



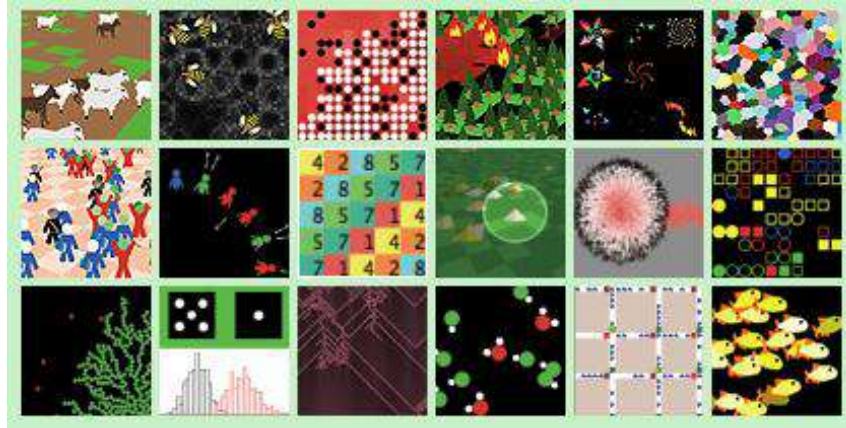
Centro de Investigación Científica y Desarrollo Tecnológico (I+D)
Universidad de Valladolid
David Poza
Manual Básico

Tabla de contenido

1. Manual de Netlogo en español	2
2. ¿Qué es Netlogo? ¿Dónde puedo obtener el programa?	2
3. Introducción al escenario de simulación de Netlogo	3
4. Vistas en Netlogo (1)	4
Ejercicio 1. Primitivas básicas en la ventana de comandos.	5
Ejercicio 2. Procedimientos	7
Ejercicio 3. Botones	8
Ejercicio 4. Propiedades de los agentes (1)	10
Ejercicio 5. Propiedades de los agentes (2)	12
Ejercicio 6. Propiedades de los agentes (3)	14
Ejercicio 7. Propiedades de los agentes (4)	15
Ejercicio 8. Variables globales. Variables de entrada.	17
Ejercicio 9. Variables globales y variables locales. Sentencias condicionales if	21
Ejercicio 10. Sentencias ifelse. Procedimientos con retorno (to-report)	24
Ejercicio 11. Propiedades adicionales de los agentes	26
Ejercicio 12. “Razas” (breeds) de agentes	29
Ejercicio 13. Bucles while	31
Ejercicio 14. Representaciones gráficas en Netlogo	33
Ejercicio 15. Listas (1)	39
Ejercicio 16. Listas (2)	43
Ejercicio 17. Operaciones sobre listas	45

1. Manual de Netlogo en español

He preparado un manual de Netlogo en español que te permitirá familiarizarte con este lenguaje de programación de una forma muy sencilla, a través de pequeños programas-ejemplo.



2. ¿Qué es Netlogo? ¿Dónde puedo obtener el programa?

Netlogo es un entorno de programación que permite la simulación de fenómenos naturales y sociales. Fue creado por Uri Wilensky en 1999 y está en continuo desarrollo por el Center for Connected Learning and Computer-Based Modeling.

Netlogo es particularmente útil para modelar sistemas complejos que evolucionan en el tiempo. Los implementadores de modelos pueden dar instrucciones a cientos o miles de agentes para que todos ellos operen de manera independiente, entre sí y con el entorno. Esto hace posible explorar la relación entre el comportamiento a bajo nivel de los individuos y los patrones macroscópicos que surgen a partir de la interacción de muchos individuos entre sí.

Netlogo permite a los usuarios abrir simulaciones y “jugar” con ellas, así como explorar su comportamiento bajo una serie de condiciones. Asimismo, permite al usuario la creación de sus propios modelos. Netlogo es lo suficientemente sencillo como para que los estudiantes y los profesores puedan ejecutar las simulaciones o incluso construir las suyas propias. Además, su grado de desarrollo actual es suficiente como para servir como una herramienta potente para investigadores en muchos ámbitos.

Existe abundante documentación y tutoriales sobre Netlogo. El programa incluye una galería de modelos (*models library*), que contiene una amplia colección de simulaciones que pueden ser ejecutadas y modificadas. Este conjunto de modelos pertenece a ámbitos muy diversos, tanto de la naturaleza como de ciencias sociales (biología, medicina, física y química, matemáticas y computación, economía y psicología social).

Existen dos maneras de ejecutar Netlogo:

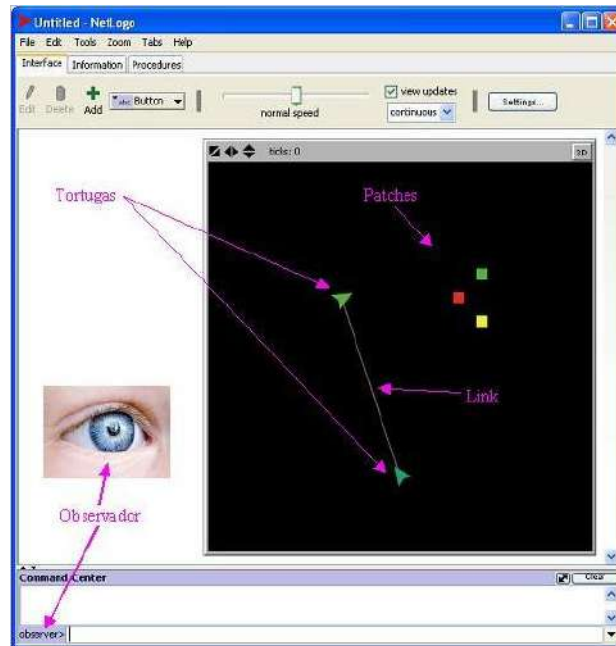
1. Descargando e instalando el programa (permite simular y editar modelos, así como la creación de modelos propios).
2. Ejecutar un applet desde una página web (permite la ejecución de los modelos, pero no editarlos ni crear modelos nuevos).

El programa puede descargarse de manera gratuita desde aquí. Para su funcionamiento, requiere tener instalada en el ordenador una máquina virtual de Java (JVM - Java Virtual Machine) versión 1.4.2 o superior. En la versión de descarga para Windows existe la opción de descargar una versión que incluye la JVM necesaria.

3. Introducción al escenario de simulación de Netlogo

Netlogo es un lenguaje de programación que sigue la filosofía del modelado basado en agentes. Concretamente, en Netlogo existen 3 tipos de agentes:

- Turtles (tortugas).
- Patches (celdas).
- Links (relaciones entre tortugas).
- Observer (observador).



Las tortugas son los agentes que se mueven por el mundo. Interaccionan entre sí y con el medio. Cada tortuga viene identificada por un identificador que es único para cada tortuga.

Netlogo denomina “mundo” (world) al terreno en el que se mueven las tortugas. Cada porción cuadrada de mundo se denomina patch. Cada patch está identificado por las coordenadas de su punto central.

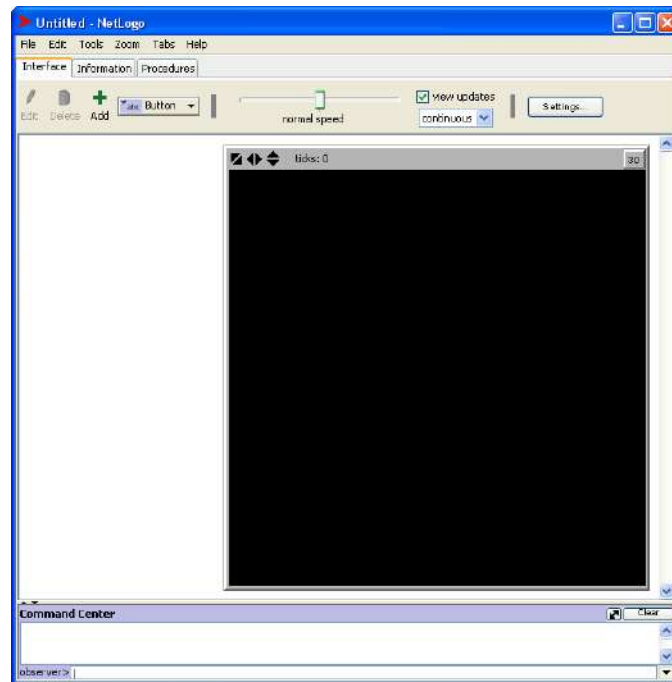
Las tortugas se mueven por el mundo (y, por tanto, por encima de los patches). Las tortugas interaccionan entre sí según unas reglas de comportamiento y con el medio (es decir, con los patches).

Se pueden modelar la relación entre distintas tortugas mediante links, que es el tercer tipo de agente presente en Netlogo. Los links se designan mediante un par (tortuga1, tortuga2), que indica las dos tortugas relacionadas mediante dicho link.

Finalmente, la última figura presente en los modelos de Netlogo es el observador. Éste no está representado en el mundo, pero puede interactuar con él (crea y destruye agentes, asigna propiedades a los agentes, etc).

4. Vistas en Netlogo (1)

Cuando arranquemos Netlogo, la pantalla de nuestro ordenador presentará el siguiente aspecto:



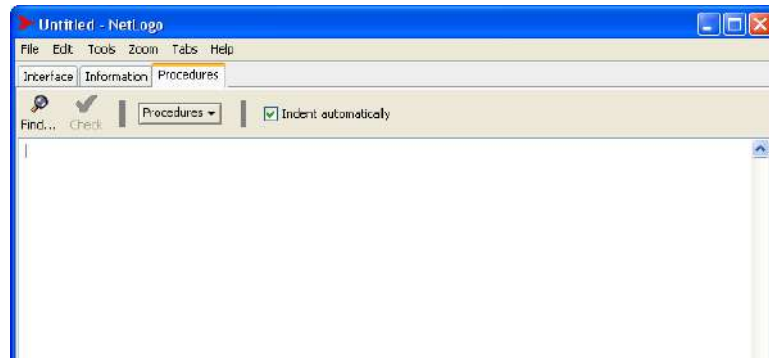
En la parte superior observamos tres pestañas: **interface** (interfaz), **information** (información) y **procedures** (procedimientos).

Aunque lo veremos con mayor detenimiento próximamente, en la primera de las pestañas (interface) será donde se represente nuestro modelo.

En la segunda pestaña (information) podremos añadir información sobre nuestro modelo para informar a los usuarios:



y en la última pestaña (procedures) escribiremos los procedimientos que se encargarán de llevar a cabo la ejecución de nuestro modelo:



Ejercicio 1. Primitivas básicas en la ventana de comandos.

Para crear tortugas, usamos la primitiva `create-turtles` (ó `crt` en abreviatura), seguido del número de tortugas que queremos crear. Ejemplo:

`crt 2` ; crea dos tortugas (equivalente: `create-turtles 2`)

Se generarán dos tortugas en el centro de coordenadas.

Podemos pedir a los agentes que realicen acciones. Para ello usamos la primitiva `ask` seguida del agente (o agentes) a los que vamos a ordenar la acción y, a continuación, entre corchetes, la acción a realizar por el / los agente(s). Algunas de estas primitivas se muestran a continuación: (en cada caso, se indica entre paréntesis el comando abreviado correspondiente, de existir)

`forward` ; (`fd`) avanzar

`back` ; (`bk`) retroceder

`left` ; (`lt`) giro a la izquierda

`right` ; (`rt`) giro a la derecha

`repeat` ; repetir un conjunto de primitivas

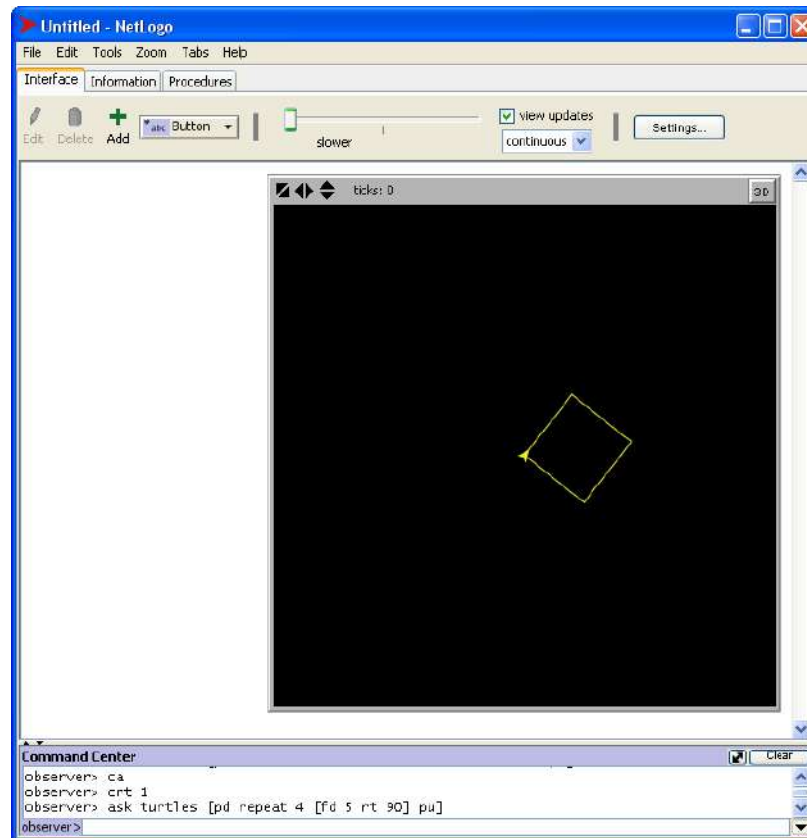
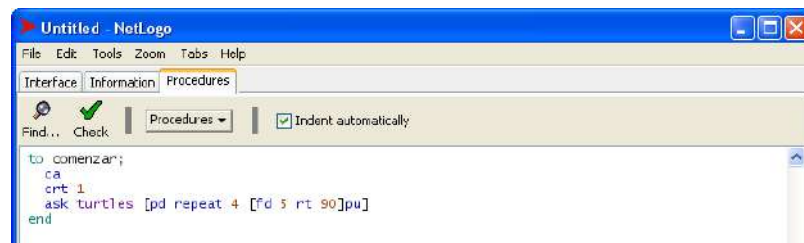
`pen-up` ; (`pu`) subir el lápiz (los agentes no dejan huella al moverse)

`pen-down` ; (`pd`) bajar el lápiz (los agentes dejan trazos al moverse)

`clear-all`; (`ca`) se resetean variables, se limpia “el mundo”, se borran gráficas, etc.

