado.

   c) Puedes intentar que cada lado sea de un color

   En mi caso, sería: `nombre.poligono [Álvaro Valdés]`, y el resultado:




o

3. Escribe un procedimiento que escriba en el Histórico de Comandos los números del 1 al 100, con diez en cada línea, es decir:

```
1  2  3  4  5  6  7  8  9  10
11  12  13  14  15  16  17  18  19  20
      ...
91  92  93  94  95  96  97  98  99  100
```

4. Recupera el procedimiento `raiz_con_prueba` de la sección 9.2, y haz que cuando el radicando sea negativo, muestre un mensaje avisando de ello en una ventana, en vez de en el Histórico de Comandos.

### 12.1.2. Propiedades del Histórico de Comandos

Esta tabla define las primitivas que permiten ajustar y preguntar las propiedades de texto del área del Histórico de Comandos, es decir, las primitivas que controlan el color y tamaño del texto en este área.

Sólo afectan a las primitivas `escribe` y `tipea`. Para `rotula` se describen en la sección siguiente.

| Primitivas | Argumentos | Uso |
|---|---|---|
| `borratexto`, `bt` | `no` | Borra el **Área de comandos**, y el área del **Histórico de comandos**. |
| `ponfuentetexto`, `pft` | `número` | Define el tamaño de la tipografía del área del **Histórico de comandos**. Sólo disponible para ser usada por la primitiva `escribe`. |
| `poncolortexto`, `pctexto` | `número o lista` | Define el color de la tipografía del área del **Histórico de comandos**. Sólo disponible para ser usada por la primitiva `escribe`. |
| `ponnombrefuentetexto`, `pnft` | `número` | Selecciona la tipografía número `n` para escribir en el área del **Histórico de comandos** con la primitiva `escribe`. Puedes encontrar la relación entre fuente y número en el menú **Herramientas → Preferencias → Fuente.** |
| `ponestilo`, `pest` | `lista o palabra` | Define los efectos de fuente para los comandos en el **Histórico de comandos**. |
| `fuentetexto`, `ftexto` | `no` | Devuelve el tamaño de la tipografía usada por la primitiva `escribe`. |
| `colortexto` | `no` | Devuelve el color de la tipografía usada por la primitiva `escribe` en el área del **Histórico de comandos**. |
| `nombrefuentetexto`, `nft` | `no` | Devuelve una lista con dos elementos. El primero es un número correspondiente a la fuente utilizada para escribir en el área del **Histórico de comandos** con la primitiva `escribe`. El segundo elemento es una lista que contiene el nombre de la fuente. |

| Primitivas | Argumentos | Uso |
|---|---|---|
| `estilo`, | `no` | Devuelve una lista que contiene todos los efectos de fuente utilizados por las primitivas `escribe` y `tipea`. |

Puedes elegir entre siete estilos:

- `ninguno`, utiliza la fuente sin ningún cambio
- `negrita`
- `cursiva`
- `tachado`
- `subrayado`
- `superindice`
- `subindice`

Si se quiere aplicar varios estilos a la vez, deben escribirse en una lista.

**Ejemplos de estilos de fuente:**

```
ponestilo [negrita subrayado] escribe "Hola
```

Devuelve:

**Hola**

```
pest "tachado tipea [Tachado] pest "cursiva tipea "\ x
             pest "superindice escribe 2
```

Devuelve:

~~Tachado~~ $x^2$

Las primitivas `caracter`, (su forma corta es `car` y cuyo argumento es `n`: un número) y `unicode "a`, devuelven, respectivamente, el carácter unicode que corresponde al número `n` y el número unicode que corresponde al carácter `a`.

**Ejemplo**:
`unicode "A` devuelve `65`
`caracter 125` devuelve `}`

### 12.1.3. Ejercicios

1. Escribe programas que, dado un valor $r$, calculen:

   a) el perímertro de una circunferencia

   b) el área de un círculo

   c) el área de una esfera

   d) el volumen de una esfera

en todos los casos de radio $r$, indicando claramente sus unidades y evitando que lea valores negativos:

`perim.circ 5`

`El perímetro de una circunferencia de radio 5 es 3.141592 m`$^2$

2. Escribe un procedimiento con dos argumentos: `peso` y `altura`. y que calcule su índice de masa corporal (`i.m.c.`) mostrando cómo se calculó. El i.m.c. se calcula con la fórmula

$$\text{i.m.c.} = \frac{\text{peso}}{\text{altura}^2}$$

`i.m.c. 78 173`

`Tu i.m.c. es 78 / 173`$^2$ `= 26.061679307694877`

Un i.m.c. muy alto indica obesidad. Los valores "normales" de i.m.c. están entre 20 y 25, pero esos lmites dependen de la edad, del sexo, de la constitución física, etc.

3. Escribe dos procedimentos que lean:

   a) una temperatura en grados Celsius y la devuelvan en Fahrenheit

   b) una temperatura en grados Fahrenheit y la devuelvan en Celsius

   La relación entre grados Celsius (°C) y grados Fahrenheit (°F) es:

$$°\text{F} - 32 = \frac{9}{5} \cdot (°\text{C})$$

   `C.a.F 35`

   `35.0 °C son 95.0 °F`

   OJO: se trata de obtener el "circulito" de "grados": °, no vale usar el símbolo "primero": º

4. Escribe un procedimiento que lea una fórmula química escrita directamente y la ponga en la forma habitual. Es decir, que convierta a los números en subíndices y deje las letras "normales":

   `fórmula.química H2SO4` $\longrightarrow$ $\mathrm{H_2SO_4}$

   Pista: Utiliza el valor unicode asociado a los números

5. ¿Cómo ampliarías el procedimiento anterior para que, además, calculara la masa molecular del compuesto escrito?

6. Escribe un procedimiento que lea una lista de varias palabras y que la reescriba en el Histórico de Comandos en Mayúsculas, distinguiendo las que ya empezaban por mayúsculas:

   `a.mayúsculas [Esto es un ejemplo]`

   `ESTO ES UN EJEMPLO`

### 12.1.4. Escritura en Pantalla

En la sección anterior controlamos la escritura en el Histórico de Comandos. Modifiquemos parámetros al escribir en el Área de Dibujo.

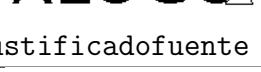
| Primitiva | Forma larga | Forma corta |
|---|---|---|
| `ponnombrefuente`, `pnf` | `número` | Modifica la tipografía con la que se escribe. Para ver la lista completa de fuentes disponibles, Menú Herramientas ⟶ Opciones ⟶ Pestaña Fuente |
| `nombrefuente`, `nf` | `número` | Devuelve el tipo de la letra con que se rotula. |
| `ponfuente`, `pf` | `número` | Modifica el tamaño de la tipografía con que se rotula. Por defecto, el tamaño es 12. |
| `fuente` | `no` | Devuelve el tamaño de la letra con que se rotula. |
| `ponjustificadofuente` | `lista` | Indica cómo alinea el texto alrededor de la tortuga. La lista debe contener dos números: el alineamiento horizontal y el vertical. |
| `justificadofuente` | `no` | Devuelve la alineación del texto en pantalla. |

Si queremos cambiar el color de las letras, debemos cambiar el color del lápiz.

Respecto a la alineación del texto, las opciones de la primitiva `ponjustificadofuente [a b]` proporcionan:

- Siendo `a` la alineación horizontal:
  - 0: A la izquierda.
  - 1: Centrado horizontalmente.
  - 2: A la derecha.
- Mientras, que `b`, la alineación vertical:
  - 0: En la zona inferior.
  - 1: Centrado verticalmente.
  - 2: En la zona superior.

Gráficamente, al teclear: `ponfuente 50 rotula "XLOGO`

| | | |
|---|---|---|
|  `ponjustificadofuente [2 0]` |  `ponjustificadofuente [1 0]` |  `ponjustificadofuente [0 0]` |
|  `ponjustificadofuente [2 1]` |  `ponjustificadofuente [1 1]` |  `ponjustificadofuente [0 1]` |
|  `ponjustificadofuente [2 2]` |  `ponjustificadofuente [1 2]` |  `ponjustificadofuente [0 2]` |

### 12.1.5. Ejercicios

1. Observa la portada de este "libro". ¿Cómo harías para conseguir la "sombra" azulada de las letras que forman la palabra xLogo? (no hace falta que busques la misma fuente).

2. En la sección 10.2, creamos tres programas distintos para conjugar el futuro simple de un verbo regular. Modifica uno de ellos para que el resultado se muestre en el Área de Dibujo dispuesto. verticalmente

3. Haz lo mismo que en el ejercicio anterior, pero con el diseñado en 10.3 que conjuga presente y pasado, y añade una "etiqueta" encima de cada serie que indique el tiempo verbal.

## 12.2. Interactuar con el teclado

Durante la ejecución del programa, se puede recibir texto ingresado por el usuario a través de 3 primitivas: `tecla?`, `leecar` y `leelista`.

- `tecla?`: Da `cierto` o `falso` según se haya pulsado o no alguna tecla desde que se inició la ejecución del programa.

- `leecar` o `leetecla`:

  - Si `tecla?` es `falso`, el programa hace una pausa hasta que el usuario pulse alguna tecla.
  - Si `tecla?` es `cierto`, devuelve el valor unicode de la última tecla que haya sido pulsada.

Estos son los valores que dan ciertas teclas:

| | | | | |
|---|---|---|---|---|
| A → 65 | B → 66 | C → 67 | ... | Z → 90 |
| ◁ → -37 ó -226 | △ → -38 ó -224 | ▷ → -39 ó -227 | ▽ → -40 ó -225 | |
| `Esc` → 27 | `F1` → -112 | `F2` → -113 | ... | `F12` → -123 |
| `SHIFT` → -16 | `ESPACIO` → 32 | `CTRL` → -17 | `ENTER` → 10 | |

En Mac, las teclas `F1` a `F12` no devuelven valor alguno, ya que el sistema las reserva para tareas "propias".

Si tienes dudas acerca del valor que da alguna tecla, puedes probar con: `es leecar`. El intérprete esperará hasta que pulses una tecla, y escribirá su valor.

- `leelista [titulo] "palabra` o `leeteclado [titulo] "palabra`: Presenta una caja de diálogo titulada `titulo`. El usuario puede escribir un texto en el área de entrada, y esta respuesta se guardará seleccionando automáticamente si en forma de número o de lista en la variable `:palabra` cuando se haga *clic* en el botón `OK`.

  **Ejemplos**:

```
para edades
  leelista [¿Qué edad tienes?] "edad
  si :edad < 18 [escribe [Eres menor]]
  si :edad > 17 [escribe [Eres adulto]]
  si :edad > 69 [escribe [Con todo respeto!!]]
fin

para dibujar
# La tortuga es controlada con las flechas del teclado.
# Se termina con Esc.
si tecla?
  [ haz "valor leecar
    si :valor=-37 [giraizquierda 90]
    si :valor=-39 [giraderecha 90]
    si :valor=-38 [avanza 10]
    si :valor=-40 [retrocede 10]
    si :valor=27 [alto] ]
 dibujar
fin
```

## 12.3. Ejercicios

1. Escribe un procedimiento que haga que la tortuga:

- dibuje en pantalla el número que se pulsa en el teclado
- rotule en pantalla las letras que se pulsen en el teclado

quedando en espera para dibujar o rotular más números o letras a la derecha de los ya introducidos. Necesitarás:

a) Hacerlo recursivo, de modo que termine al pulsar `Esc`

b) Ir desplazando a la tortuga hacia la derecha a medida que son dibujados los números o rotuladas las letras.

c) OJO: Dibujados, no rotulados. Esto implica crear 10 procedimientos, uno por cada cifra.

No obstante, para ver si la parte que lee y desplaza a la tortuga está bien diseñada, puedes usar `rotula` para mostrar el resultado en el Área de Dibujo como con las letras.

d) Para cambiar el tamaño de las letras, usa `ponfuente` (sección 12.1.4)

2. Escribe un procedimiento que lea una lista de varias palabras y que la reescriba en el Histórico de Comandos del siguiente modo:

a) Si la palabra empieza por vocal, en cursiva (itálica)

b) Si es un número:

1) Como subíndice si es par
2) Como superíndice si es impar

c) Si la palabra empieza por consonante:

1) Si está entre `b` y `l`, tachada
2) Si está entre `m` y `z`, subrayada

d) Si, además, empieza por mayúscula, en negrita

Pista: Observa que usando `unicode`, las letras mayúsculas `A - Z` devuelven valores consecutivos entre 65 y 90, mientras que las minúsculas `a - z` lo hacen entre 97 y 122. Debes tener cuidado, eso sí, con los caracteres especiales del castellano: vocales acentuadas y "eñes".

3. Modifica el procedimiento `dibujar` para que suba o baje el lápiz al pulsar la barra espaciadora, borre al pulsar "borrar", ... y/o cualquier otra opción que se te ocurra (cambiar el color al pulsar la inicial del mismo, ...)

4. **Un pequeño juego**: Diseña un juego tal que el programa elija un número entre 0 y 32 aleatoriamente (recuerda la primitiva `azar` (página 62). A continuación, abra un cuadro de diálogo que pida al usuario que introduzca un número.

Si este número entero es igual al guardado, muestra "GANASTE"en la zona de texto. En caso contrario, el programa indica si el númer guardado en mayor o menor que el introducido por el usuario y vuelve a abrir el cuadro de diálogo.

El programa terminará cuando el usuario haya dado el número correcto.

UNA AYUDA:

- El número elegido por XLOGO se almacena en una variable llamada `numero`.
- El cuadro de diálogo se llamará "Dame un número, por favor"
- El número elegido por el usuario se almacena en una variable llamada `intento`.
- El procedimiento principal se llamará `juego`.

**Algunas posibles mejoras**:

- Mostrar el número de intentos.
- Que XLOGO elija un número entre 0 y 2000.
- Comprobar que el usuario introduce un número válido. (Recuerda la primitiva `número?`, página 76).

## 12.4. Interactuar con el ratón

Durante la ejecución del programa, se pueden recibir eventos del ratón a través de tres primitivas: `leeraton`, `raton?` y `posraton`.

- `leeraton`: el programa hace una pausa hasta que el usuario hace un *clic* o un movimiento. Entonces, da un número que representa ese evento. Los diferentes valores son:
  - 0 → El ratón se movió.
  - 1 → Se hizo un *clic* izquierdo.
  - 2 → Se hizo un *clic* central (Linux).
  - 3 → Se hizo un *clic* derecho.

  En Mac, sólo se reciben 0 y 1, ya que el ratón no tiene botón derecho, y la opción de mantener pulsado `Ctrl` no es interpretada por XLOGO como "botón derecho".
- `posraton`: Da una lista que contiene la posición actual del ratón.
- `raton?`: Devuelve `cierto` o `falso` según toquemos o no el ratón desde que comienza la ejecución del programa

**Ejemplos**:

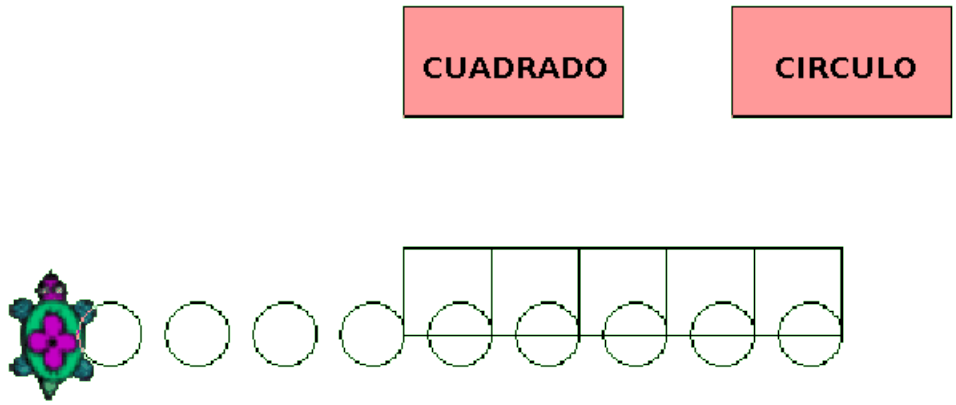
En este primer procedimiento, la tortuga sigue los movimientos del ratón por la pantalla.

```
para seguir
# cuando el raton se mueve, la tortuga cambia de posicion.
  si leeraton=0 [ponpos posraton]
  seguir
fin
```

Este segundo procedimiento es similar, pero hay que hacer *clic* izquierdo para que la tortuga se mueva.

```
para seguir2
  si leeraton = 1 [ponpos posraton]
  seguir2
fin
```

En este tercer ejemplo, hemos creado dos botones rosa. Si hacemos *clic* izquierdo, la tortuga dibuja un cuadrado de lado 40. Si hacemos *clic* derecho, la tortuga dibuja un pequeño círculo. Por último si hacemos *clic* derecho en el botón derecho, se detiene el programa.




```
para boton
# crea un boton rectangular color rosa, de 50 x 100.
  repite 2 [
   avanza 50 giraderecha 90 avanza 100 giraderecha 90 ]
  giraderecha 45 subelapiz avanza 10
```

```
   bajalapiz poncolorlapiz [255 153 153]
   rellena retrocede 10 giraizquierda 45 bajalapiz poncolorlapiz 0
fin

para empieza
  borrapantalla boton subelapiz ponpos [150 0] bajalapiz boton
  subelapiz ponpos [30 20] bajalapiz rotula "Cuadrado
  subelapiz ponpos [180 20] bajalapiz rotula "Circulo
  subelapiz ponpos [0 -100] bajalapiz
  raton
fin

para raton
# ponemos el valor de leeraton en la variable ev
# ponemos la primer coordenada en la variable x
# ponemos la segunda coordenada en la variable y
  haz "ev leeraton
  haz "x elemento 1 posraton
  haz "y elemento 2 posraton
# si hay clic izquierdo
  si :ev=1 & :x>0 & :x<100 & :y>0 & :y<50 [cuadrado]
# si hay clic derecho
  si :x>150 & :x<250 & :y>0 & :y<50 [
  si :ev=1 [circulo]
  si :ev=3 [alto] ]
  raton
fin

para circulo
  repite 90 [av 1 gi 4]
  giraizquierda 90 subelapiz avanza 40 giraderecha 90 bajalapiz
fin

para cuadrado
  repite 4 [avanza 40 giraderecha 90]
  giraderecha 90 avanza 40 giraizquierda 90
fin
```

## 12.5. Ejercicios

1. Diseña un procedimiento que dibuje un tablero de ajedrez, y determine el color de una casilla al hacer "*clic*" sobre ella.

2. Observa el funcionamiento del ejemplo anterior, y piensa cómo podrías utilizarlo para diseñar un procedimiento que cargue en el área de dibujo un mapa "mudo" (Deberás utilizar la primitiva `cargaimagen`, sección 13.6.2) y el alumno tenga que ir haciendo "*clic*" en un área según se le vaya mostrando el nombre de la misma (por ejemplo, Países, Comunidades Autónomas, Concejos, ...).

   Simplifica lo más posible el planteamiento, ya veremos más adelante cómo "pulirlo" para tener en cuenta la forma de las fronteras.

3. Plantea un procedimiento que haga que la tortuga "persiga al ratón". Para ello, la tortuga debe quedarse en un punto hasta que se hace *clic* en otro, hacia donde se dirigirá despacio después de haberse orientado hacia allí. Puedes ralentizar el movimiento con un procedimiento del tipo:

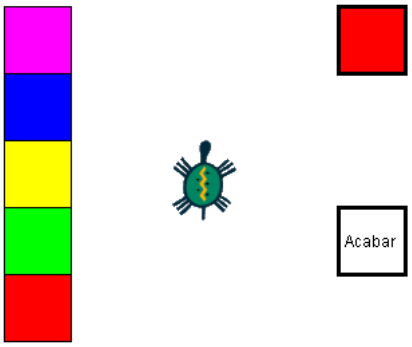
```
para avance.lento :distancia
  repite :distancia
    [ avanza 1 espera 10 ]
fin
```

   Explicaremos `espera` en el capítulo 18.

4. Plantea un procedimiento que:

   a) Dibuje varios cuadrados apilados y los rellene con distintos colores
   b) Dibuje otros dos, dejando uno en blanco y en el otro rotula "Acabar"
   c) Al hacer *clic* en uno de los cuadrados coloreados, el que está en blanco adquirirá el color de aquél.

      Necesitarás la primitiva `encuentracolor`, que se explica en el capítulo siguiente.
   d) Al hacer *clic* en el que dice "Acabar", el programa terminará




5. **Este problema implica conocimientos de Física**: Plantea un procedimiento que calcule el potencial electrostático y/o gravitatorio en el origen de coordenadas de un sistema de cargas o masas ubicadas haciendo *clic* con el ratón.

   Con cada "*clic*" del ratón, se debe dibujar un círculo centrado en ese punto.

## 12.6. Componentes Gráficos

Desde la versión 0.9.90, xLogo permite añadir componentes gráficos en el Área de dibujo (botones, menús, ...)

Las primitivas que permiten crear y modificar estos componentes terminan con el sufijo `igu` (Interfaz Gráfica de Usuario – *Graphical User Interface, gui* son sus siglas inglesas).

### 12.6.1. Crear un componente gráfico

La secuencia de pasos que debes seguir es: **Crear** → **Modificar** sus propiedades o características → **Mostrar**lo en el Área de dibujo.

#### Crear un Botón

Usaremos al primitiva `botonigu`, cuya sintaxis es:

```
# Esta primitiva crea un boton llamado b
# y cuya leyenda es: Clic
botonigu "b "Clic
```

#### Crear un Menú

Disponemos de la primitiva `menuigu`, cuya sintaxis es:

```
# Esta primitiva crea un menu llamado m
# y que contiene 3 opciones: opcion1, opcion2 y opcion3
menuigu "m [opcion1 opcion2 opcion3]
```

#### Modificar las propiedades del componente gráfico

`posicionigu` determina las coordenadas donde se colocará el elemento gráfico. Por ejemplo, para colocar el botón definido antes en el punto de coordenadas `(20 , 100)`, escribiremos:

```
posicionigu "b [20 100]
```

Si no se especifica la posición, el objeto será colocado por defecto en la esquina superior izquierda del Área de dibujo.

#### Eliminar un componente gráfico

La primitiva `eliminaigu` elimina un componente gráfico. Para eliminar el botón anterior

```
eliminaigu "b
```

### Definir acciones asociadas a un componente gráfico

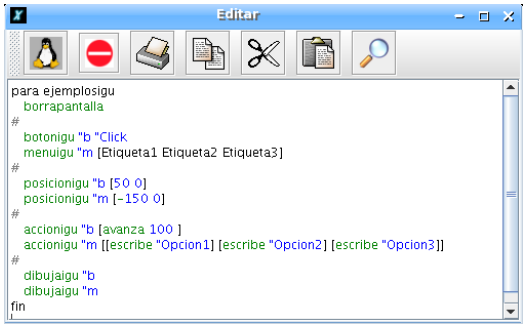
La primitiva `accionigu`, define una acción asociada al componente, y que se realizará cuando el usuario interactúa con él.

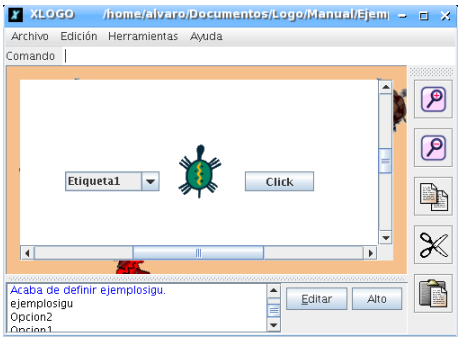
```
# Que la tortuga avance 100 al pulsar el boton "b
accionigu "b [avanza 100 ]
# En el menu, cada opcion indica su accion
accionigu "m [[escribe "Opcion1] [escribe "Opcion2] [escribe "Opcion3]]
```

### Dibujar el componente gráfico

La primitiva `dibujaigu`, muestra el componente gráfico en el Área de dibujo. Para mostrar el botón que estamos usando como ejemplo:

```
dibujaigu "b
```

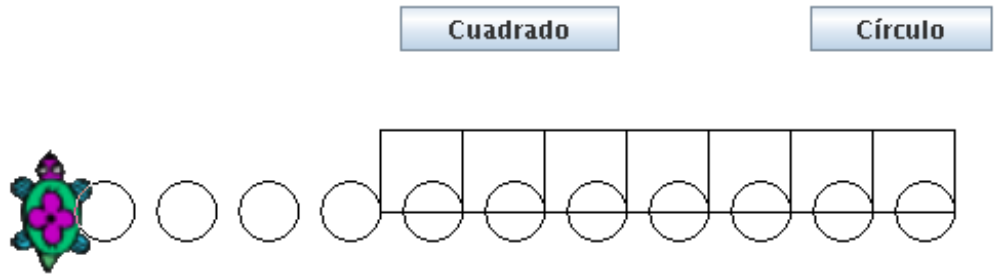



Cambiemos el ejemplo anterior utilizando las nuevas primitivas:

```
para empieza
  botonigu "Boton_Circ "Circulo
  botonigu "Boton_Cuad "Cuadrado
  posicionigu "Boton_Circ   [50 100]
  posicionigu "Boton_Cuad [-150 100]
  accionigu "Boton_Circ [ circunferencia ]
  accionigu "Boton_Cuad [ cuadrados ]
  dibujaigu "Boton_Circ
  dibujaigu "Boton_Cuad
fin

para circunferencia
  repite 90 [av 1 gi 4]
  giraizquierda 90 subelapiz avanza 40 giraderecha 90 bajalapiz
fin
```

```
para cuadrado
  repite 4 [avanza 40 giraderecha 90]
  giraderecha 90 avanza 40 giraizquierda 90
fin
```




## 12.7. Ejercicios

1. a) Crea cuatro botones: `Av`, `Re`, `Gd` y `Gi`, cuyas etiquetas sean, respectivamente: `Avanzar`, `Retroceder`, `Girar Derecha` y `Girar Izquierda`

   b) Ubica los botones anteriores en las posiciones `[-350 25]`, `[-360 -25]`, `[-300 0]` y `[-440 0]`

   c) Asígnales las acciones: `avanza 10`, `retrocede 10`, `giraderecha 90` y `giraizquierda 90`

   d) Dibuja los cuatro botones en pantalla y comprueba que "funcionan" correctamente.

   e) Crea dos menús `Cl` y `Gl`, cuyas opciones sean, respectivamente: `[Color Lápiz Azul Rojo Amarillo Verde Negro Blanco]` y `[Grosor Lápiz 1 2 3 4 5]`

   f) Asígnales las acciones:

      1) `[ ] [poncolorlapiz azul] [poncolorlapiz rojo] [poncolorlapiz amarillo] [poncolorlapiz verde] [poncolorlapiz negro] [poncolorlapiz blanco]`
      2) `[ ] [pongrosor 1] [pongrosor 2] [pongrosor 3] [pongrosor 4] [pongrosor 5]`

      Observa que, en ambos casos, la primera opción NO hace nada

   g) Ubícalos en las posiciones `[-350 100]` y `[-350 70]`

   h) Dibuja los dos menús en pantalla y comprueba que "funcionan" correctamente.

   Compara el resultado de los ejercicios con el xLogo con que iniciamos el curso. ¿Serías capaz de diseñar, asignar y dibujar los elementos que faltan?

2. Crea un programa sobre conjugación de verbos que:

   a) Lea un verbo con `leelista`, analizando previamente si ya ha sido definido uno o no

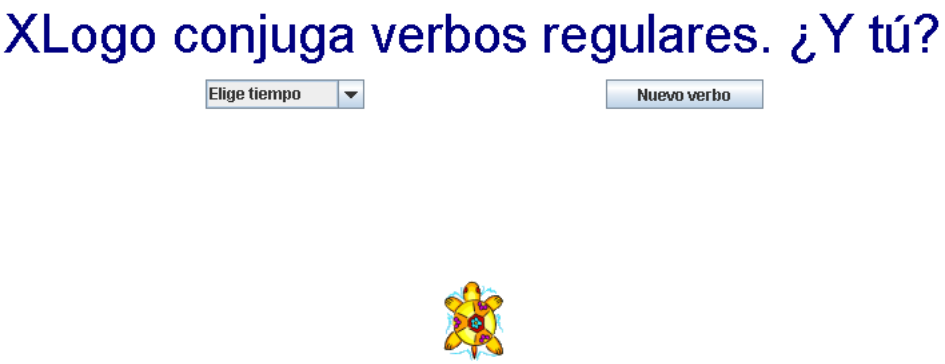
   b) Conjugue correctamente en presente, pasado y futuro simple (sólo verbos regulares)

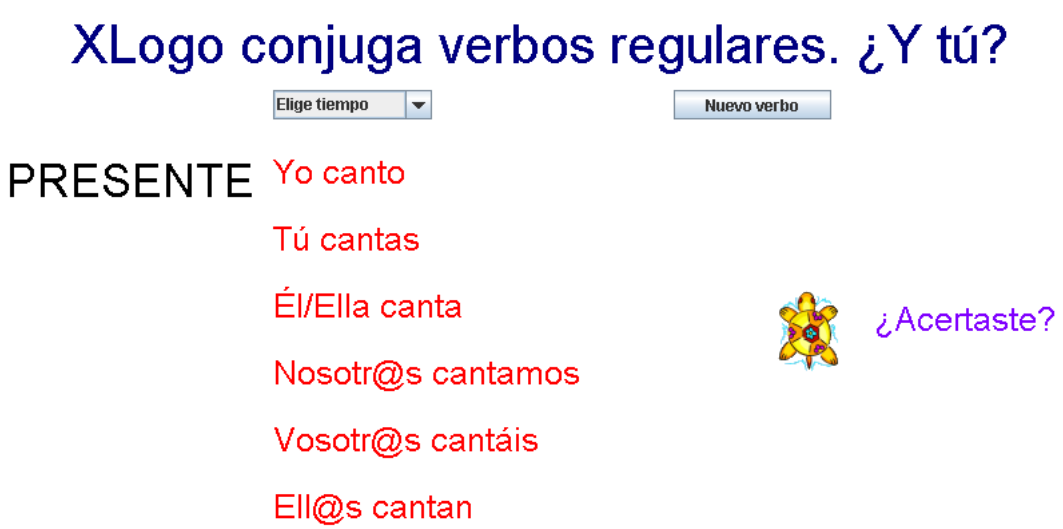
   c) El tiempo verbal se elija con un menú contextual

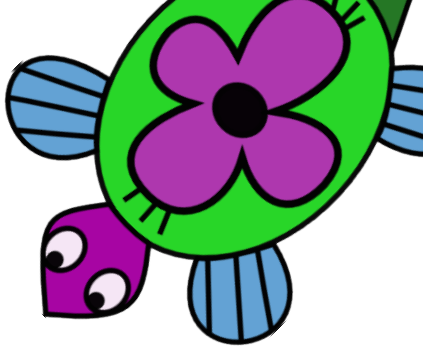
   d) Rotule en pantalla el tiempo verbal elegido

   e) Rotule `¿Acertaste?` tras mostrar el timepo verbal

   f) Se pueda cambiar el verbo con un botón





# Capítulo 13

# Técnicas avanzadas de dibujo

## 13.1. Más opciones para la tortuga

En la sección 4.4 presentamos algunas primitivas que controlan las opciones de la tortuga. Ampliemos las opciones:

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| `muestratortuga`, `mt` | `no` | Hace que la tortuga se vea en pantalla. |
| `ocultatortuga`, `ot` | `no` | Hace invisible a la tortuga. |
| `pongrosor` | `número` | Define el grosor del trazo del lápiz (en pixels). Por defecto es 1. |
| `ponformalapiz`, `pfl` | `0` ó `1` | Fija la forma del lápiz: `pfl 0`: cuadrada; `pfl 1`: ovalada. Por defecto la forma es cuadrada. |
| `grosorlapiz`, `gl` | `no` | Devuelve el grosor del lápiz. |
| `formalapiz`, `fl` | `no` | Devuelve la forma del lápiz. |

Entre otros, `ocultatortuga` es interesante en dos casos:

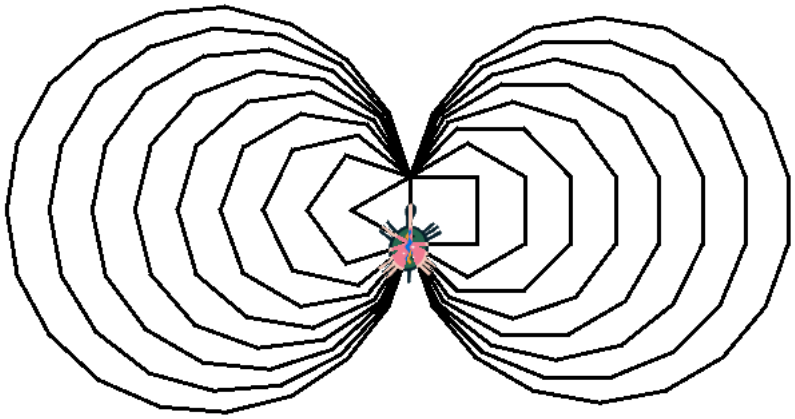
- No tapar parte del dibujo
- Dibujar más rápido

Analiza el procedimiento siguiente:

```
para poli.poligonos :cuantos
 repite :cuantos
```

```
    [ hazlocal "angulo 360/(suma cuentarepite 2)
      si (0 = resto cuentarepite 2)
        [ haz "angulo (-:angulo)]
      repite (suma cuentarepite 2)
        [ avanza 50 giraderecha :angulo ] ]
fin
```

respuesta a un problema planteado en el capítulo 11.4:



Prueba a ejecutarlo de dos formas distintas:

```
borrapantalla poli.poligonos 100

borrapantalla ocultatortuga poli.poligonos 100
```

y observa cómo la segunda vez el dibujo se consigue mucho antes. (En el capítulo 18 veremos como medir exactamente el tiempo de ejecución de un programa).

Respecto a las demás primitivas, vamos a estudiar la siguiente hipótesis:

"*Si pongo un grosor n y avanza n pasos, dibujaré un cuadrado*"

Teclea las órdenes adecuadas y razona si la hipótesis es cierta o no.

Compara el resultado con el obtenido con la siguiente secuencia de comandos:

```
borrapantalla ponformalapiz 1 pongrosor 200 punto posicion

borrapantalla ponformalapiz 0 pongrosor 200 punto posicion
```

¿Qué observas? ¿Influye en el resultado del análisis anterior?