Nota: En Netlogo, para introducir un comentario detrás de una línea de código, se emplea punto y coma (;) antes del comentario.

Prueba a escribir lo siguiente en la ventana de comandos (línea a línea) y observa lo que ocurre:

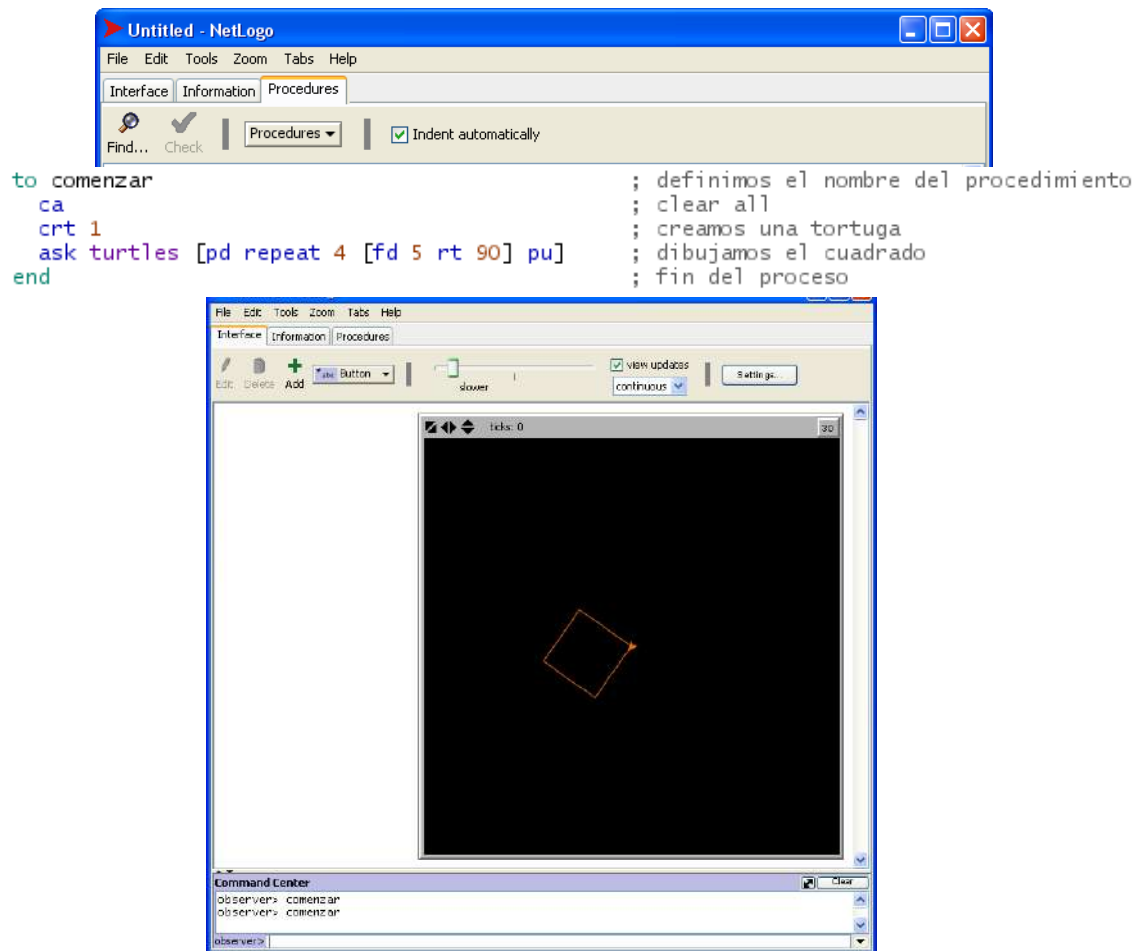


Ejercicio 2. Procedimientos

Un procedimiento es un nuevo comando definido por el programador que engloba un conjunto de comandos de NetLogo (esto es análogo al concepto de función en otros lenguajes de programación).

A continuación crearemos un nuevo procedimiento llamado “comenzar”. Dicho procedimiento englobará los comandos que utilizamos en el ejercicio anterior, es decir, creará una tortuga, y ésta dibujará un cuadrado.

Seleccionamos la pestaña “procedures” en la pantalla de NetLogo y escribimos lo siguiente:



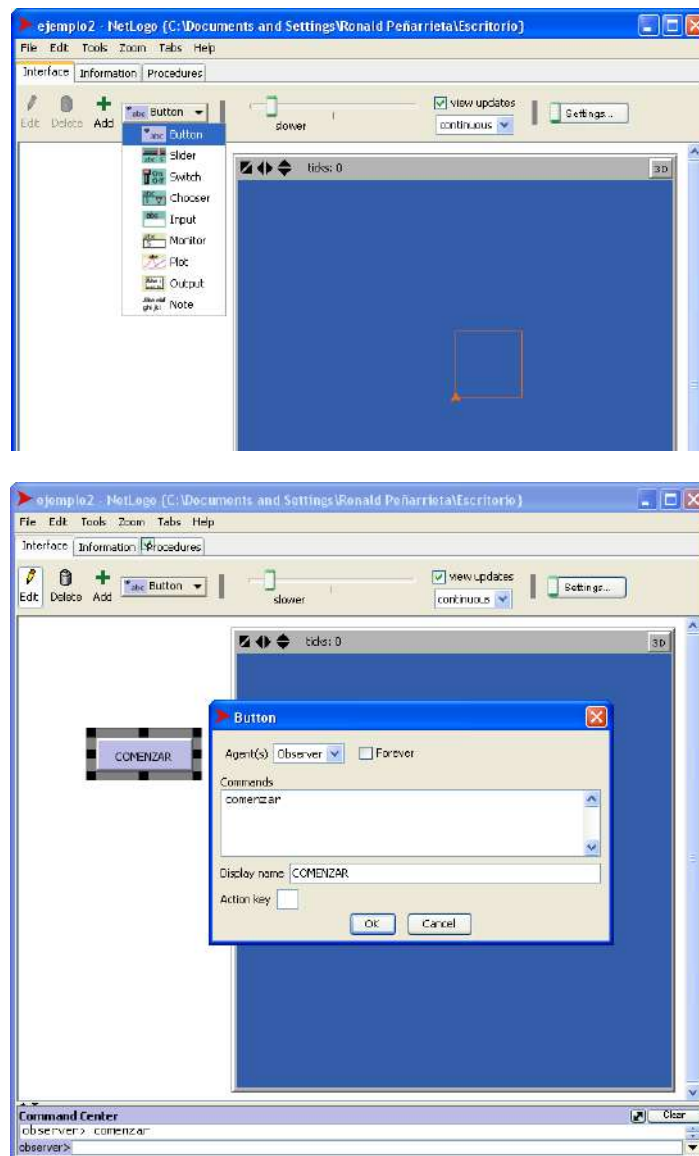
Observa que para iniciar la definición de un proceso utilizamos la palabra “to” seguida del nombre que queramos dar al procedimiento. Después introducimos el conjunto de comandos que componen la primitiva y finalmente cerramos la definición del procedimiento con la palabra “end”.

Para ejecutar el comando, debes ir a la pestaña “interface”, y, en la línea de comandos, teclear el nombre del procedimiento que acabas de crear (comenzar). Pulsa *enter* y observa lo que ocurre.

Ejercicio 3. Botones

El objetivo de este ejercicio es crear un botón para que al pulsarlo se ejecute el procedimiento creado en el ejercicio anterior. En la pestaña *interface*, en el menú desplegable de la parte superior de la pantalla, elegimos *button*. A continuación hacemos clic en el punto de la pantalla en el que deseemos colocar el botón.

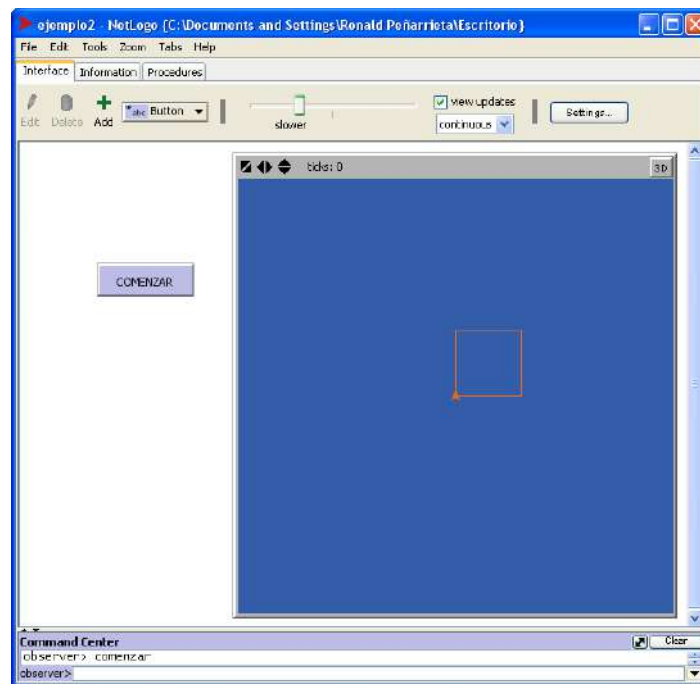
En el campo de comandos (*commands*) escribimos los comandos que queremos que se ejecuten al presionar el botón (en este caso escribiremos *comenzar*, sin comillas, para que se ejecute el procedimiento que acabamos de crear). Opcionalmente, en el campo *display name* puede escribirse el nombre que queremos que se muestre sobre el botón. También podemos asignar al botón una tecla del teclado para no tener que utilizar el ratón. Para ello, introduce la tecla deseada en el campo *action key*. La pantalla debería mostrar un aspecto semejante al que se muestra a continuación:



Marcar la casilla *Forever* provocará que el procedimiento *comenzar* se repitiera una y otra vez, hasta que pulsáramos de nuevo el botón *COMENZAR*, momento en el que el programa terminará. En cambio, si dejamos esta casilla sin marcar, cada vez que pulsemos el botón *COMENZAR*, se llamará al procedimiento *comenzar* una sola vez, por lo que se dibujará el

cuadrado una sola vez y el programa terminará. Esto es útil para simulaciones paso a paso, y es lo que emplearemos en el programa que estamos elaborando.

Cuando hayas terminado, pulsa ok. Aprieta el botón y observa lo que ocurre.



Ejercicio 4. Propiedades de los agentes (1)

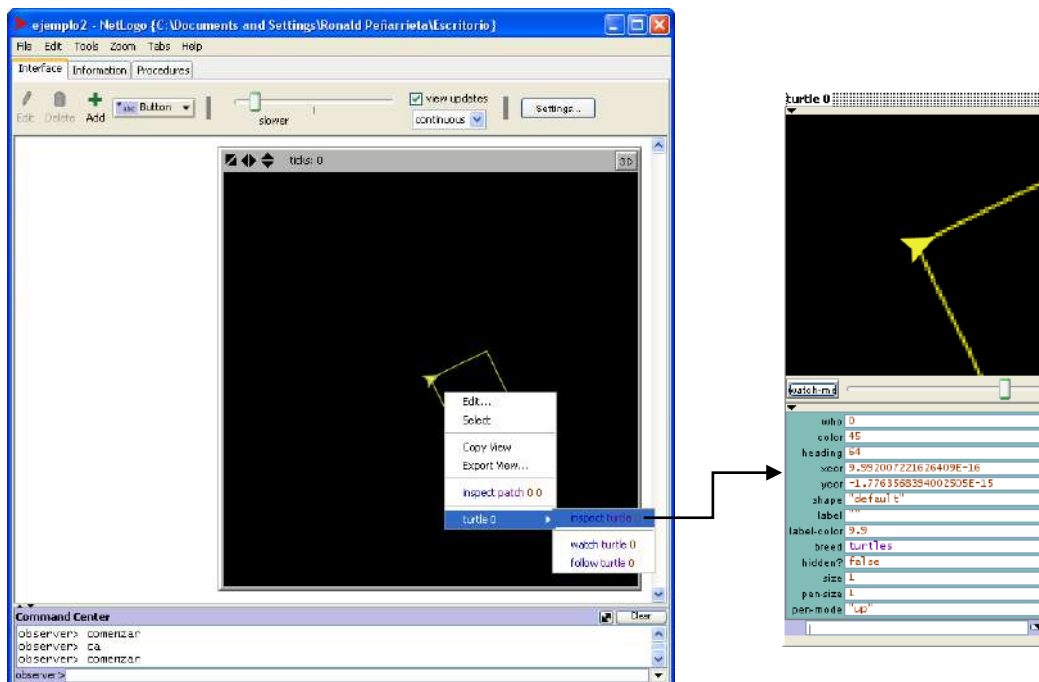
Cada uno de los agentes que creamos, ya sean tortugas, patches o links, tienen una serie de propiedades. Algunas de estas propiedades vienen predefinidas en NetLogo.

Por ejemplo, las tortugas tienen las siguientes propiedades predefinidas:

who	:: identificador (no se puede modificar)
color	:: color
heading	:: orientación
xcor	:: coordenada x
ycor	:: coordenada y
shape	:: forma
label	:: etiqueta
label-color	:: color de la etiqueta
breed	::raza
hidden?	:: ¿visible o no visible?
size	:: tamaño
pen-size	:: tamaño del trazo al desplazarse (cuando pen-mode=down)
pen-mode	:: ¿dejar trazo al desplazarse o no?

Para ver las propiedades de cada tortuga en cada momento, no tenemos más que colocar el puntero del ratón sobre la tortuga y hacer clic en el botón derecho del ratón. Aparecerá una ventana. Debemos seleccionar turtle x, y, a continuación, inspect turtle x (x es el identificador de la tortuga sobre la que hemos colocado el puntero del ratón).

Emergerá una ventana con las propiedades de la tortuga seleccionada en el momento actual:



Más adelante veremos cómo añadir nuevas propiedades a los agentes.

Los patches también tienen sus propiedades:

`pxcor` ;; coordenada x del patch
`pycor` ;; coordenada y del patch
`pcolor` ;; color del patch
`plabel` ;; etiqueta del patch
`plabel-color` ;; color de la etiqueta del patch

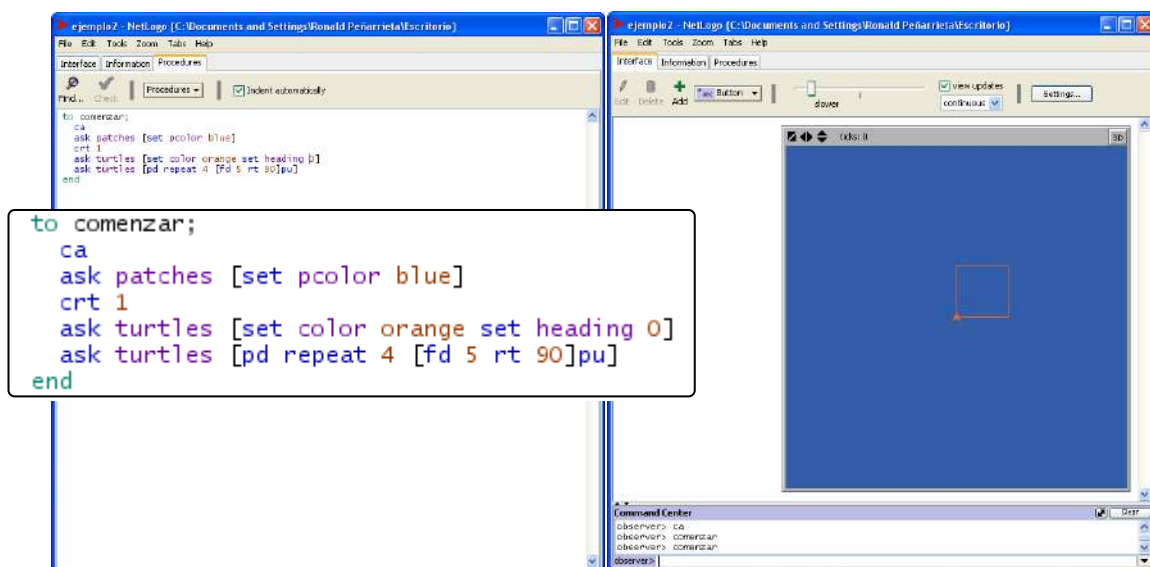
En este ejercicio, modificaremos el procedimiento del ejercicio 3 para que el cuadrado esté alineado con los ejes de coordenadas. El color del cuadrado no puede ser aleatorio, sino que debe ser definido de antemano. Además, podremos el mundo de color azul (es decir, modificaremos la propiedad `pcolor` de todos los patches para que éstos sean de color azul).

Debemos partir del ejercicio anterior y modificar el procedimiento comenzar (en la pestaña `procedures`).

```

to comenzar;
  ca
  ask patches [set pcolor blue]
  crt 1
  ask turtles [set color orange set heading 0]
  ask turtles [pd repeat 4 [fd 5 rt 90]pu]
end
  
```

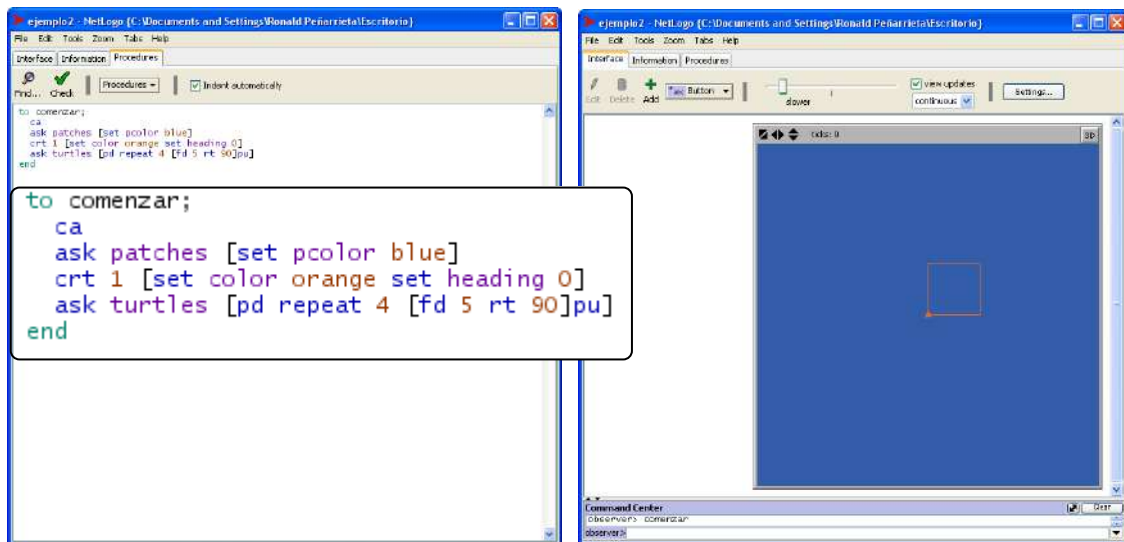
El resultado debería ser semejante al que se muestra a continuación:



Observación: También es posible asignar propiedades a los agentes justo en el momento de crearlos. Comparemos las siguientes órdenes:

OPCIÓN 1:	OPCIÓN 2:
<pre> crt 1 ask turtles [set color orange set heading 0] </pre>	<pre> crt 1 [set color orange set heading 0] </pre>

En el caso de la opción 1, en primer lugar se crea la tortuga (y se le asignan los parámetros por defecto, es decir, color y orientación aleatorias) y, después, con la segunda línea de código, se modifican las propiedades color y heading. En cambio, con la opción 2, la tortuga se está creando directamente con las propiedades de color y heading definidas por el programador.



Ejercicio 5. Propiedades de los agentes (2)

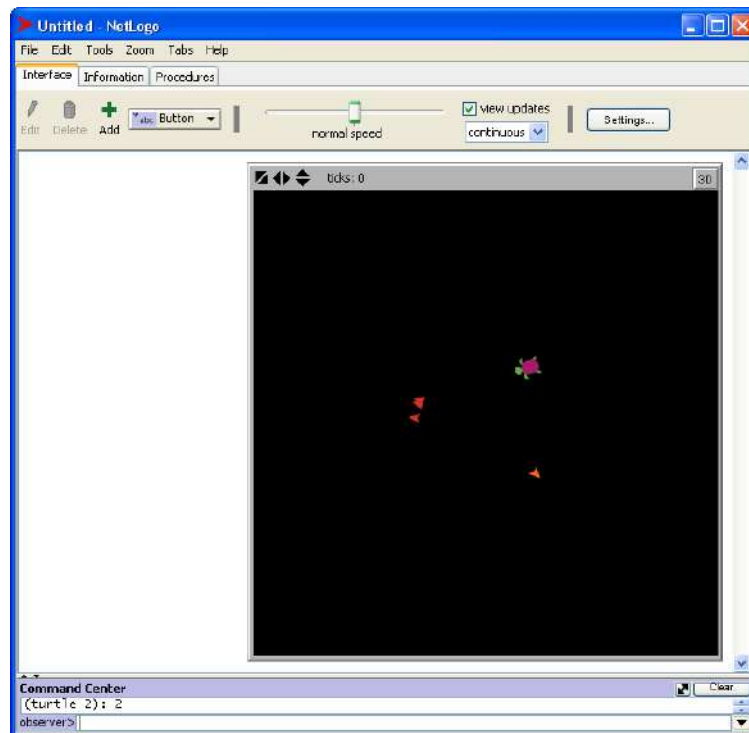
En este ejercicio vamos a repasar rápidamente cómo modificar las propiedades de los agentes. Además, vamos a distinguir cómo dar órdenes a un agente o a un conjunto de agentes. Practica escribiendo las siguientes órdenes en la **ventana de comandos** de Netlogo:

- En primer lugar crearemos 5 tortugas y las desplazaremos 5 unidades:
crt 5
ask turtles [fd 5]

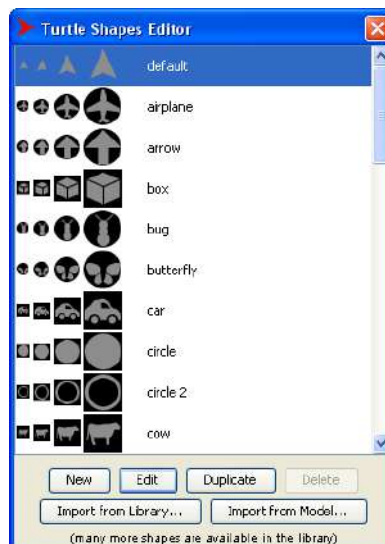
Nota: Hasta ahora hemos visto cómo dar órdenes a todas las tortugas que hay en el mundo. Ahora consideraremos cómo dar órdenes a una tortuga en concreto. Como ya hemos visto, cada tortuga que creamos en Netlogo está representada por un identificador (who). La primera tortuga que creamos tendrá el identificador 0 (who=0), la segunda 1 (who=1) y así sucesivamente.

En este último ejemplo, hemos creado 5 tortugas, por lo que para referirnos a cada una de ellas deberemos emplear **turtle 0**, **turtle 1**, **turtle 2**, **turtle 3** ó **turtle 4** en función de a qué tortuga nos estemos refiriendo.

- Vamos a colocar la tortuga 2 en el punto x = 3 ; y = 4
ask turtle 2 [set xcor 3 set ycor 4] ; equivalente: ask turtle 2 [setxy 3 4]
- Vamos a dar forma de tortuga a la tortuga 2:
ask turtle 2 [set shape "turtle"]
- Aumentamos el tamaño de esta misma tortuga:
ask turtle 2 [set size 2]
- Pedimos a la tortuga 2 que nos muestre su identificador
ask turtle 2 [show who]



Nota: Para ver las formas que pueden adoptar las tortugas (y también para crear las nuestras propias) debemos seleccionar tools en la barra de menú, y a continuación en turtle shapes editor.