**Prueba** distintos dibujos modificando simultáneamente el grosor y la forma, observando claramente las diferencias entre un lápiz cuadrado y ovalado, cúando se aprecia mejor esa diferencia, ...

## 13.2. Control del color

### 13.2.1. Primitivas que controlan los colores

Conozcamos ahora las primitivas que controlan el color del trazo, del papel (fondo) y del texto:

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| Cambiar el `color` del `lápiz` | `poncolorlapiz n` | `poncl n` |
| Cambiar el `color` del `papel` | `poncolorpapel n` | `poncp n` |
| `Invertir` el `lápiz` | `inviertelapiz` | `ila` |
| Averiguar el `color` del `lápiz` | `colorlapiz` | `cl` |
| Averiguar el `color` del `papel` | `colorpapel` | `cp` |
| Mirar el `color` de la posición `[X Y]` | `encuentracolor [X Y]` | `ec [X Y]` |
| `Borrar` por donde pasa | `goma` | `go` |
| Para volver a dibujar, debe usarse `poncolorlapiz` | | |

¿Qué es "*invertir el lápiz*"? Como veremos en la sección siguiente y detallaremos en 13.6, cada color en xLogo está codificado usando tres valores: rojo, verde y azul, o `R V A` (RGB en inglés). Deberíamos conocer mínimamente la teoría del color asociada a la luz, que difiere de la que conocemos al tratar con pinturas:
`http://w3.cnice.mec.es/eos/MaterialesEducativos/mem2000/color/Intro/indice.htm`

Si trabajo sobre una paleta de pintor, sabemos que los colores básicos son tres, rojo, amarillo y azul; pero si son "colores de luz" los colores básicos cambian, y pasan a ser rojo, azul y verde y que si mezclo los colores obtengo:

| Mezcla | Paleta | Luz |
|---|---|---|
| rojo + azul = | violeta | magenta |
| rojo + amarillo = | naranja | amarillo |
| azul + amarillo = | verde | cyan |
| azul + rojo + amarillo = | negro | blanco |

Utiliza las opciones del menú para VER (literalmente) lo que ocurre al sumar colores en xLogo. Menú Herramientes → Elegir color de lápiz (o de fondo) → pestaña `RVA` y "juega" con las barras de desplazamiento para comprobar lo que acabamos de contar.

Pues bien, al "invertir el lápiz":

- si la zona del papel por la que pasa está en blanco, pinta del color activo
- si la zona del papel tiene un color traza, no la suma, sino la **resta** de colores:

  rojo – azul = verde  rojo – verde = azul  azul – verde = rojo

Respecto a la primitiva `encuentracolor`, devuelve la lista `[R V A]` asociada al color, y que detallamos a continuación.

### 13.2.2. Descripción de los colores

El color en XLOGO está especificado por una lista de tres números `[r v a]` comprendidos entre 0 y 255. El número `r` es el componente rojo, `v` el verde y `a` el azul (`[r g b]` en inglés). XLOGO tiene 17 colores predefinidos, que pueden ser indicados con su lista `[r v a]`, con un número o con una primitiva. Las primitivas correspondientes son:

| Número | Primitiva | [R V A] | Color |
|---|---|---|---|
| 0 | `negro` | `[0 0 0]` | |
| 1 | `rojo` | `[255 0 0]` | |
| 2 | `verde` | `[0 255 0]` | |
| 3 | `amarillo` | `[255 255 0]` | |
| 4 | `azul` | `[0 0 255]` | |
| 5 | `magenta` | `[255 0 255]` | |
| 6 | `cyan` | `[0 255 255]` | |
| 7 | `blanco` | `[255 255 255]` | |
| 8 | `gris` | `[128 128 128]` | |
| 9 | `grisclaro` | `[192 192 192]` | |
| 10 | `rojooscuro` | `[128 0 0]` | |
| 11 | `verdeoscuro` | `[0 128 0]` | |
| 12 | `azuloscuro` | `[0 0 128]` | |
| 13 | `naranja` | `[255 200 0]` | |
| 14 | `rosa` | `[255 175 175]` | |
| 15 | `violeta` | `[128 0 255]` | |
| 16 | `marron` | `[153 102 0]` | |

**Ejemplo:** Estas tres órdenes son la misma:

```
poncolorlapiz naranja
poncolorlapiz 13
poncolorlapiz [255 200 0]
```

### 13.2.3. Función avanzada de relleno

Las primitivas `rellena` y `rellenazona` permiten pintar una figura. Se pueden comparar a la función "`rellena`" disponible en la mayoría de los programas de dibujo. Esta funcionalidad se extiende hasta los márgenes del área de dibujo. Hay tres reglas a tener en cuenta para usar correctamente estas primitivas:

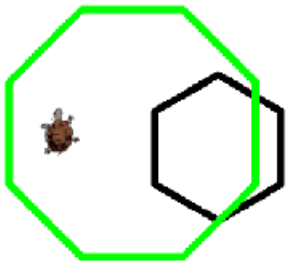
1. El lápiz debe estar bajo (`bl`).

2. La tortuga no debe estar sobre un punto del mismo color que se usará para rellenar. (Si quieres pintar rojo, la tortuga no puede estar sobre un punto rojo).

3. Observar si `cuadricula` está o no activada.

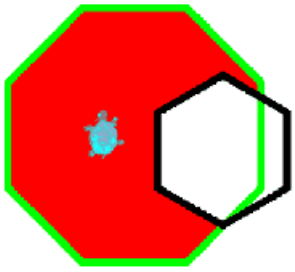
Veamos un ejemplo para explicar la diferencia entre estas dos primitivas:

Los píxeles por donde pasa la tortuga son, en este momento, blancos. La primitiva `rellena` va a colorear todos los píxeles blancos vecinos con el color elegido para el lápiz hasta llegar a una frontera de cualquier color (incluida la cuadrícula):



```
poncolorlapiz 1
rellena
```

produce:

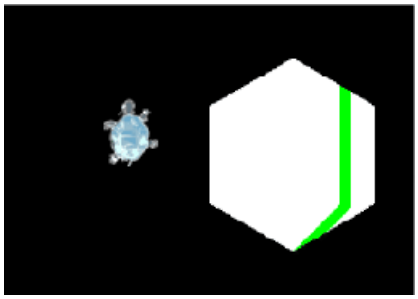


es decir, ha coloreado de rojo la región cerrada en la que se encuentra la tortuga.

Sin embargo, si hacemos:

```
   poncolorlapiz 0
   rellenazona
```

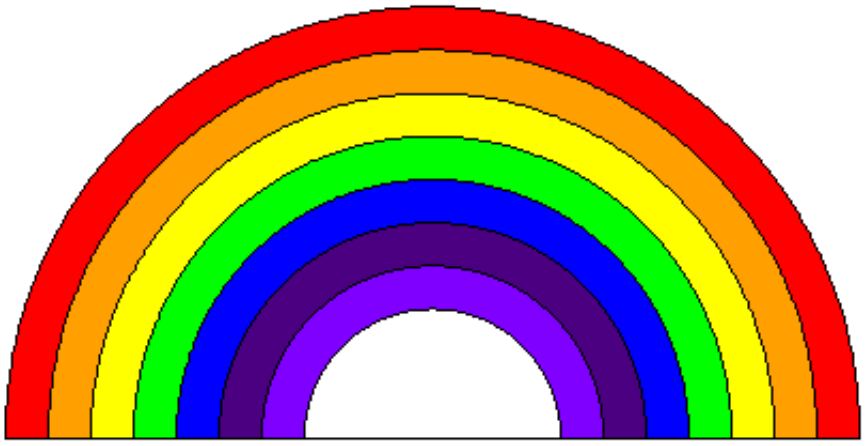
se obtiene:



es decir, rellena todos los píxeles vecinos hasta encontrar una "frontera" del color activo.

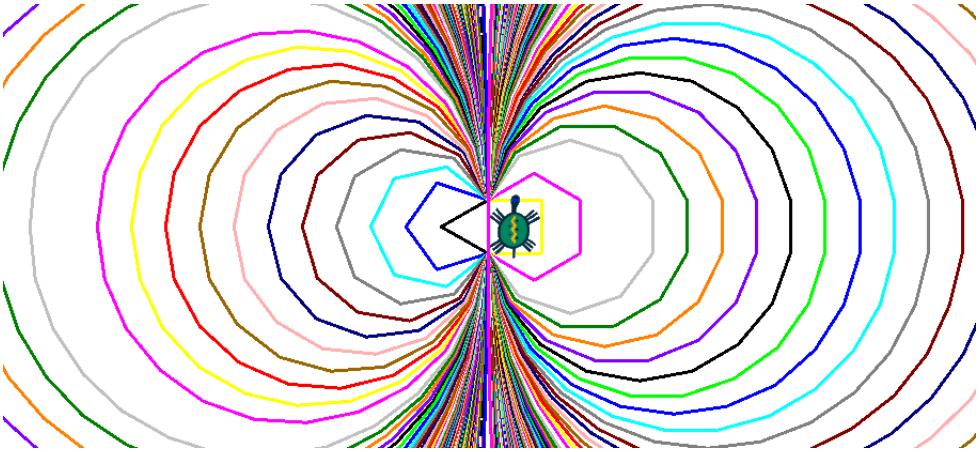
Este es un buen ejemplo para usar la primitiva `rellena`:

```
para mediocirc :c
# dibuja un semicirculo de diametro :c
  repite 180 [
    avanza :c * tan 0.5
    giraderecha 1 ]
  avanza :c * tan 0.5
  giraderecha 90 avanza :c
fin

para arcohueco :c
# Utiliza el procedimiento mediocirc para dibujar un arcoiris sin colores
  si :c < 100 [alto]
  mediocirc :c
  giraderecha 180 avanza 20 giraizquierda 90
  arcohueco :c - 40
fin

para arcoiris
 borrapantalla ocultatortuga arcohueco 400
 subelapiz giraderecha 90 retrocede 150
 giraizquierda 90 avanza 20 bajalapiz
 haz "color [ [255 0 0] [255 160 0] [255 255 0] [0 255 0] [0 0 255]
        [75 0 130] [128 0 255] ]
 repitepara [colores 1 7]
  [ poncolorlapiz elemento :colores :color rellena
    subelapiz giraderecha 90 avanza 20 giraizquierda 90 bajalapiz ]
fin
```

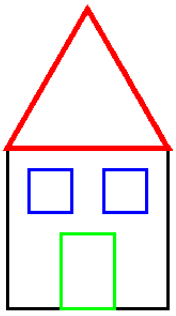


## 13.3. Ejercicios

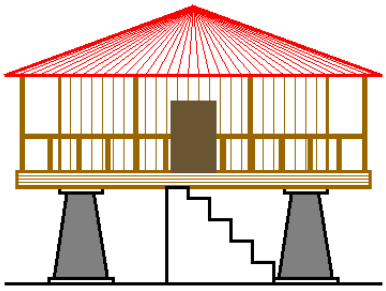
1. Modifica el procedimiento `poli.poligono` para obtener:



2. Combina adecuadamente los procedimientos `cuadrado`, `rectangulo` y `triangulo` que hemos ido diseñando a lo largo del libro para obtener la casa coloreada con la que presentamos a la tortuga "hace tiempo":



3. ¿Te atreves con este hórreo asturiano con corredor?



4. En el capítulo anterior hablamos de usar un mapa mudo en el que el alumno tuviera que ir haciendo "*clic*" en un área concreta según se le fueran mostrando nombres de la misma.

   a) ¿Cómo harías para tener en cuenta las fronteras reales?

   b) ¿Qué requisitos debería tener entonces el mapa para poder hacerlo?

## 13.4. Control del Área de dibujo

En esta sección podemos distinguir dos tipos de primitivas, las que controlan el tamaño del área de dibujo y las que determinan aspectos del dibujo:

### 13.4.1. Control del dibujo

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| `ponzoom` o `ponlupa` | `número` | Acerca o aleja el Área de dibujo. En concreto, el valor de `n` es el factor de escala respecto a la imagen original: `(n >1)` acerca el Área de dibujo; `(0 <n <1)` aleja el Área de dibujo. |
| `zoom` o `lupa` | `no` | Devuelve el valor del escalado anterior. |
| `modoventana` | `no` | La tortuga puede salir del área de dibujo (pero no dibujará nada). |
| `modovuelta` | `no` | Si la tortuga sale del área de dibujo, vuelve a aparecer en el lado opuesto |
| `modojaula` | `no` | La tortuga queda confinada al área de dibujo. Si intenta salir, aparecerá un mensaje de error avisando cuántos pasos faltan para el punto de salida. |
| `poncalidaddibujo`, `pcd` | `0, 1 ó 2` | Fija la calidad del dibujo: `pcd 0`: normal; `pcd 1`: alta; `pcd 2`: baja; |

| | | |
|---|---|---|
| `calidaddibujo`, `cdib` | `no` | Devuelve la calidad del dibujo |

Ajusta el tamaño de la ventana (**Herramientas** → **Preferencias** → **Opciones** → **Tamaño de la ventana**) a 500 por 300, copia el siguiente procedimiento:

```
para modos
  borrapantalla subelapiz ponx -170 bajalapiz
  repite 360 [avanza 3 giraderecha 1]
fin
```

y determina qué pasará en cada uno de los tres modos: `ventana`, `vuelta` y `jaula`.

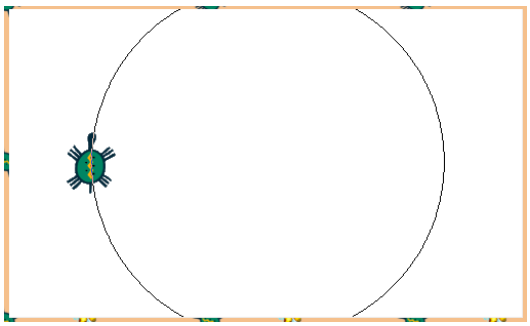
¿Has podido? Este es el resultado:

Obviamente, en `modovuelta` no se ha completado el dibujo y aparece el mensaje de error:

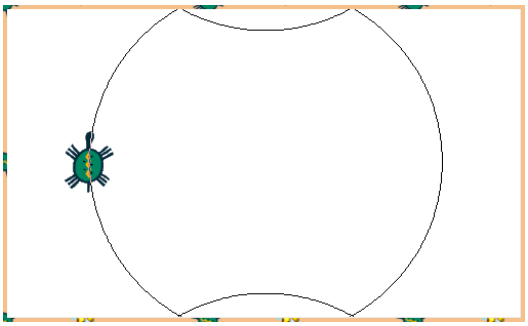
```
En modos, línea 2:
La tortuga sale de la pantalla.
Número de pasos antes de salir:0
```

mientras que para `modovuelta` y `modoventana` nos encontramos con:

Modo ventana

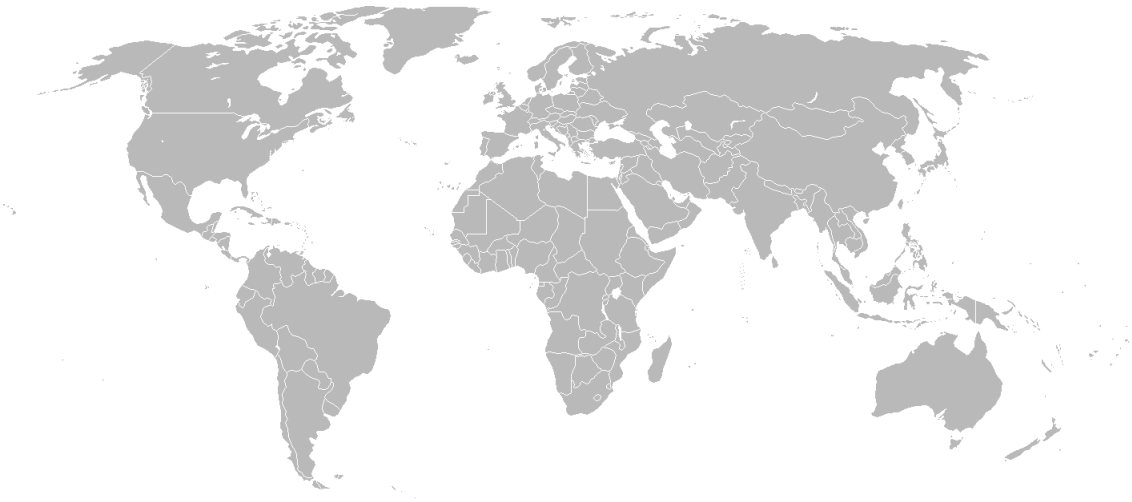


Modo vuelta



pero, cuidado, NO es que la tortuga haya rebotado en la "pared" superior, sino que vuelve a aparecer por la parte de abajo.

**Prueba** a cambiar el número de pasos que avanza la tortuga o ralentízala, para ver claramente cómo se comporta en el modo vuelta. ¿Se te ocurre para qué podríamos usar este modo? Una pista:



(Recuerda la primitiva `cargaimagen`)

### 13.4.2. Control de las dimensiones

| Primitiva | Forma larga | Forma corta |
|---|---|---|
| `pontamañopantalla ptp` | `lista` | Fija el tamaño de la pantalla. |
| `tamañopantalla, tpant` | `no` | Devuelve una lista que contiene el tamaño de la pantalla |
| `tamañoventana, tv, esquinasventana` | `no` | Devuelve una lista con cuatro elementos, las coordenadas de la esquina superior izquierda y de la esquina inferior derecha. |
| `ponseparacion, ponsep` | `número comprendido entre 0 y 1` | Determina la proporción de pantalla ocupada por el **Área de Dibujo** y el **Histórico de Comandos**. |
| `separacion` | `no` | Devuelve el valor de la proporción de pantalla ocupada por el **Área de Dibujo** y el **Histórico de Comandos**. |

El control de la separación también puede hacerse con el ratón, simplemente arrastrando la separación entre el Área de dibujo y el Histórico de comandos hacia arriba o hacia abajo. En `ponseparacion`, si `n` vale 1, el **Área de Dibujo** ocupará toda la pantalla. Si `n` vale 0, será el **Histórico** quien la ocupe.

## 13.5. Ejercicios

1. Vamos a calcular el valor de $\pi$. Para ello, usa el procedimiento que dibuja una circunferencia (o mejor dicho un Trihectahexacontágono)[1] y:

   a) Permite que el avance sea una variable (p.e. `:lado`)

   b) Haz que la tortuga se desplace a través del diámetro hasta alcanzar el punto diametralmente opuesto al de partida, que localizará con `encuentracolor`

   c) Determine la distancia hasta el punto de partida

   d) Muestre nuestra estimación de $\pi$, el resultado de dividir:

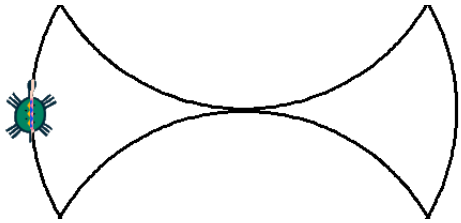
   $$\pi \simeq \frac{360 \times \texttt{:lado}}{\texttt{distancia}}$$

   Prueba para distintos valores de `:lado` y observa los resultados

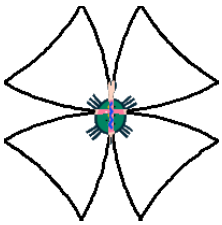
2. En el ejemplo sobre `modojaula`, `modovuelta` y `modoventana`, ¿qué ocurre si usamos `circulo` para dibujar la circunferencia en vez del procedimiento anterior?

3. Observa el dibujo que obtuvimos con `modovuelta`. ¿A qué te recuerda? ¿Podrías usarlo para dibujar, por ejemplo, un murciélago?

4. ¿Cómo harías para que la circunferencia se divida y sea tangente **a sí misma**?

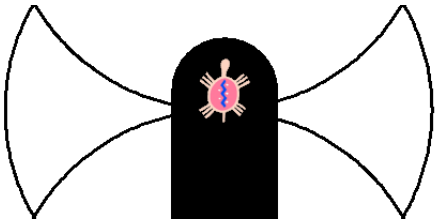


   ¿Y si hacemos que la pantalla sea cuadrada?:

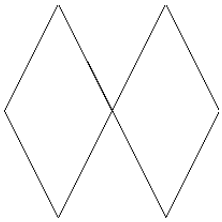


5. Aprovecha ese dibujo para que nuestra tortuga "hable" a través de dos altavoces:

[1] Trihectahexacontágono: tri = 3, hecta = 100, hexaconta = 60, gono = ángulo



6. Intenta dibujar estos dos rombos:



(El tamaño de la pantalla es `200 * 200`)

7. ¿Podrías dibujar un rectángulo (o un cuadrado) **sin que haya giros de 90°**?
(No vale sumar ángulos hasta que den 90°)

8. Dibuja los ejes cartesianos en pantalla

**CUIDADO:** Si pretendes usar estos dibujos como parte de otros redimensionando la pantalla a mitad de un dibujo, NO LO HAGAS. Al igual que `cuadricula`, `ejes`, ... xLogo borra la pantalla al modificar sus dimensiones. Debes guardarlo como imagen y cargarla depués.

## 13.6. Manejando imágenes

### 13.6.1. Introducción

Primero, algunas aclaraciones: Habrás visto en la sección 13.2.2 que el comando `poncolorlapiz` puede tomar como argumento tanto un número como una lista. Aquí nos centraremos en codificar valores RVA. Cada color en xLogo está codificado usando tres valores: rojo, verde y azul, de ahí RVA (RGB en inglés).

Estos tres números conforman una lista que es argumento de la primitiva `poncl`, por lo que representan respectivamente los componentes rojo, verde y azul de un color. Esta manera de codificar no es muy intuitiva, así que para tener una idea del color que obtendrás puedes usar la caja de diálogo **Herramientas → Elegir color del lápiz**.

Sin embargo, usando esta forma de codificar colores, se hace muy fácil transformar una imagen. Por ejemplo, si quieres convertir una foto color en escala de grises, puedes cambiar

cada punto (píxel) de la imagen a un valor promedio de los 3 componentes RVA. Imagina que el color de un punto de la imagen está dado por `[0 100 80]`. Calculamos el promedio: `(0 + 100 + 80)/3 = 60`, y asignamos el color `[60 60 60]` a este punto. Esta operación debe ser realizada para cada punto de la imagen.

### 13.6.2. Práctica: Escala de grises

Vamos a transformar una imagen color de 100 por 100 a escala de grises. Esto significa que tenemos `100 * 100 = 10000` puntos a modificar.

La imagen de ejemplo utilizada aquí está disponible en la siguiente dirección:

```
http://xlogo.tuxfamily.org/images/transfo.png
```

Utilizaremos la primitiva `cargaimagen` o `ci` que carga el archivo de imagen indicado con una palabra.

La esquina superior izquierda de dicha imagen se ubica en la posición actual de la tortuga. Los únicos formatos soportados son jpg y png. La ruta debe especificarse previamente con `pondirectorio` (capítulo 15) y debe ser absoluta, empezando en el nivel superior del árbol de directorios.

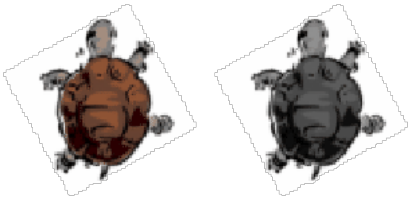
**Ejemplo**:

```
pondirectorio [/home/alumnos/mis\ imagenes]
cargaimagen "turtle.jpg
```

Así es como vamos a proceder: primero, nos referiremos al punto superior izquierdo como `[0 0]`. Luego, la tortuga examinará los primeros 100 puntos (pixeles) de la primera línea, seguidos por los primeros 100 de la segunda, y así sucesivamente. Cada vez tomaremos el color del punto usando `encuentracolor`, y el color será cambiado por el promedio de los tres `[r v a]` valores. Aquí está el código principal: (¡No olvides cambiar la ruta del archivo en el procedimiento!)

```
para transform
# Debes cambiar la ruta de la imagen transfo.png
# Ej: cargaimagen [/home/usuario/imagenes/transfo.png]
  borrapantalla ocultatortuga
  pondirectorio "/home/usuario/imagenes
  cargaimagen "transfo.png
  escalagris
fin

para escalagris
  repitepara [y 0 -100 -1]
```

```
    [ repitepara [x 0 100]
# asignamos el promedio de color del punto al color del lapiz
      [ poncolorlapiz pixel encuentracolor lista :x :y
# convertimos el punto escala de grises
        punto lista :x :y ] ]
fin

para pixel :lista1
# devuelve el promedio de los 3 numeros [r v a]
  haz "r primero :lista1
  haz "lista1 menosprimero :lista1
  haz "v primero :lista1
  haz "lista1 menosprimero :lista1
  haz "a primero :lista1
  haz "color redondea (:r+:v+:a)/3
  devuelve frase :color frase :color :color
fin
```

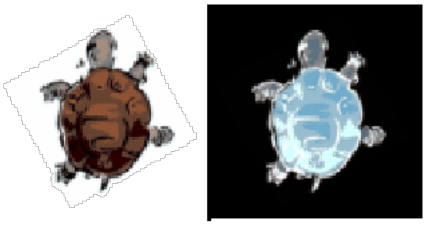



### 13.6.3. Negativo

Para cambiar una imagen a su negativo, se puede usar el mismo proceso de la escala de grises, excepto que en lugar de hacer el promedio de los números `[r v a]`, los reemplazamos por su complemento, o sea la diferencia a 255.
**Ejemplo**: Si un punto (píxel) tiene un color `[2 100 200]`, lo reemplazamos con `[253 155 55]`. Podríamos usar el mismo código que en el ejemplo anterior, cambiando únicamente el procedimiento `pixel`, pero veamos un procedimiento recursivo:

```
para transform2
# Debes cambiar la ruta de la imagen transfo.png
# Ej: c:\Mis Documentos\Mis imagenes\transfo.png
  borrapantalla
  ocultatortuga
  pondirectorio "c:\\Mis\ Documentos\\Mis\ imagenes
  cargaimagen "transfo.png
  negativo 0 0
```

```
fin

para negativo :x :y
  si :y = -100
   [ alto ]
   [ si :x = 100
     [ haz "x 0 haz "y :y-1]
     [ poncolorlapiz pixel2 encuentracolor lista :x :y
       punto lista :x :y ] ]
  negativo :x+1 :y
fin

para pixel2 :lista1
# devuelve el promedio de los 3 numeros [r v a]
  haz "r primero :lista1
  haz "lista1 menosprimero :lista1
  haz "v primero :lista1
  haz "lista1 menosprimero :lista1
  haz "a primero :lista1
  devuelve frase (255 - :r) frase (255 - :v) (255 - :a)
fin
```




## 13.7. Ejercicios

1. Carga la imagen de un laberinto (o mejor aún, que la tortua dibuje uno) y puedas guiar a la tortuga por el camino de salida. Utiliza la primitiva `encuentracolor` para evitar que "atraviese" las paredes del mismo.

   Intenta que la tortuga dibuje un rastro (recuerda, que debes dibujar el rastro después de encontrar el color del papel) de un color si el camino es el correcto y de otro cuando debes retroceder.

2. Busca o dibuja una casa para nuestra tortuga (no muy grande). Busca también imágenes de casas, parques, bancos, edificios gubernamentales y comerciales, . . . , y:

a) Redimensiónalos hasta un tamaño de unos 50 píxeles.

b) Carga esas imágenes formando una "ciudad"

c) Diseña "recorridos", con "tareas" que debe realizar la tortuga, y controla que esas tareas se realizan sin desplazarse por fuera de las "calles".

d) Puedes dibujar puntos de distintos colores en las entradas de cada lugar que se debe visitar para controlar que los sitios a los que se desplaza son los correctos.

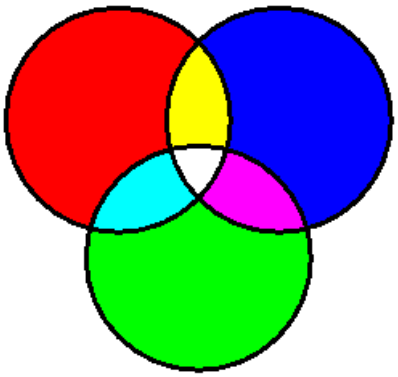
3. Vamos a dibujar un "Diagrama de Venn" de colores. Plantea un procedimiento que:

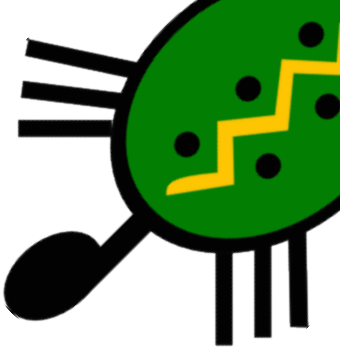
   a) Dibuje dos círculos de radios variables y los rellene con distintos colores

   b) Analice si se cortan (dos circunferencias se cortan si la distancia entre sus centros es menor que la suma de los radios) y, en caso afirmativo:

      1) encuentre el color de cada círculo
      2) sitúe a la tortuga en la intersección de ambas circunferencias (por ejemplo, el punto medio entre sus centros)
      3) rellene esa intersección con el color resultante de sumar o promediar (tú eliges) ambos

   ¿Sabrías ampliarlo a tres circunferencias?





# Capítulo 14

# Modo multitortuga y Animación

## 14.1. Multitortuga

Se pueden tener varias tortugas activas en pantalla. Nada más iniciarse xLogo, sólo hay una tortuga disponible y su número es 0.

### 14.1.1. Las primitivas

Estas son las primitivas que se aplican al modo multitortuga:

| Primitiva | Argumentos | Uso |
|---|---|---|
| `ponforma`, `pforma` | `número` | Puedes elegir tu tortuga preferida en la segunda etiqueta del menú **Herramientas** → **Preferencias**, pero también es posible con `ponforma`. El número `n` puede ir de 0 a 6. (0 es la forma triangular del Logo tradicional). |
| `forma` | `no` | Devuelve un número que representa la forma actual de la tortuga. |
| `pontortuga`, `ptortuga` | `número` | La tortuga número `n` es ahora la tortuga activa. Por defecto, cuando xLogo comienza, está activa la tortuga número 0. |
| `tortuga` | `no` | Da el número de la tortuga activa. |
| `tortugas` | `no` | Da una lista que contiene todos los números de tortuga actualmente en pantalla. |
| `eliminatortuga` | `número` | Elimina la tortuga número `n` |
| `ponmaximastortugas`, `pmt` | `número` | Fija el máximo número de tortugas |
| `maximastortugas`, `maxt` | `no` | Devuelve el máximo número de tortugas |

Si quieres "crear" una nueva tortuga, puedes usar la primitiva `pontortuga` seguida del número de la nueva tortuga. Para evitar confusiones, la nueva tortuga se crea en el centro y es invisible (tienes que usar `muestratortuga` para verla). Así, la nueva tortuga es la activa, y será la que obedezca las clásicas primitivas mientras no cambies a otra tortuga con `pontortuga`.

El máximo número de tortugas disponibles también puede fijarse en el menú **Herramientas → Preferencias**.

### 14.1.2. Ejemplo. Curva de persecución

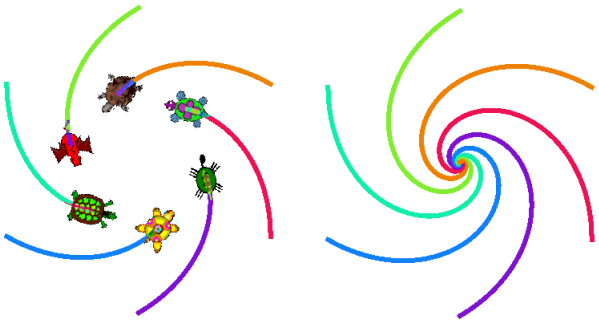
En este ejemplo vamos reproducir la **curva de persecución**. Vamos a distribuir `n` tortugas en los vértices de un polígono regular, y haremos que cada una se dirija hacia la posición de la tortuga situada a su derecha.

Como queremos ver a todas las tortugas en movimiento, vamos a hacer *un poco de trampa* y utilizar la primitiva `animacion` antes de explicarla. El motivo: al moverse las tortugas, la velocidad de refresco de la imagen no alcanza la velocidad de movimiento de las tortugas, y se genera un parpadeo bastante molesto:

```
para empieza :n
  borrapantalla ocultatortuga subelapiz
  animacion                              # Cierto, todavia no la explicamos
  inicio :n
  mientras [(distancia [0 0]) > 2 ]           # Funciona hasta que "chocan"
    [ repite :n
        [ pontortuga cuentarepite
          haz "mipos pos                      # Miramos donde esta la tortuga n
          pontortuga cuentarepite+1
          si cuentarepite+1>:n [pontortuga 1]
          ponrumbo hacia :mipos               # Orientamos la tortuga n +1
          avanza 2]
      refresca ]                              # Hacemos visibles los trazos
  detieneanimacion
  repite :n [
    pontortuga cuentarepite
    ocultatortuga ]          # Ocultamos todas las tortugas en la imagen final
fin

para inicio :n
  si :n <2 [escribe [Necesitas mas de una tortuga!] alto] # Control de error
  repite :n
```

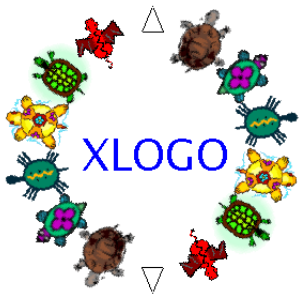
```
    [ pontortuga cuentarepite                          # Activamos tortuga
      ponforma cuentarepite                            # Cambiamos el tipo de tortuga
      muestratortuga                                   # Mostramos tortuga
      pongrosor 6
      haz "angulo cuentarepite*360/:n
      poncolorlapiz angcol :angulo subelapiz     # El color depende de la ubicacion
      ponxy (190*sen :angulo) (190*cos :angulo) bajalapiz]
fin

para angcol :x
  haz "r 127.5 *(1 + sen (:x))
  haz "g 127.5 *(1 + sen (:x + 120))
  haz "b 127.5 *(1 + sen (:x + 220))
  devuelve frase lista :r :g :b
fin
```

Si ejecutamos `empieza 6`, iremos viendo cómo las seis tortugas (cada una con una forma) van acercándose entre ellas, hasta juntarse en el centro. En ese momento, desparecen todas (la imagen de todas ellas superpuestas no es muy... *elegante*).