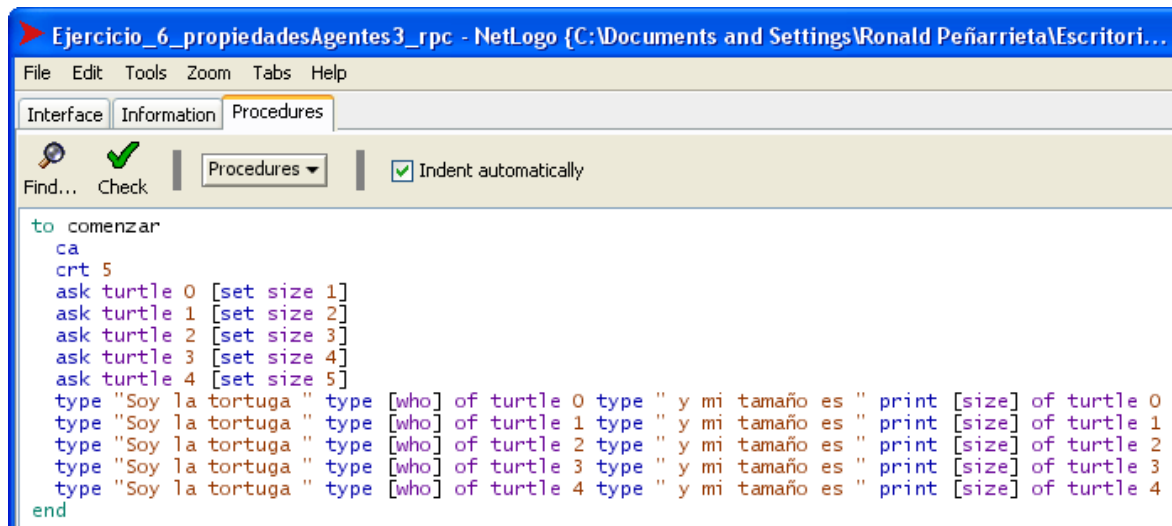


Ejercicio 6. Propiedades de los agentes (3)

Acceso al contenido de las propiedades de un agente.

En este ejercicio vamos a aprender a acceder a las propiedades de un agente.

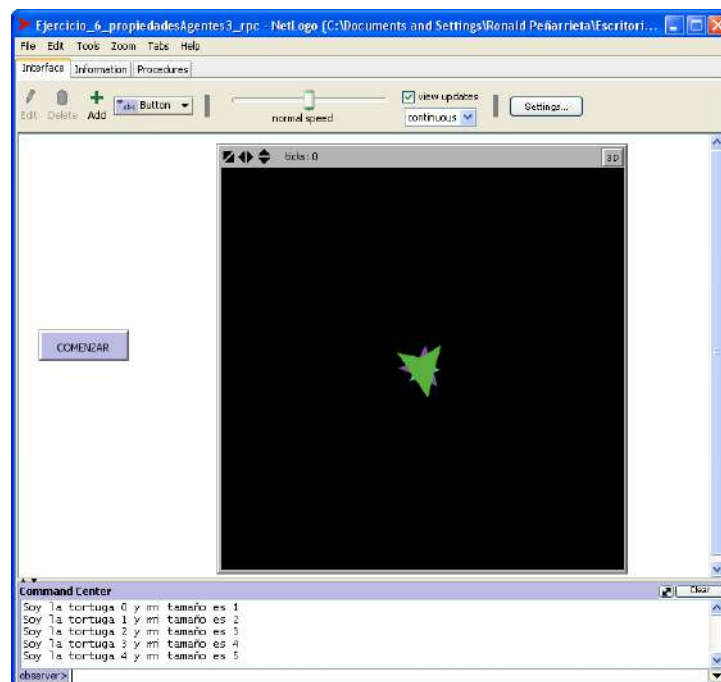
Escribe un procedimiento que cree 5 tortugas, cada una de ellas de un tamaño distinto. Al ejecutar el procedimiento, en la ventana de comandos cada tortuga debe “decir” su identificador y qué tamaño tiene.



```
to comenzar
  ca
  crt 5
  ask turtle 0 [set size 1]
  ask turtle 1 [set size 2]
  ask turtle 2 [set size 3]
  ask turtle 3 [set size 4]
  ask turtle 4 [set size 5]
  type "Soy la tortuga " type [who] of turtle 0 type " y mi tamaño es " print [size] of turtle 0
  type "Soy la tortuga " type [who] of turtle 1 type " y mi tamaño es " print [size] of turtle 1
  type "Soy la tortuga " type [who] of turtle 2 type " y mi tamaño es " print [size] of turtle 2
  type "Soy la tortuga " type [who] of turtle 3 type " y mi tamaño es " print [size] of turtle 3
  type "Soy la tortuga " type [who] of turtle 4 type " y mi tamaño es " print [size] of turtle 4
end
```

Comentarios:

1. Sintaxis para acceder a las propiedades de un agente: **[propiedad] of agente**
2. Para mostrar un mensaje en la ventana de comandos: **type “mensaje”** y **print “mensaje”**. La diferencia entre ambas primitivas es que *print* hace un retorno de carro al final del mensaje y *type* no.



Ejercicio 7. Propiedades de los agentes (4)

Discriminación de agentes por sus propiedades.

Hasta el momento, para identificar a los agentes, o bien les hemos “llamado” a todos a la vez:

```
ask turtles ["comandos"]  
;; pedimos a todas las tortugas que ejecuten ciertos comandos
```

```
ask patches ["comandos"]  
;; pedimos a todos los patches que ejecuten ciertos comandos
```

o bien les hemos “llamado” por su identificador (who en el caso de las tortugas, [pxcor pycor] en el caso de los patches):

```
ask turtle 5 [fd 5]  
;; pedimos que sólo la tortuga 5 avance 5 posiciones
```

```
ask patch 3 4 [set pcolor blue]  
;; pedimos que sólo el patch [3 4] se ponga azul
```

¿Cómo seleccionamos un conjunto de agentes que tienen una propiedad concreta? Es decir, ¿cómo seleccionaríamos por ejemplo todas las tortugas de color rojo o todos los patches que tengan coordenadas x positivas?

Para hacer esta distinción de los agentes a partir de sus propiedades, se utiliza la primitiva with.

La sintaxis es la siguiente:

```
ask "agentes" with ["propiedad"] ["acciones"]
```

A continuación se muestran unos ejemplos de cómo hacer uso de esta primitiva. Escríbelos paso a paso en la ventana de comandos para familiarizarte con el uso de la primitiva with.

PASO 1. Creamos 2 tortugas de tamaño 1, las situamos en una posición aleatoria del mundo y las pintamos de rojo.

```
crt 2 [set shape "turtle" set size 1 set xcor random-pxcor set ycor random-pycor set color red]
```

PASO 2. Creamos 2 tortugas de tamaño 2, las situamos en una posición aleatoria del mundo y las pintamos de azul

```
crt 2 [set shape "turtle" set size 2 set xcor random-pxcor set ycor random-pycor set color blue]
```

PASO 3. Creamos 2 tortugas de tamaño 3, las situamos en una posición aleatoria del mundo y las pintamos de blanco

```
crt 2 [set shape "turtle" set size 3 set xcor random-pxcor set ycor random-pycor set color white]
```

PASO 4. Pedimos a las tortugas rojas que avancen 5 posiciones.

```
ask turtles with [color = red] [fd 5]
```

PASO 5. Pedimos a las tortugas azules que adquieran tamaño 4.

```
ask turtles with [color = blue] [set size 4]
```

PASO 6. Pedimos a las tortugas blancas que muestren “soy blanca” como etiqueta.

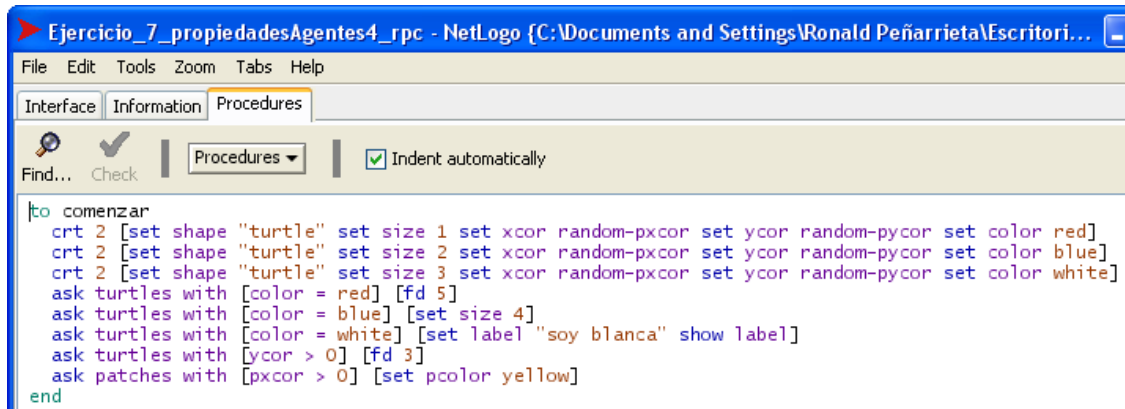
```
ask turtles with [color = white] [set label "soy blanca" show label]
```

PASO 7. Pedimos a las tortugas que tengan coordenada y positiva que avancen 3 posiciones.

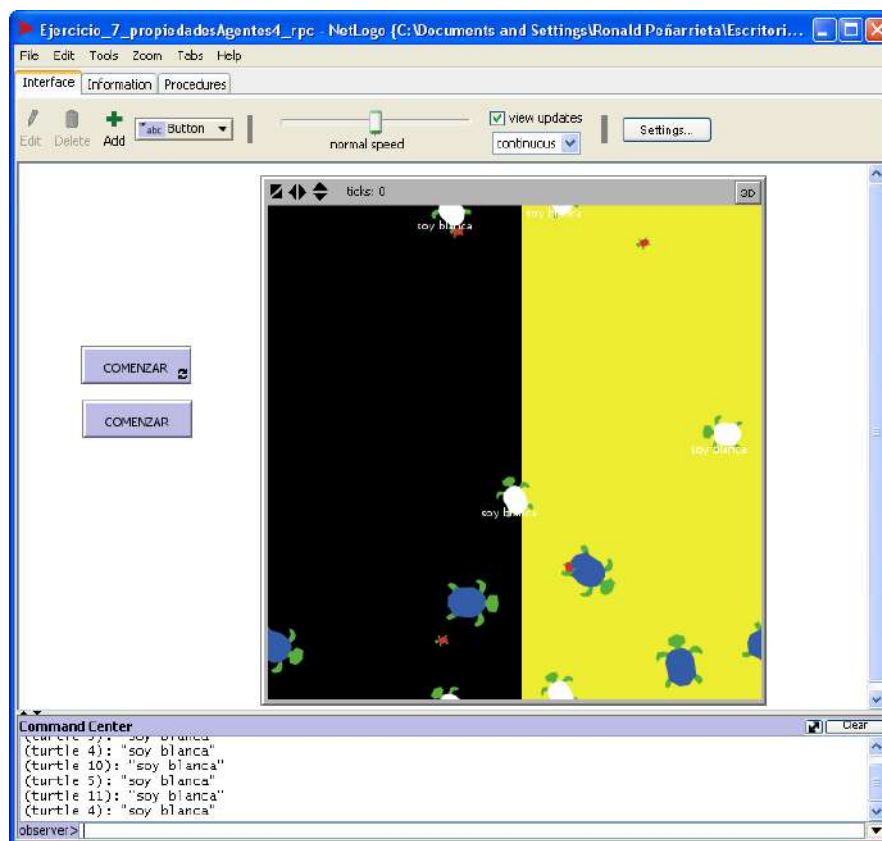
```
ask turtles with [ycor > 0] [fd 3]
```

PASO 8. Pedimos a los patches que tengan coordenada x negativa que se vuelvan de color amarillo.

`ask patches with [pxcor <0] [set pcolor yellow]`



```
to comenzar
  crt 2 [set shape "turtle" set size 1 set xcor random-pxcor set ycor random-pycor set color red]
  crt 2 [set shape "turtle" set size 2 set xcor random-pxcor set ycor random-pycor set color blue]
  crt 2 [set shape "turtle" set size 3 set xcor random-pxcor set ycor random-pycor set color white]
  ask turtles with [color = red] [fd 5]
  ask turtles with [color = blue] [set size 4]
  ask turtles with [color = white] [set label "soy blanca" show label]
  ask turtles with [ycor > 0] [fd 3]
  ask patches with [pxcor > 0] [set pcolor yellow]
end
```



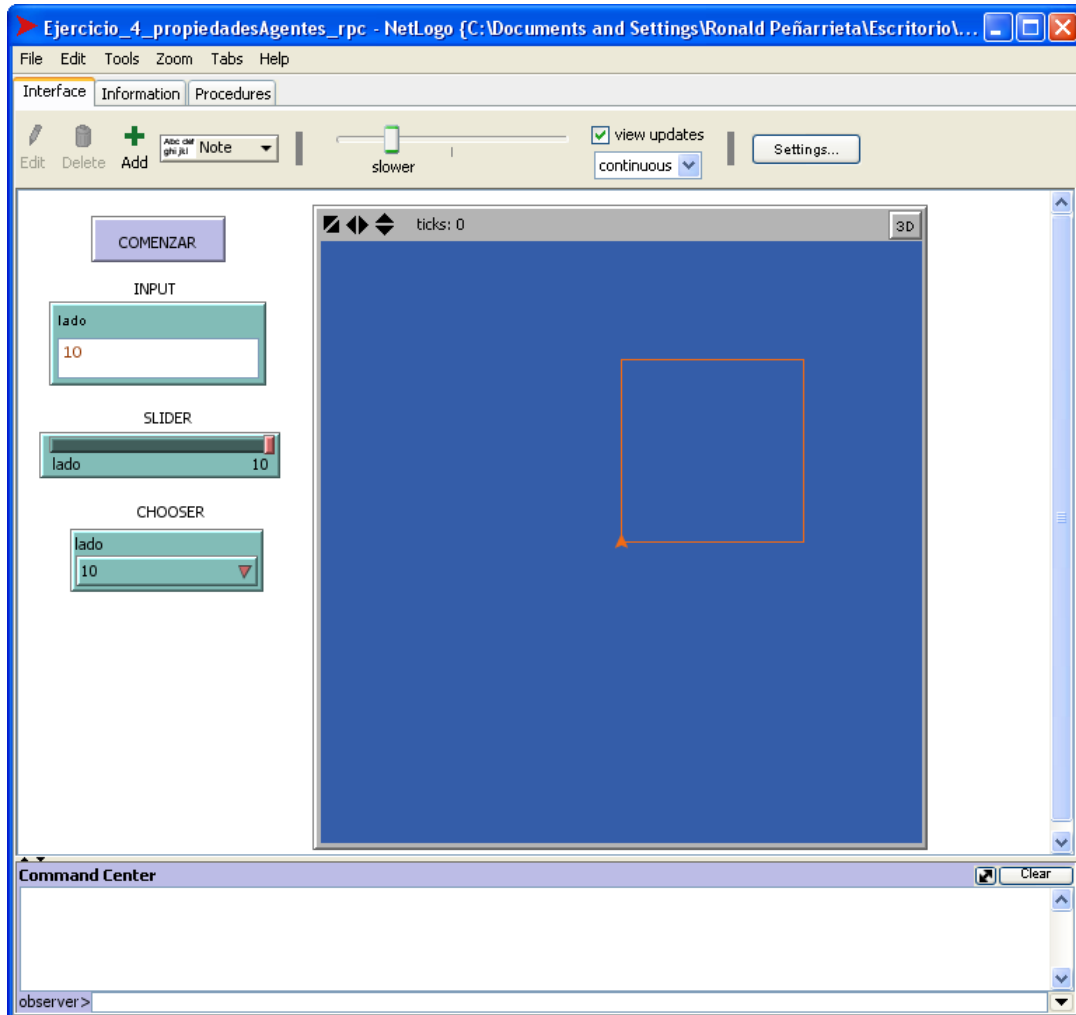
Ejercicio 8. Variables globales. Variables de entrada.

Una variable global es aquella cuyo valor es accesible por todos los procedimientos que se creen dentro de un mismo programa de NetLogo. Un caso particular de variable global son las variables de entrada, denominadas así porque son introducidas por el usuario desde la interfaz del programa.

En este ejercicio vamos a modificar el código del procedimiento del ejercicio 4 para permitir que la longitud del lado del cuadrado pueda ser elegido por el usuario en la vista “interfaz”:

Por tanto, puede resultarte útil partir del archivo correspondiente al ejercicio 4.

En concreto, lo que buscamos es que la longitud del cuadrado, en lugar de ser fija como en el ejercicio 4, pueda ser introducida mediante:

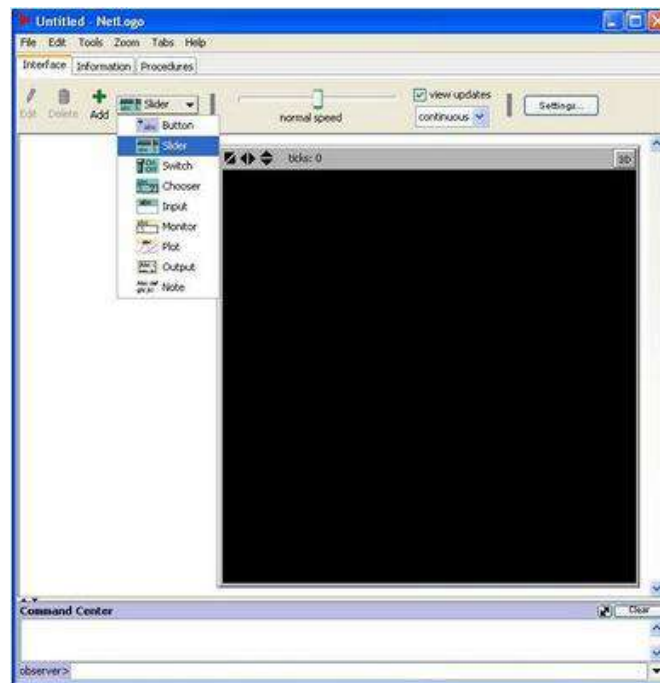


Ejercicio 8.nlogo

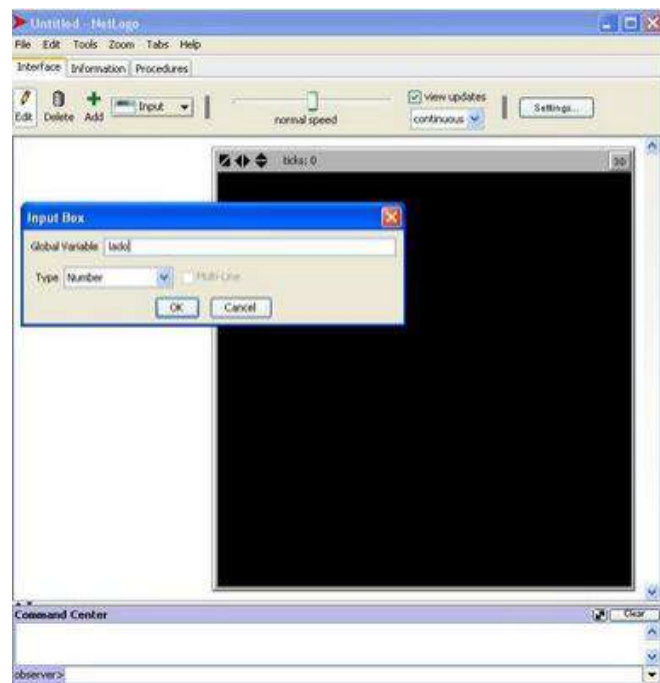
- Un botón *input*
- Una barra desplazadora (*slider*)

- Un botón *chooser*

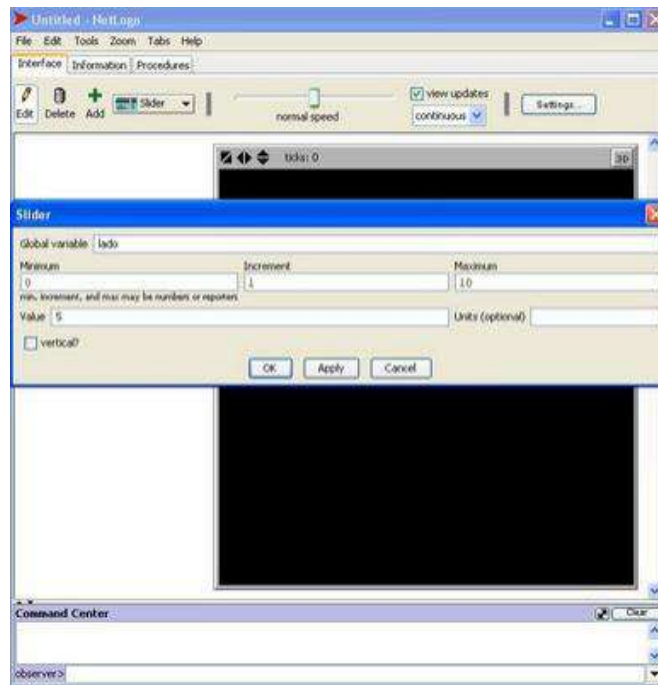
Introduce estos 3 tipos de botones en la vista interfaz (*interface*) de la misma manera que introdujiste el botón del ejercicio 4. Observa la siguiente captura de pantalla:



Al colocar el botón *input*, se te pedirá que le des un nombre a la variable global asignada a dicha entrada. Introduce el nombre “lado”, tal como se indica en la siguiente figura. Puesto que se trata de un número, elige “number” en el campo type:

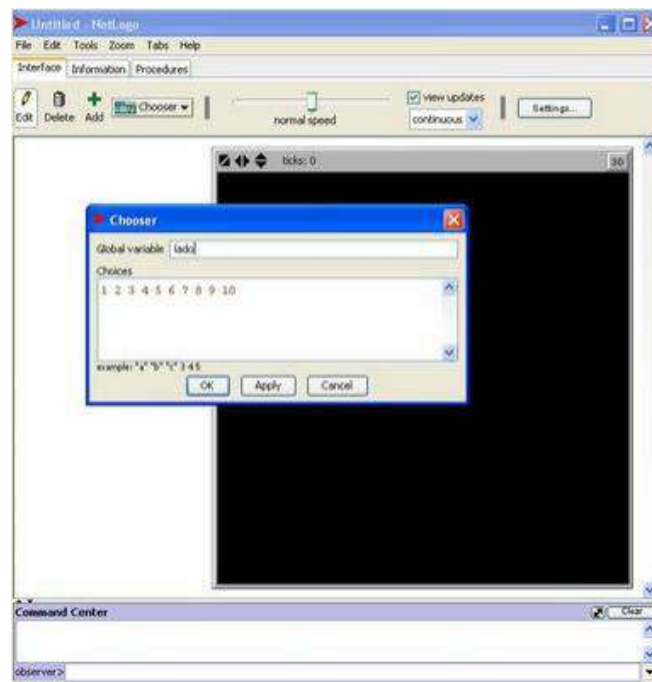


Ahora introduciremos una barra desplazadora (*slider*) de la misma manera:

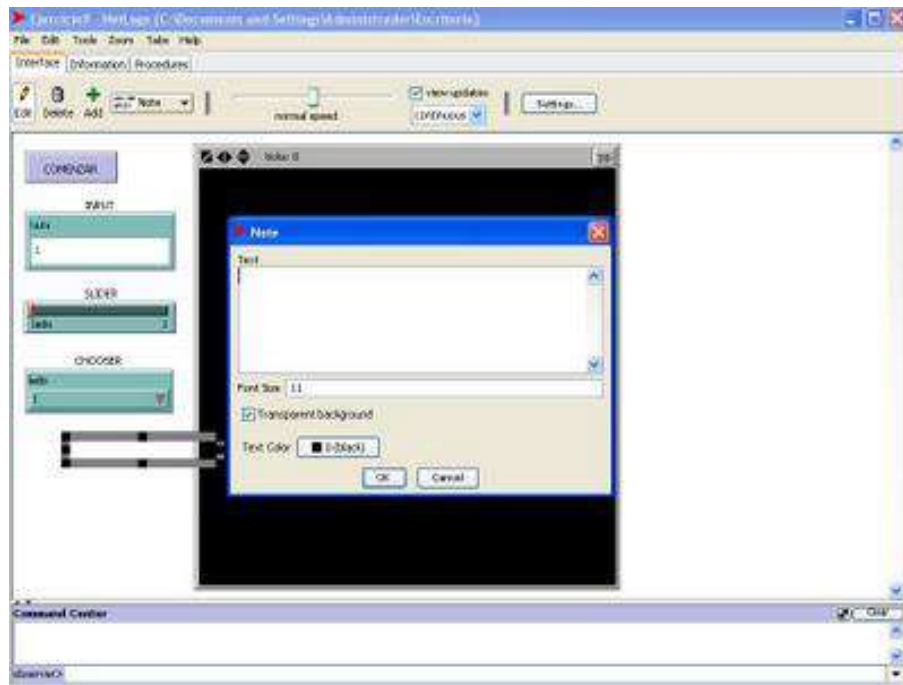


En la ventana que aparece, escribimos “lado” en el campo “*global variable*” y en la siguiente línea el intervalo en el que puede elegirse el lado del cuadrado. Escoge un mínimo de 1, un máximo de 10 y un incremento de 1, para poder incrementar el lado del cuadrado de uno en uno. Finalmente, en el campo *value*, escoge 5 (este es el valor que se tomará la variable “lado” por defecto).