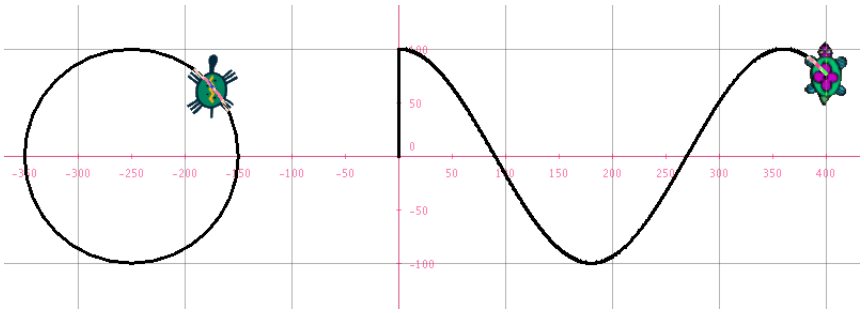


## 14.2. Ejercicios

1. Plantea un procedimiento que dibuje la "rosca" que aparece en la portada:



2. **Este problema implica conocimientos de Física**: Plantea un procedimiento que represente la muchas veces explicada analogía entre movimiento circular uniforme y movimiento armónico simple:



Para ello, necesitaremos tres tortugas:

a) Una que describa el movimiento circular

b) Otra que represente el paso del tiempo

c) Una tercera que describa el movimiento armónico

La primera tortuga irá trazando una circunferencia a la par que la segunda avanza un pequeño número de pasos (debe ajustarse para que se vea bien el movimiento) y la tercera se desplaza de modo que:

- Su ordenada coincida con la de la tortuga 1
- Su abscisa coincida con la de la tortuga 2

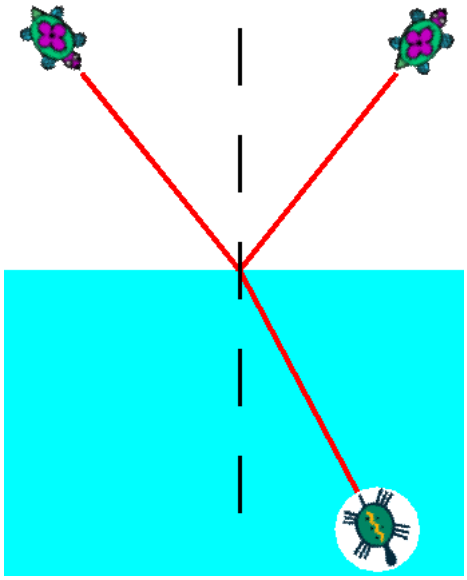
3. **Este problema implica conocimientos de Física**: Plantea un procedimiento que simule la reflexión y refracción de la luz. Para ello, necesitaremos dos tortugas, una para cada medio.

   Debemos trazar una línea que represente la separación de medios, pudiendo colorear al menos uno para distinguirlos bien. Una tortuga se desplazará desde un punto (que puede dejarse como argumento) hasta el origen (orientada correctamente con `ponrumbo`) y allí

   - Ella se "reflejará", invirtiendo su desplazamiento vertical
   - Apareceá la segunda tortuga que modificará su desplazamiento horizontal

   en ambos casos, siguiendo las leyes de Snell:

$$\begin{aligned} \text{Reflexión:} \quad & \alpha_{\text{inc}} = \alpha_{\text{refl}} \\ \text{Refracción:} \quad & n_{\text{inc}} \cdot \operatorname{sen} \alpha_{\text{inc}} = n_{\text{refr}} \cdot \operatorname{sen} \alpha_{\text{refr}} \end{aligned}$$



Puede "mejorarse" cambiando la forma de la tortuga para distinguir los dos "haces"

4. **Este problema implica conocimientos de Física**: Plantea un procedimiento que simule las fuerzas electrostáticas y gravitatorias de un sistema de masas o de cargas. Para ello:

   a) Ubicará $n$ tortugas en posiciones que se indicarán como argumento

   b) Ubique otra tortuga con el ratón, con una "forma" distinta

   c) Determine vectorialmente la fuerza total en ese punto, y se desplace 3 ó 4 pasos en esa dirección

   $$\vec{F} = K \cdot \frac{Q \cdot q}{r^2} \vec{u}_r$$

   d) Se detenga cuando la distancia a una de las tortugas sea, por ejemplo, 10

| | |
|---|---|
|  | Estás comprobando cómo no es "demasiado" complicado realizar simulaciones de Física con xLogo. Por supuesto, con esto entramos en temas que requieren conocimientos "avanzados" de Ciencias, pero podemos plantearnos usarlos en lugar de buscar software específico para cada tema. Como ves, las posibilidades son "infinitas". |

## 14.3. Aplicación didáctica: lanzamiento de dos dados

Cuando se lanzan dos dados y se calcula la suma de los puntos de cada uno de ellos, se obtiene un resultado comprendido entre 2 y 12. En esta actividad vamos ver la distribución de frecuencias de las distintas tiradas y a representarla en un sencillo gráfico.

### 14.3.1. Simular el lanzamiento de un dado

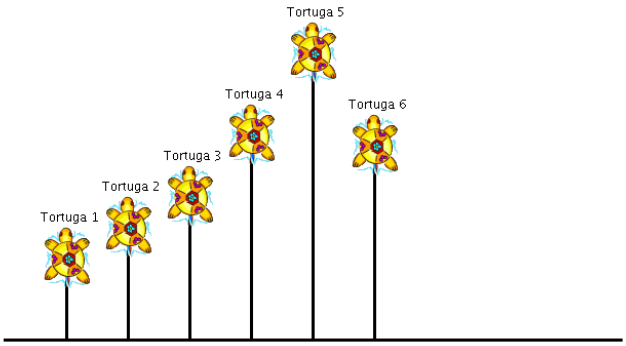
Para simular el lanzamiento de un dado, vamos a utilizar la primitiva `azar` (sección 7.3). `azar 6`, devuelve un número aleatorio comprendido entre 0 y 5. Por tanto, `(azar 6) + 1` devuelve una cantidad elegida aleatoriamente del conjunto `{1,2,3,4,5,6}`.

Como ya explicamos (página 65), debemos utilizar paréntesis; si no, xLogo leería `azar 7`. Para evitar los paréntesis, se puede escribir `1 + azar 6`. Se define así el procedimiento `lanzar`, que simula el lanzamiento de un dado.

```
para lanzar
  devuelve 1 + azar 6
fin
```

### 14.3.2. El programa

Vamos a utilizar el modo *multitortuga* que acabamos de explicar, para así disponer de varias tortugas sobre la pantalla. La imagen siguiente nos indica qué queremos conseguir:




El objetivo es que cada tortuga, numerada de 2 a 12, avance un *paso de tortuga* cuando el resultado de la suma de la tirada de los dos dados coincida con su número. Por ejemplo, si la tirada de dados suma 8, la tortuga número 8 avanzará un paso.

La separación horizontal entre las tortugas es de 30 pasos de tortuga, y se colocarán a las tortugas con ayuda de los datos.

- Se colocará a la tortuga número 2 en `(-150 ; 0)`
- Se colocará a la tortuga número 3 en `(-120 ; 0)`
- Se colocará a la tortuga número 4 en `(-90 ; 0)`
- Se colocará a la tortuga número 5 en `(-60 ; 0)`
- ...

es decir:

```
pontortuga 2 ponpos [-150 0]
pontortuga 3 ponpos [-120 0]
pontortuga 4 ponpos [-90 0]
pontortuga 5 ponpos [-60 0]
pontortuga 6 ponpos [-30 0]
```

En lugar de copiar 11 veces prácticamente la misma línea de órdenes, usaremos `repitepara`, con la variable `:i` tomando los valores `2, 3, 4, ..., 12`.
Para colocar a las tortugas, creamos el procedimiento `inicia`

```
para inicia
  borrapantalla
  ocultatortuga
  repitepara [i 2 12]
   [ # coloca la tortuga
    pontortuga :i ponpos lista -150 + (:i - 2) * 30 0
     # escribe el numero de la tortuga justo debajo
    subelapiz retrocede 15
     rotula :i
    avanza 15 bajalapiz ]
fin
```

Observa la expresión `-150 + (:i - 2) * 30`. Con ello hacemos que el primer valor para la abscisa sea `-150`, y a cada nueva tortuga se añaden 30 (probar con distintos valores de `:i` si no se ve bien).

Finalmente, se obtiene el siguiente programa:

```
para lanzar
  devuelve 1 + azar 6
fin

para inicia
  borrapantalla
  ocultatortuga
  repitepara [i 2 12]
    [ # coloca la tortuga
     pontortuga :i ponpos lista -150 + (:i - 2)*30 0
      # escribe el numero de la tortuga justo debajo
     subelapiz retrocede 15
       rotula :i
     avanza 15 bajalapiz ]
fin
```

```
para empezar
  inicia
# Hacemos 1000 intentos
  repite 1000
    [ haz "suma lanzar+lanzar
      pontortuga :suma avanza 1 ]
 # indicamos las frecuencias de tirada
  repitepara [i 2 12]
    [ pontortuga :i
  # la ordenada de la tortuga representa el numero de tiradas
      hazlocal "frecuencia ultimo pos
      subelapiz avanza 10 giraizquierda 90
        avanza 10 giraderecha 90 bajalapiz
      rotula :frecuencia/1000*100 ]
fin
```

Veamos ahora una generalización de este programa. Aquí, se pedirán al usuario el número de dados deseados así como el número de lanzamientos a efectuar.

```
para lanzar :dados
  hazlocal "suma 0
  repite :dados
    [ hazlocal "suma :suma + 1 + azar 6 ]
  devuelve :suma
fin

para inicia
  borrapantalla ocultatortuga
  ponmaximastortugas :max + 1
  repitepara frase lista "i :min :max
   [ # coloca la tortuga
     pontortuga :i
     ponpos lista (:min - :max)/2*30 + (:i - :min)*30 0
     # escribe el numero de la tortuga justo debajo
     subelapiz retrocede 15
       rotula :i
     avanza 15 bajalapiz ]
fin

para empezar
  leeteclado [Numero de dados:] "dados
  si no numero? :dados
   [ es [largoetiqueta No es un numero!]
```

```
     alto ]
  haz "min :dados
  haz "max 6*:dados
  leeteclado [Numero de lanzamientos a realizar] "tiradas
  si no numero? :tiradas
   [ es [largoetiqueta El numero introducido no es valido!]
     alto ]
  inicia
# Debemos ajustar el paso para que no se salga de pantalla
  haz "paso :dados * 500/:tiradas
# Hacemos un numero de intentos igual a :tiradas
  repite :tiradas
    [ pontortuga lanzar :dados avanza :paso ]
# indicamos las frecuencias de tirada
  repitepara frase lista "i :min :max
    [ pontortuga :i
  # la ordenada de la tortuga representa el numero de tiradas
      hazlocal "frecuencia ultimo pos
  # normalizamos entre 0,1
      subelapiz avanza 10 giraizquierda 90
        avanza 10 giraderecha 90 bajalapiz
  # en caso de numeros grandes, los decimales son ... terribles
      rotula (redondea 10000*:frecuencia/:tiradas)/100 ]
fin
```

**Piensa cómo podemos utilizar el sistema mostrado en el ejemplo para leer una lista de datos discretos y que las tortugas vayan representando un diagrama de barras con ellos. Pista: Deberás hacer más de una pasada a la lista, para determinar cuántos elementos distintos hay y ordenarlos de menor a mayor antes de releerla para hacer avanzar a las tortugas**

## 14.4. Animación

Existen dos primitivas llamadas `animacion` y `refrescar` que permiten escribir órdenes sin que la tortuga las realice. `animacion` hace que la tortuga dibuje pero no lo muestre, es decir, a nuestros ojos no hace nada; al recibir la orden `refrescar` muestra todo el trabajo almacenado en memoria.

| Primitivas | Uso |
|---|---|
| `animacion` | Se accede al modo de animación. |

| Primitivas | Uso |
|---|---|
| `detieneanimacion` | Detiene el modo animación, retornando al modo *normal*. |
| `refrescar` | En modo de animación, ejecuta las órdenes y actualiza la imagen |

Mientras se escriben las órdenes en el modo de animación (una cámara de cine aparece a la izquierda del **Histórico de Comandos**), éstas no son ejecutadas en el **Área de Dibujo** sino que son almacenadas en memoria hasta que se introduce la orden `refrescar`.

Haciendo *clic* en este icono, se detiene el modo de animación, sin necesidad de usar la primitiva `detieneanimacion`.

Esto es muy útil para crear animaciones o conseguir que los dibujos se realicen rápidamente.

### 14.4.1. Ejemplo

Vamos a conseguir que un "camión" se desplace de izquierda a derecha de la pantalla. Empezamos por dibujar un camión (puedes cambiar el modelo, si este no te gusta) muy sencillo, un cuadrado como remolque, dos ruedas y una cabina simple:

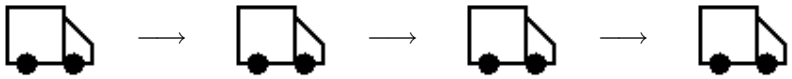
```
para camion
  pongrosor 2 ocultatortuga
  repite 4
     [avanza 30 giraizquierda 90]
  giraderecha 90 avanza 15 giraizquierda 90 avanza 10
  giraizquierda 45 avanza (rc 2)*15 giraizquierda 135 avanza 25
  giraizquierda 90 goma avanza 5 ponlapiz circulo 5 rellena
  retrocede 20 goma retrocede 5 ponlapiz circulo 5 rellena
fin
```

y concluimos con la parte asociada a la animación:

```
para moviendose
  animacion
  repitepara [ lugar -300 +300 2]
```

```
   [ borrapantalla subelapiz ponx :lugar bajalapiz
     coche refrescar ]
  detieneanimacion
fin
```

El efecto final es el del camión desplazándose desde el punto `[-300 0]` hasta el `[300 0]`



Si aún tienes dudas, en la sección 18 mostraremos otra animación, esta vez sobre las cifras de la calculadora (cuenta atrás).

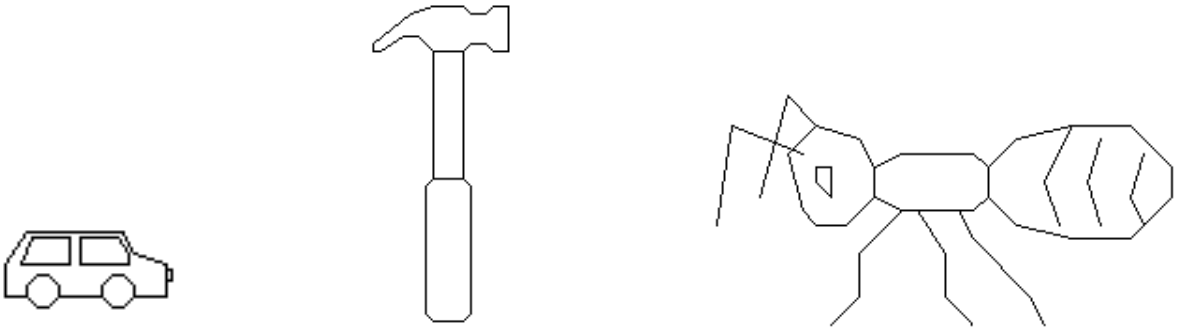
Este tipo de problemas pueden asociarse al estudio de los números complejos. Recordemos que la interpretación geométrica de un complejo permite sistematizar el análisis de:

- traslaciones
- rotaciones
- dilataciones

es decir, las **Homotecias**. En:

`http://neoparaiso.com/logo/numeros-complejos-aplicaciones.html`

muestran tres ejemplos de ello con las figuras de un coche, un martillo y una hormiga:



## 14.5. Ejercicios

Intenta reproducir con xLogo las tres animaciones propuestas:

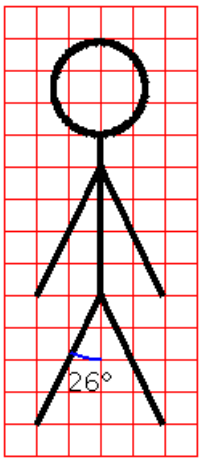
1. Desplazamiento de un coche
2. Giro de un martillo

3. Escalado de una hormiga

**PERO** usando correctamente las operaciones entre complejos.

Para representar los complejos con XLOGO deberás crear una lista con dos elementos, las coordenadas X e Y, y usar correctamente `primero` y `ultimo` para efectuar las operaciones.

## 14.6. El increible *monigote* creciente




En primer lugar, vamos a definir un procedimiento `monigote` que dibujará el monigote representado arriba, con un tamaño de nuestra elección.

```
para monigote :c
   giraizquierda 154 avanza 2.2*:c retrocede :c*2.2
   giraizquierda  52 avanza 2.2*:c retrocede :c*2.2
   giraizquierda 154 avanza 2*:c
   giraizquierda 154 avanza 2.2*:c retrocede :c*2.2
   giraizquierda  52 avanza 2.2*:c retrocede :c*2.2
   giraizquierda 154 avanza :c/2
   giraizquierda  90 repite 180
        [avanza :c/40 giraderecha 2]
   giraderecha 90
fin
```

Vamos ahora con la animación que creará la ilusión de que el monigote crece poco a poco. Para ello, escribimos `monigote 1`, despues `monigote 2` `monigote 3` ... hasta `monigote 75`. Entre cada trazado, se borrará la pantalla. Se obtienen los procedimientos siguientes:

```
para monigote :c
  si :c=75 [alto]
  giraizquierda 154 avanza 2.2*:c retrocede :c*2.2
```

```
  giraizquierda  52 avanza 2.2*:c retrocede :c*2.2
  giraizquierda 154 avanza 2*:c
  giraizquierda 154 avanza 2.2*:c retrocede :c*2.2
  giraizquierda  52 avanza 2.2*:c retrocede :c*2.2
  giraizquierda 154 avanza :c/2
  giraizquierda  90 repite 180
        [avanza :c/40 giraderecha 2]
  giraderecha 90
  borrapantalla ocultatortuga monigote :c+1
fin

para empezar
   borrapantalla ocultatortuga monigote 0
fin
```

Por último, para suavizar todo el proceso, vamos a servirnos del modo `animacion` y de la primitiva `refrescar`.

```
para monigote :c
  giraizquierda 154 avanza 2.2*:c retrocede :c*2.2
  giraizquierda  52 avanza 2.2*:c retrocede :c*2.2
  giraizquierda 154 avanza 2*:c
  giraizquierda 154 avanza 2.2*:c retrocede :c*2.2
  giraizquierda  52 avanza 2.2*:c retrocede :c*2.2
  giraizquierda 154 avanza :c/2
  giraizquierda  90 repite 180
        [avanza :c/40 giraderecha 2]
  giraderecha 90
  refresca
  borrapantalla ocultatortuga monigote :c+1
fin

para empezar
   borrapantalla ocultatortuga animacion
   monigote 0
   detieneanimacion
fin
```

# Capítulo 15

# Manejo de Archivos

## 15.1. Las primitivas

Como siempre, vamos a clasificar las primitivas en función de su objetivo:

### 15.1.1. Navegación por el sistema de archivos

Al iniciar xLogo, la ruta de trabajo será el directorio personal (`/home/nombre/` en Linux y Mac) o el directorio raíz `C:\\` en Windows. Para desplazarnos por el disco duro y ejecutar programas externos, disponemos de:

| Primitivas | Argumentos | Uso |
|---|---|---|
| `catalogo`, `cat` | `no` | Lista el contenido del directorio actual. (Equivalente al comando `ls` de Linux, `dir` de DOS) |
| `pondirectorio`, `pondir` | `lista` | Especifica el directorio actual. La ruta debe ser absoluta. El directorio debe especificarse dentro de una lista, y la ruta no debe contener espacios. |
| `cambiadirectorio`, `cd` | `palabra o lista` | Cambia el directorio de trabajo desde el directorio actual (ruta relativa). Puede utilizarse `..` para referirse a la ruta del directorio superior. |
| `directorio`, `dir` | `no` | Da el directorio actual. |

Supongamos la siguiente estructura de directorios en el disco duro de un alumno:

```
/home/alumno
   --> /Documentos
         --> /Clase
         --> /xLogo
              --> /Programas
```

```
            --> /Capturas
 --> /Escritorio
 --> /Imagenes
```

Al iniciar xLogo, la ruta de trabajo será `/home/alumno/`. Para llegar al directorio `Programas` podemos escribir alternativamente:

```
pondirectorio "/home/alumno/Documentos/xLogo/Programas
cambiadirectorio "Documentos/xLogo/Programas
```

Una vez en él, podemos ir hacia atrás de forma absoluta o relativa:

```
pondirectorio "/home/alumno/Documentos/xLogo
cambiadirectorio ".. 
```

y listar su contenido, que se mostrará en el Histórico de Comandos separando los directorios de los archivos:

```
catalogo
--> Directorio(s):
    Programas Capturas
    Archivo(s):
    manual.pdf prueba.lgo tortuga.png transfo.png xlogo.jar
```

Hay dos posibilidades de que esto no sea exactamente así:

- cuando se abre un archivo `.lgo` desde el Menú Archivo, el directorio donde se encontraba éste queda establecido como directorio de trabajo. Por ejemplo, si abrimos el fichero:

  ```
  /home/alumno/Documentos/xLogo/rosa.lgo
  ```

  el directorio de trabajo será

  ```
  /home/alumno/Documentos/xLogo
  ```

- Si no es la primera vez que se trabaja con xLogo. Al terminar una "sesión", se guarda la última ruta de trabajo y se mantiene para ejecuciones posteriores.

Todo lo anterior afecta al guardado en disco de procedimientos o la carga desde disco duro de procedimientos e imágenes, NO a las capturas de pantalla que pueden hacerse desde el Menú Archivo ⟶ Capturar la imagen.

### 15.1.2. Carga y guardado de procedimientos

Finalizando el capítulo 5 te enseñamos a guardar en el disco duro usando las opciones de Menú:

- Menú Archivo ⟶ Guardar como ....
- Menú Archivo ⟶ Guardar

que difieren en que uno te permite asignar un nombre, y el otro sobreescribe el archivo abierto (si lo hay). Podemos también abrir procedimientos con:

Menú Archivo ⟶ Abrir
y disponemos de primitivas con las que conseguir lo mismo:

| Primitivas | Argumentos | Uso |
|---|---|---|
| `carga` | `palabra` | Abre y lee el archivo indicado por `palabra`. |
| `guarda` | `palabra`, `lista` | Guarda en el archivo indicado por `palabra` los procedimientos especificados en `lista`, en el directorio actual. (Ver ejemplo) |
| `guardatodo` | `palabra` | Guarda en el archivo indicado por la `palabra` todos los procedimientos definidos, en el directorio actual. (Ver ejemplo) |

**Ejemplos**:

- `carga "prueba.lgo` carga un archivo llamado `prueba.lgo` del directorio actual, SIN ABRIR el Editor de Comandos.

  Difiere de la opción `Abrir` del Menú Archivo en que este sí abre el Editor de Comandos.
- `guarda "trabajo.lgo [proc1 proc2 proc3]` guarda en el directorio actual un archivo llamado `trabajo.lgo` que contiene los procedimientos `proc1`, `proc2` y `proc3`.
- `guardatodo "trabajo.lgo` guarda en el directorio actual un archivo llamado `trabajo.lgo` que contiene la totalidad de los procedimientos actualmente definidos.

  Es equivalente a usar la opción `Guardar` del Menú Archivo.

En ambos casos, si no se indica la extensión `.lgo`, será añadida. La palabra especifica una ruta relativa a partir del directorio corriente. No funciona colocar una ruta absoluta.

Para borrar todos los procedimientos definidos y cargar el archivo `trabajo.lgo`, debes usar:

```
borratodo carga "trabajo.lgo
```

La palabra especifica una ruta relativa a partir del directorio corriente. No funciona colocar una ruta absoluta, es decir:

```
borratodo carga "/home/alumno/xLogo/trabajo.lgo
```

no producirá ningún efecto.

### 15.1.3. Modificando archivos

| Primitivas | Argumentos | Uso |
|---|---|---|
| abreflujo | número nombre | Para poder leer o escribir en un fichero, es necesario crear un flujo hacia él. El argumento nombre debe ser su nombre, que se refiere al directorio de trabajo. El argumento n es el número que identifica a ese flujo. |
| cierraflujo | número | Cierra el flujo n. |
| listaflujos | lista | Carga una lista con los flujos abiertos indicando su identificador |
| leelineaflujo | número | Abre el flujo cuyo identificador es n, y lee una línea del fichero |
| leecarflujo | número | Abre el flujo cuyo identificador es n, después lee un caracter del fichero. Esta primitiva devuelve el número correspondiente al caracter unicode (como leecar - sec. 12.2) |
| escribelineaflujo | número lista | Escribe la línea de texto indicada en lista al principio del fichero indicado por el flujo n. Atención: la escritura no se hace efectiva hasta que se ciera el fichero con cierraflujo. |
| agregalineaflujo | número lista | Escribe la línea de texto indicada en lista al final del fichero indicado por el flujo n. Atención: la escritura no se hace efectiva hasta que se ciera el fichero con cierraflujo. |
| finflujo? | número | Devuelve cierto si se ha llegado al final del fichero, y falso en caso contrario. |

**Ejemplo**:

El objetivo es crear el fichero ejemplo en el directorio personal: /home/tu_nombre, en Linux, C:\, en Windows que contiene:

```
ABCDEFGHIJKLMÑNOPQRSTUVWXYZ
abcdefghijklmñnopqrstuvwxyz
0123456789
```

Para ello:

```
# abre un flujo hacia el fichero indicado
# identificara el flujo con el numero 2
pondirectorio "/home/tu_nombre
abreflujo 2 "ejemplo
# escribe las lineas que quiero
escribelineaflujo 2 [ABCDEFGHIJKLMNÑOPQRSTUVWXYZ]
escribelineaflujo 2 [abcdefghijklmnñopqrstuvwxyz]
escribelineaflujo 2 [0123456789]
# cerramos el flujo para acabar la escritura
cierraflujo 2
```

Ahora, comprobamos que está bien escrito:

```
# abre un flujo hacia el fichero indicado
# identificara el flujo con el numero 0
pondirectorio "/home/tu_nombre
abreflujo 0 "ejemplo
# lee las lineas del fichero consecutivamente
escribe leelineaflujo 0
escribe leelineaflujo 0
escribe leelineaflujo 0
# cerramos el flujo
cierraflujo 0
```

Si queremos que nuestro fichero termine con la línea `Formidable!`:

```
pondirectorio "c:\\
abreflujo 1 "ejemplo
agregalineaflujo 1 [Formidable!]
cierraflujo 1
```

## 15.2. Ejecutando programas externos

Si quisiéramos ejecutar un program externo a xLogo, disponemos de la primitiva `comandoexterno`. Su argumento debe ser una lista de sub–listas, que contienen en este orden:

- El comando que lanza el programa
- Opcionalmente, las opciones del mismo

Por ejemplo:

```
comandoexterno [ [gedit] ]
```

Ejecuta el program GEDIT (en Linux) sin opciones.

```
comandoexterno [ [notepad] ]
```

Ejecuta el program BLOCK DE BOTAS (en Windows) sin opciones.

```
comandoexterno [ [gedit] [/home/xlogo/ejemplo.txt] ]
```

Abre el archivo `ejemplo.txt` con GEDIT (en Linux).

```
comandoexterno [ [notepad] [c:\ejemplo.txt]]
```

Abre el archivo `ejemplo.txt` con el BLOCK DE NOTAS (en Windows).

Esta sintaxis tan "especial" permite llamar al sistema con los espacios en blanco adecuados para que no haya errores.

## 15.3. Obtención aproximada de $\pi$ (2)

Un resultado conocido de teoría de los números pone de manifiesto que la probabilidad que dos números tomados aleatoriamente sean primos entre ellos es de $\frac{6}{\pi^2} \simeq 0{,}6079$. Para intentar encontrar este resultado, vamos a:

- Tomar dos números al azar entre 0 y 1 000 000.
- Calcular su m.c.d.
- Si su m.c.d. vale 1, añadir 1 a una variable contador.
- Repetir 1000 veces
- La frecuencia de los pares de números primos entre ellos se obtendrá dividiendo la variable contador por 1000 (el número de pruebas)

### 15.3.1. Noción de m.c.d. (máximo común divisor)

Dados dos números enteros, el máximo común divisor define al mayor divisor común de ambos. Por ejemplo:

- 42 y 28 tienen como m.c.d. 14, ya que es el número más grande por el que es posible dividir a la vez 28 y 42
- el m.c.d. de 25 y 55 es 5
- 42 y 23 tienen m.c.d. igual a 1

Cuando dos números tienen m.c.d. 1, se dice que son primos entre sí. Así en el ejemplo anterior, 42 y 23 son primos entre sí. Eso significa que no tienen ningún divisor común excepto 1 (¡por supuesto, hablamos de división entera!).

### 15.3.2. Algoritmo de Euclides

Para determinar del m.c.d. de dos números, se puede utilizar un método llamado **algoritmo de Euclides**: (Aquí no se demostrará la validez de este algoritmo, se admite que funciona).

El mecanismo de este método es: "dados dos números naturales `a` y `b`, analizamos si `b` es nulo.:

- En caso afirmativo, entonces el m.c.d. es igual a `a`.
- Si no, se calcula `r`, el resto de la división de `a` entre `b`.
- Se sustituyen `a` por `b` y `b` por `r`, y se reinicia el método.

Calculemos por ejemplo, el m.c.d. de 2160 y 888 por este algoritmo con las siguientes etapas:

| a | b | r |
|---|---|---|
| 2160 | 888 | 384 |
| 888 | 384 | 120 |
| 384 | 120 | 24 |
| 120 | 24 | 0 |
| 24 | 0 | |

El m.c.d. de 2160 y 888 es, por tanto, 24. 24 es el mayor entero que divide simultáneamente a los dos números. De hecho, `2160 = 24 * 90` y `888 = 24 * 37`. El m.c.d. es, por tanto, el último resto no nulo.

### 15.3.3. Calcular un m.c.d. en xLogo

Un pequeño algoritmo recursivo permite calcular el m.c.d. de dos números, `:a` y `:b`.

```
para mcd :a :b
   si (resto :a :b) = 0
      [devuelve :b]
      [devuelve mcd :b resto :a :b]
fin

escribe mcd 2160 888
```

proporciona como resultado `24`

Nota: Nos vemos obligados a poner paréntesis en `resto :a :b`, si no el intérprete intentará evaluar `b = 0`. Para evitar este problema de paréntesis, podemos escribir: `si 0 = resto :a :b`

### 15.3.4. Avanzando con el programa

Tras la introducción matemática, recordemos nuestro objetivo:

- Tomar dos números al azar entre 0 y 1 000 000.
- Calcular su m.c.d.
- Si su m.c.d. vale 1, añadir 1 a una variable contador.
- Repetir 1000 veces
- La frecuencia de los pares de números primos entre ellos se obtendrá dividiendo la variable contador por 1000 (el número de pruebas)

```
para prueba
# Inicializamos la variable contador a 0
  haz "contador 0
  repite 1000
    [ si (mcd azar 1000000 azar 1000000) = 1
       [haz "contador :contador+1] ]
  escribe [Frecuencia:]
  escribe :contador/1000
fin
```

Comprobamos la validez del método:

```
prueba
0.609
prueba
0.626
prueba
0.597
```

y vemos que se obtienen valores próximos al valor teórico de 0,6097. Lo que es notable es que esta frecuencia es un valor aproximado de $\frac{6}{\pi^2} \simeq 0,6079$.

Si tenemos en cuenta `f`, la frecuencia encontrada, se tiene, entonces:

$$f \simeq \frac{6}{\pi^2}$$

Despejando:

$$\pi^2 \simeq \frac{6}{f} \quad \text{y así:} \quad \pi \simeq \sqrt{\frac{6}{f}}$$

Añadimos esta aproximación al programa, y transformamos el final del procedimiento `prueba`:

```
  para prueba
  # Inicializamos la variable contador a 0
    haz "contador 0
    repite 1000
      [ si (mcd azar 1000000 azar 1000000) = 1
         [ haz "contador :contador+1 ] ]
  # Tras calcular la frecuencia
    haz "f :contador/1000
  # Mostramos el valor aproximado de pi
    escribe frase [Aproximacion de pi:] raizcuadrada (6/:f) fin
  fin
```

que proporciona:

```
prueba
Aproximacion de pi: 3.164916190172819
prueba
Aproximacion de pi: 3.1675613357997525
prueba
Aproximacion de pi: 3.1008683647302115
```

Por último, modifiquemos el programa de modo que cuando lo lancemos, podamos indicar cuántas pruebas con números aleatorios deseamos.

```
  para prueba :repeticiones
  # Inicializamos la variable contador a 0
    haz "contador 0
    repite :repeticiones
      [ si (mcd azar 1000000 azar 1000000) = 1
         [ haz "contador :contador+1 ] ]
  # Tras calcular la frecuencia
    haz "f :contador/:repeticiones
  # Mostramos el valor aproximado de pi
    escribe frase [Aproximacion de pi:] raizcuadrada (6/:f)
  fin
```

Probamos con 10000 repeticiones, y obtenemos en las tres primeras tentativas:

```
prueba 10000
Aproximacion de pi: 3.1300987144363774
prueba 10000
Aproximacion de pi: 3.1517891481565017
prueba 10000
Aproximacion de pi: 3.1416626832299914
```

No está mal, ¿verdad?

## 15.4. Compliquemos un poco más: π que genera π...

¿Qué es un número aleatorio? ¿Es que un número tomado aleatoriamente entre 1 y 1.000.000 es un número realmente aleatorio?

Deberías darte cuenta rápidamente que nuestro modelo no hace más que aproximarse al modelo ideal. Bien, es precisamente sobre el modo de generar el número aleatorio sobre el que vamos a efectuar algunos cambios .... No vamos utilizar más la primitiva `azar`, sino que utilizaremos la secuencia de los decimales de π.

Me explico: los decimales de π siempre han intrigado a los matemáticos por su falta de regularidad, las cifras de 0 a 9 parecen aparecer en cantidades aproximadamente iguales y de manera aleatoria. No se pueden predecir los decimales siguientes basándonos en los anteriores.