A continuación, introduciremos un menú desplegable (*chooser*), de una manera totalmente similar:



En el campo *global variable*, escribe “lado”. En el campo *choices*, escribe los posibles valores que pueden tomar el lado del cuadro (en este caso 1, 2, 3, 4, 5, 6, 7, 8, 9 ó 10). Finalmente, introduciremos una nota de texto sobre cada botón introducido. Para ello, elige la opción “*note*” en el menú desplegable de dónde has sacado el resto de los botones:



Por último, modifica el procedimiento que creaste en el ejercicio 4 de la siguiente forma:

```

Ejercicio_4_propiedadesAgentes_rpc - NetLogo [C:\Documents and Settings\Ronald Peñarrieta\Escritorio\...
File Edit Tools Zoom Tabs Help
Interface Information Procedures
Find... Check Procedures Indent automatically

to comenzar;
  ca;
  ask patches [set pcolor blue];
  crt 1;
  ask turtles [set color orange set heading 0];
  ask turtles [pd repeat 4 [fd lado rt 90] pu];
end;

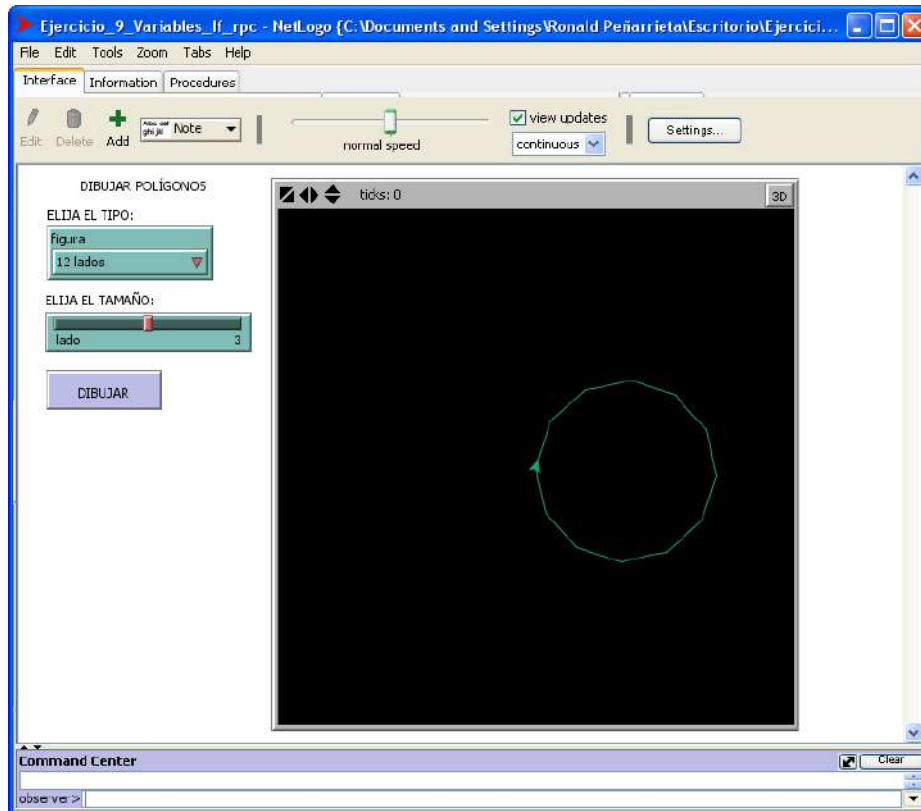
Define el nombre del procedimiento como 'comenzar'
clear-all
mundo azul
crea una tortuga
Color tortuga naranja, orientación hacia arriba
dibuja el cuadrado (trayecto de la tortuga)
fin del procedimiento

```

Vuelve a la pestaña interface, haz clic en el botón comenzar y observa lo que ocurre.

Ejercicio 9. Variables globales y variables locales. Sentencias condicionales if

El objetivo de este ejercicio es crear un procedimiento que dibuje un polígono de n lados y con una longitud de lado dada. Aprovecharemos este ejemplo para distinguir entre variables globales y locales en NetLogo, así como para presentar la sentencia condicional if.



¿Qué diferencia hay entre una variable global y una variable local?

Las variables globales son accesibles para todos los procedimientos del programa, mientras que las variables locales sólo tienen validez dentro de un mismo procedimiento. Otra diferencia es que las variables globales hay que declararlas (es decir, tenemos que decirle a NetLogo al principio del programa que vamos a utilizar una variable global de cierto nombre, aunque el valor se lo asignemos posteriormente). Las variables locales, en cambio, no se declaran al principio del programa, sino que se crean y se les asigna un valor según se va necesitando a lo largo del programa.

Para declarar una variable global, utilizaremos la primitiva *globals*, y, entre corchetes, el nombre de las variables globales que vayamos a utilizar en nuestro programa:

```
globals [  
  variable_global_1  
  variable_global_2  
  variable_global_n ]
```

Para asignar un valor a una variable global, se utiliza la primitiva *set* (recuerda: la misma que se utilizaba para dar valores a las propiedades de los agentes):

```
set variable_global_1 5 ;; asignamos el valor 5 a la variable variable_global_1
```

Para asignar un valor a una variable local, **por primera vez**, se utiliza la primitiva `let`. Si a lo largo del mismo procedimiento debemos asignar un nuevo valor a esta variable, entonces utilizaremos la primitiva `set`:

```
let variable_local 5 ;; asignamos el valor 5 a la variable
;; comandos
```

```
set variable_local 7 ;; asignamos el valor 7 a la variable
```

Nota: Si recuerdas, en el ejercicio 8 ya utilizamos variables globales. Entonces dijimos que la variable “lado” era una variable global. Sin embargo, en aquel caso no la teníamos que declarar dentro del código del programa, ya que ésta quedaba definida dentro de un botón (input, slider, chooser), por lo que el programa se encargaba de declararla de forma interna.

Una vez hechas estas observaciones, comenzaremos a programar nuestro modelo. El código se muestra a continuación:

El código del procedimiento se muestra a continuación:

```
;; Declaración de variables Globales
globals [n] ;; declaración de la variable global n (número de lados del polígono)

;; Comienza el procedimiento

to comenzar
  ca ;; borrar pantalla, variables...
  crt 1 ;; creamos la tortuga que dibujará el polígono

  ;; asignamos un valor a n en función del número de lados del polígono elegido

  if figura = "triángulo"
  [
    set n 3
  ]

  if figura = "cuadrado"
  [
    set n 4
  ]

  if figura = "pentágono"
  [
    set n 5
  ]

  if figura = "hexágono"
  [
    set n 6
  ]

  if figura = "recta"
  [
    set n 1
  ]

  if figura = "12 lados"
  [
    set n 12
  ]
  ;; Definimos el ángulo interior del polígono

  let angulo 360 / n

  ;; Dibujamos el polígono

  ask turtles [repeat n [pd fd lado rt angulo]]
end
```

Observa la sintaxis de las sentencias condicionales *if*:

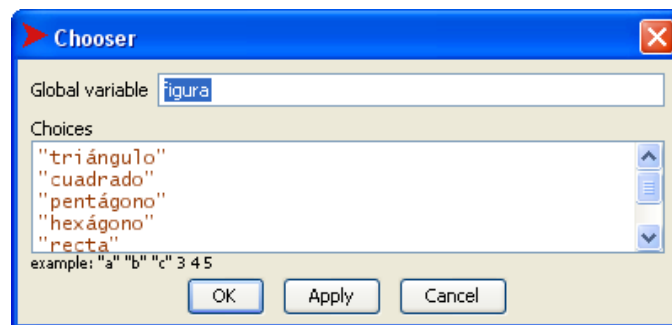
```
if condición
[
  ;; comandos
]
```

Si se cumple la condición que sigue a *if*, entonces se ejecutarán el conjunto de comandos que escribamos a continuación, entre corchetes. Si no se cumple, no se ejecutarán dichos comandos.

Nota: El hecho de emplear *n* como una variable global en este ejemplo es simplemente ilustrativo: ya que sólo existe un procedimiento en nuestro programa, podríamos haber utilizado una variable local en su lugar.

Una vez escrito el procedimiento, en la vista interfaz deberás introducir un botón que inicialice el procedimiento, una barra desplazadora para elegir el lado del polígono y un menú desplegable para elegir el polígono a dibujar, de la misma manera que lo hemos hecho en ejercicios anteriores.

Los valores de lado deberán ser 1, 2, 3, 4, 5 ó 6. Presta atención a la configuración del botón chooser. Las posibles opciones han de escribirse entre comillas:



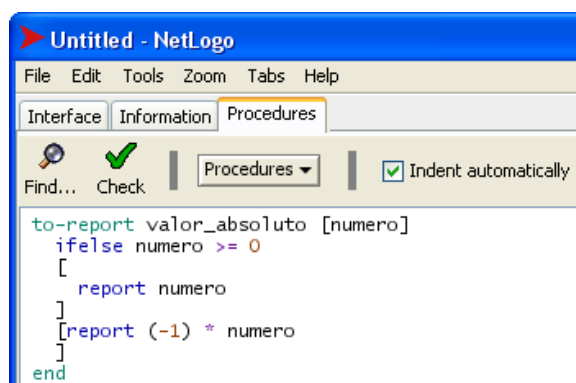
Ejecuta el programa probando los distintos polígonos y tamaños.

Ejercicio 10. Sentencias ifelse. Procedimientos con retorno (to-report)

En el ejercicio 9 introdujimos las sentencias condicionales if. En este ejercicio utilizaremos la sentencia condicional ifelse.

Si recordamos, cuando empleábamos la sentencia if, si se cumplía la condición que proseguía a if, se ejecutaban un conjunto de comandos que escribíamos entre corchetes, y, si no se cumplía, no se hacía nada: el programa ignoraba los comandos escritos entre corchetes.

La filosofía de la sentencia ifelse consiste en que si se cumple una determinada condición, se ejecutarán una serie de comandos (al igual que ocurría con if) pero si no se cumple, en lugar de no hacer nada, se ejecutarán otra serie de comandos. Su sintaxis es la que se muestra a continuación:



Con procedimientos con retorno, nos referimos a aquellos que devuelven cierto valor al procedimiento que les ha llamado.

Un procedimiento con retorno (a diferencia de un procedimiento convencional) comienza con la palabra clave to-report y termina con la palabra end. La variable o valor que devuelve al procedimiento llamante va precedido de la palabra clave report.

La sintaxis genérica de este tipo de procedimientos es la que se muestra a continuación:

```
to-report nombre_del_procedimiento [valor_recibido_del_procedimiento_llamante]
  ;; comandos
  ;; [...]
  report valor_resultante
end
```

Aclaremos todo esto con un programa sencillo: elaboraremos un procedimiento que recibirá un número y nos devolverá su valor absoluto.

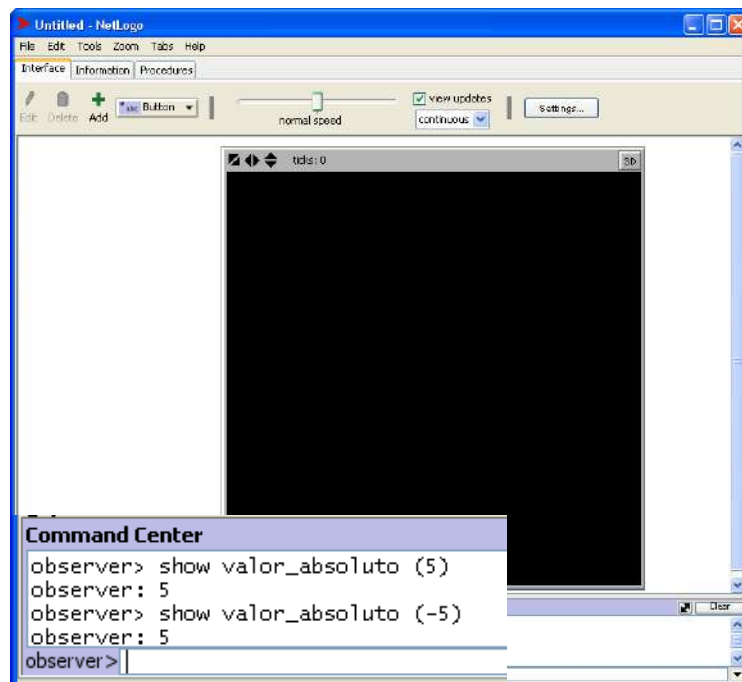
El código es el siguiente (escríbelo en la pantalla de procedimientos):

```
to-report valor_absoluto [numero]
  ifelse numero >= 0
  [
    report numero
  ]
  [
    report (-1) * numero
  ]
end
```

A continuación, escribe lo siguiente en la ventana de comandos, para probar el correcto funcionamiento del procedimiento que has elaborado:

show valor_absoluto (5)

how valor_absoluto (-5)



Ejercicio 11. Propiedades adicionales de los agentes

En el ejercicio 4 vimos que cada agente (ya sea tortuga, patch o link) tiene una serie de propiedades intrínsecas. Por ejemplo, las tortugas tienen las siguientes propiedades por defecto:

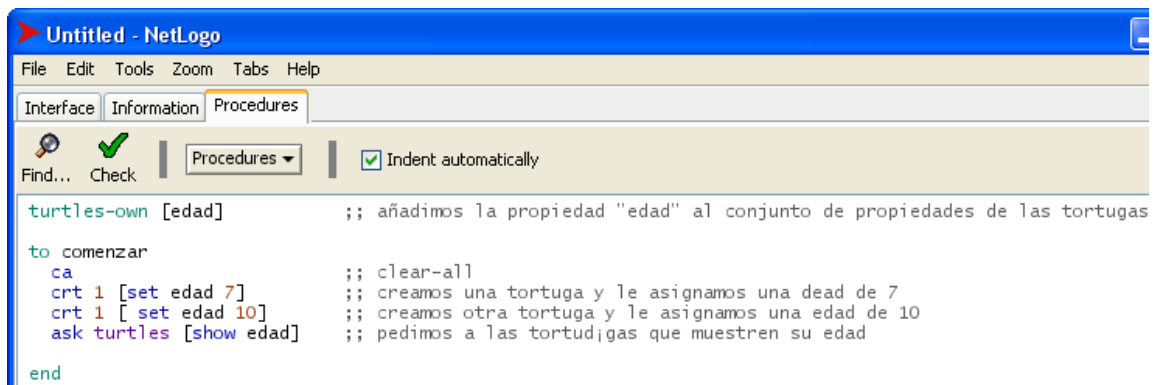
```
who      ;; identificador (no se puede modificar)
color    ;; color
heading  ;; orientación
xcor     ;; coordenada x
ycor     ;; coordenada y
shape    ;; forma
label    ;; etiqueta
label-color ;; color de la etiqueta
breed    ;; raza
hidden?  ;; ¿visible o no visible?
size     ;; tamaño
pen-size  ;; tamaño del trazo al desplazarse (cuando pen-mode=down)
pen-mode  ;; ¿dejar trazo al desplazarse o no?
```

Nosotros podemos añadir propiedades adicionales a los agentes. Por ejemplo, supón que estás haciendo un estudio sobre tortugas y que te resulta interesante poder asignarle a cada tortuga una edad.