Vamos a ver a continuación como generar un número aleatorio con ayuda de los decimales de π. En primer lugar, debes obtener los primeros decimales de π (por ejemplo un millón). Existen dos programas que los calculan bastante bien. Aconsejamos `PiFast` para Windows y `Schnell-Pi` para Linux.

Puedes acceder a Internet para conseguir un fichero de texto:

`http://3.14159265358979323846264338327950288419716939937510582097494459 2.com`

También en Internet, en la *web* de xLogo:

`http://downloads.tuxfamily.org/xlogo/common/millionpi.txt`

Para crear los números aleatorios, agrupamos de 8 en 8 cifras la serie de decimales de π. Es decir, el fichero empieza así:

$$\underbrace{3{,}1415926}_{1^{\text{er}}\ \text{numero}}\underbrace{53589793}_{2^{\text{o}}\ \text{numero}}\underbrace{23846264}_{3^{\text{er}}\ \text{numero}}\underbrace{33832795}_{\dots}0288419716939\dots$$

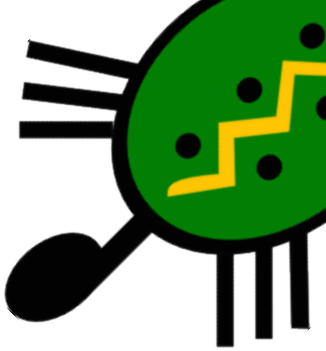
Retiro la coma "," del `3.14...` que podría equivocarnos al extraer los decimales. Bien, todo está preparado, creamos un nuevo procedimiento llamado `azarpi` y modificamos ligeramente el procedimiento `prueba` para llamar al procedimiento `azarpi`:

```
 para azarpi :n
   hazlocal "numero "
   repite :n [
 # Si no hay ningun caracter en la linea
   si 0=cuenta :linea
    [haz "linea primero leelineaflujo 1]
 # Asignamos a la variable :caracter el valor de primer caracter de la linea
   haz "caracter primero :linea
 # despues eliminamos el primer caracter de la linea
   haz "linea menosprimero :linea
   haz "numero palabra :numero :caracter ]
   devuelve :numero
fin
```

El resultado es:

```
prueba 10
Aproximacion de pi: 3.4641016151377544
prueba 100
Aproximacion de pi: 3.1108550841912757
prueba 1000
Aproximacion de pi: 3.081180112566604
prueba 10000
Aproximacion de pi: 3.1403714651066386
```

Encontramos pues una aproximación del número $\pi$ ¡con ayuda de sus propios decimales!



# Capítulo 16

# Geometría de la tortuga en 3-D

## 16.1. La tortuga en Tres Dimensiones

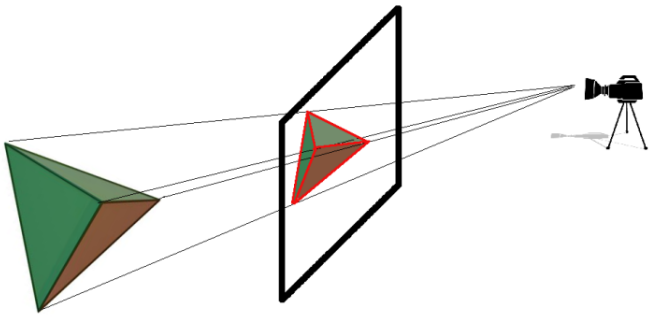
Desde la versión 0.9.92, nuestra tortuga puede dejar el plano para trasladarse a un espacio en tres dimensiones (3D). Para cambiar a esta modalidad, Usaremos la primitiva `perspectiva`. ¡Bienvenido a un mundo en 3D!

La tortuga cambia de forma y orientación, indicándonos en qué modo nos encontramos:

Para recuperar el modo bidimensional (2D), debemos indicarle que vuelva a uno de los modos "planos": `modojaula`, `modoventana` o `modovuelta`.

### 16.1.1. La proyección en perspectiva

Para representar un espacio 3D en un plano 2D, xLogo utiliza una proyección en perspectiva. Es equivalente a tener una cámara grabando la escena en 3D, y mostrando en la pantalla la imagen de la proyección. Veamos un esquema gráfico para explicarlo mejor:



Disponemos de primitivas para fijar la posición de la cámara, mientras que la pantalla de proyección se encuentra en el punto medio entre la cámara y el objeto.

### 16.1.2. Entender la orientación en el mundo tridimensional

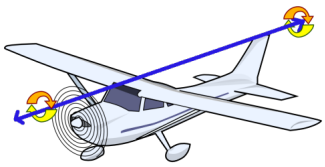
En el plano, la orientación de la tortuga se define únicamente por su rumbo. Sin embargo, en el mundo tridimensional la orientación de la tortuga necesita de tres ángulos. Si usamos la orientación por defecto de la tortuga en 3D (en el plano `XY` mirando hacia el semieje `Y` positivo):

**Balanceo:** la inclinación alrededor del eje $OY$

**Cabeceo:** la inclinación según el eje $OX$

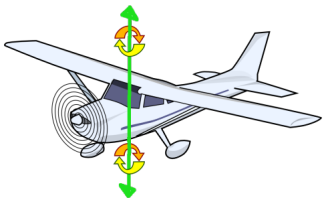
**Rumbo:** la inclinación según el eje $OZ$

De hecho, para moverse en el mundo tridimiensional, la tortuga se comportará de modo muy similar a un avión. De nuevo, ilustremos con una imagen los 3 ángulos:



**Balanceo**

**Cabeceo**



**Rumbo**

Parece bastante complicado la primera vez que se estudia, pero veremos que muchas cosas son similares a los movimientos en el plano bidimensional.

### 16.1.3. Primitivas

Estas son las primitivas básicas para moverse en el mundo 3D:

| Primitiva | Forma larga |
|---|---|
| Pasar al modo 3D | `perspectiva` |
| Salir del modo 3D | `modojaula`, `modovuelta` o `modoventana` |
| Baja el "morro" `n` grados | `cabeceaabajo n` ó `bajanariz n`, `bn n` |
| Sube el "morro" `n` grados | `cabeceaarriba n` ó `subenariz n`, `sn n` |

| | |
|---|---|
| Sube "ala" izquierda y baja "ala" derecha `n` grados | `balanceaderecha n`, `bd n` |
| Sube "ala" derecha y baja "ala" izquierda `n` grados: | `balanceaizquierda n`, `bi n` |
| Borrar Pantalla (y tortuga al centro, orientada "hacia nosotros" con el "timón de cola" hacia arriba) | `borrapantalla`, `bp` |
| Devuelve la inclinación de las "alas" | `balanceo` |
| Devuelve la inclinación del "morro" | `cabeceo` |

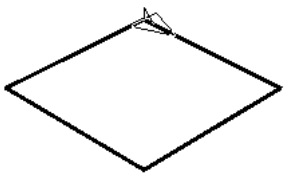
Además de las anteriores, podemos usar algunas de nuestras "viejas conocidas":

- `avanza`, `av`
- `retrocede`, `re`
- `giraderecha`, `gd`
- `giraizquierda`, `gi`

que realizan los mismos movimientos que en el "mundo 2D".
Por ejemplo, en el plano bidimensional, para dibujar un cuadrado de `200` pasos de tortuga, escribimos:

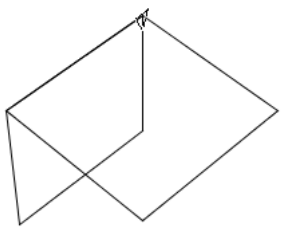
```
repite 4 [ avanza 200 giraderecha 90 ]
```

Estas órdenes siguen existiendo el mundo 3D, y el cuadrado puede dibujarse perfectamente en modo `perspectiva`:



Si la tortuga baja "el morro" 90 grados, podemos dibujar otro cuadrado, y obtenemos:

```
borrapantalla
repite 4 [ avanza 200 giraderecha 90 ]
bajanariz 90
repite 4 [ avanza 200 giraderecha 90 ]
```



Puedes (debes) probar otros ejemplos para entender perfectamente la orientación de la tortuga y el uso de los ángulos y ¡convertirte en un experto!

También debes entender que las tres primitivas que controlan la rotación en 3D están relacionadas entre sí; por ejemplo, al ejecutar:

```
borrapantalla
balanceaizquierda 90 subenariz 90 balanceaderecha 90
```

El movimiento de la tortuga es equivalente a:

```
giraizquierda 90
```

es decir, que como ocurre en 2D, no todas las primitivas son necesarias, por ejemplo:

```
bajanariz 90
```

es equivalente a:

```
balancea 90 giraderecha 90 balanceaizquierda 90
```

(Puedes probar con tu mano si no lo entiendes bien)

## 16.2. Primitivas disponibles tanto en 2D como 3D

Las siguientes primitivas están disponibles en el plano o en el mundo 3D. La única diferencia son los argumentos admitidos por las primitivas. Estas precisan de los mismos argumentos que en el plano:

| | | |
|---|---|---|
| `circulo` | `arco` | `centro` |
| `ponx` | `pony` | `coordenadax` |
| `coordenaday` | `rumbo` | `ponrumbo` |
| `rotula` | `largoetiqueta` | |

Las siguientes primitivas siguen esperando una lista como argumento, pero ahora debe contener **tres** argumentos, correspondientes a las tres coordenadas de un punto en el espacio: `[x y z]`.

| | | |
|---|---|---|
| `hacia` | `distancia` | `pos`, `posicion` |
| `ponpos`, `ponposicion` | `punto` | |

## 16.3. Primitivas sólo disponibles en 3D

- `ponxyz` Esta primitiva mueve a la tortuga al punto elegido. Esta primitiva espera tres argumentos que representan las coordenadas del punto.

  `ponxyz` es muy similar a `ponposicion`, pero las coordenadas no están escritos en una lista.

  Ejemplo, `ponxyz -100 200 50` traslada a la tortuga hasta el punto `x = -100`; `y = 200`; `z = 50`

- `ponz` Esta primitiva mueve a la tortuga al punto de "altura" (desconozco si el término *applikate* usado en Alemania tiene traducción al castellano más allá de **tercera coordenada**) dada. `ponz` recibe un número como argumento, de modo idéntico a `ponx` y `pony`

- `coordenadaz` o `coordz` Devuelve la "altura" de la tortuga. Es equivalente a `ultimo posicion`, pero simplifica la escritura.

- `ponorientacion` Fija la orientación de la tortuga. Esta primitiva espera una lista que contiene tres números: `[balanceo cabeceo rumbo]`

  Ejemplo: `ponorientacion [100 0 58]`: la tortuga tendrá balanceo: 100 grados, cabeceo: 0 grados y rumbo: 58 grados.

  Por supuesto, el orden de los números es importante. Si, por ejemplo, el valor de la orientación es `[100 20 90]`, esto significa que si quieres esa misma orientación partiendo del origen (después de un `borrapantalla`, por ejemplo) deberás escribir la siguiente secuencia:

  ```
  cabeceaderecha 100
  subenariz 20
  giraderecha 90
  ```

  Si en esta instrucción cambiamos el orden, no obtendremos la orientación deseada.

- `orientacion` Devuelve la orientación de la tortuga en una lista que contiene: `[balanceo cabeceo rumbo]`

- `ponbalanceo` La tortuga gira en torno a su eje longitudinal y adquiere el ángulo de balanceo elegido.

- `balanceo` Devuelve el valor actual del balanceo

- `ponbalanceo` La tortuga gira en torno a su eje transversal, y se orienta con el ángulo de cabeceo indicado.

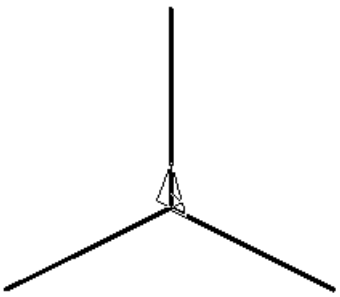
- `balanceo` Devuelve el valor actual del cabeceo

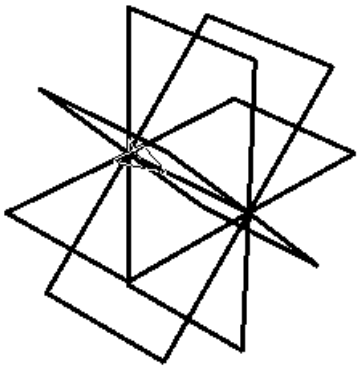
Podemos dibujar los semiejes en la posición habitual tecleando, por ejemplo:

**semieje y+:** `borrapantalla avanza 200 retrocede 200`

**semieje x+:** `giraderecha 90 avanza 200 retrocede 200`

**semieje z+:** `subenariz 90 avanza 200 retrocede 200`





Podemos dibujar 8 planos verticales, cada uno girado respecto del anterior 45 grados, así:

```
borrapantalla
repite 8
  [ balanceaderecha 45
    repite 4
      [ avanza 100 giraderecha 90 ] ]
```

## 16.4. Ejercicios

1. Dibuja un taburete cuadrado con una pata en cada esquina.

   **Idea:** repetir 4 veces "lado – pata".

2. A partir del taburete anterior, dibuja una silla con el respaldo inclinado.

3. Reforma la silla anterior para dibujar una tumbona.

4. Dibuja una pirámide cuadrangular regular.

   **Idea**: proceder como si fuera un taburete invertido, con las patas inclinadas 45 grados respecto del plano horizontal.

5. Dibuja un octaedro a partir de la pirámide anterior

## 16.5. El Visor 3D

xLogo incluye un visor 3D que permite visualizar los dibujos realizados en tres dimensiones. Este módulo usa las librerías de Java3D, por lo tanto es necesario tener instalado todo el Java3D.

### 16.5.1. Reglas

Las reglas a tener en cuenta para utilizar el Visor 3D son:

Al crear una figura geométrica sobre el Área de Dibujo, hay que indicar al Visor 3D qué formas desea grabar para una futura visualización. Es posible grabar polígonos (superficies), líneas, puntos o texto. Para utilizar esta función, las primitivas son:

- `empiezapoligono`, `definepoligono`: Los movimientos de la tortuga posteriores a esta llamada se guardan para crear un polígono.
- `finpoligono`: Desde la ejecución de `definepoligono`, la tortuga habrá pasado por varios vértices. Este polígono se habrá "registrado" y su color se definirá en función del color de todos sus vértices.

  Esta primitiva finaliza el polígono.
- `empiezalinea`, `definelinea`: Los movimientos de la tortuga posteriores a esta llamada se guardan para crear una banda (una línea).
- `finlinea`: Desde la ejecución de `definelinea`, la tortuga habrá pasado por varios vértices. Se guardará esta línea y su color se definirá en función del color de todos sus vértices.

  Esta primitiva finaliza la banda
- `empiezapunto`, `definepunto`: Los movimientos sigueintes de la tortuga se guardan para crear un conjunto de puntos.
- `finpunto`: Esta primitiva finaliza el conjunto de puntos.
- `empiezatexto`, `definetexto`: Cada vez que el usuario muestre un texto sobre el Área de Dibujo con la primitiva `rotula`, se almacenará y luego será representado por el visor 3D.
- `fintexto`: Esta primitiva finaliza la grabación de texto.
- `vista3d`, `vistapoligono` Inicia el visor 3D, todos los objetos guardados se dibujan en una nueva ventana.

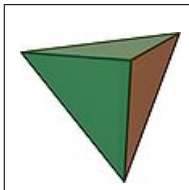
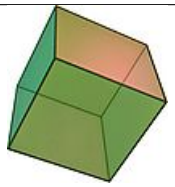
  Disponemos de controles para mover la "cámara" que muestra la escena:

  - Para hacer rotar la imagen haciendo *clic* con el botón izquierdo del ratón y arrastrando.

- Para desplazar la imagen haciendo *clic* con el botón derecho del ratón y arrastrando.
- Para hacer *zoom* sobre la escena, usaremos la rueda del ratón

### 16.5.2. Poliedros. Los sólidos platónicos

Los sólidos platónicos son el tetraedro, el cubo (o hexaedro regular), el octaedro, el dodecaedro y el icosaedro. También se conocen como cuerpos platónicos, cuerpos cósmicos, sólidos pitagóricos, sólidos perfectos, o **poliedros regulares convexos**. Son poliedros convexos cuyas caras son polígonos regulares iguales en cuyos vértices se unen el mismo número de caras, y reciben este nombre en honor al filósofo griego Platón (ca. 427 adC/428 adC – 347 adC).

|  |  |  |  |  |
|---|---|---|---|---|
| Tetraedro | Hexaedro o cubo | Octaedro | Dodecaedro | Icosaedro |

Esta lista es exhaustiva, ya que es imposible construir otro sólido diferente que cumpla todas las propiedades exigidas, es decir, convexidad y regularidad.

Para su construcción seguiremos un mecanismo general:

- Crearemos un procedimiento que trace el polígono que define sus caras entre las primitivas `empiezapoligono` y `finpoligono`.
- Comenzando desde una cara, iremos recorriendo los vértices dibujando las caras anexas.
- En caras dibujadas en el punto anterior, repetimos el proceso hasta concluir el poliedro.

En todo momento, la mayor dificultad será determinar la orientación adecuada, es decir, el ángulo diedro:.

| Tetraedro | Hexaedro o cubo | Octaedro |
|---|---|---|
| $\varphi = \arccos\frac{1}{3} \simeq 70{,}52878^\circ$ | $90^\circ$ | $\varphi = \arccos\frac{-1}{\sqrt{3}} \simeq 109{,}47122^\circ$ |

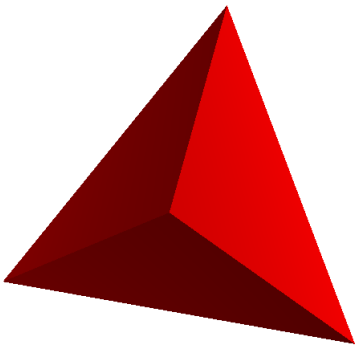
| Dodecaedro | Icosaedro |
|---|---|
| $\varphi = \arccos\frac{-1}{\sqrt{5}} \simeq 116{,}56505$ | $\varphi = \arccos\frac{-\sqrt{5}}{3} \simeq 138{,}189685^\circ$ |

### Dibujando un tetraedro

El poliedro más simple, una vez dibujada la base, basta recorrerla con el balanceo adecuado:

```
para triangulo :lado
   empiezapoligono
    repite 3 [ avanza :lado giraderecha 120 ]
   finpoligono
fin

para tetraedro :lado
  borrapantalla perspectiva
  poncolorlapiz rojo
  haz "angulo arcocoseno 1/3
# Base triangular
  triangulo :lado
# caras laterales
  repite 3
    [ balanceaizquierda :angulo
      triangulo :lado
      balanceaderecha :angulo avanza :lado giraderecha 120 ]
  vista3d
fin
```

Ejecutando: `tetraedro 500`:



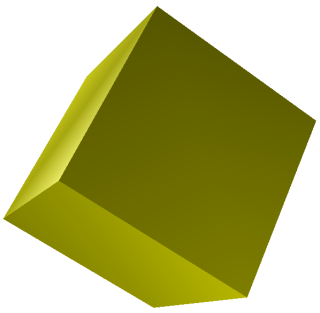
### Dibujando un cubo

El caso más simple en lo referente al ángulo diedro es el cubo: son todos de 90 grados. A cambio, debemos recorrer más vértices.

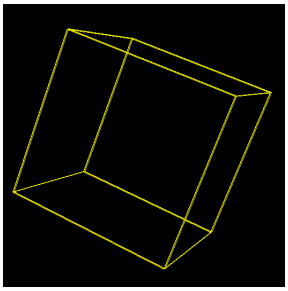
```
 para cuadrado :lado
 # Grabamos los vertices del cuadrado
```

```
   empiezapoligono
   repite 4 [ avanza :lado giraderecha 90 ]
   finpoligono
 fin

para cubosimple :lado
 # Cubo Amarillo
   borrapantalla perspectiva
   poncolorlapiz amarillo
 # Caras laterales
   repite 4
     [ cuadrado :lado subelapiz
       giraderecha 90 avanza :lado giraizquierda 90
       balanceaderecha 90 bajalapiz ]
 # Parte inferior
   bajanariz 90 cuadrado :lado subenariz 90
 # Cara Superior
   avanza :lado bajanariz 90 cuadrado :lado
 # Visualizacion
   vista3d
fin
```

Estamos listos para ejecutar el comando: `cubosimple 400`:



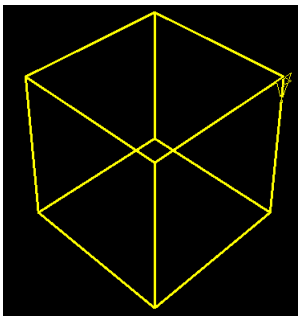
Al sustituir en el procedimiento `cuadrado`, `empiezapoligono` por `empiezalinea`, y `finpoligono` por `finlinea`:



Si hubiéramos usado `empiezapunto` y `finpunto` en lugar de `empiezalinea` y `finlinea`, deberíamos ver en la pantalla sólo los ocho vértices del cubo.

Estas primitivas son muy útiles para mostrar el conjunto de puntos en el espacio 3D.

En todos los casos, en el Área de Dibujo se muestran las aristas del cubo que luego se verá "macizo" con el Visor:

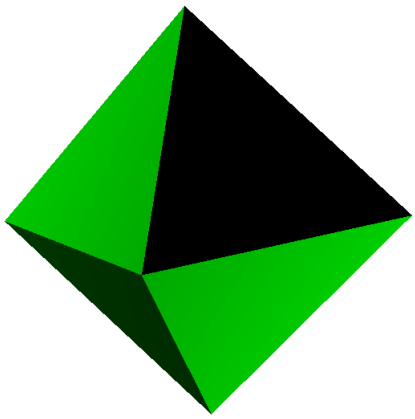


### Dibujando un octaedro

Este poliedro tampoco presenta una dificultad excesiva. Recorremos el cuadrado central dos veces, una vez balanceados hacia la derecha y otra a la izquierda:

```
para triangulo :lado
   empiezapoligono
    repite 3 [ avanza :lado giraderecha 120 ]
   finpoligono
fin

para octaedro :lado
  borrapantalla perspectiva
  poncolorlapiz verde
  haz "angulo 0.5*arcocoseno (-1)/3
  repite 4
    [ balanceaizquierda :angulo
         triangulo :lado
      balanceaderecha 2*:angulo
         triangulo :lado
      balanceaizquierda :angulo
      avanza :lado giraderecha 90 ]
  vista3d
fin
```

El resultado:

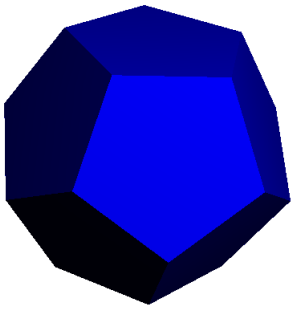


### Dibujando un dodecaedro

Gracias a Juan Casal por "prestarnos" este programa. En él vemos como se introduce un procedimiento `cambiacara` para simplificar el código, ya que en este caso también se hace necesario modificar el cabeceo de la tortuga:

```
para pentagono :lado
  empiezapoligono
   repite 5
    [ avanza :lado giraderecha 72 ]
  avanza :lado
  finpoligono
fin

para cambiacara
  giraizquierda 18
  bajanariz 63.44
  giraizquierda 18
fin

para dodecaedro :lado
  limpia
  perspectiva
  poncolorlapiz azul
  pentagono :lado          #cara sup
  giraizquierda 180
  balanceaderecha 63.44    #pos ini p corona sup
  repite 5                   #corona sup
  [ pentagono :lado cambiacara ]
  subelapiz
```

```
     balanceaizquierda 63.44 giraderecha 180
    cambiacara
     avanza :lado giraderecha 72 avanza :lado
    cambiacara
     avanza :lado giraderecha 72
  bajalapiz
  repite 5
   [ pentagono :lado cambiacara ]
  balanceaizquierda 63.44
  giraizquierda 180
  retrocede :lado
  pentagono :lado
#
  vista3d
fin
```



### Dibujando un icosaedro

Finalmente, y adaptando un programa de Tom Lynn, conseguimos el más complejo de los sólidos platónicos. Dibujaremos cinco bloques de cuatro triángulos, y nos ayudamos de dos procedimientos para movimientos básicos:

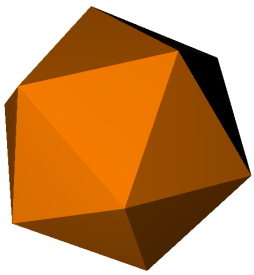
- `proxarista`, que desplaza a la tortuga hacia la siguiente arista en sentido horario sobre la cara actual
- `carasig`, que la desplaza hacia la cara adyacente por la derecha

  `repite 5 [carasig]` devuelve la tortuga a la posición original

```
para icosaedro :lado
  borrapantalla
  perspectiva
  poncolorlapiz naranja
# Inicializamos los angulos
```

```
  haz "phi (1 + raizcuadrada 5) / 2
  haz "beta arcoseno (:phi / raizcuadrada 3)
  haz "alpha 180-2*:beta
  repite 5 [cuatrotrian :lado]
  vistapoligono
fin

para triangulo :lado
  bajalapiz # los procedimientos auxiliares lo dejan arriba
  empiezapoligono
   repite 3 [avanza :lado giraderecha 120]
  finpoligono
  carasig :lado
fin

para cuatrotrian :lado
  repite 2 [triangulo :lado] proxarista :lado
  repite 2 [triangulo :lado]
  repite 2 [ repite 2 [carasig :lado] proxarista :lado]
  carasig :lado
fin

para proxarista :lado
  subelapiz avanza :lado giraderecha 120
fin

para carasig :lado
  subelapiz giraderecha 60 avanza :lado giraizquierda 120 balanceaderecha :alpha
fin
```

El resultado:

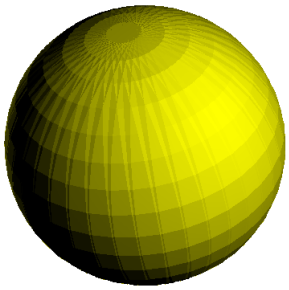


### 16.5.3. La esfera

El paso final es extender las `n` caras al caso *infinito*, es decir, cómo obtener una esfera. Evidentemente no podemos dibujar *infinitos* polígonos, así que vamos a aproximar la esfera

mediante cuadrados:

```
para esfera
  borrapantalla
  perspectiva
  poncolorlapiz amarillo
  repite 20
    [ retrocede 25 giraizquierda 90 avanza 25 giraderecha 90
      repite 36
        [ cuadrado 50 avanza 50 cabeceaarriba 10 ]
      giraizquierda 90 retrocede 25
      giraderecha 90 avanza 25
      giraderecha 9 ]
  vistapoligono
fin

para cuadrado :lado
  empiezapoligono
  repite 4 [ avanza :lado giraderecha 90 ]
  finpoligono
fin
```



**Intenta crear una esfera usando la primitiva `circulo`. ¿Qué observas? ¿Por qué crees que es? Plantéate las mismas preguntas al ver el resultado cuando utilizas un procedimiento como el del `360-gono` que vimos en la sección 4.6.3 (pág. 36)**

## 16.6. Ejercicios

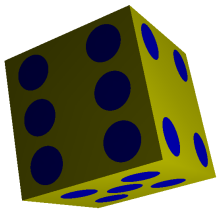
1. Crea un procedimiento `prisma_rect` que dibuje un prisma de base rectangular cuyas aristas midan los valores introducidos con tres variables `:a`, `:b` y `:c`.

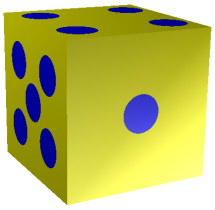
2. Diseña un procedimiento `prisma_reg` que dibuje un prisma cuya base sea un polígono regular y lea tres variables: `:n` (número de lados del polígono), `:l` (lado del polígono) y `:h` (altura).

   Generaliza el procedimiento anterior para obtener un cilindro cuya base tenga radio `r` y altura `h`.

   Observa como cambia el aspecto que tiene el cilindro (la calidad del mismo) a medida que aumenta el número de lados de la base.

3. Transforma el cubo creado en 16.5.2 en un dado.





   Recuerda que las caras opuestas de un dado suman siete.

4. Crea un procedimiento `piramide` que dibuje una pirámide de base rectangular, cuyos parámetros sean las variables `:a`, `:b` (las dimensiones del rectángulo de la base) y `:h` (la altura de la pirámide).

5. Modifica el procedimiento anterior para obtener una pirámide regular que acepte tres valores: `n`, número de lados de la base; `l`, lado de la base y `h`, altura.

   Generaliza el procedimiento anterior para obtener un cono de radio de base `r` y altura `h`.

   Observa como cambia el aspecto que tiene el cono (la calidad del mismo) a medida que aumenta el número de lados de la base.

6. Combina los procedimientos creados antes para obtener:

   a) Una casa en 3D combinando un prisma y una pirámide.

   b) Un silo en 3D combinando un cilindro y un cono.

7. Modifica los procedimientos `cono` y `piramide` para obtener un cono truncado y una pirámide truncada, respectivamente. Ten en cuenta que debes introducir nuevas variables.

8. Combina los procedimientos creados en el ejercicio anterior para crear una casa de base rectangular cuyo tejado termine con uan pirámide truncada.

9. Combina los procedimientos `cono`, `cilindro` y `esfera` para dibujar un "árbol" por superposición de:

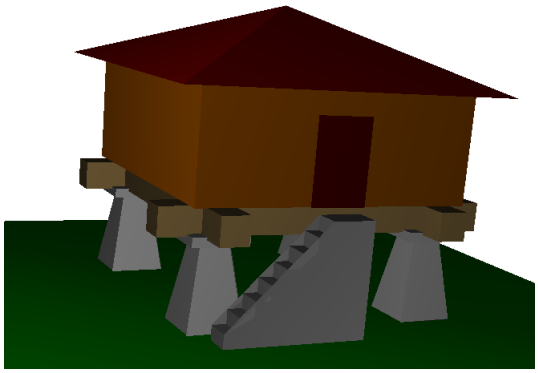
   a) Un cono "pegado" al suelo.

   b) Un cilindro de radio algo menor como tronco.

   c) Una esfera como copa del árbol.

10. Ya sabes crear "casas" y "árboles". ¿Te atreves a crear un pequeño pueblo en 3-D? Puedes hacer que las calles sean perpendiculares entre sí, y que las dimensiones de los edificios sean aleatorias, usando `azar`, para darle más realismo.

    Puedes incluir la versión en 3-D del hórreo asturiano (página 130):



    cuyo código, así como otros ejemplos, puede encontrarse en:

    `http://xlogo.tuxfamily.org/sp/html/ejemplos/3D.html`

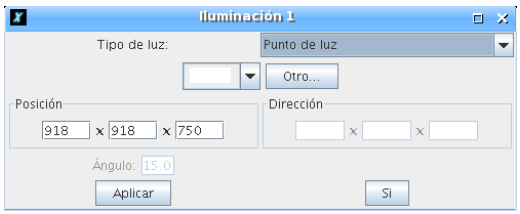
## 16.7. Efectos de luz y niebla

Desde la versión 0.9.93 se pueden añadir efectos artísticos a las imágenes generadas en el Visor. Estos pueden ser efectos de luz y de niebla, y se accede a ellos con los botones presentes en el visor 3D.

### Efectos de luz

Se pueden utilizar cuatro tipos de luz en las imágenes en tres dimensiones, a las que se accede haciendo *clic* en uno de los cuatro botones mostrados bajo la leyenda `Iluminación`.

Al trazar por primera vez una imagen en 3D sólo se utilizan dos tipos de luz, ambos **Luz Puntual**, pero pulsando en cualquiera de los cuatro botones de `Iluminación`, aparece el siguiente cuadro de diálogo:




donde podemos elegir entre los siguientes tipos de luz:

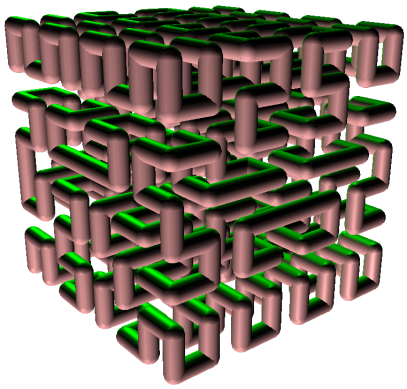
**Luz Ambiente:** Luz uniforme de la que sólo puede modificarse el color

**Luz Direccional:** Se genera respecto a una dirección fija. Se parece a la luz ambiente cuando la fuente está muy lejos del objeto (por ejemplo, el sol)

**Punto de Luz:** La fuente está en una posición determinada, como en el caso de un faro.

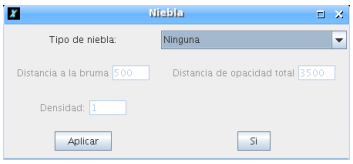
**Foco:** Es como el punto de luz, pero el haz de luz se abre formando un cono cuya abertura debe fijarse.

La mejor forma de entenderlo, es practicar con ello.



### Efectos de niebla

Se pueden añadir efectos de niebla en la imagen tridimensional. Pulsa el botón con "nubes" y obtendrás este cuadro de diálogo:



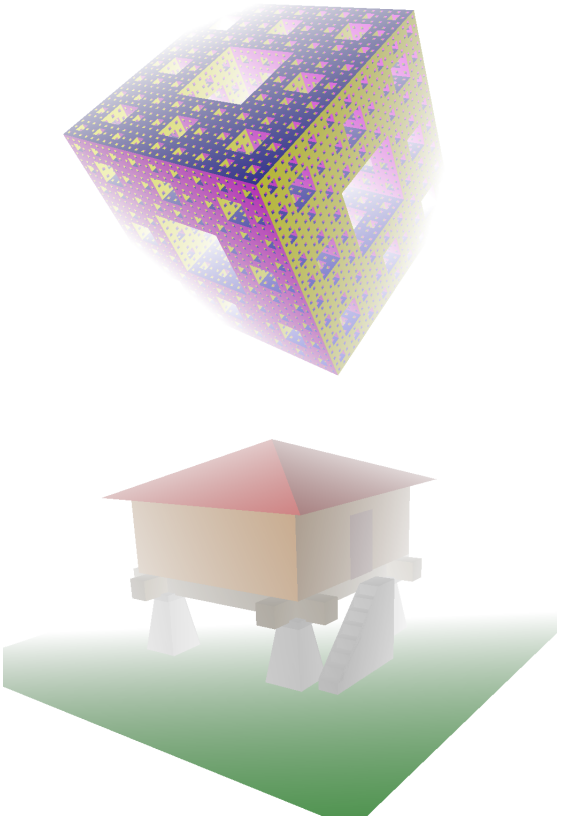

Disponemos de dos tipos de niebla:

**Niebla Lineal o progresiva:** La imagen se va difuminando de modo lineal, pudiendo variar dos parámetros:

- La distancia a la que empieza la niebla
- La distancia a la que la niebla no deja ver nada (opacidad total)

**Niebla Densa:** La niebla es uniforme en toda la escena, y sólo necesitamos especificar la densidad de la misma.

Este es un ejemplo con niebla lineal:

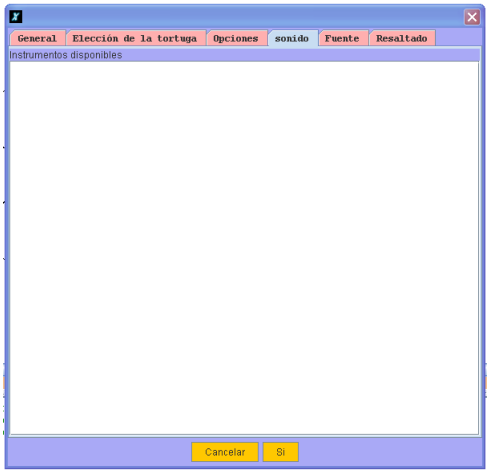


# Capítulo 17

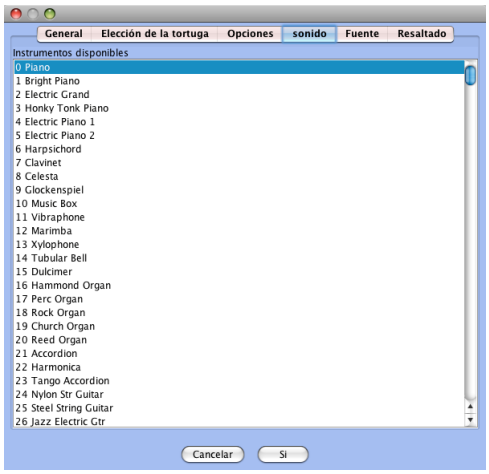
# Tocar música (MIDI)

Ya comentamos anteriormente (sección 1.2.1) que la versión Windows de `jre` no incorpora las API (*Application Programming Interface* – Interfaz de Programación de Aplicaciones) que contienen los instrumentos y que deben ser instaladas manualmente. Es importante recordarlo porque, si no lo haces, con la instalación por defecto de Java no tendrás instrumentos disponibles:

Instalación por defecto de Java




En Windows




En Linux y Mac

Puedes hacer pruebas con los distintos bancos de sonidos y ver las diferencias entre ellos para elegir el que más te guste, si bien el mínimo es más que suficiente para nuestros objetivos.

Ten en cuenta que el intérprete Java selecciona automáticamente el banco de mayor "calidad", así que vete probando de "menor a mayor", sin borrar el anterior hasta que te hayas decidido.

## 17.1. Las primitivas

| Primitivas | Argumentos | Uso |
|---|---|---|
| `secuencia`, `sec` | `lista` | Carga en memoria la secuencia incluída en la lista. Siguiendo a esta tabla, se indica cómo escribir una secuencia de notas musicales. |
| `tocamusica` | `no` | Toca la secuencia de notas en memoria. |
| `instrumento`, `instr` | `no` | Da el número que corresponde al instrumento actualmente seleccionado. |
| `poninstrumento`, `pinstr` | `número` | Queda seleccionado el instrumento número `n`. Puedes ver la lista de instrumentos disponibles en el menú **Herramientas** → **Preferencias** → **Sonido**. |
| `indicesecuencia`, `indsec` | `no` | Da la posición del puntero en la secuencia corriente. |
| `ponindicesecuencia`, `pindsec` | `número` | Pone el puntero en la posición `n` de la secuencia corriente. |
| `borrasecuencia`, `bos` | `no` | Elimina de memoria la secuencia corriente. |

Para tocar música, primero hay que poner en memoria una lista de notas llamada *secuencia*. Para crear una secuencia, puedes usar la primitiva `sec` o `secuencia`. Para crear una secuencia válida, hay que seguir las siguientes reglas:

- `do re mi fa sol la si`: Las notas usuales de la primera octava.
- Para hacer un re sostenido, anotamos `re +`
- Para hacer un re bemol, anotamos `re -`
- Para subir o bajar una octava, usamos "`:`" seguido de "`+`" o "`-`".

  **Ejemplo**:

  Después de `:++` en la secuencia, todas las notas sonarán dos octavas más altas.

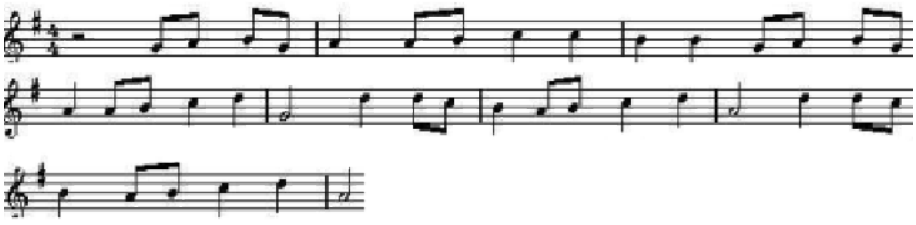
Por defecto, todas las notas tienen una duración uno. Si quieres aumentar o disminuir la duración, debes escribir un número correspondiente.

**Ejemplos**:

```
secuencia [sol 0.5 la si]
```

tocará sol con la duración 1 y la y si con la duración 0.5 (el doble de rápido).

Otro ejemplo:



```
para partitura
# crea la secuencia de notas
  secuencia [0.5 sol la si sol 1 la 0.5 la si 1 :+ do do :- si si 0.5 sol la
    si sol 1 la 0.5 la si 1 :+ do re 2 :- sol ]
  secuencia [:+ 1 re 0.5 re do 1 :- si 0.5 la si 1 :+ do re 2 :- la ]
  secuencia [:+ 1 re 0.5 re do 1 :- si 0.5 la si 1 :+ do re 2 :- la ]
  secuencia [0.5 sol la si sol 1 la 0.5 la si 1 :+ do do :- si si 0.5 sol la
    si sol 1 la 0.5 la si 1 :+ do re 2 :- sol ]
fin
```

Para escuchar la música, ejecuta los comandos: `partitura tocamusica`.

Ahora veamos una aplicación interesante de la primitiva `pindsec`:

```
borrasecuencia   # elimina toda secuencia de memoria
partitura        # pone en memoria las notas
pindsec 2        # pone el cursor en el segundo "la"
partitura        # pone en memoria las mismas notas, pero movidas 2 lugares.
tocamusica       # Grandioso!
```

En el caso en que tengas más de una secuencia en el programa, debes borrar la anterior antes de reproducir la siguiente.

También puedes elegir un instrumento con la primitiva `poninstrumento` o en el menú **Herramientas** → **Preferencias** → **Sonido**. Encontrarás la lista de instrumentos disponibles con su número asociado.

## 17.2. Ejercicios

¿Recuerdas el juego del capítulo 3? Modifícalo en los siguientes puntos:

1. En vez de círculos, que cargue imágenes de piedras "reales" y de un lago cercano a la zona donde vives (recuerda la primitiva `cargaimagen`.

- Quizá sea necesario que las redimensiones con algún programa externo, aunque existen formas de hacerlo con xLogo
- Por comodidad, la imagen del lago deberá tener un color uniforme

2. Distribuye aleatoriamente las imágenes de las piedras por la pantalla
3. Ubica el lago en una esquina y a la tortuga en la contraria.
4. Elige una melodía (te aconsejamos una sencilla, pero si te gusta mucho una, eres libre de usarla) y haz que suene cuando la tortuga llegue al lago.
5. Crea los botones `derecha` e `izquierda` (ya lo hiciste en 12.7), pero modifica el de `avanza` y `retrocede` para que llamen a un procedimiento que vaya mirando el color según se mueve la tortuga
   - Si ese color es el del lago, que aparezca un mensaje de felicitación y que suene la melodía.
   - Si, por el contrario, no es el del fondo o el del lago, que aparezca un mensaje de "Inténtalo de nuevo" y devuelva la tortuga al inicio.

## 17.3. Cargando archivos de música

Otra forma de escuchar música es cargando un archivo `mp3`:

- `escuchamp3`, cuyo argumento debe ser una palabra, inicia la reproducción del archivo `mp3` indicado.
- `detienemp3`, sin argumentos, detiene la reproducción del archivo `mp3` que está sonando.

# Capítulo 18

# Gestión de tiempos

xLogo dispone de varias primitivas que permiten conocer la hora y la fecha o utilizar un cronómetro descendente (útil para repetir una tarea a intervalos fijos).

## 18.1. Las primitivas

| Primitivas | Argumentos | Uso |
|---|---|---|
| `espera` | `número entero` | Hace una pausa en el programa, la tortuga espera `(n/60)` segundos. |
| `cronometro`, `crono` | `número entero` | Inicia un conteo descendiente de `n` segundos. Para saber que la cuenta ha finalizado, disponemos de la primitiva `fincrono?` |
| `fincronometro?`, `fincrono?` | `no` | Devuelve `"cierto` si no hay ningún conteo activo. Devuelve `"falso` si el conteo no ha terminado. |
| `fecha` | `no` | Devuelve una lista compuesta de 3 números enteros que representan la fecha del sistema. El primero indica el día, el segundo el mes y el último el año. `[día mes año]` |
| `hora` | `no` | Devuelve una lista compuesta de 3 números enteros que representan la hora del sistema. El primero representa las horas, el segundo los minutos y el último los segundos. `[horas minutos segundos]` |
| `tiempo` | `no` | Devuelve el tiempo, en segundos, transcurrido desde el inicio de xLogo. |

En las sección 13.1 comentábamos que los dibujos eran más rápidos con la tortuga oculta que con ella en pantalla. Si añadimos las siguientes líneas al principio y al final del procedimiento:

```
para nombre.proc :a :b ...
```

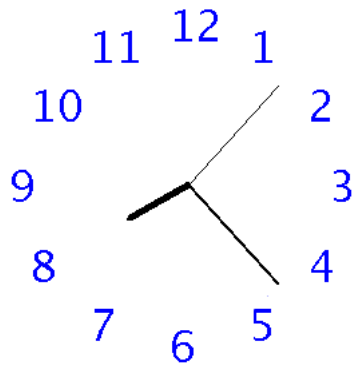
```
  haz "tardanza tiempo
   ...
   ...
  escribe :tardanza - tiempo
fin
```

obtendremos los segundos que tardó en ejecutarse el procedimiento.

Veamos otro ejemplo:

```
para reloj
 # muestra la hora en forma numerica (actualizada cada 5 segundos)
  si fincrono? [
   bp ponfuente 75 ot
   haz "ho hora
   haz "h primero :ho
   haz "m elemento 2 :ho
 # muestra dos cifras para los minutos (completando el 0)
   si :m - 10 < 0 [
    haz "m palabra 0 :m ]
   haz "s ultimo :ho
 # muestra dos cifras para los segundos
   si :s - 10 < 0 [
    haz "s palabra 0 :s ]
    rotula palabra palabra palabra palabra :h ": :m ": :s crono 5 ]
  reloj
fin
```

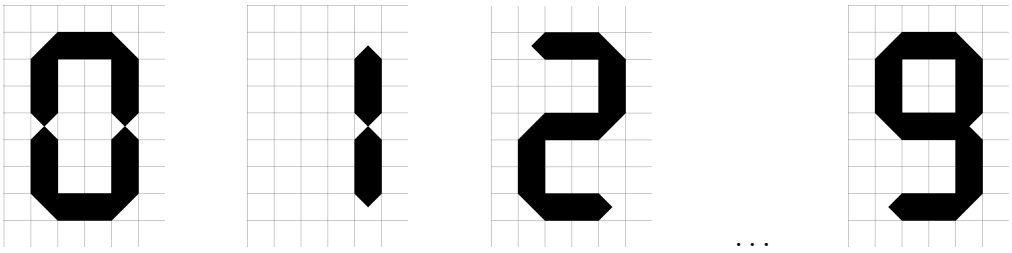
Podríamos plantearnos la forma de representar un reloj analógico:




¿Se te ocurre cómo hacerlo?

## 18.2. Actividad sobre las cifras de una calculadora

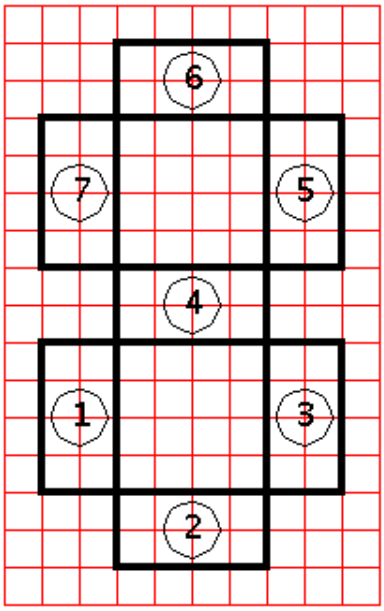
Al hablar de cifras de calculadora, nos referimos a las formas "tradicionales":



aunque nada impediría hacerlo con las formas *pixeladas* más modernas. De todas formas, simplificaremos aún más las cifras para centrarnos en los aspectos relacionados con el tiempo.

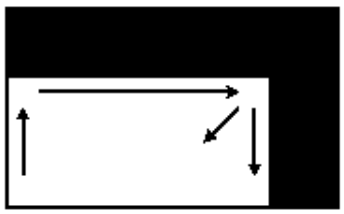
Esta actividad se basa en el hecho de que todos los números de calculadora pueden obtenerse con ayuda de un patrón sí–no:

- para dibujar un "4", "encenderemos" los rectángulos 3, 4, 5 y 7, pero no los 1, 2 y 6
- para dibujar un "8", "encenderemos" los rectángulos 1, 2, 3, 4, 5, 6 y 7.
- para dibujar un "3", encenderemos los rectángulos 2, 3, 4, 5 y 6, pero no los 1 y 7
- ...




### 18.2.1. El programa

Crearemos un rectángulo sólido de modo recursivo:



```
para rectangulo :alto :ancho
  si :ancho = 0 | :alto =0 [alto]
  repite 2
    [ avanza :alto giraderecha 90
      avanza :ancho giraderecha 90]
  rectangulo :alto-1 :ancho-1
fin
```

Supondremos aquí que la tortuga parte de la esquina inferior izquierda. Vamos a definir un procedimiento llamado `cifra` que lee 7 argumentos `:a`, `:b`, `:c`, `:d`, `:e`, `:f` y `:g`.

- Cuando `:a` vale `1`, dibuja el rectángulo 1. Si `:a` vale `0`, no se dibuja.
- Cuando `:b` vale `1`, dibuja el rectángulo 2. Si `:b` vale `0`, no se dibuja.
- Cuando `:c` vale `1`, dibuja el rectángulo 1. Si `:c` vale `0`, no se dibuja.
- ...

Se obtiene el siguiente procedimiento:

```
para cifra :a :b :c :d :e :f :g
# Dibujamos el rectangulo 1
  si :a=1 [rectangulo 160 40]
# Dibujamos el rectangulo 2
  si :b=1 [rectangulo 40 160]
  sl gd 90 av 120 gi 90 bl
# Dibujamos el rectangulo 3
  si :c=1 [rectangulo 160 40]
  sl av 120 bl
# Dibujamos el rectangulo 5
  si :e=1 [rectangulo 160 40]
# Dibujamos el rectangulo 4
  gi 90 sl re 40 bl
  si :d=1 [rectangulo 160 40]
# Dibujamos el rectangulo 6
  gd 90 sl av 120 gi 90 bl
  si :f=1 [rectangulo 160 40]
# Dibujamos el rectangulo 7
  sl av 120 gi 90 re 40 bl
  si :g=1 [rectangulo 160 40]
fin
```

### 18.2.2. Creación de una pequeña animación

Vamos a simular una cuenta atrás, que consiste en hacer aparecer sucesivamente las cifras de `9` a `0` en orden decreciente.

```
para cuentatras
   bp ot cifra 0 1 1 1 1 1 1 espera 60
   bp ot cifra 1 1 1 1 1 1 1 espera 60
   bp ot cifra 0 0 1 0 1 1 0 espera 60
   bp ot cifra 1 1 1 1 0 1 1 espera 60
   bp ot cifra 0 1 1 1 0 1 1 espera 60
   bp ot cifra 0 0 1 1 1 0 1 espera 60
   bp ot cifra 0 1 1 1 1 1 0 espera 60
   bp ot cifra 1 1 0 1 1 1 0 espera 60
   bp ot cifra 0 0 1 0 1 0 0 espera 60
   bp ot cifra 1 1 1 0 1 1 1 espera 60
fin
```

Pequeño problema: hay un efecto de parpadeo desagradable durante la creación de cada cifra. Para suavizarlo se van a utilizar las primitivas `animacion` y `refrescar`.
Se obtiene el siguiente programa modificado:

```
para cuentatras
# Entramos en modo animacion
  animacion
  bp ot cifra 0 1 1 1 1 1 1 refrescar espera 60
  bp ot cifra 1 1 1 1 1 1 1 refrescar espera 60
  bp ot cifra 0 0 1 0 1 1 0 refrescar espera 60
  bp ot cifra 1 1 1 1 0 1 1 refrescar espera 60
  bp ot cifra 0 1 1 1 0 1 1 refrescar espera 60
  bp ot cifra 0 0 1 1 1 0 1 refrescar espera 60
  bp ot cifra 0 1 1 1 1 1 0 refrescar espera 60
  bp ot cifra 1 1 0 1 1 1 0 refrescar espera 60
  bp ot cifra 0 0 1 0 1 0 0 refrescar espera 60
  bp ot cifra 1 1 1 0 1 1 1 refrescar espera 60
# Volvemos al modo de dibujo habitual
  detieneanimacion
fin
```

## 18.3. Ejercicios

1. Diseñemos un juego. A ver quién "cuenta" mejor un minuto mentalmente.

   a) Debe haber un botón de "Inicio" (sólo nos preocupa el *clic* del ratón)

b) Al hacer *clic* sobre el botón, la tortuga puede:

- Guardar el valor de `tiempo`
- Guardar la `hora` actual

c) Permanece en espera hasta que se hace un segundo *clic*. En ese momento, calcula la diferencia entre el `tiempo` o la `hora` almacenados y muestra si te has pasado, te has quedado corto o has acertado.

Puedes añadir cualquier efecto que deje claro cualquiera de los tres resultados (color de pantalla, de letra, música, ...)

2. Diseña un programa que se comporte como la alarma de un reloj:

   a) Abra una ventana que pida la hora a la que debe sonar

   b) Compruebe la hora recursivamente hasta que coincida con la introducida

   c) Cuando coincidan la hora real y la introducida, que suene una melodía

3. Recupera el procedimiento `bisiesto?`, y haz que xLogo te diga si el año actual es bisiesto o no.

4. Diseña un procedimiento que te pida la fecha de tu cumpleaños y determine cuántos días faltan para él.

   Si hoy es tu cumpleaños, que suene el "Cumpleaños Feliz".

5. Diseña un procedimiento que determine qué día de la semana es hoy.

   **(Dificultad alta)** Amplíalo para que te pida una fecha cualquiera y diga qué día de la semana le corresponde. Para ello:

   a) Elige una fecha de referencia de la que sepas qué día de la semana es.

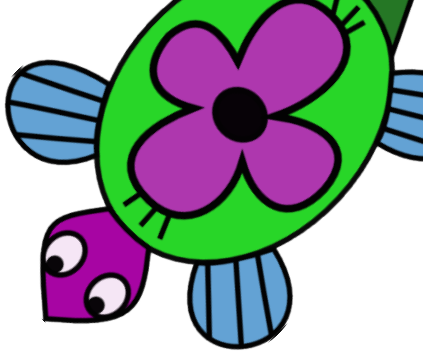
   b) Observa que los 365 días que tiene un año son:

   $$365 = 364 + 1 = 7 \times 52 + 1$$

   es decir, cada año normal desplaza un día el día de la semana de una fecha fija: por ejemplo, el 23 de Enero de 2006 fue Lunes, pero el mismo día de 2007 fue Martes.

   c) Ten en cuenta que los años bisiestos desplazan dos días en vez de uno

   d) Finalmente, haz un razonamiento análogo para cada mes, estudiando sus respectivos días



# Capítulo 19

# Utilización de la red con xLogo

## 19.1. La red: ¿cómo funciona eso?

En primer lugar es necesario explicar los conceptos básicos de la comunicación en una red para usar correctamente las primitivas de xLogo.

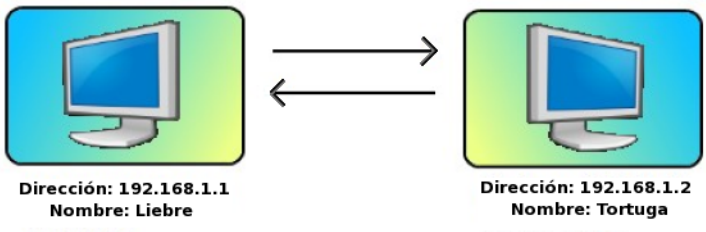



Figura: noción de red

Dos ordenadores pueden comunicarse a través de una red si están equipados con una tarjeta de red (llamada también tarjeta *ethernet*). Cada ordenador se identifica por una dirección personal: su dirección I.P. Esta dirección IP consta de cuatro números enteros comprendidos entre 0 y 255 separados por puntos. Por ejemplo, la dirección IP del primer ordenador del esquema de la figura es `192.168.1.1`

Dado que no es fácil acordarse de este tipo de dirección, también se puede hacer corresponder a cada dirección IP un nombre más fácil de recordar. Sobre el esquema anterior, podemos comunicar con el ordenador de la derecha bien llamándolo por su dirección IP: `192.168.1.2`, o llamándolo por su nombre: `tortuga`.

No nos extendamos más sobre el significado de estos números. Añadamos únicamente una cosa que es interesante saber, el ordenador local en el cual se trabaja también se identifica por una dirección: `127.0.0.1`. El nombre que se asocia con él es habitualmente `localhost`.

## 19.2. Primitivas orientadas a la red

xLogo dispone de 4 primitivas que permiten comunicarse a través de una red: `escuchatcp`, `ejecutatcp`, `chattcp` y `enviatcp`.

En los ejemplos siguientes, mantendremos el esquema de red de la subsección anterior.

- `escuchatcp`: esta primitiva es la base de cualquier comunicación a través de la red. No espera ningún argumento. Permite poner al ordenador que la ejecuta a la espera de instrucciones dadas por otros ordenadores de la red.

- `ejecutatcp`: esta primitiva permite ejecutar instrucciones sobre otro ordenador de la red.

  Sintaxis: `ejecutatcp palabra lista` → La palabra indica la dirección IP o el nombre del ordenador de destino (el que va a ejecutar las órdenes), la lista contiene las instrucciones que hay que ejecutar.

  Ejemplo: desde el ordenador `liebre`, deseo trazar un cuadrado de lado 100 en el otro ordenador. Por tanto, hace falta que desde el ordenador `tortuga` ejecute la orden `escuchatcp`. Luego, desde el ordenador `liebre`, ejecuto:

  ```
  ejecutatcp "192.168.2.2 [repite 4 [avanza 100 giraderecha 90]]
  ```

  o

  ```
  ejecutatcp "tortuga [repite 4 [avanza 100 giraderecha 90]]
  ```

- `chattcp`: permite chatear entre dos ordenadores de la red, abriendo una ventana en cada uno que permite la conversación.

  Sintaxis: `chattcp palabra lista` → La palabra indica la dirección IP o el nombre del ordenador de destino, la lista contiene la frase que hay que mostrar.

  Ejemplo: `liebre` quiere hablar con `tortuga`.
  `tortuga` ejecuta `escuchatcp` para ponerse en espera de los ordenadores de la red.
  `liebre` ejecuta entonces: `chattcp "192.168.1.2 [buenos días]`.
  Una ventana se abre en cada uno de los ordenadores para permitir la conversación

- `enviatcp`: envía datos hacia un ordenador de la red.

  Sintaxis: `enviatcp palabra lista` → La palabra indica la dirección IP o el nombre del ordenador de destino, la lista contiene los datos que hay que enviar. Cuando xLogo recibe los datos en el otro ordenador, responderá `Si` , que podrá asignarse a una variable o mostrarse en el **Histórico de comandos**.

Ejemplo: `tortuga` quiere enviar a `liebre` la frase `"3.14159 casi el número pi"`. `liebre` ejecuta `escuchatcp` para ponerse en espera de los ordenadores de la red. Si `tortuga` ejecuta entonces: `enviatcp "liebre [3.14159 casi el número pi]`, `liebre` mostrará la frase, pero en `tortuga` aparecerá el mensaje:

```
Qué hacer con [ Si ] ?
```

Deberíamos escribir:

```
es enviatcp "liebre [3.14159 casi el número pi]
```

o

```
haz "respuesta enviatcp "liebre [3.14159 casi el número pi]
```

En el primer caso, el **Histórico de comandos** mostrará `Si`, y en el segundo `"respuesta` contendrá la lista `[ Si ]`, como podemos comprobar haciendo

```
es lista? :respuesta
cierto
es :respuesta
Si
```

Con esta primitiva se puede establecer comunicación con un robot didáctico a través de su interfaz de red. En este caso, la respuesta del robot puede ser diferente, y se podrán decidir otras acciones en base al contenido de `:respuesta`.

Un pequeño truco: lanzar dos veces xLogo en un mismo ordenador.

- En la primera ventana, ejecuta `escuchatcp`.
- En la segunda, ejecuta

```
ejecutatcp "127.0.0.1 [repite 4 [avanza 100 giraderecha 90]]
```

¡Así puedes mover a la tortuga en la otra ventana! (¡Ah sí!, esto es así porque `127.0.0.1` indica tu dirección local, es decir, de tu propio ordenador)

## 19.3. Robótica

Desde 2006, xLogo es capaz de comandar una interfaz externa para robótica, (y así se presentó en el congreso Cafeconf - Aulas Libres). Las primitivas de comunicación por red que acabamos de ver, además de ser utilizadas para la tareas grupales en red, *chat*, comandar la tortuga en una PC desde otra PC, etc. permiten experimentar en robótica.

Me duele reconocer que mucho del trabajo realizado por Marcelo en este área se nos ha ido con él, pero intentaré explicar las cosas de forma lo bastante clara para que cualquiera pueda seguir el tema sin problemas.

Es aquí más que en ningún otro sitio del curso, donde agradeceré cualquier ayuda y/o corrección que usted, lector o lectora, sea capaz de proporcionar.

### 19.3.1. Presentación

**Prólogo**: Prof. María Cristina Moreno.

En los libros de ciencia ficción, el escritor, como un alquimista, le dió vida a los robots que hoy se incorporan a nuestra vida cotidiana; desde una simple cafetera, hasta el Mars Rover.

El objetivo es que tecnología y educación recorran juntas el camino hacia el futuro.
La propuesta es lograr que en casa y en la escuela se construyan y manejen pequeños robots didácticos.

Con estas premisas, Marcelo Duschkin nos dejó el proyecto **Mi Primer Robot**, basado en una interfaz electrónica simple y económica, TortuRob, que todos pueden construir libremente, al ser liberada bajo Licencia Creative Commons.

### 19.3.2. La electrónica

El currículo de la asignatura de **Tecnología** en la Enseñanza Secundaria nos presenta un bloque de control de procesos y robótica, pero habitualmente no se acompaña de una dotación adecuada en lo referente al material necesario para experimentar en el taller. Lo más habitual es aprovechar las tarjetas controladoras antiguas (tipo Inves, Enconor, CNICE, ladrillos Lego,...), con el problema que ello conlleva: su incompatibilidad con los sistemas Windows actuales. Por otro lado, tampoco es raro el caso de disponer de pocas unidades de los equipos citados, lo que añade un sobrecoste económico a la asignatura.

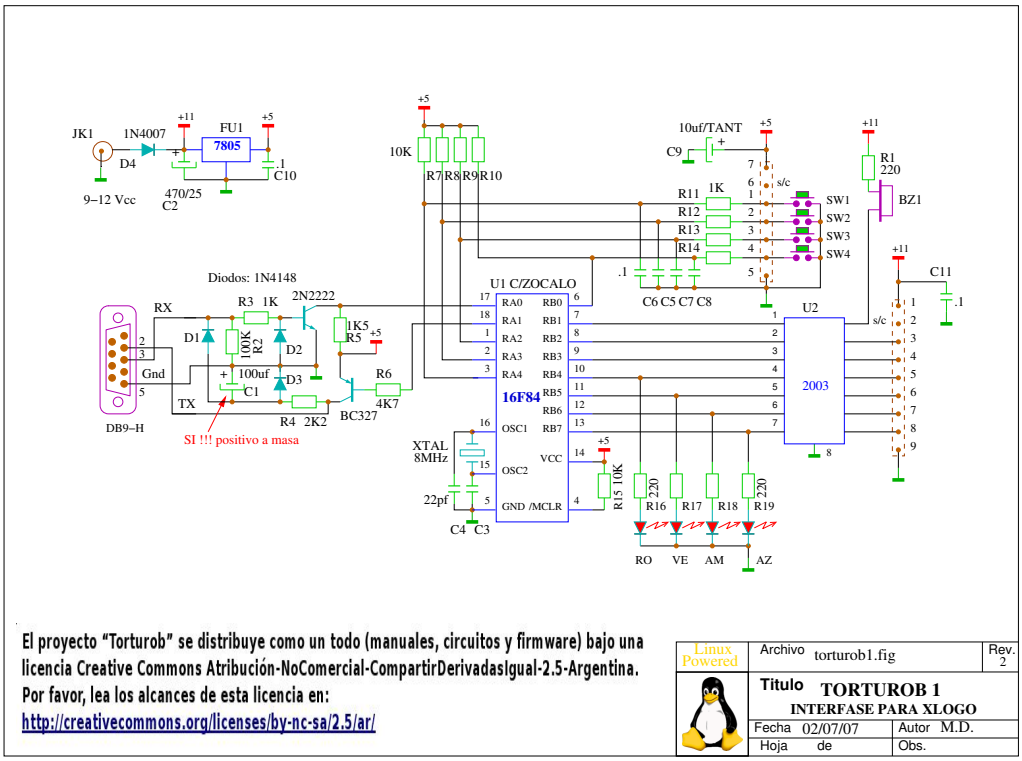
Nuestra propuesta de interfaz electrónica es el proyecto TortuRob.

#### El proyecto TortuRob

Es un circuito basado en un microprocesador PIC16F628A, que recibe comandos desde el PC, y contiene puertos de entrada y salida para controlar una mecánica de robot. También están incluídos 4 pulsadores y cuatro LEDs, con el objeto de usarlos en la etapa de aprender a usar el sistema. De esta manera, las primeras pruebas no necesitan de una mecánica a controlar.

La placa es de reducidas dimensiones (9,5 x 7 cm) y es muy fácil de armar por el aficionado a la electrónica. Los detalles técnicos para su construcción pueden descargarse de:

`http://downloads.tuxfamily.org/xlogo/robotica/TortuRob.zip`




La placa tiene interfaz serie, por ser el sistema más económico y sencillo, pero como XLOGO utiliza la conexión `ethernet`, es necesario hacer algún tipo de conversión. Lo más simple es una conversión por *software*. En Linux puede utilizarse un *script* Tcl/Tk liberado bajo licencia GPL:

`http://downloads.tuxfamily.org/xlogo/robotica/tcptty-es.zip`

que se ejecuta "*por detrás*" (*background*) de XLOGO.

Por desgracia, la versión en Visual Basic para Windows no es GPL y no puede proporcionarse un enlace al mismo (por razones obvias).

La otra solución es utilizar un módulo de *hardware* externo al PC, es decir, un convertidor `ethernet/rs232` de los hay muchos en el mercado. Esta solución, por supuesto, es más cara, pero tiene una aplicación interesante:

> En un aula con muchos ordenadores en red, se coloca un sólo módulo convertidor con el robot, y cualquier alumno lo comanda desde su puesto.

Finalmente, decir que tiene seis salidas (la séptima dedicada a un *buzzer*) y soporta cuatro sensores digitales, que pueden ser: contactos, pulsadores, fotorresistores u optoacopladores.

### 19.3.3. El lenguaje de la TortuRob

Evidentemente, para poner ejemplos sobre robótica debemos elegir un lenguaje. De acuerdo a nuestra elección sobre la placa, explicaremos el lenguaje de la TortuRob. Obviamente, entendiendo esta terminología, podremos utilizar los ejemplos en las tarjetas de que disponga el lector en su Centro.

El **protocolo de comunicación** con la TortuRob parte de las tres premisas siguientes:

1. La placa TortuRob nunca inicia una comunicación, solo responde.
2. Todo comando enviado recibe una respuesta.
3. Todos los comandos y respuestas tienen la estructura: `$Lx...x%[CR]`, donde:

   `$` → Identificador de inicio de comando.

   `%` → Identificador de fin de comando.

   `L` **y** `x...x` → `L` es una letra que identifica la acción (mayúscula si es comando, minúscula si es respuesta), mientras que `x...x` son los datos asociados a las órdenes/respuestas anteriores y su longitud es variable.

Con esta breve información, veamos unos ejemplos antes de entrar en detalle:

#### El semáforo

La TortuRob posee colores, concretamente, las salidas 3, 4 y 5 se corresponden con los LED rojo, verde y amarillo, respectivamente. ¿No invita eso a construir un semáforo?

En este programa sólo damos órdenes a la placa, concretamente a los LED y al zumbador. Los códigos asociados a los LED comienzan con `A` seguido del número del LED y un `0` ó `1` según queramos apagarlo o encenderlo, mientras que activamos el *buzzer* con `B` seguido del número de zumbidos que queremos que haga:

```
para semaforo
  haz "estado enviatcp "localhost [$A40%]  # apaga verde
  haz "estado enviatcp "localhost [$A50%]  # apaga amarillo
  haz "estado enviatcp "localhost [$A31%]  # enciende rojo
  haz "estado enviatcp "localhost [$B1%]   # un beep (semaforo para ciegos)
  espera 100
#
  haz "estado enviatcp "localhost [$A30%]  # apaga rojo
  haz "estado enviatcp "localhost [$A50%]  # apaga amarillo
  haz "estado enviatcp "localhost [$A41%]  # enciende verde
  haz "estado enviatcp "localhost [$B2%]   # dos beep (semaforo para ciegos)
```

```
  espera 100
#
  haz "estado enviatcp "localhost [$A30%]  # apaga rojo
  haz "estado enviatcp "localhost [$A40%]  # apaga verde
  haz "estado enviatcp "localhost [$A51%]  # enciende amarillo
  haz "estado enviatcp "localhost [$B3%]   # tres beep (semaforo para ciegos)
  espera 30
#
  semaforo                                 # recursivo, volvemos a empezar
fin
```

Simple, ¿no?