Para añadir una propiedad adicional a un agente, se utiliza la primitiva `own`. Veamos cómo se añade la propiedad “edad” a una tortuga:

`turtles-own [edad]` ;; añadimos la propiedad “edad” al conjunto de propiedades de las tortugas.

Para fijar ideas, crea este sencillo procedimiento:

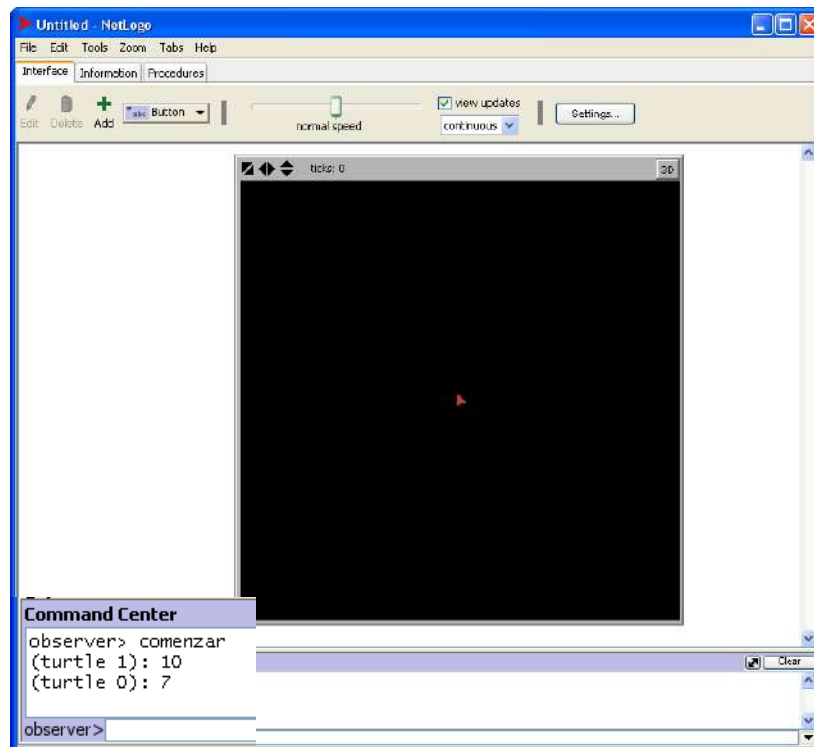


```
Untitled - NetLogo
File Edit Tools Zoom Tabs Help
Interface Information Procedures
Find... Check Procedures Indent automatically

turtles-own [edad] ;; añadimos la propiedad "edad" al conjunto de propiedades de las tortugas

to comenzar
  ca ;; clear-all
  crt 1 [set edad 7] ;; creamos una tortuga y le asignamos una edad de 7
  crt 1 [set edad 10] ;; creamos otra tortuga y le asignamos una edad de 10
  ask turtles [show edad] ;; pedimos a las tortugas que muestren su edad
end
```

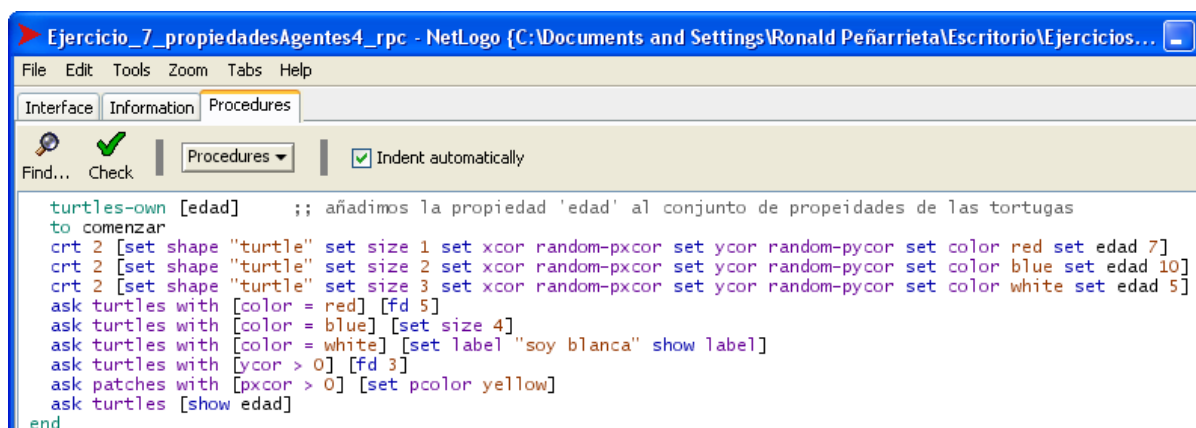
Después, en la ventana de comandos escribe “comenzar” para ejecutar el procedimiento que acabas de crear. Verás que se muestra, en primer lugar, el agente al que hemos pedido que ejecute la acción y, a continuación, la propiedad que le hemos mandado mostrar (en este caso “edad”).

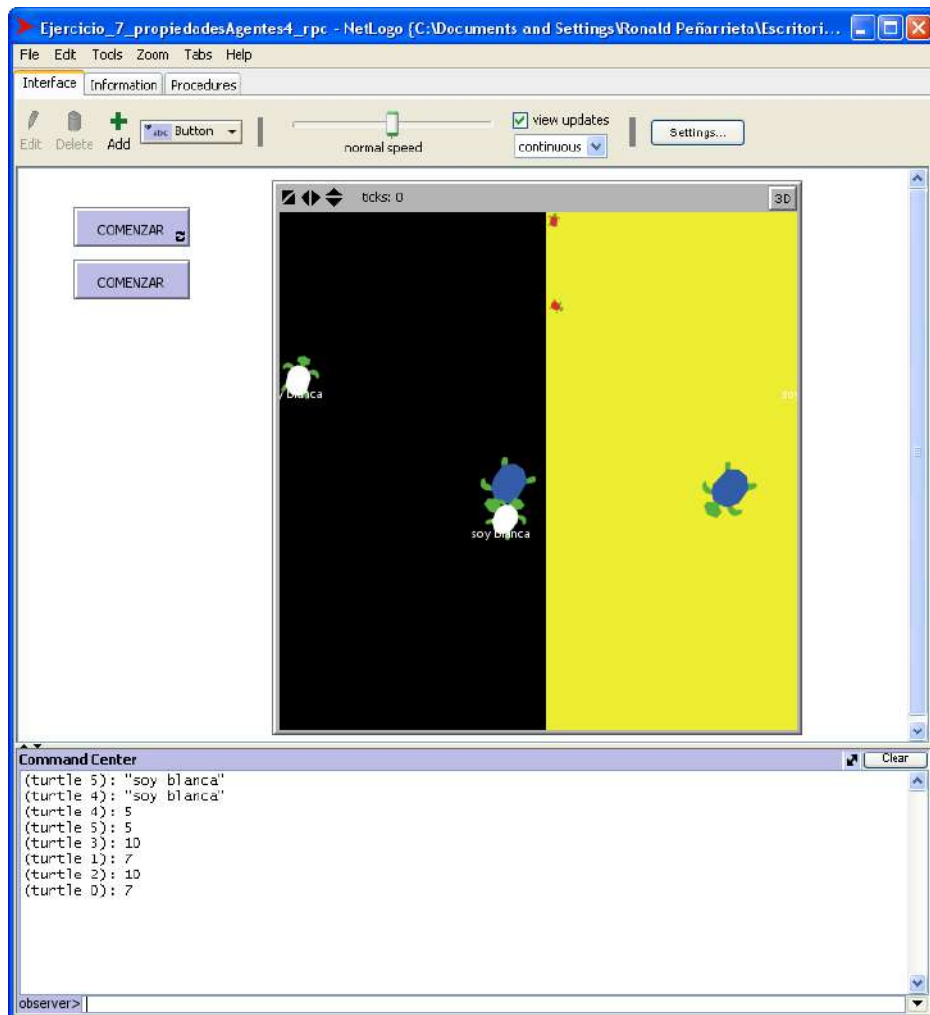


Si recuerdas, en casos anteriores, para mostrar las propiedades de un agente en la ventana de comandos empleábamos la primitiva `print` (ver ejercicio 6). La diferencia entre `print` y `show` es que con `show` se muestra el agente llamado, mientras que con `print` no. Ambas dejan un retorno de carro al final.

Para comprobar esta diferencia, puedes sustituir la línea de código `ask turtles [show edad]` por la siguiente:
`ask turtles [print edad]`

Como alternativa, se propone 'colocarle la edad' a las tortugas del ejercicio 7. Vea que sucede y analice los cambios en las edades de las tortugas: ¿Porqué tienen edades diferentes a las asignadas?





Ejercicio 12. “Razas” (breeds) de agentes

Hasta el momento, hemos supuesto que todas las tortugas que empleamos en el modelo van a tener el mismo tipo de propiedades. Por ejemplo, en el ejercicio 11 creamos una propiedad adicional (edad) para todas las tortugas. Sin embargo, supongamos que en nuestro modelo necesitamos dos tipos de agentes lo suficientemente diferenciados como para que cada uno de ellos tenga unas propiedades diferentes.

A cada tipo de agente con propiedades diferenciadas, Netlogo lo denomina “raza” (*breed*). Para definir una raza, utilizaremos la primitiva *breed* y, a continuación, entre corchetes, indicaremos el nombre del conjunto de agentes de la nueva raza (en plural) y el nombre del agente de la nueva raza (en singular). Por ejemplo, para definir la raza “coches” emplearemos la siguiente línea de código:

breed [coches coche]

Para asignar nuevas propiedades a la nueva raza de agentes, procederemos de la misma forma que en el ejercicio 11, utilizando la primitiva *own*. Por tanto, para añadir la propiedad “edad” a la raza coche, emplearemos la siguiente sentencia:

coches-own [edad]

Una vez creada la nueva raza, podemos decirle a Netlogo la forma que debe dar a los agentes de esta raza cada vez que le mandemos crear alguno de ellos. Para ello se emplea la primitiva *set-default-shape*. Por ejemplo, para que los agentes de la raza coche tengan forma de coche por defecto cada vez que los creemos, deberemos usar la siguiente sentencia:

set-default-shape coches “car”

Para crear los agentes de la nueva raza, usaremos la primitiva *create* seguida de un guión y el nombre de la raza. Por ejemplo, para crear 20 coches emplearemos la siguiente codificación:

create-coches 20

Para pedir a los agentes de una raza concreta que realicen alguna acción, empleamos la primitiva *ask* seguido de la raza y, a continuación, entre corchetes, las acciones a realizar. Por ejemplo, para pedir a los coches que avancen 10 posiciones, emplearemos:

ask coches [fd 10]