### Alarma

Creemos un botón de alarma. Al pulsarlo, aparecerá escrito en pantalla un aviso:

**ALARMA!**

En este programa vamos a activar la zona de los pulsadores y leer la respuesta que da la controladora. El código asociado a los pulsadores lleva asociada la letra `D`. El programa resulta ser:

```
para alarma
  borrapantalla
  poncolorlapiz 1                          # color lapiz -> rojo
  ponfuente 30
  repite 1000
    [ haz "puls enviatcp "localhost [$D0%]  # lee pulsadores
      si :puls= [$d1000%]                   # respuesta
        [ rotula "ALARMA!
          ocultatortuga
          alto ]
      espera 10 ]
fin
```

La tortuga rotulará en pantalla `ALARMA!` al apretar uno de los pulsadores de la controladora (concretamente, el SW1).

### Moviendo a la tortuga

En la misma línea del anterior, vamos a utilizar un programa recursivo con el que controlemos los movimientos de la tortuga mediante los cuatro pulsadores de la TortuRob, de modo similar al programa de ejemplo de la sección 3.1 en la página 20:

- El botón 1 hará que la tortuga avance 10 pasos.
- El botón 2 hará que la tortuga se dé la vuelta.
- El botón 3 hará que la tortuga gire 90 grados a la izquierda.
- El botón 4 hará que la tortuga gire 90 grados a la derecha.

```
para tortu
  haz "puls enviatcp "localhost [$D0%]  # lee pulsadores
  si :puls = [$d1000%]                  # Pulsaron el boton 1
    [ avanza 10 espera 5 ]      # esperamos 5/60 seg para evitar rebotes
  si :puls = [$d0100%]                  # Pulsaron el boton 2
    [ giraderecha 180 espera 5 ]
  si :puls = [$d0010%]                  # Pulsaron el boton 3
    [ giraizquierda 90 espera 5 ]
  si :puls = [$d0001%]                  # Pulsaron el boton 4
    [ giraderecha 90 espera 5 ]
  tortu
fin
```

Vemos que las respuestas de la controladora son muy simples: tras inicializarla con `$D0%`, las respuestas vienen en la posición correspondiente al pulsador, y es un `1` ó un `0` según esté abierto o cerrado.

### 19.3.4. Los comandos de la TortuRob

Hemos visto tres ejemplos sencillos de la controladora. Veamos ahora todo lo que puede hacer. Recordemos:

1. La placa TortuRob nunca inicia una comunicación, solo responde.
2. Todo comando enviado recibe una respuesta.
3. Todos los comandos y respuestas tienen la estructura: `$Lx...x%[CR]`, donde:

   `$` → Identificador de inicio de comando.

   `%` → Identificador de fin de comando.

   `L` **y** `x...x` → `L` es una letra que identifica la acción (mayúscula si es comando, minúscula si es respuesta), mientras que `x...x` son los datos asociados a las órdenes/respuestas anteriores y su longitud es variable.

   Los valores que pueden tomar son:

   `$Gssssssss%` → Apaga o activa las salidas correspondientes:

   - `s1`: Apaga/activa la salida 1 (CO1, pin 3).

- `s2`: Apaga/activa la salida 2 (CO2, pin 4).
- `s3`: Apaga/activa la salida 3, LED rojo (CO1, pin 5).
- `s4`: Apaga/activa la salida 4, LED verde (CO1, pin 6).
- `s5`: Apaga/activa la salida 5, LED amarillo (CO1, pin 7).
- `s6`: Apaga/activa la salida 6, LED azul (CO1, pin 8).
- `s7`: Apaga/activa la salida 7, *buzzer*.

`s` puede tomar los valores

`0:` apagar.

`1:` encender permanentemente.

`a-z:` encender durante un tiempo que depende de la letra introducida (desde medio segundo a 8 segundos).

La respuesta a las órdenes anteriores viene en la forma:

- `$0%`: El comando se ejecutó correctamente.
- `$1%`: Error, no se pudo ejecutar el comando.

**Ejemplo**: `$G00100dy%`:

– Apaga las salidas 1, 2, 4 y 5.

– Activa la salida 3 de forma permanente.

– Activa la salida 6 durante `d` fracciones de segundo.

– Activa la salida 7 durante `y` fracciones de segundo.

`$Asx%` → Apaga o activa la salida `s` durante `x` fracciones de segundo. El valor de `s` puede ser:

- `1`: Salida 1 (CO1, pin 3).
- `2`: Salida 2 (CO2, pin 4).
- `3`: Salida 3, LED rojo (CO1, pin 5).
- `4`: Salida 4, LED verde (CO1, pin 6).
- `5`: Salida 5, LED amarillo (CO1, pin 7).
- `6`: Salida 6, LED azul (CO1, pin 8).
- `7`: Salida 7, *buzzer*.

mientras que `x` puede tomar los valores:

`0:` Apaga la salida.

`1:` Enciende la salida permanentemente.

`a-z:` Enciende la salida un tiempo que oscila entre medio segundo y 8 segundos.

La respuesta a las órdenes anteriores viene, como antes, en la forma:

- `$0%`: El comando se ejecutó correctamente
- `$1%`: Error, no se pudo ejecutar el comando.

**Ejemplo**: `$A4y%`: Activa el LED verde durante `y` fracciones de segundo.

`$Bn%` → Activa el *buzzer* o zumbador, `n` veces.
La respuesta es, de nuevo:

- `$0%`: El comando se ejecutó correctamente
- `$1%`: Error, no se pudo ejecutar el comando.

**Ejemplo**: `$B3%`: El *buzzer* emite 3 *beeps*.

`$Dn%` → Lee el estado de entradas y salidas. `n` puede ser:

`0:` Grupo de 4 pulsadores o conector CO2.

`1:` Contenido del contador binario asociado a CO2, pin1 (SW1).

`2:` Estado de salidas activadas.

La respuesta viene precedida de una `E`:

- `$E%`: para cualquier valor de `n` si hubo error.
- `$dwxyz%`: si `n` fue 0. `wxyz` hacen referencia a los pulsadores SW1, SW2, SW3 y SW4 (o a masa CO2, pin1, pin2, pin3 y pin4) respectivamente, y serán `0` si el pulsador está cerrado y `1` si está abierto.
- `$cxyz%`: si `n` fue 1, `xyz` es un valor entre `000` y `255`, representando el valor del contador binario asociado a CO2, pin1 (SW1).
- `$assssssss%`: si `n` fue 2, hace referencia a las 7 salidas ya explicadas, y seá `0` si la salida correspondiente está apagada y `1` si está activa.

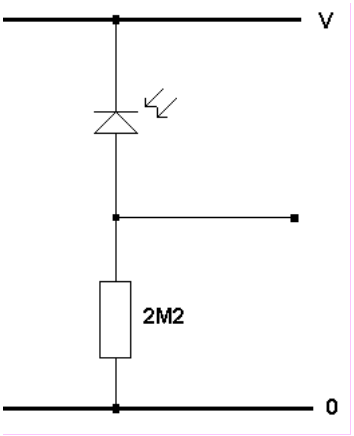
**Ejemplos**:
`$d0010%`: Pulsadores SW1, SW2 y SW4 abiertos, SW3 cerrado.
`$c035%`: El contador registró 35 cierres de SW1 desde la última lectura.
`$a0010000%`: Solo la salida 3 está activa (*timer* 3 contando).

`[CR]` → Cierre. Lo asigna el sistema de comunicaciones (es decir, el *Enter*)

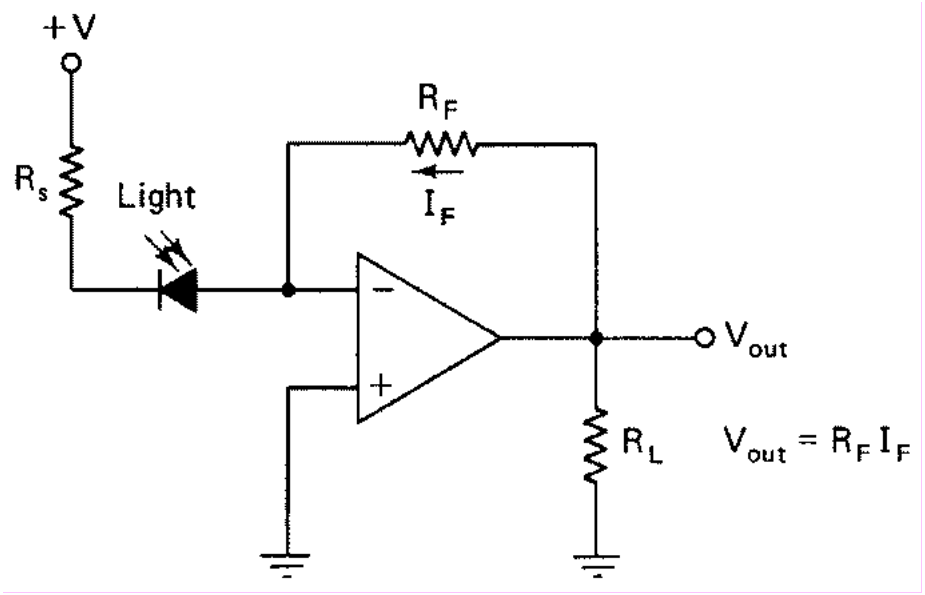
### Sigue la luz

Analicemos ahora un *proyecto* con mayor complejidad. Vamos a hacer que nuestra tortuga responda a estímulos luminosos (en una habitación oscura y con una linterna, por ejemplo), sustituyendo los pulsadores por fotodiodos.

El fotodiodo es un diodo que, conectado en inversa, genera una intensidad proporcional al nivel de luz recibida; aunque podríamos intentar disparar directamente los sensores mediante la intensidad del diodo, el esquema sería muy susceptible a la luz ambiente, por lo que es mejor conectarlo a masa mediante una resistencia *pull–down*, de forma que la tensión en la misma será directamente proporcional a la intensidad/luz recibida:




Por supuesto el concepto básico puede refinarse tanto como queramos, ya sea mejorando el circuito de disparo (ver figura) o sustituyendo el tipo de sensor de activación, por ejemplo haciendo que la tortuga reaccione a sonidos en lugar de luces:




La respuesta de la tortuga a los sensores deberá ser:

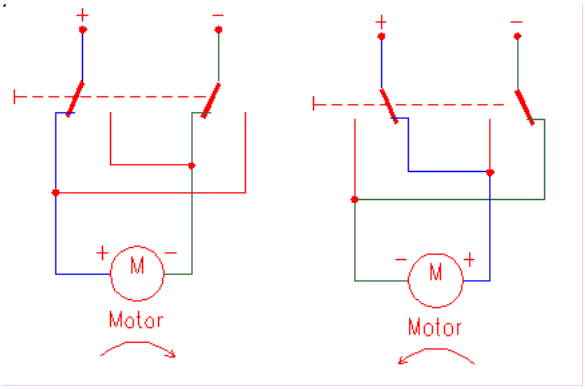
- Si le da la luz por un lateral, haremos que gire 90°.
- Si se ilumina por delante o por detrás, irá en esa dirección.

El programa se detendrá cuando se pulse la tecla “Escape”.

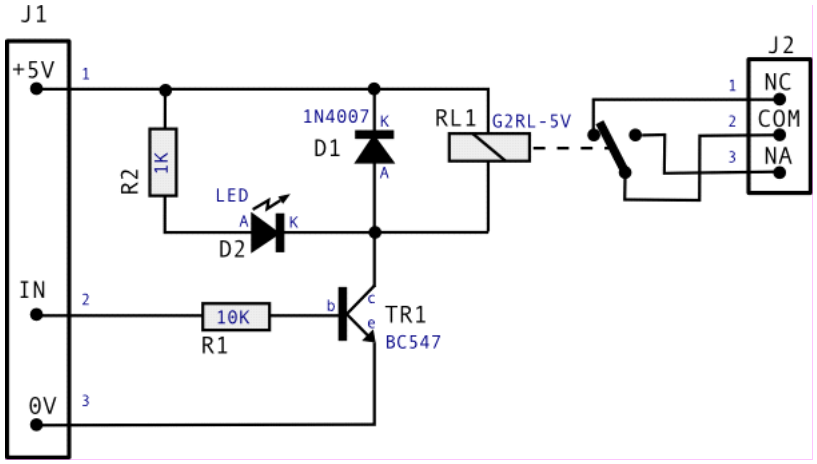
Necesitamos montar la placa en un robot que tenga una base con dos motores y cuatro sensores, uno a cada lado, otro en el frontal y el cuarto en la parte posterior.

Los motores se montarán a ambos lados de la base, de modo que:

- Activando los dos, se moverá hacia delante.
- Activando solo el derecho, girará a la izquierda y viceversa.
- Para lograr la marcha atrás, conectamos un inversor de giro en `CO1`. Invertiremos la fase de los motores con los dos contactos del relé (ver figura).




Como tenemos que controlar dos motores, y el inversor de giro afecta a ambos a la vez, los pondremos en paralelo. Además, conectaremos en serie con cada motor el contacto del relé que regula su puesta en marcha, permitiendo que trabajen independientemente.
Por último, y dado que no es conveniente intentar activar los relés directamente, deberemos agregar un transistor en modo interruptor a cada puerto de disparo, si es que el entrenador no está instalado.




Ha llegado el momento de realizar el programa. Elegimos que el LED "rojo" mueva el motor izquierdo y "LED verde" el derecho; encender ambos a la vez hace que el robot avance, y activando `CO1` llamamos al inversor de giro. Por tanto:

- el fotodiodo asociado al pulsador 1 irá detrás (marcha atrás),
- el fotodiodo 2 en el costado derecho (enciende rojo → motor izquierdo en solitario),

- el fotodiodo 3 en el lado izquierdo (enciende verde → motor derecho en solitario)
- el cuarto fotodiodo en el frontal (ambos encendidos → hacia delante).

Traducido a lenguaje TortuRob:

`[$d1000%]` → `[$Ga0aa000%]`: Pulsador 1, activa CO1, rojo y verde

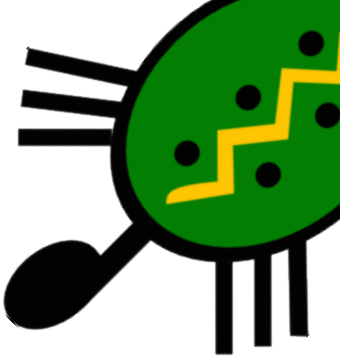
`[$d0100%]` → `[$G00a0000%]`: Pulsador 2, activa rojo

`[$d0010%]` → `[$G000a000%]`: Pulsador 3: activa verde

`[$d0001%]` → `[$G00aa000%]`: Pulsador 4: activa rojo y verde

El programa resulta::

```
para sigueluz
  haz "puls enviatcp "localhost [$D0%]
  si :puls = [$d1000%]
    [ es enviatcp "localhost [$Ga0aa000%] ]      # Activar inversion
  si :puls = [$d0100%]
    [ es enviatcp "localhost [$G00a0000%] ]      # Activar izquierdo
  si :puls = [$d0010%]
    [ es enviatcp "localhost [$G000a000%] ]      # Activar derecho
  si :puls = [$d0001%]
    [ es enviatcp "localhost [$G00aa000%] ]      # Activar ambos
  si tecla? [ si leetecla = 27 [alto] ]
  espera 10
  sigueluz
fin
```



# Capítulo 20

# Acerca de xLogo

## 20.1. El sitio

Para conseguir la última versión y corrección de errores, visita el sitio de xLogo de vez en cuando:

`http://xlogo.tuxfamily.org`

Siéntete libre de contactar al autor si tienes algún problema con la instalación o el uso. Toda sugerencia será bienvenida.

## 20.2. Acerca de este documento

### xLogo: Manual del Usuario

- Original en francés revisado por Loïc (8 de julio de 2009).
- Traducido al español por Marcelo Duschkin y Álvaro Valdés (9 de diciembre de 2010)
- Guy Walker: Traducción al inglés.
- Alexandre Soares: Traducción al Portugués.
- Miriam Abresch, Michael Malien. Traducción al Alemán
- Michel Gaillard. Traducción al Esperanto
- Marco Bascietto. Traducción al Italiano

## Agradecimientos


- Quiero empezar por agradecer a Loïc su fantástico trabajo con xLogo.
- También quiero enviar un sentido recuerdo a Marcelo Duschkin, fallecido en abril de este año 2010, que me animó y me dio todo su apoyo siempre que lo necesité.
- A todos los traductores/as del proyecto xLogo:
  - Inglés: Guy Walker
  - Portugués: Alexandre Soares
  - Alemán: Michael Malien, Miriam Abresch.
  - Esperanto: Michel Gaillard, Carlos Enrique Carleos Artime
  - Árabe: El Houcine Jarad
  - Gallego: Justo freire
  - Asturiano: Montserrat G.ª Fueyo
  - Griego: Anastasios Drakopoulos
  - Italiano: Marco Bascietto
  - Catalán: David Arso
  - Húngaro: József Varga
- José Higinio Cernuda Menéndez, por su ayuda con el tema de Robótica.
- Olivier SC: por sus sugerencias, y por las invalorables pruebas que permitieron depurar el intérprete xLogo.
- Daniel Ajoy, por sus sugerencias en cuanto a la compatibilidad de las primitivas en español y su valiosa colaboración en pruebas de esa versión.
- Eitan Gurari, por la extensión LaTeX `tex4ht`, que permite convertir el manual desde TeX hacia varios formatos:

  `www.cse.ohio-state.edu/ gurari/TeX4ht`
- A varios proyectos de *software libre* muy importantes para poder mantener xLogo:
  - Java y Java3D:

    `https://java3d.dev.java.net`
  - Java Help:

    `http://java.sun.com/javase/technologies/desktop/javahelp`
  - Eclipse: `http://www.eclipse.org`
  - JLayer: `http://www.javazoom.net/javalayer/javalayer.html`
- Finalmente, un grandísimo **¡GRACIAS!** a TuxFamily, por la calidad de su alojamiento *web* y su importante contribución al *software libre*:

  `http://www.tuxfamily.org`

# Capítulo 21

# Carnaval de Preguntas – Artimañas – Trucos que conocer

## 21.1. Preguntas frecuentes

### Por más que borro un procedimiento en el Editor, reaparece todo el tiempo!

Cuando se sale del **Editor**, éste se limita únicamente a guardar o poner al día los procedimientos definidos en él. La única forma de borrar un procedimiento en xLogo es utilizar la primitiva `borra` o `bo`.

**Ejemplo**: `borra "toto` → borra el procedimiento `toto`

### ¿Cómo hago para escribir rápidamente una orden utilizada previamente?

- Primer método: con el ratón, haz *click* en la zona del **Histórico** sobre la línea deseada. Así reaparecerá inmediatamente en la **Línea de Comando**

- Segundo método: con el teclado, las flechas Arriba y Abajo permiten navegar por la lista de los comandos anteriores (más práctico)

### En la opción Sonido del cuadro de diálogo Preferencias, no hay disponible ningún instrumento

Mira la solución en la sección del capítulo sobre música, 1.2.1

### Utilizo la versión en Esperanto, pero no puedo escribir los caracteres especiales

Cuando escribes en la **Línea de comandos** o en el **Editor**, si haces *click* con el botón derecho del ratón, aparece un menú contextual. En ese menú se encuentran las funciones habituales de Edición (copiar, cortar, pegar) y los caracteres especiales del Esperanto cuando se selecciona ese idioma.

### Utilizo la versión en Español, y no puedo utilizar las primitivas `animacion`, `division`, `separacion` y `ponseparacion`

Corregido desde la versión `0.9.20e`. Para versiones anteriores de xLogo:

- `animacion`, se escribe `animacicn`, con "`c`" en lugar de "`o`"
- `division`, `separacion` y `ponseparacion` se escriben con tilde: `división`, `separación` y `ponseparación`.

Obviamente, el mejor consejo es que actualices a la versión más moderna de xLogo.

### Uso Windows XP y tengo correctamente instalado y configurado el JRE; pero hago doble *clic* en el icono de xLogo y ¡no pasa nada!

Debes estar usando una versión muy antigua de xLogo. Dos opciones:

- Utiliza una versión más actual de `xlogo.jar`.
- Si presionas `Alt+Contrl+Supr` y en el **Gestor de Procesos** "matas" el correspondiente a `javaw`, se inicia xLogo. Desde ese momento, funciona correctamente haciendo "doble *click*" sobre el icono del archivo `xlogo.jar`.

## 21.2. ¿Cómo puedo ayudar?

- Informándome de todos los errores ("*bugs*") que encuentres. Si puedes reproducir sistemáticamente un problema detectado, mejor aún
- Toda sugerencia dirigida a mejorar el programa es siempre bienvenida
- Ayudando con las traducciones, en particular el inglés ...
- Las palabras de ánimo siempre vienen bien

# Capítulo 22

# Configuración avanzada de xLogo

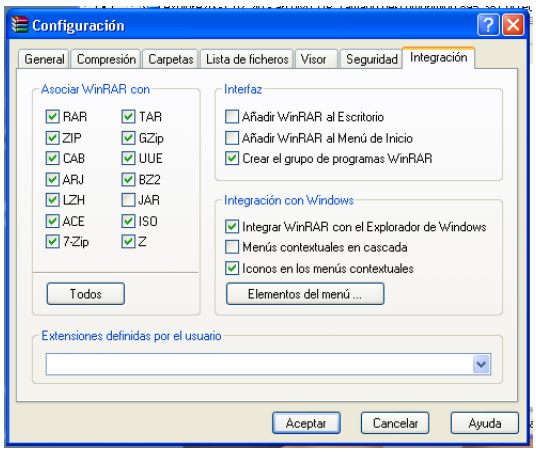
En la sección 1.2 ya vimos las formas rápidas de configurar xLogo en Windows, Linux y Mac. A veces en Windows se necesita una mayor intervención para configurarlo.

## 22.1. Asociar los archivos `.jar` con Java

Si tu ordenador abre el WinZip (o similar), entonces debes modificar la asociación de este programa con los archivos `.jar`

### 22.1.1. Configuración del compresor

Configurar WinZIP o WinRAR es fácil, sólo tienes que buscar la opción *Asociaciones de Archivo* o*Integración* y deshabilitar la casilla asociada a la extensión `.jar`:




Después de esto, si Java está correctamente instalado, debería ejecutarse con cualquiera de las opciones explicadas en 1.2.1

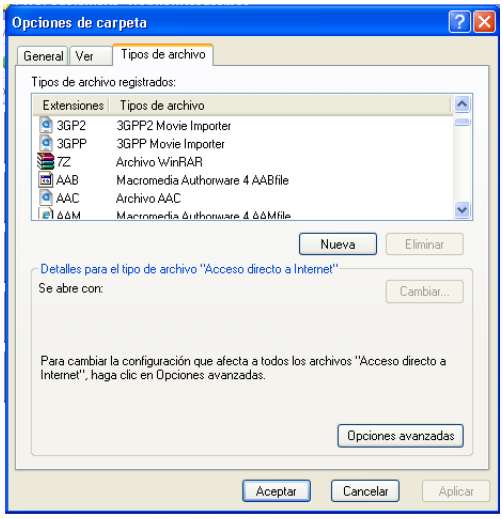
### 22.1.2. Configuración de Windows

Si aún no consigues que XLOGO funcione, debes seguir los siguientes pasos (que pueden variar según que versión de Windows tengas):

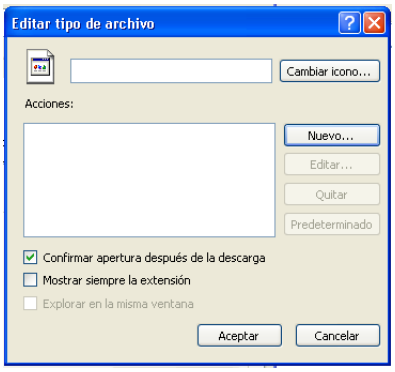
Si usas Windows XP, en primer lugar, debes tener visible una opción explorador que sólo se muestra en la *Vista Clásica*:

*Inicio* → *Panel de Control* → *Cambiar a Vista Clásica*

Desde este momento, en cualquier ventana del Explorador de Windows puedes hacer *clic* en el menú *Opciones de carpeta* → *Tipos de Archivo*:




1. Busca en la lista de archivos registrados la asociación a los `.jar`
2. *Clic* en *Tipo de archivo*, luego *opciones Avanzadas ...*
3. En la ventana que se abre, *clic* en *Abrir*, luego *Editar*




4. *Clic* en *Examinar ...* y navegar hasta encontrar `javaw.exe` que podría estar en

```
c:\Archivos de Programa\java\jre6\bin\javaw.exe
```

5. Haciendo doble *clic* en el archivo, esa ruta aparecerá en el campo *Aplicación usada ...*

6. Finalmente, cierra todos los cuadros de diálogo. Ahora debe ser posible ejecutar xLogo haciendo doble *clic* en su icono

Si estos pasos no funcionan, existe otra posibilidad. Abre una ventana MS-DOS (en Windows XP: Inicio → Todos los Programas → Accesorios → Símbolo del Sistema), y teclea la orden siguiente:

```
c:\Archivos de Programa\java\jre6\bin\javaw -jar \ubicacion_de_xLogo\xlogo.jar
```

Por ejemplo, si eliges descargar xLogo en el Escritorio, en Windows XP sería:

```
c:\Documents and settings\usuario\Escritorio\xlogo.jar
```

Para no repetir este proceso cada vez que quieras ejecutar xLogo, escríbelo en un archivo de texto y guárdalo como `xlogo.bat`. Ahora, simplemente haciendo doble *clic* en este archivo, arrancará xLogo.

## 22.2. Asociar archivos `.lgo` con xLogo

### 22.2.1. En Windows

Los archivos `.lgo` no son reconocidos inicialmente por el ordenador, y si haces doble *clic* sobre ellos, aparecerá un cuadro de diálogo preguntando con qué aplicación abrirlos.

Tienes que elegir *Otros ...* y poner la ruta a `javaw.exe` que, como dijimos ya varias veces, podría estar en

```
c:\Archivos de Programa\java\jre6\bin\javaw.exe
```

Deberás indicar un nombre para designar los archivos `.lgo`, por ejemplo: `Archivos Logo`.

Para que esto quede registrado en Windows, debes seguir los siguientes pasos (que pueden variar según que versión de Windows tengas):

1. *Inicio* ⟶ *Panel de Control* ⟶ *Cambiar a Vista Clásica* ⟶ *Opciones de carpeta*

2. *Clic* en *Tipos de archivos*

3. Buscar en la lista de archivos registrados la asociación a los `.jar`

4. *Clic* en *Tipo de archivo*, luego *Nuevo*

5. Escribe la extensión `.jar` en el cuadro *Extensión de Archivo*, luego *OK*

6. *Clic* en el recién agregado *LOG* y luego *clic* en *Avanzado ...*

7. Aparece una ventana, *clic* en *Nuevo ...*

8. En *Acción*, poner *Abrir*, y hacer *clic* en *Examinar ...* navegar hasta encontrar `javaw.exe` que podría estar en

   `c:\Archivos de Programa\java\jre6\bin\javaw.exe`

9. *Clic* en *Abrir* y agregar la ruta en el cuadro de *Acción* de *Editar tipo de archivo.*

10. *Clic* en *Abrir*, luego en *Editar*

11. Esa ruta debe aparecer en el campo *Aplicación usada ...* Lo que tienes que hacer es completar la línea para que se vea:

    `"c:\Archivos de Programa\java\jre6\bin\javaw.exe -jar xlogo.jar" "%1" %*`

    (El espacio a ambos lados de `-jar` es importante)

12. Finalmente, cierras todos los cuadros de diálogo. Ahora debería ser posible abrir los archivos `.lgo` con XLOGO

### 22.2.2. En Linux

Como verás, y contrariamente a lo que se dice, la configuración es mucho más fácil e intuitiva (y las cosas funcionan mejor)

#### En KDE

#### Crear un icono de acceso directo a xLogo

1. *Clic* con el botón derecho en el Escritorio: *Crear Nuevo* $\longrightarrow$*Enlace a Aplicación*

2. En la pestaña *General*, elige el nombre. XLOGO es una buena opción

3. En la pestaña *Aplicación*, navega hasta *Comando*, y escribe:

   ```
   java -jar /home/tu_nombre/xlogo.jar
   ```

   También puedes rellenar los campos *Descripción*, *Comentario* y *Ruta de trabajo*. Por ejemplo:

   a) *Descripción*: xLogo
   b) *Comentario*: Intérprete Java de Logo
   c) *Ruta de trabajo*: `/home/tu_nombre/`

4. Finalmente cierra el cuadro de diálogo. Ahora debe ser posible ejecutar XLOGO haciendo doble *clic* en su icono, que puedes elegir y modificar si no te gusta el predeterminado.

**Asociar los archivos `.lgo` con xLogo**

1. En el Gestor de Archivos (**Konqueror**), haz *clic* con el botón derecho sobre un archivo `.lgo`
2. Selecciona *Editar tipo de Archivo*
3. En *Aplicación*, *Orden de preferencia*, escribe:

   ```
   java -jar /home/tu_nombre/xlogo.jar
   ```

   También puedes poner un icono para asociar el tipo de archivos.

**En GNOME**

**Crear un icono de acceso directo a xLogo**

1. *Clic* con el botón derecho en el Escritorio: *Crear lanzador*
2. En el cuadro de diálogo, completa los apartados:
   - *Nombre:* xLogo es una buena opción
   - *Comentario*: si quieres una pequeña explicación, que aparecerá en forma de globo al pasar el ratón por encima del icono.

     Por ejemplo: Intérprete Java de Logo
   - *Comando*:

     ```
     java -jar /home/tu_nombre/xlogo.jar
     ```
   - *Icono*: Elige entre los del sistema o uno personalizado
3. Finalmente cierra el cuadro de diálogo. Ahora debe ser posible ejecutar xLogo haciendo doble *clic* en su icono.

**Asociar los archivos `.lgo` con xLogo**

1. En el Gestor de Archivos (**Nautilus**), haz *clic* con el botón derecho sobre un archivo `.lgo`
2. Selecciona *Abrir con otra Aplicación*
3. Selecciona *Utiliza un Comando Personalizado* y escribe:

   ```
   java -jar /home/tu_nombre/xlogo.jar
   ```

# Bibliografía


[1] Manual de referencia de xLogo. Autor: **Loïc Le Coq**. Traducción: **Marcelo Duschkin y Álvaro Valdés**.

`http://downloads.tuxfamily.org/xlogo/downloads-sp/manual-sp.pdf`

[2] Diez lecciones para descubrir el lenguaje xLogo. Autor: **Loïc Le Coq**. Traducción: **Álvaro Valdés**.

`http://downloads.tuxfamily.org/xlogo/downloads-sp/tutorial.pdf`

[3] Wikipedia

`http://es.wikipedia.org`

[4] Paraíso de Logo. Autor: **Daniel Ajoy**.

`http://neoparaiso.com/logo/`

[5] Cuaderno de ejercicios de Logo. Autor: **Pedro José Fernández**.

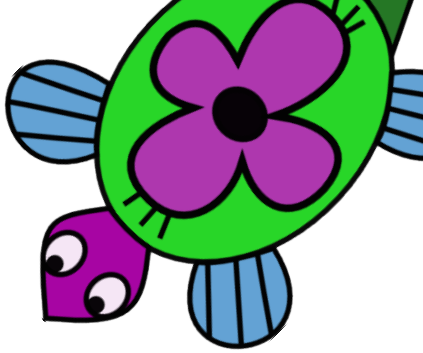
`http://neoparaiso.com/logo/documentos/ejercicios-logo.pdf`

[6] Informática y Matemática. Asignatura optativa (y libre configuración). Autores: **Eugenio Roanes Macías** y **Eugenio Roanes Lozano**

`http://www.ucm.es/info/secdealg/ApuntesLogo/INF_MATN_2006.PDF`

[7] Introducción a la Programación con Python. Autor: **Bartolomé Sintes Marco**

`http://www.mclibre.org/consultar/python/`



# Apéndice A

# Iniciando xLogo desde la línea de comandos

La sintaxis del comando que ejecuta xLogo es:

```
java-jar xlogo.jar [-a] [-lang en] [archivo1.lgo archivo2.lgo ...]
```

Las opciones disponibles son:

- El atributo `-lang`: este atributo especifica el idioma con que ejecutar xLogo.

  El parámetro indicado con esta opción se superpone al existente en el fichero `.xlogo` de configuración personal. En la siguiente tabla se muestran los parámetros asociados a todos los idiomas disponibles:

| Francés | Inglés | Español | Alemán | Portugués |
|---|---|---|---|---|
| `fr` | `en` | `es` | `de` | `pt` |

| Árabe | Esperanto | Gallego | Asturiano | Griego |
|---|---|---|---|---|
| `ar` | `eo` | `gl` | `as` | `el` |

| Italiano | Catalán | Húngaro | | |
|---|---|---|---|---|
| `it` | `ca` | `hu` | | |

- Atributo `-a`: este atributo determina que se ejecute el **Comando de Inicio** (ver sección 1.7) que figura en el archivo cargado sin necesidad de pulsar el botón de inicio (sección 1.6) al iniciarse xLogo.
- Atributo `-memory`: este atributo cambia el tamaño de memoria reservado para xLogo
- `[archivo1.lgo archivo2.lgo ...]`: estos archivos de extensión `.lgo` se cargan al iniciar xLogo.

  Los archivos pueden ser locales (estar en el disco duro del ordenador) o remotos (en la *web*). Por lo tanto, se puede especificar una dirección local o una dirección web.

Veamos ejemplos:

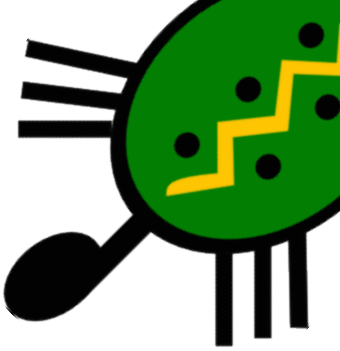
- `java -jar xlogo.jar -lang es prog.lgo`

  los archivos `xlogo.jar` y `prog.lgo` están en el directorio actual. Este comando ejecuta xLogo cambiando el idioma a "español" y a continuación carga el fichero `prog.lgo`. Por tanto, es necesario que este archivo sea un programa Logo escrito en Español.

- `java -jar xlogo.jar -a -lang en http://xlogo.tuxfamily.org/prog.lgo`:

  Este comando ejecuta xLogo en Inglés, estando `xlogo.jar` en el directorio de trabajo, y se carga el fichero `http://xlogo.tuxfamily.org/prog.lgo` desde la *web*.

  Por último, el Comando de Inicio de este fichero se ejecuta en el arranque.



# Apéndice B

# Ejecutando xLogo desde la *web*

## B.1. El problema

Supongamos que usted es administrador/a de un sitio *web*. En este sitio, habla de xLogo y quiere ofrecer algunos de los programas que ha creado.

Podría distribuir el fichero Logo en formato `.lgo` y esperar que los visitantes descarguen primero xLogo, después el programa y, por último, lo carguen para ver qué hace.

La alternativa es que el usuario pueda ejecutar xLogo *en línea*, y probar su programa sin apenas esfuerzo.

Para lanzar xLogo desde una página *web*, utilizaremos la tecnología Java Web Start. Con ello, sólo necesita poner en su web un enlace hacia un archivo con extensión `.jnlp`, que iniciará la ejecución de xLogo.

## B.2. Cómo crear un fichero `.jnlp`

Veamos cómo hacerlo con un archivo de ejemplo. De hecho, el siguiente ejemplo es el que se usa en la sección "**Ejemplos**" de la página de xLogo en francés.

Este archivo carga el programa que dibuja un dado en la sección 3D. Veamos primero el archivo, y veremos después las explicaciones sobre su contenido.

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.5+" codebase="http://downloads.tuxfamily.org/xlogo/common/webstart">
<information>
  <title>xLogo</title>
  <vendor>xlogo.tuxfamily.org</vendor>
  <homepage href="http://xlogo.tuxfamily.org"/>
  <description>Logo Programming Language</description>
```

```
  <offline-allowed/>
</information>

<security>
  <all-permissions/>
</security>

<resources>
  <j2se version="1.4+"/>
  <jar href="xlogo.jar"/>
</resources>

<application-desc main-class="Lanceur">
  <argument>-lang</argument>
  <argument>fr</argument>
  <argument>-a</argument>
  <argument>http://xlogo.tuxfamily.org/fr/html/examples-fr/3d/de.lgo</argument>
</application-desc>
</jnlp>
```

Este archivo está escrito en formato XML, y las cuatro líneas más importantes son:

```
  <argument>-lang</argument>
  <argument>fr</argument>
  <argument>-a</argument>
  <argument>http://xlogo.tuxfamily.org/fr/html/examples-fr/3d/de.lgo</argument>
```

Esas líneas especifican los parámetros de inicio de xLogo:

- Las líneas 1 y 2 fuerzan a xLogo a ejecutarse en francés.
- La última línea indica la ruta del fichero a cargar.
- La línea 3 indica que se ejecute el Comando de Inicio de este programa al iniciarse xLogo.

Una última sugerencia (petición): Dado que el servidor de Tuxfamily no puede aceptar "infinitas" conexiones, le pedimos que tenga una copia del archivo `xlogo.jar` en su web. Para ello, cambie la dirección en la segunda línea del archivo `.jnlp`, donde dice `codebase=` por:

```
<jnlp spec="1.5+" codebase="http://mi.direccion.web/directorio/xLogo">
```

# Apéndice C

# Programación Estructurada y Diseño Modular

A finales de los años sesenta surgió una nueva forma de programar que no solamente daba lugar a programas fiables y eficientes, sino que además estaban escritos de manera que facilitaba su comprensión posterior.

## C.1. Programación Estructurada

El **Teorema de Dijkstra**, demostrado por Edsger Dijkstra en los años sesenta, demuestra que todo programa puede escribirse utilizando únicamente las tres instrucciones de control siguientes:

- Secuencia
- Instrucción condicional.
- Iteración (bucle de instrucciones) con condición inicial

Sólamente con estas tres estructuras se pueden escribir programas, si bien ya hemos visto que existen otras estructuras de control: los procedimientos.

## C.2. Diseño Modular

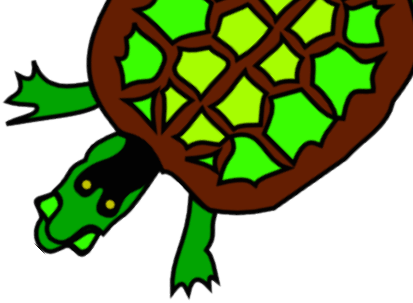
El uso de procedimientos y subprocedimientos permite dividir un problema grande en *subproblemas* más pequeños. Si el objetivo es elaborar un programa para resolver dicho problema grande, cada subproblema (menos complejo) podrá ser resuelto por un *módulo* (subprocedimiento) relativamente fácil de implementar

¿Por qué descomponer un problema en partes? Experimentalmente está comprobado que:

- Un problema complejo cuesta más de resolver que otro más sencillo

- La complejidad de un problema global es mayor que la suma de las complejidades de cada una de sus partes por separado.

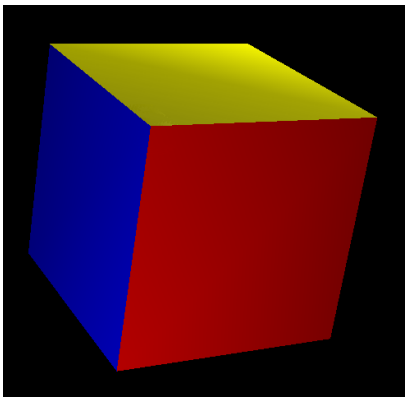
La combinación de las instrucciones de control con los procedimientos permite diseñar programas de forma rápida, sencilla y eficiente.



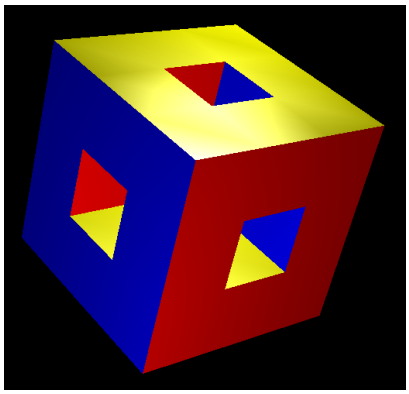
# Apéndice D

# La esponja de Menger

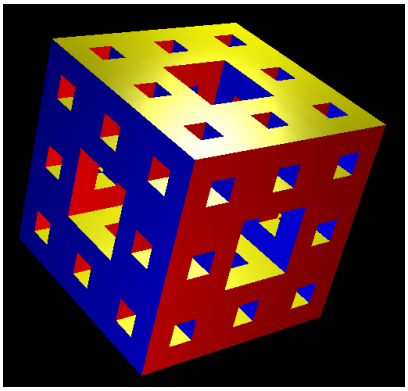
En este capítulo vamos a construir un sólido fractal llamado **la esponja de Menger**, cuyas primeras iteraciones son:



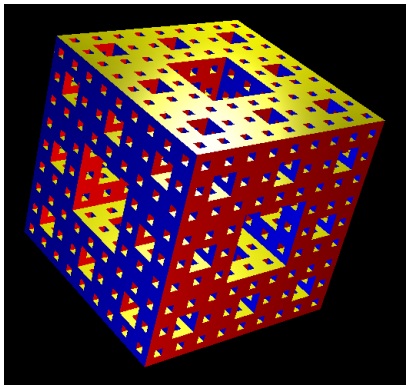
**Estado inicial (paso 0)**



**Paso 1**



**Paso 2**



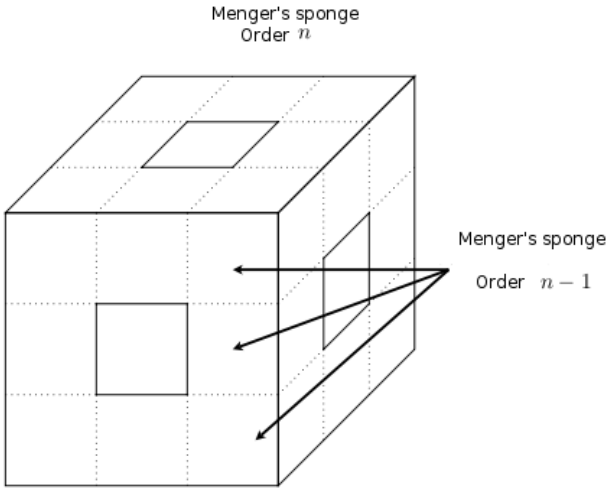
**Paso 3**

Este tema se divide en dos partes:

- Iniciciación, donde mostramos cómo crear el sólido usando la recursividad.
- Desarrollo y optimización, donde dibujaremos una esponja de Menger de orden 4.

## D.1. Utilizando recursividad

Consideremos una esponja de Menger de orden $n$ cuyo lado mide $L$.




El dibujo muestra que, realmente, esta *esponja* está formada por 20 esponjas de Menger de orden $n-1$, cada una de lado $\frac{L}{3}$. Hemos encontrado la estructura recursiva de la esponja.

**El programa**

```
para cubo :l
  si :contador=10000 [vistapoligono]
  # color de las caras laterales
  hazlocal "colores [amarillo magenta cyan azul]
# caras laterales
  repite 4
    [ poncolorlapiz ejecuta elemento contador :colores
      cuadrado :l giraderecha 90 avanza :l
      giraizquierda 90 balanceaderecha 90]
  poncolorlapiz rojo
  cabeceaabajo 90 cuadrado :l cabeceaarriba 90
  avanza :l cabeceaabajo 90
  poncolorlapiz verde
  cuadrado :l cabeceaarriba 90 retrocede :l
fin

para cuadrado :c
  haz "contador :contador+1
  empiezapoligono
    repite 4 [avanza :c giraderecha 90]
  finpoligono
fin
```

```
# Esponja de Menger
# p: nivel de recursividad
# l: arista del cubo final (grande)
para menger :l :p
  si :p=0 [cubo :l]
   [ hazlocal "p :p-1
     hazlocal "l :l/3
     repite 3 [ menger :l :p avanza :l ]
     retrocede 3*:l
     giraderecha 90 avanza :l giraizquierda 90
     menger :l :p avanza 2*:l menger :l :p retrocede 2*:l
     giraderecha 90 avanza :l giraizquierda 90
     repite 3 [menger :l :p avanza :l] retrocede 3*:l
  # lado derecho
    cabeceaabajo 90 avanza :l cabeceaarriba 90
     menger :l :p  avanza 2*:l menger :l :p retrocede 2*:l
     cabeceaabajo 90 avanza :l cabeceaarriba 90
     repite 3 [menger :l :p avanza :l]
     retrocede 3*:l giraizquierda 90 avanza :l giraderecha 90
     menger :l :p  avanza 2*:l menger :l :p retrocede 2*:l
     giraizquierda 90 avanza :l giraderecha 90
     repite 3 [menger :l :p avanza :l]
     retrocede 3*:l cabeceaabajo 90 retrocede :l cabeceaarriba 90
     menger :l :p  avanza 2*:l menger :l :p retrocede 2*:l
     cabeceaabajo 90 retrocede :l cabeceaarriba 90
   ]
fin

para esponja :p
  borrapantalla ocultatortuga
  haz "contador 0
  perspectiva poncolorpapel 0
  menger 800 :p
  mecanografia [Numero de poligonos: ] escribe :contador
  vistapoligono
fin
```

Este programa consta de cuatro procedimientos:

- `cuadrado :c`

  Este procedimiento dibuja un cuadrado de lado `:c`, almacenándolo para dibujarlo en 3-D. La variable `contador` guardará el número de polígonos dibujados.

- `cubo :l`

  Este procedimiento dibuja un cubo de arista `:l`, llamando al procedimiento `cuadrado`.

- `menger :l :p`

  Llegamos al más importante del programa; dibuja el *motivo* de Menger de orden $p$ y lado $l$. Este *motivo* se crea de modo recursivo, como acabamos de explicar.

- `esponja :p`

  Crea una esponja de Menger de orden $p$ y lado 800 y la dibuja en el visor 3D.

La mayor dificultad de este programa proviene de su consumo de memoria. La cantidad de memoria asignada por defecto para xLogo (ver pág. 17) no permite dibujar una esponja de orden 3 (`esponja 3`), ya que requiere dibujar **48 000** polígonos. Debemos aumentar la memoria asignada a xLogo hasta un mínimo de 256 Mb ... u optimizar el código.

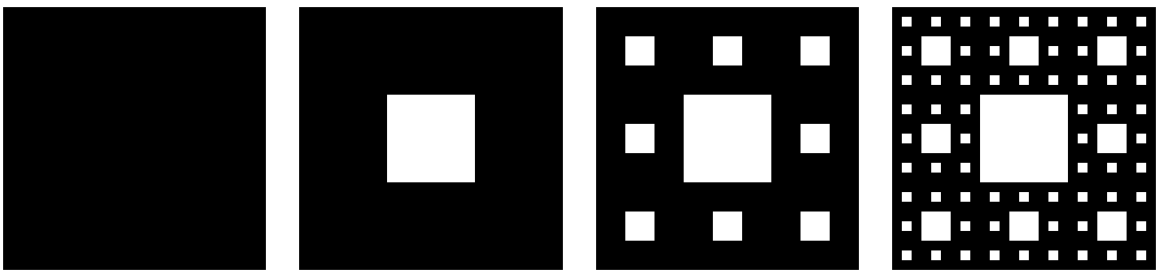
## D.2. Segunda aproximación. Sólido de orden 4

La principal característica del programa anterior es su utilización de la estructura recursiva del sólido fractal. podemos observar que es similar al método utilizado para dibujar el copo de Koch en la página 100. La principal ventaja de utilizar recursividad es que el código del program es el más pequeño posible. A cambio, la principal desventaja es el número de polígonos que se deben crear que, como dijimos, alcanza los 48.000. Esto hace inviable plantearse una esponja de orden superior, pues los requisitos de memoria crecerán muy rápido.

Si queremos dibujar una esponja de Menger de orden 4, debemos replanearnos el programa y olvidar la recursividad.

### D.2.1. La alfombra de Sierpinski

La esponja de Menger es la generalización en 3 dimensiones de una figura plana llamada **la alfombra de Sierpinski**, cuyas primeras iteraciones son:



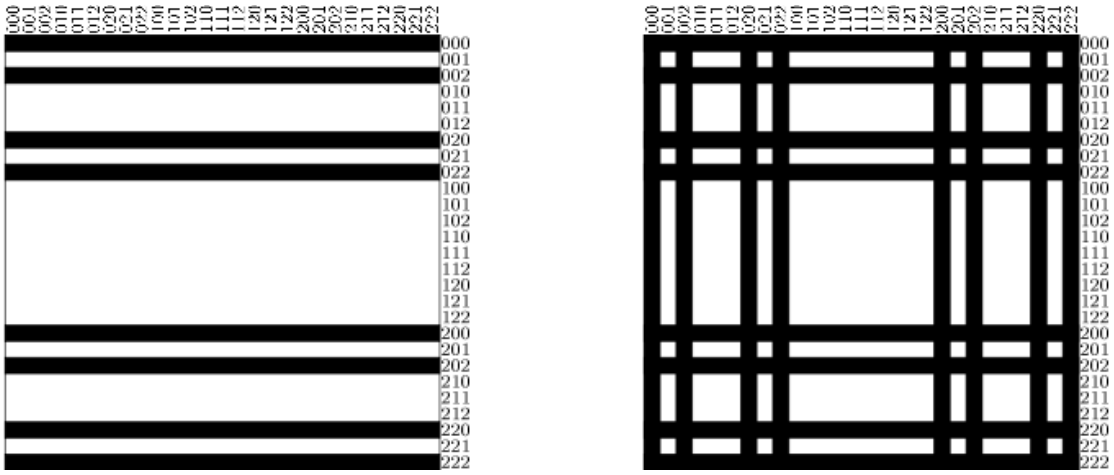
**Paso 0** **Paso 1** **Paso 2** **Paso 3**

Observamos que cada cara de la esponja de Menger es una alfombra de Sierpinski.

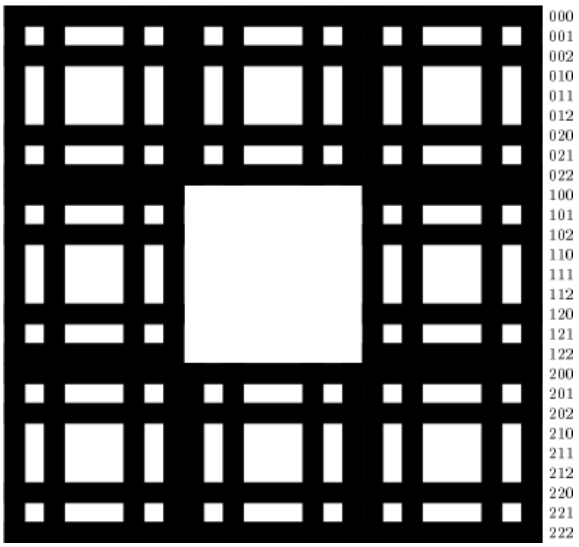
### D.2.2. Dibujando una alfombra de Sierpinski de orden $p$

El objetivo es reducir al mímino el número de polígonos necesarios para dibujar una alfombra de Sierpinski. El ejemplo explica cómo dibujar una alfombra de Sierpinski de orden 3. Aquí, el cuadrado inicial consta de $3^3 = 27$ filas y 27 columnas. Escribimos en base 3 el número de cada fila y cada columna.:

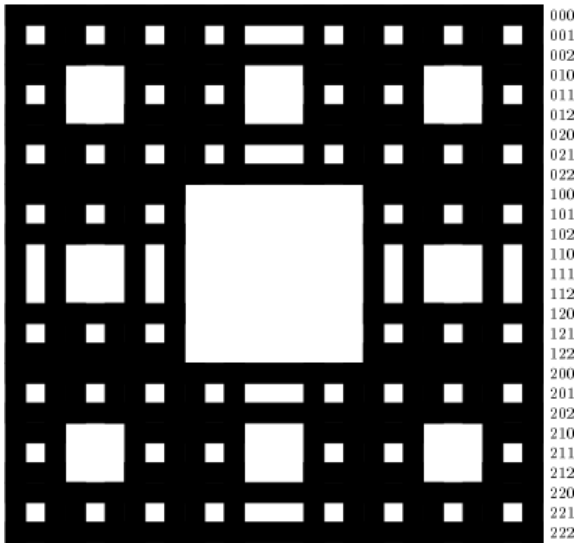
**Primera etapa:** Para cada fila cuyo número no contiene ningún 1, trazamos una línea de 27 unidades. Por simetría, realizamos la misma operación con las columnas.



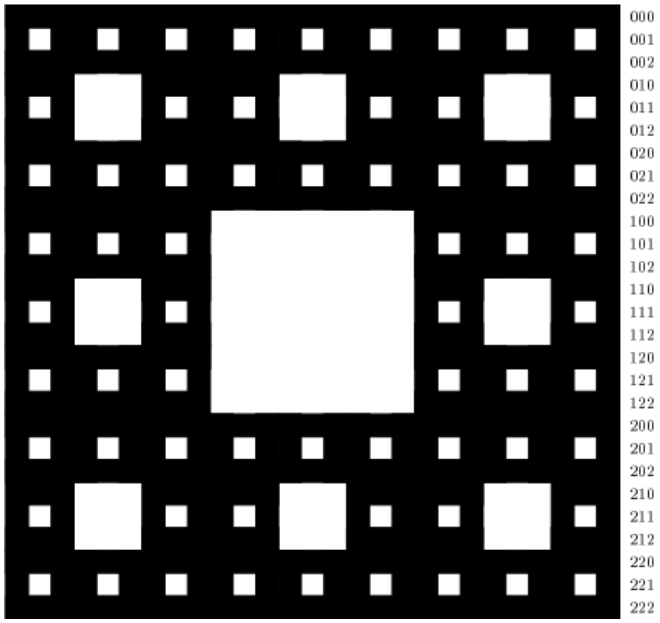

**Segunda etapa:** Ahora buscamos las filas que sólo tienen un 1 en la primera posición. Dibujaremos alternativamente rectángulos de longitud 9 unidades. Como antes, la simetría nos obliga a repetir esta operación con las columnas.




**Tercera etapa:** Buscamos las filas que sólo tienen un 1 en la segunda posición. Dibujaremos los rectángulos de acuerdo a la serie [3 3 6 3 6 3 3]. (dibujamos 3, 3 en blanco, dibujamos 6, etc). Repetimos con las columnas.




**Última etapa:** Miramos ahora las líneas que contienen un doble 1 en las dos primeras posiciones. La serie de rectángulos se dibuja de acuerdo al esquema [3 3 3 9 3 3 3], y repetimos en las columnas.




La construcción de la alfombra de Sierpinski de orden 3 está completa. Para dibujarlo, sólo necesitamos $16 + 16 + 32 + 16 = 80$ polígonos.

### D.2.3. Otros esquemas posibles usando columnas

Para recapitular la construcción anterior, estos son los diferentes tipos de esquemas por columnas, de acuerdo al número de líneas (* representa 0 ó 2):

| Número de línea | Esquema a aplicar |
|---|---|
| *** | 27 |
| 1** | 9 9 9 |
| *1* | 3 3 6 3 6 3 3 |
| 11* | 3 3 3 9 3 3 3 |

Del mismo modo, para construir una alfombra de orden 4, necesitamos un cuadrado de $3^4 = 81$ unidades. Los números de línea y de columna tendrán 4 cifras al descomponerlos a base 3. Para cada tipo de número de línea, el esquema a aplicar será (* representa 0 ó 2):

| Número de línea | Esquema a aplicar |
|---|---|
| **** | 81 |
| 1*** | 27 27 27 |
| *1** | 9 9 18 9 18 9 9 |
| **1* | 3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3 |
| *11* | 3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3 |
| 1*1* | 3 3 6 3 6 3 3 27 3 3 6 3 6 3 3 |
| 11** | 9 9 9 27 9 9 9 |
| 111* | 3 3 3 9 3 3 3 27 3 3 3 9 3 3 3 |

De aquí se deduce que hacen falta 496 polígonos para dibujar una alfombra de Sierpinski de orden 4.

Por si alguien tiene la duda, este es el esquema para el caso $n = 2$:

| Número de línea | Esquema a aplicar |
|---|---|
| ** | 9 |
| 1* | 3 3 3 |

## D.2.4. El programa

```
# Dibuja una alfombra de Sierpinski de orden :p y arista :lado
para alfombra :lado :p
haz "unidad :lado/(potencia 3 :p)
  si :p=0 [ rec :lado :lado alto]
  si :p=1 [repite 4 [rec :lado :unidad avanza :lado giraderecha 90 ] alto]
  repitepara (lista "x 1 potencia 3 :p)
    [ hazlocal "cantorx cantor :x :p []
# no dibujan los elementos que tengan un 1 en ultima posicion
      si  no (1=ultimo :cantorx)
       [ hazlocal "nom evalua menosultimo :cantorx "
         dibujacolumna :x devuelvepropiedad "map :nom ]
    ]
fin
```

```
# Devuelve la descomposicion en base 3 del numero x
# p indice de profundidad 3^p
# :lista lista vacia inicialmente

para cantor :x :p :lista
  si :p=0 [devuelve :lista ]
  hazlocal "a potencia 3 :p-1
  si :x<= :a
    [ devuelve cantor  :x :p-1  frase :lista 0]
    [ si :x<=2*:a
        [devuelve cantor  :x-:a :p-1  frase :lista 1]
      devuelve cantor :x-2*:a :p-1 frase :lista 0]
fin

# trazado de la columna numero x de acuerdo al esquema de
#    construccion definido en la lista
para dibujacolumna :x :lista
  subelapiz
    giraderecha 90 avanza (:x-1)*:unidad giraizquierda 90
  bajalapiz
  des :lista
  subelapiz giraizquierda 90 avanza (:x-1)*:unidad
  giraderecha 90 avanza :x*:unidad giraderecha 90 bajalapiz des :lista
  subelapiz
    giraizquierda 90 retrocede :x*:unidad
  bajalapiz
fin

# trazado de un rectangulo con las medidas dadas
# el pologono se guarda para verlo en 3d
para rec :lo :la
  haz "contador :contador+1
  empiezapoligono
    repite 2
     [avanza :lo giraderecha 90 avanza :la giraderecha 90]
  finpoligono
fin

# Inicializa las diferentes columnas posibles para la alfombra de orden 1 a 4
para initmap
  ponpropiedad "map 111 [3 3 3 9 3 3 3 27 3 3 3 9 3 3 3]
  ponpropiedad "map 110 [9 9 9 27 9 9 9]
  ponpropiedad "map 101 [3 3 6 3 6 3 3 27 3 3 6 3 6 3 3]
```

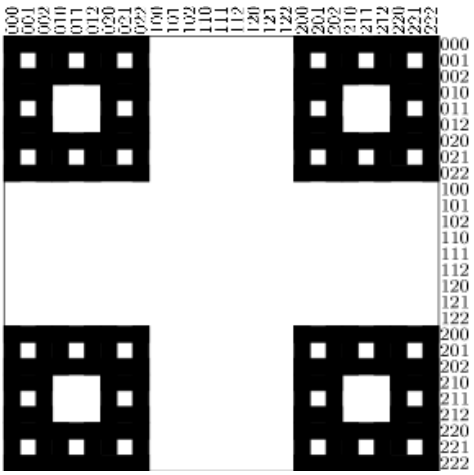
```
  ponpropiedad "map 011 [3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3]
  ponpropiedad "map 000 [81]
  ponpropiedad "map 100 [27 27 27]
  ponpropiedad "map 010 [9 9 18 9 18 9 9]
  ponpropiedad "map 001 [3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3]
  ponpropiedad "map 01 [3 3 6 3 6 3 3]
  ponpropiedad "map 00 [27]
  ponpropiedad "map 10 [9 9 9]
  ponpropiedad "map 11 [3 3 3 9 3 3 3]
  ponpropiedad "map 1 [3 3 3]
  ponpropiedad "map 0 [9]
fin

# si la decomposicion es [1 0 1] --> devuelve 101
para evalua :lista :motivo
  si vacio? :lista [devuelve :motivo]
  [ hazlocal "motivo palabra :motivo primero :lista
    devuelve evalua menosprimero :lista :motivo ]
fin

# trazado de los grupos de rectangulos de cada columna alternativamente
para des :lista
  hazlocal "suma 0
  repitepara (lista "i 1 cuenta :lista)
    [ hazlocal "elemento elemento :i :lista
      hazlocal "suma :elemento+:suma
      si par? :i
        [ subelapiz avanza :elemento*:unidad bajalapiz ]
        [ rec :elemento*:unidad :unidad
          avanza :elemento*:unidad]
    ]
  subelapiz retrocede :suma * :unidad bajalapiz
fin

# prueba si un numero es par
para par? :i
  devuelve 0 = resto :i 2
fin

para sierpinski :p
  borrapantalla perspectiva ocultatortuga initmap
  haz "contador 0
  alfombra 810 :p
```

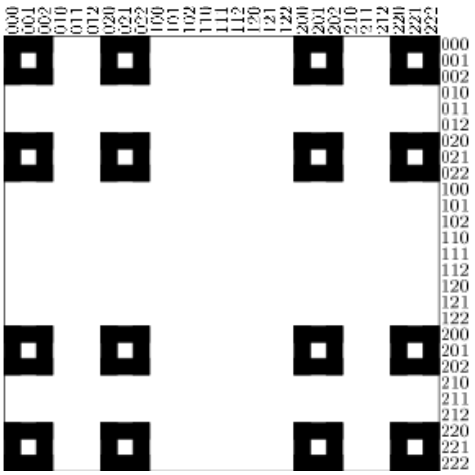
```
  mecanografia "Numero\ de\ poligonos:\  escribe :contador
  vistapoligono
fin
```

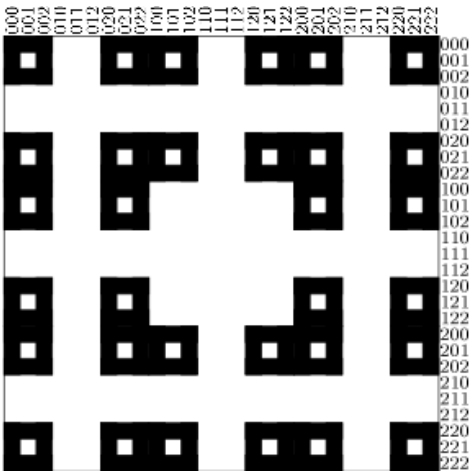
`sierpinski 3` dibuja una alfombra de Sierpinski de orden 3 y lado 810. Primera etapa conseguida, podemos volver a la esponja de Menger.

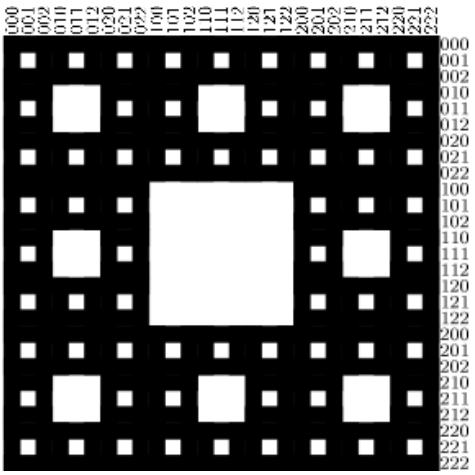
### D.2.5. La esponja de Menger de orden 4

La esponja de Menger possee múltiples propiedades de simetría. Para generar la esponja, vamos a trazar las diferentes secciones a lo largo del plano $(xOy)$ y las repetiremos sobre los planos $(yOz)$ y $(xOz)$. Para explicar qué pasa, echemos un vistazo a la esponja de orden 3. Cuando cortamos la esponja con un plano vertical, podemos obtener cuatro motivos distintos:









Para trazar una esponja de orden 3, vamos a recorrer los números de 1 a 27, es decir, de 001 a 222 en base 3. Para cada número, aplicaremos la sección adecuada que nos generará la figura en las 3 direcciones: $(Ox)$, $(Oy)$ y $(Oz)$.

### El código

Este programa dibujará los sólidos de Menger de orden 0, 1, 2, 3 y 4. El número de procedimientos es importante, y precisa de algunas explicaciones.

```
# Comando de Inicio: esponja 3
#
# Dibuja una alfombra de Sierpinski de orden :p y arista :lado
para alfombra :lado :p
  haz "unidad :lado/(potencia 3 :p)
  si :p = 0 [ rec :lado :lado alto ]
  si :p = 1 [ repite 4 [rec :lado :unidad avanza :lado giraderecha 90 ] alto]
  repitepara (lista "x 1 potencia 3 :p)
    [ hazlocal "cantorx cantor :x :p []
# no traza elementos que tienen un 1 en la ultima posicion
      si  no (1 = ultimo :cantorx)
        [ hazlocal "nom evalue menosultimo :cantorx "
          dibujacolumna :x devuelvepropiedad "map :nom ]
    ]
fin

# devuelve la decomposicion en base 3 del numero x
# p indica de profundidad 3^p
# :lista lista vacia al principio

para cantor :x :p :lista
  si :p = 0 [devuelve :lista]
  hazlocal "a potencia 3 :p-1
  si :x<= :a
    [ devuelve cantor  :x :p-1  frase :lista 0 ]
    [ si :x<=2*:a [devuelve cantor  :x-:a :p-1  frase :lista 1]
      devuelve cantor :x-2*:a :p-1 frase :lista 2]
fin

# trazado de la columna numero x respecto al esquema de construccion
#    definido en la lista
para dibujacolumna :x :lista
  subelapiz
    giraderecha 90 avanza (:x-1)*:unidad giraizquierda 90
```

```
  bajalapiz
  des :lista
  subelapiz
    giraizquierda 90 avanza (:x-1)*:unidad giraderecha 90
    avanza :x*:unidad giraderecha 90
  bajalapiz
  des :lista
  subelapiz giraizquierda 90 retrocede :x*:unidad bajalapiz
fin

# dibuja un rectangulo con las  dimensiones dadas
# el poligono se guarda para el visor 3D
para rec :lo :la
  haz "contador :contador+1
  empiezapoligono
    repite 2 [avanza :lo giraderecha 90 avanza :la giraderecha 90]
  finpoligono
fin

# Inicializa las diferentes columnas posibles para las alfombras de orden 1 a 4
para initmap
  ponpropiedad "map 111 [3 3 3 9 3 3 3 27 3 3 3 9 3 3 3]
  ponpropiedad "map 110 [9 9 9 27 9 9 9]
  ponpropiedad "map 101 [3 3 6 3 6 3 3 27 3 3 6 3 6 3 3]
  ponpropiedad "map 011 [3 3 3 9 3 3 6 3 3 9 3 3 6 3 3 9 3 3 3]
  ponpropiedad "map 000 [81]
  ponpropiedad "map 100 [27 27 27]
  ponpropiedad "map 010 [9 9 18 9 18 9 9]
  ponpropiedad "map 001 [3 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 6 3 3]
  ponpropiedad "map 01 [3 3 6 3 6 3 3]
  ponpropiedad "map 00 [27]
  ponpropiedad "map 10 [9 9 9]
  ponpropiedad "map 11 [3 3 3 9 3 3 3]
  ponpropiedad "map 1 [3 3 3]
  ponpropiedad "map 0 [9]
fin

# si la decomposicion es [1 0 1] --> devuelve 101
# si la decomposicion es [1 0 2] --> devuelve 100
# los elementos de la lista son concatenados en una palabra.
#  los 2 son reemplazados por dos ceros
para evalue :lista :mot
  si vacio? :lista
```

```
    [ devuelve :mot ]
    [ hazlocal "primera primero :lista
      si :primera=2 [hazlocal "primera 0 ]
      hazlocal "mot palabra :mot :primera
      devuelve evalue menosprimero :lista :mot
    ]
fin

# dibujamos los grupos de rectangulos de cada columna, alternativamente
para des :lista
  hazlocal "suma 0
 repitepara (lista "i 1 cuenta :lista)
   [ hazlocal "elemento elemento :i :lista
     hazlocal "suma :elemento + :suma
     si par? :i
      [ subelapiz avanza :elemento*:unidad bajalapiz ]
      [ rec :elemento*:unidad :unidad avanza :elemento*:unidad]
   ]
   subelapiz retrocede :suma * :unidad bajalapiz
fin

# prueba si un numero es par
para par? :i
  devuelve 0 = resto :i 2
fin

para sierpinski :p
  borrapantalla perspectiva ocultatortuga
  initmap
  haz "contador 0
  alfombra 810 :p
  mecanografia "numero\ de\ poligonos:\  escribe :contador
  vistapoligono
fin

# suprime el ultimo 1 de la lista :lista
para borraultimouno :lista
  repitepara (lista "i cuenta :lista 1 cambiasigno 1)
    [ hazlocal "elemento elemento :i :lista
      si :elemento = 1
        [ hazlocal "lista reemplaza :lista :i 0 alto ]
        [ si :elemento = 2 [alto]]
    ]
```

```
  devuelve :lista
fin

# Esponja de Menger de arista dada y de profundidad :p

para menger :lado :p
  haz "unidade :lado/(potencia 3 :p)
  repitepara (lista "z 1 potencia 3 :p)
    [ hazlocal "cantorz cantor :z :p []
      hazlocal "last ultimo :cantorz
      hazlocal "cantorz menosultimo :cantorz
      si :last = 0
        [ hazlocal "orden evalue borraultimouno :cantorz "]
        [ hazlocal "orden evalue :cantorz "]
      hazlocal "orden palabra "corte :orden
      draw3alfombra :lado :orden :z
      subelapiz cabeceaarriba 90 avanza :unidad cabeceaabajo 90 bajalapiz
    ]
    draw3alfombra :lado :orden (potencia 3 :p)+1
fin

# trazado de las alfombras de sierpinski de orden :p
# a lo largo de cada eje (ox), (oy) y (oz)
# a la altitud :z
para draw3alfombra :lado :orden :z
  subelapiz centro
  cabeceaarriba 90 avanza (:z-1)*:unidade cabeceaabajo 90 bajalapiz
  poncolorlapiz azul ejecuta :orden :lado
  subelapiz centro
  balanceaizquierda 90 avanza (:z-1)*:unidade cabeceaabajo 90  bajalapiz
  poncolorlapiz amarillo ejecuta :orden :lado
  subelapiz centro
  cabeceaarriba 90 avanza :lado giraderecha 90
  avanza (:z-1)*:unidade cabeceaabajo 90 bajalapiz
  poncolorlapiz magenta ejecuta :orden :lado
fin


# procedimiento principal
# dibuja una esponja de Menger d profundidad p
para esponja :p
  borrapantalla perspectiva ocultatortuga
  hazlocal "tiempo tiempo
```

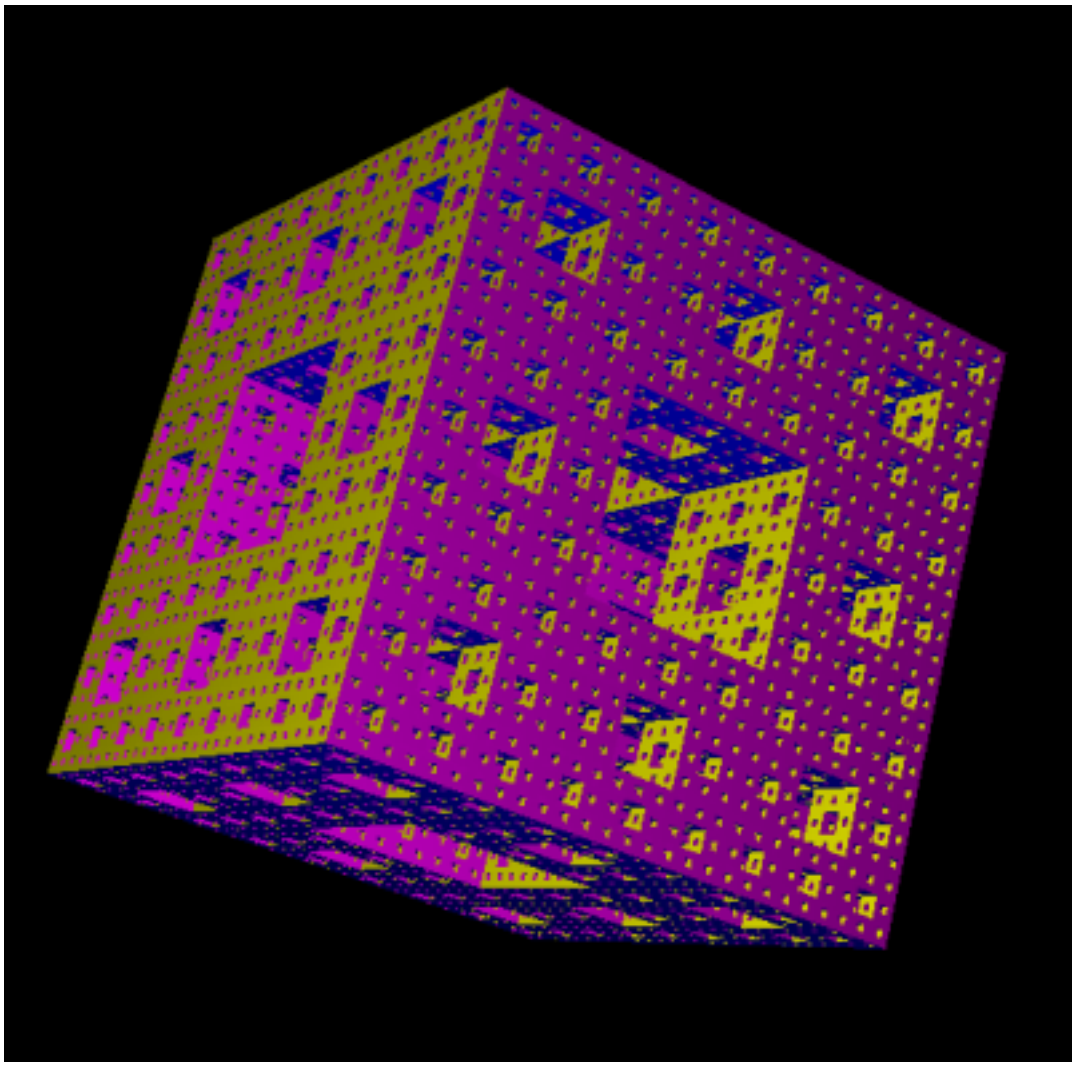
```
  initmap
  haz "contador 0
  si :p=0 [cubo 405] [menger 405 :p]
# muestra el tiempo y establece el numero de poligonos necesarios
#    para la construccion
  mecanografia "Numero\ de\ poligonos:\  escribe :contador
  mecanografia "Tiempo\ transcurrido:\  escribe tiempo -:tiempo
  vistapoligono
fin

# seccion para Menger de orden 2
para corte1 :lado
  repite 4
    [alfombra :lado/3 1 subelapiz avanza :lado giraderecha 90 bajalapiz]
fin

para corte0 :lado
  alfombra :lado 2
fin

# seccion para Menger de orden 3
para corte10 :lado
  repite 4
    [ alfombra :lado/3 2 subelapiz avanza :lado giraderecha 90 bajalapiz]
fin

para corte01 :lado
  repite 4
    [ repite 2 [corte1 :lado/3 subelapiz avanza :lado/3 bajalapiz]
      avanza :lado/3 giraderecha 90 ]
fin

para corte11 :lado
  repite 4
    [ corte1 :lado/3 subelapiz avanza :lado giraderecha 90 bajalapiz]
fin

para corte00 :lado
  alfombra :lado 3
fin

# seccion para Menger de orden 4
para corte000 :lado
```

```
  alfombra :lado 4
fin

para corte100 :lado
  repite 4
    [ alfombra :lado/3 3 subelapiz avanza :lado giraderecha 90 bajalapiz]
fin

para corte010 :lado
  repite 4
   [ repite 2 [corte10 :lado/3 subelapiz avanza :lado/3 bajalapiz]
     avanza :lado/3 giraderecha 90]
fin

para corte001 :lado
  repite 4
    [ repite 2 [corte01 :lado/3 subelapiz avanza :lado/3 bajalapiz]
      avanza :lado/3 giraderecha 90]
fin

para corte110 :lado
  repite 4
    [ corte10 :lado/3 subelapiz avanza :lado bajalapiz giraderecha 90 ]
fin

para corte111 :lado
  repite 4
    [ corte11 :lado/3 subelapiz avanza :lado giraderecha 90 bajalapiz]
fin

para corte101 :lado
  repite 4
    [ corte01 :lado/3 subelapiz avanza :lado giraderecha 90 bajalapiz]
fin

para corte011 :lado
  repite 4
    [ repite 2 [corte11 :lado/3 subelapiz avanza :lado/3 bajalapiz]
      avanza :lado/3 giraderecha 90]
fin

para corte :lado
  alfombra :lado 1
```

```
fin

para cubo :lado
  repite 2
    [ poncolorlapiz azul rec :lado :lado
      subelapiz avanza :lado cabeceaabajo 90 bajalapiz
      poncolorlapiz amarillo rec :lado :lado
      subelapiz avanza :lado cabeceaabajo 90  bajalapiz ]
  poncolorlapiz magenta
  subelapiz
     balanceaizquierda 90 giraizquierda 90 avanza :lado giraderecha 90
  bajalapiz rec :lado :lado
  subelapiz
     giraderecha 90 avanza :lado giraizquierda 90 balanceaderecha 90
     giraderecha 90 avanza :lado giraizquierda 90 balanceaderecha 90
  bajalapiz rec :lado  :lado
  balanceaizquierda 90 giraizquierda 90 avanza :lado giraderecha 90
fin

para cubos
  borrapantalla perspectiva ocultatortuga
  hazlocal "temps tiempo
  initmap
  haz "contador 0
  repite 4
    [ si contador=1 [cubo 405]
                    [menger 405 contador-1]
     subelapiz avanza 1000 giraderecha 90 bajalapiz ]
# muestra el tiempo y el numero de poligonos necesarios en la construcion
  mecanografia "Numero\ de\ poligonos:\  escribe :contador
  mecanografia "Tiempo\ empleado:\  escribe tiempo -:temps
  vistapoligono
fin
```

Con todo lo anterior, y tras ajustar la memoria disponible para xLogo a 640 Mb, si tecleamos

```
esponja 4:
```



# Apéndice E

# El sistema de Lindenmayer

En este tema hemos utilizado materiales procedentes de:

- La Wikipedia: `http://es.wikipedia.org/wiki/Sistema-L`, en sus ediciones española, inglesa y francesa.
- El libro "**The Algorithmic Beauty of Plants**" escrito por Przemyslaw Prusinkiewicz y Aristid Lindenmayer.

Esta lección va a tratar de la noción de **Sistema de Lindenmayer** o **Sistema–L** inventado en 1968 por el biólogo húngaro Aristid Lindenmayer. Un Sistema-L es un conjunto de reglas y símbolos que modelan el proceso de crecimiento de seres vivos como las plantas o las células. El concepto central de los Sistemas–L es la noción de re-escritura. La re-escritura es una técnica para construir objetos complejos por reemplazo a partir de un objeto inicial sencillo utilizando reglas de re-escritura.

Para ello, las células se modelizan mediante símbolos. En cada generación, las células se dividen, es decir, un símbolo se sustituye por uno o varios otros símbolos que forman una palabra.

## E.1. Definición formal

Un Sistema–L es una gramática formal que comprende:

1. Un alfabeto $V$ : conjunto de variables del Sistema–L. $V*$ es el conjunto de "palabras" que podemos generar con cualquiera de los símbolos de $V$, y $V+$ el conjunto de palabras con al menos un símbolo.
2. Un conjunto de valores constantes $S$. Algunos de estos símbolos son comunes a todos los Sistemas–L, en particular cuando los utilizamos con la tortuga).
3. Un axioma de partida $\omega$, elegido de $V+$, es el estado inicial.

4. Un conjunto de reglas o producciones, denominado $P$, que definen la forma en la que las variables pueden ser reemplazadas por combinaciones de constantes y otras variables.

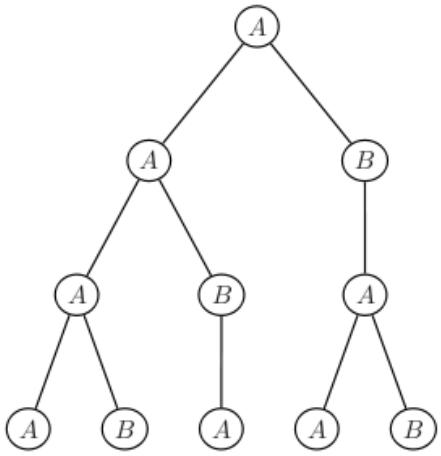
   Una producción está formada por dos cadenas. el predecesor y el sucesor.

Un Sistema–L definido así, es una tupla $\{V, S, \omega, P\}$.

Consideremos el Sistema–L siguiente:

- Alfabeto: $V = \{A, B\}$
- Constantes: $S = \{\emptyset\}$
- Axioma inicial: $\omega = A$
- Reglas : $\boxed{\begin{array}{l} A \to AB \\ B \to A \end{array}}$

Las dos reglas de producción dadas son las reglas de re-escritura del sistema. En cada etapa, $A$ es sustituida por $AB$, y $B$ es sustituido por $A$. Estas son las primeras iteraciones de este sistema de Lindemayer:




- Iteración 1: $A$
- Iteración 2: $AB$
- Iteración 3: $ABA$
- Iteración 4: $ABAAB$

Bien, bien pero ... ¿y en qué se concreta esto? Pasemos a la siguiente sección.

## E.2. Interpretación por la tortuga

Este primer ejemplo ayuda a entender la noción de sistema de Lindenmayer y, posiblemente, cómo vamos a utilizarla de manera eficaz con la tortuga.

Aquí es donde se pone interesante: Cada palabra así construida no tiene ningún significado especial. Vamos a definir para cada letra de la secuencia, una acción (comando) que ejecutará la tortuga y generará dibujos en 2D o 3D.

### E.2.1. Simbología

- $F$ : Avanzar un paso (unitario) ($\in V$)
- $+$ : Girar un cierto ángulo $\alpha$ a la izquierda ($\in S$).
- $-$ : Girar un cierto ángulo $\alpha$ a la derecha ($\in S$).
- & : Girar un cierto ángulo $\alpha$ hacia abajo ($\in S$).
- ^: Girar un cierto ángulo $\alpha$ hacia arriba ($\in S$).
- \: Girar sobre sí mismo hacia la izquierda un ángulo $\alpha$ ($\in S$).
- / : Girar sobre sí mismo hacia la derecha un ángulo $\alpha$ ($\in S$).
- | : Efectuar una media vuelta. Con xLogo: `giraderecha 180`

Por ejemplo, con $\alpha = 90°$ y un desplazamiento unitario de 10 pasos de tortuga, obtennemos:

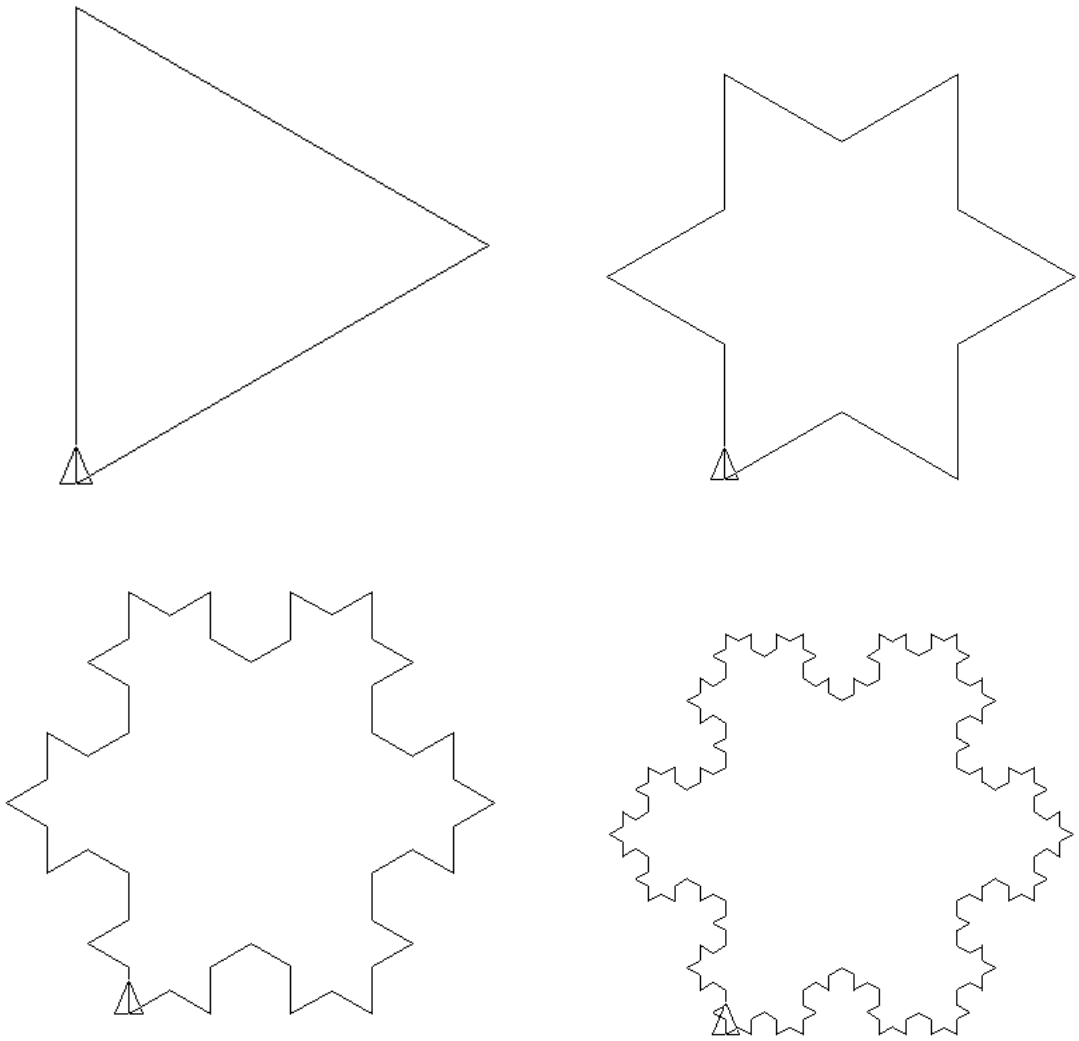
| Símbolo | $F$ | $+$ | $-$ | & | ^ | \ | / | \| |
|---|---|---|---|---|---|---|---|---|
| Comando xLogo | `av 10` | `gi 90` | `gd 90` | `sn 90` | `bn 90` | `bi 90` | `bd 90` | `gd 180` |

### E.2.2. Copo de Koch

Consideremos el sistema–L:

- Estado inicial: $F--F--F--$
- Regla de producción: $F \to F+F--F+F$
- Ángulo $\alpha = 60°$, el paso unitario se divide por 3 en cada iteración.

Primeras iteraciones:



cuyo programa en xLogo es:

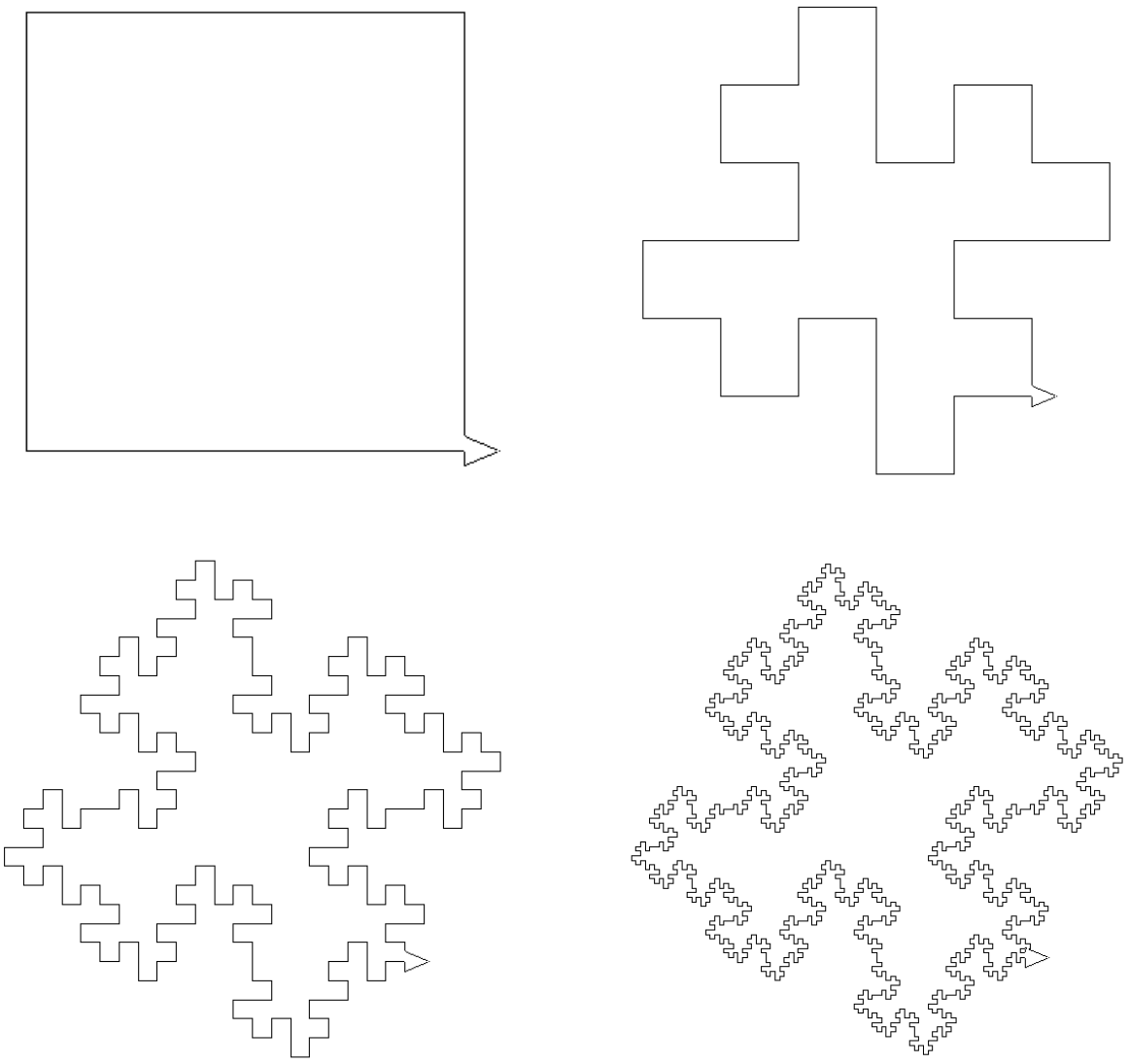
```
para coponieve :p
  borrapantalla
  haz "unidad 300/potencia 3 :p-1
  repite 3 [f :p-1 giraderecha 120]
fin

para f :p
  si :p=0 [avanza :unidad alto]
  f :p-1 giraizquierda 60
  f :p-1 giraderecha 120
  f :p-1 giraizquierda 60
  f :p-1
fin
```

### E.2.3. Curva de Koch de orden 2

Fijémonos ahora en el sistema–L siguiente:

- Estado inicial: $F - F - F - F$
- Regla de producción: $F \rightarrow F - F + F + FF - F - F + F$

Las primeras representaciones utilizando $\alpha = 90$ y ajustando el paso unitario para que la figura tenga un tamaño constante:



Ahora es muy fácil crear el programa Logo para generar esas figuras:

```
# p indica la iteracion
```

```
para koch :p
  # entre cada iteracion, la distancia unitaria se divide entre 4
  # asi, la figura final tendra unas dimensiones maximas de 600x600
  haz "unidad 300/potencia 4 :p-1
  repite 3 [ f :p-1 giraizquierda 90]
  f :p-1
fin

# la cadena de re-escritura
para f :p
  si :p=0 [avanza :unidad alto]
  f :p-1 giraizquierda 90
  f :p-1 giraderecha 90
  f :p-1 giraderecha 90
  f :p-1 f :p-1 giraizquierda 90
  f :p-1 giraizquierda 90
  f :p-1 giraderecha 90 f :p-1
fin
```

### E.2.4. Curva del dragón

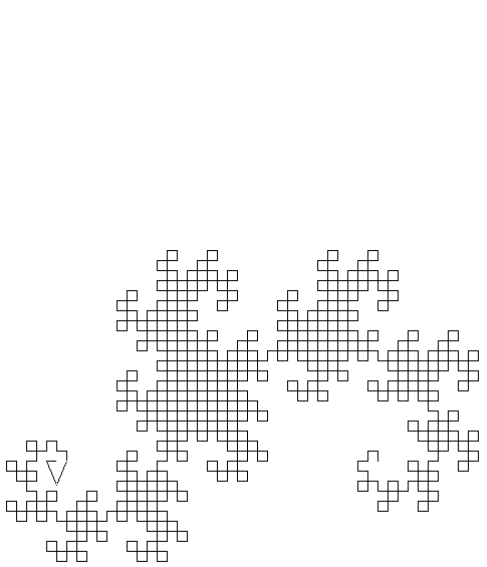
Terminamos esta serie de ejemplos con la **curva dragón**, cuyas condiciones son:

- Estado inicial: $F$
- Regla de producción: $\boxed{\begin{array}{l} A \rightarrow A + B+ \\ B \rightarrow -A - B \end{array}}$

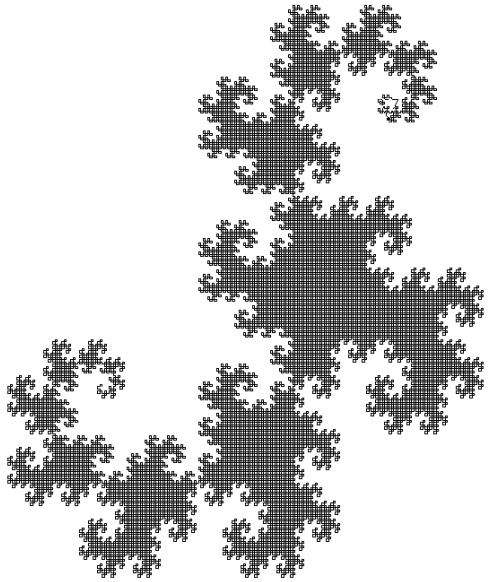
El programa resulta:

```
para dragon :p
  haz "unidad 300/8/ :p
  a :p
fin

para a :p
  si :p=0 [avanza :unidad alto]
  a :p-1 giraizquierda 90 b :p-1 giraizquierda 90
fin

para b :p
  si :p=0 [avanza :unidad alto]
  giraderecha 90 a :p-1 giraderecha 90 b :p-1
fin
```

y los resultados son:



dragon 10



dragon 15

### E.2.5. Curva de Hilbert en 3D

El ejemplo siguiente generará la curva de Hilbert en el espacio. Esta es una curva que presenta la propiedad de reemplazar perfectamente un cubo al aumentar el número de iteraciones.

El sistema–L asociado es:

- Estado inicial: $A$
- Ángulo $\alpha = 90°$, la longitud unitaria se divide entre dos en cada iteración
- Regla de produción:

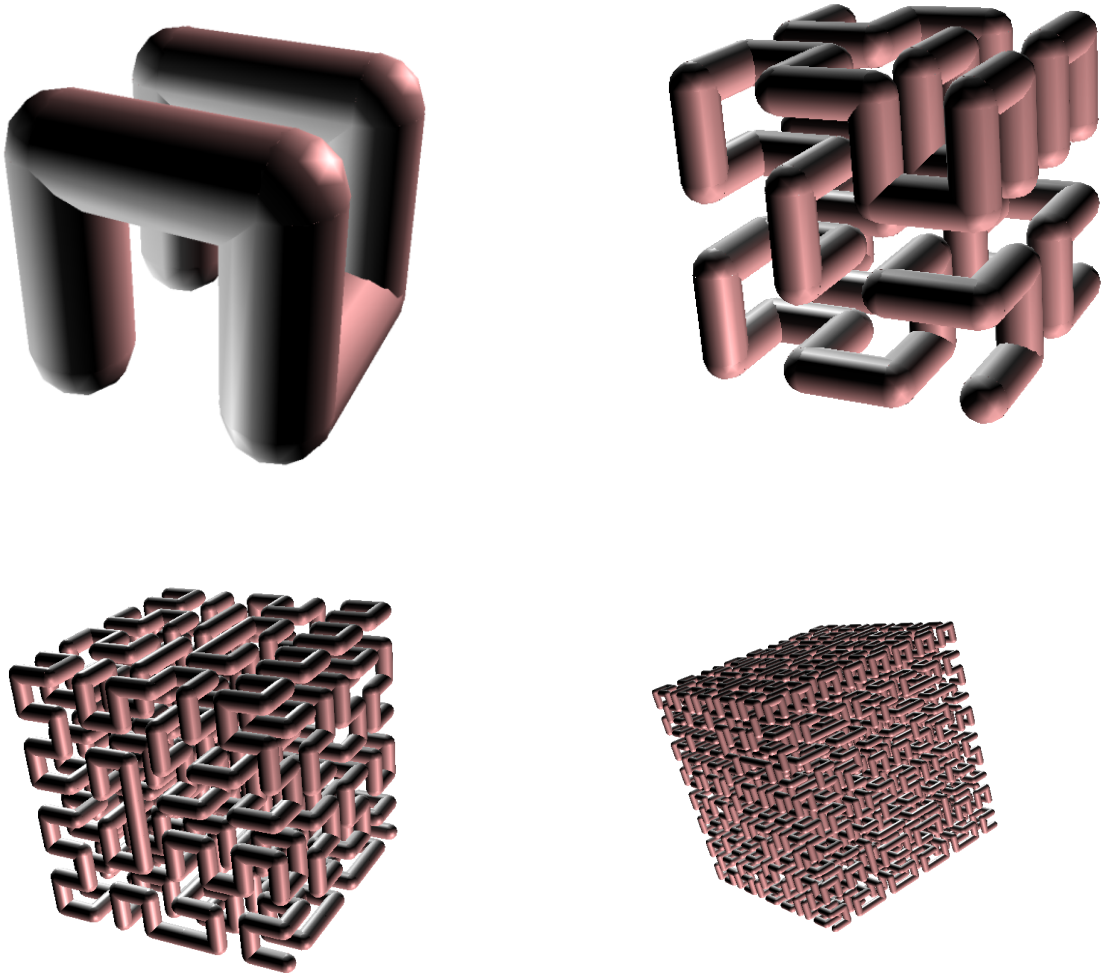
$$\begin{array}{l} A \rightarrow B - F + CFC + F - D\&F\hat{}D - F + \&\&CFC + F + B// \\ B \rightarrow A\&F\hat{}CFB\hat{}F\hat{}D\hat{}\hat{} - F - D\hat{}|F\hat{}B|FC\hat{}F\hat{}A// \\ C \rightarrow |D\hat{}|F\hat{}B - F + C\hat{}F\hat{}A\&\&FA\&F\hat{}C + F + B\hat{}F\hat{}D// \\ D \rightarrow |CFB - F + B|FA\&F\hat{}A\&\&FB - F + B|FC// \end{array}$$

```
para hilbert :p
  borrapantalla perspectiva
  haz "unidad 400/potencia 2 :p
  empiezalinea
    pongrosor :unidad/2 a :p
  finlinea
  vistapoligono
fin
```

```
para a :p
  si :p=0 [alto]
  b :p-1 giraderecha 90 avanza :unidad giraizquierda 90
  c :p-1 avanza :unidad
  c :p-1 giraizquierda 90 avanza :unidad giraderecha 90
  d :p-1 cabeceaabajo 90 avanza :unidad cabeceaarriba 90
  d :p-1 giraderecha 90 avanza :unidad giraizquierda 90 cabeceaabajo 180
  c :p-1 avanza :unidad
  c :p-1 giraizquierda 90 avanza :unidad giraizquierda 90
  b :p-1 balanceaderecha 180
fin

para b :p
  si :p=0 [alto]
  a :p-1 cabeceaabajo 90 avanza :unidad cabeceaarriba 90
  c :p-1 avanza :unidad
  b :p-1 cabeceaarriba 90 avanza :unidad cabeceaarriba 90
  d :p-1 cabeceaarriba 180 giraderecha 90 avanza :unidad giraderecha 90
  d :p-1 cabeceaarriba 90 giraderecha 180 avanza :unidad cabeceaarriba 90
  b :p-1 giraderecha 180 avanza :unidad
  c :p-1 cabeceaarriba 90 avanza :unidad cabeceaarriba 90
  a :p-1 balanceaderecha 180
fin

para c :p
  si :p=0 [alto]
  giraderecha 180
  d :p-1 cabeceaarriba 90 giraderecha 180 avanza :unidad cabeceaarriba 90
  b :p-1 giraderecha 90 avanza :unidad giraizquierda 90
  c :p-1 cabeceaarriba 90 avanza :unidad cabeceaarriba 90
  a :p-1 cabeceaabajo 180 avanza :unidad
  a :p-1 cabeceaabajo 90 avanza :unidad cabeceaarriba 90
  c :p-1 giraizquierda 90 avanza :unidad giraizquierda 90
  b :p-1 cabeceaarriba 90 avanza :unidad cabeceaarriba 90
  d :p-1 balanceaderecha 180
fin

para d :p
  si :p=0 [alto]
  giraderecha 180
  c :p-1 avanza :unidad
  b :p-1 giraderecha 90 avanza :unidad giraizquierda 90
```

```
  b :p-1 giraderecha 180 avanza :unidad
  a :p-1 cabeceaabajo 90 avanza :unidad cabeceaarriba 90
  a :p-1 cabeceaabajo 180 avanza :unidad
  b :p-1 giraderecha 90 avanza :unidad giraizquierda 90
  b :p-1 giraderecha 180 avanza :unidad
  c :p-1 balanceaderecha 180
fin
```

En las primeras iteraciones obtenemos:



Hermoso, ¿verdad?

# Índice alfabético