En el siguiente ejemplo crearemos dos razas de agentes distintas: por un lado una raza “coches” con la propiedad “edad” y una raza “flores” con la propiedad “planta”. Asignaremos formas distintas a los agentes de las dos razas y finalmente ordenaremos acciones distintas a cada raza de agentes.

```

Untitled - NetLogo
File Edit Tools Zoom Tabs Help
Interface Information Procedures
Find... Check Procedures Indent automatically

;; DEFINICIÓN DE RAZAS
breed [coches coche]           ;; definimos el primer tipo de agente
breed [flores flor]            ;; definimos el segundo tipo de agente

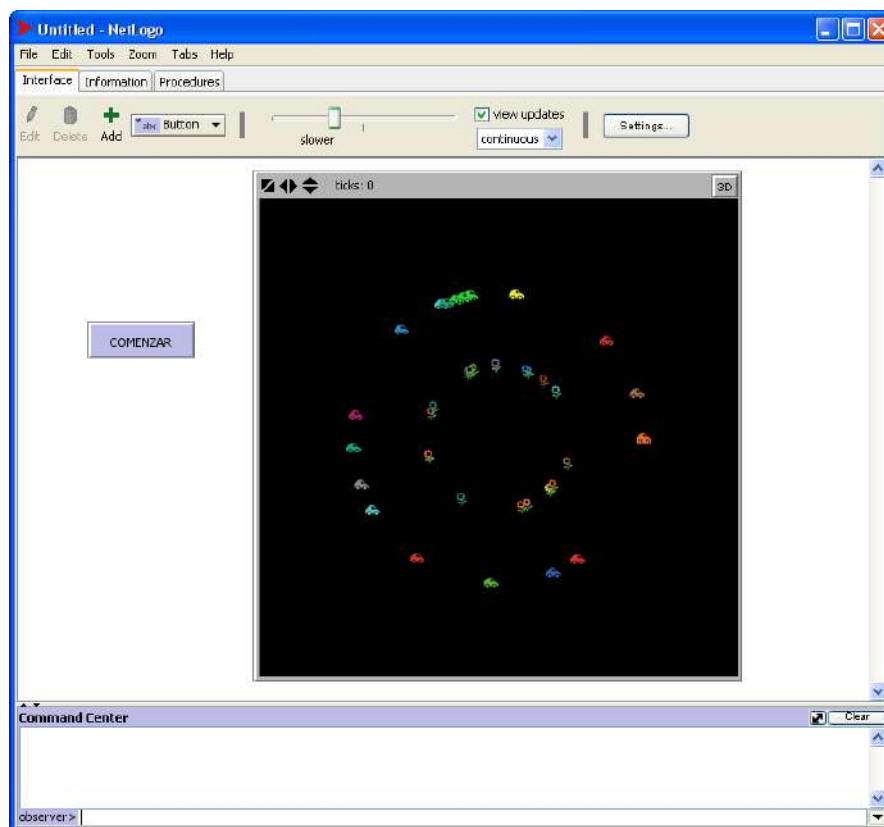
;;DEFINICIÓN DE PROPIEDADES
coches-own [edad]              ; añadimos una propiedad adicional a los coches
flores-own [planta]            ; añadimos una propiedad adicional a las plantas

;;PROCEDIMIENTOS
to comenzar
  ca
  set-default-shape coches "car" ; establecemos una forma por defecto para el tipo de agente "coche"
  set-default-shape flores "flower" ; establecemos una forma por defecto para el tipo de agente "flor"
  create-coches 20                ; creamos 20 coches
  create-flores 20                ; creamos 20 flores
  ask coches [set edad 15]        ; definimos la edad de todos los coches
  ask flores [set planta "castaño"] ; definimos la planta de la que rproducen las hojas

  ask coches [fd 10]              ; pedimos a los coches que avancen 10 posiciones
  ask flores [fd 5]              ; pedimos a las flores que avancen 10 posiciones
end

```

Al 'ejecutar' el modelo la aplicación tenemos:



Ejercicio 13. Bucles while

En este ejercicio introduciremos la primitiva *while*. Esta primitiva sirve para repetir un conjunto de acciones mientras se cumpla una determinada condición. En el ejercicio 1 ya empleamos una primitiva que nos permitía repetir un número de comandos: la primitiva *repeat*. Sin embargo, con *repeat*, el número de veces que se repiten los comandos ha de ser un número fijo (no se evalúa ninguna condición para ejecutar el conjunto de comandos a repetir), mientras que con *while* sí que se evalúa una condición: si se cumple la condición se entra en el bucle si no se cumple se sale del bucle.

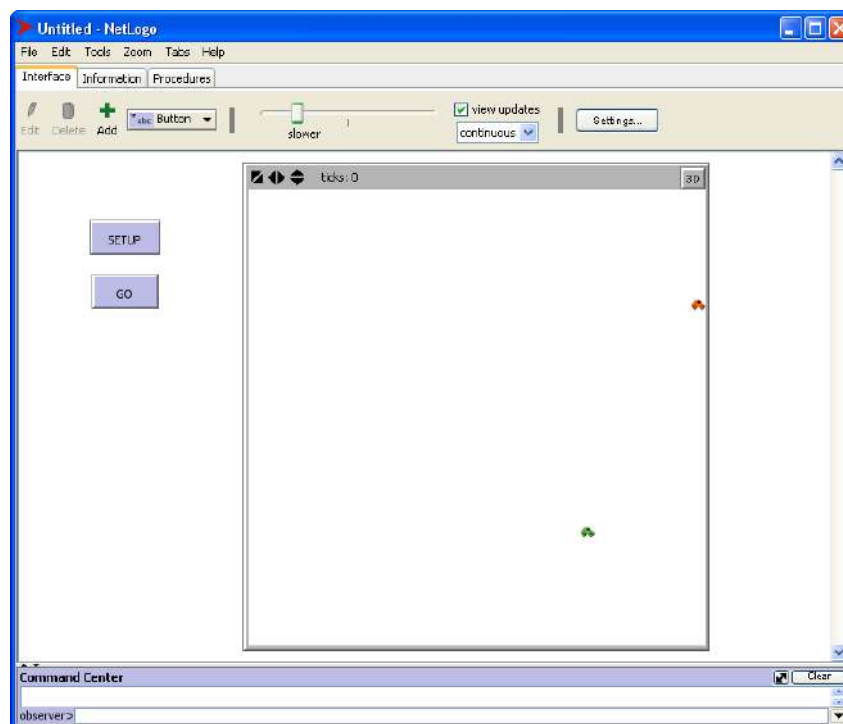
Su sintaxis es la que se muestra a continuación:

```
while condición  
  
;; comandos  
  
]
```

Compárala con la sintaxis de la primitiva *repeat*:

```
repeat n ;; n es el número de veces que queremos que se repita el conjunto de comandos  
  
[  
  
;; comandos  
  
]
```

Para ilustrar el funcionamiento de los bucles *while*, vamos a construir un pequeño modelo. En él crearemos dos coches que, partiendo del lado izquierdo del “mundo” deberán alcanzar el lado derecho del “mundo” avanzando una distancia aleatoria en cada iteración.



El código se muestra a continuación:

```

;;DECLARACIÓN DE RAZAS
breed [coches coche]

;; PROCEDIMIENTO SETUP (preparación)
to setup
  ca
  set-default-shape coches "car"           ;; clear-all
  ask patches [set pcolor white]           ;; la raza de coches tendrá forma de coche (car) por defecto
  create-coches 2                           ;; el mundo será de color blanco
  [                                         ;; creamos 2 coches
    set xcor min-pxcor + 1                 ;; los colocamos en la parte izquierda del mundo
    set heading 90                         ;; definimos su orientación "mirando"
  ]
  ask coche 0 [set ycor (- world-height / 4) set color green] ;; definimos el color y la coordenada 'y' inicial del primer coche
  ask coche 1 [set ycor world-height / 4 set color orange]   ;; definimos el color y la coordenada 'y' inicial del segundo coche
end

;; PROCEDIMIENTO GO (ejecución)
to go
  while [all? coches [xcor < max-pxcor]] ;; (*) mientras la coordenada 'x' de todas las tortugas sean inferiores a la coordenada 'x'
  ;; del patch situado más a la derecha se entra en el bucle (while) y los
  ;; coches seguirán desplazándose una cantidad aleatoria
  [
    let distancia_coche_1 random 3        ;; en esta iteración, el coche 1 avanzará una dist. aleatoria entre 0 y 2
    let distancia_coche_2 random 3        ;; en esta iteración, el coche 2 avanzará una dist. aleatoria entre 0 y 2
    ask coche 0 [fd distancia_coche_1]    ;; pedimos al primer coche avance la cantidad aleatoria obtenida
    ask coche 1 [fd distancia_coche_2]    ;; pedimos al segundo coche avance la cantidad aleatoria obtenida
  ]
end

```

En primer lugar definimos la raza “coches” con la palabra clave *breed*.

En este modelo tenemos dos procedimientos, el primero de ellos llamado **setup** y el segundo llamado **go**. Esto es muy común en los sistemas modelados en Netlogo. En general, el procedimiento *setup* inicializa las variables, los agentes, las gráficas, la interfaz, etc. y prepara el terreno para que el procedimiento *go* ejecute la simulación del modelo.

En nuestro caso, el procedimiento *setup* define la forma por defecto de la raza “coches”, pinta el mundo de color blanco, crea dos coches, cada uno de un color y los coloca en la parte izquierda del mundo.

El procedimiento *go* se encarga de hacer avanzar los coches hasta que uno de ellos alcanza la parte derecha del mundo.

Observa que la condición para que el bucle *while* se siga ejecutando es que todos los coches tengan una coordenada *x* inferior a la coordenada *x* del patch situado más a la derecha del mundo (*max-pxcor*)

Observa también cómo funciona la primitiva *all?*, que sirve para evaluar las propiedades de todo un conjunto de agentes.

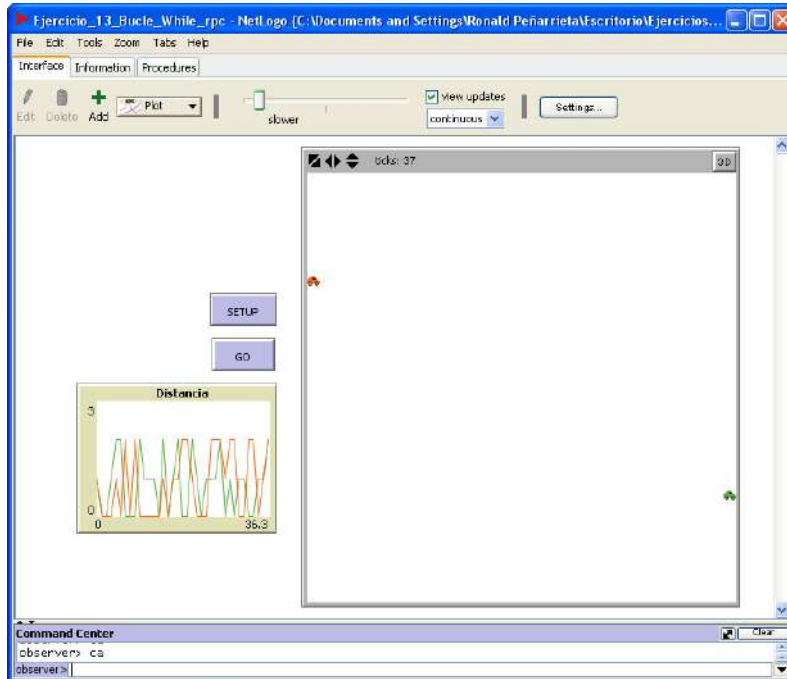
No olvides crear dos botones en la vista interfaz: uno que ejecute el procedimiento *setup* y otro que ejecute el procedimiento *go*.

Ejecuta el modelo, para ello pulsa primeramente el botón *setup*, para preparar el modelo. Finalmente pulsa el botón *go*, para iniciar la simulación. Es posible que tengas que bajar la velocidad de la simulación para poder ver cómo se desplazan los coches a lo largo del mundo.

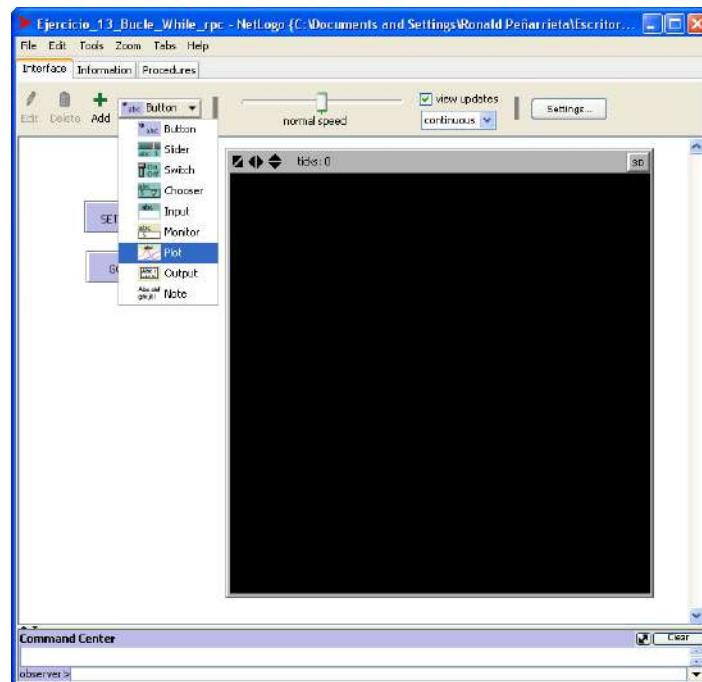
Puedes descargarte el código fuente del programa en la sección de 'Archivos adjuntos', al final de esta página.

Ejercicio 14. Representaciones gráficas en Netlogo

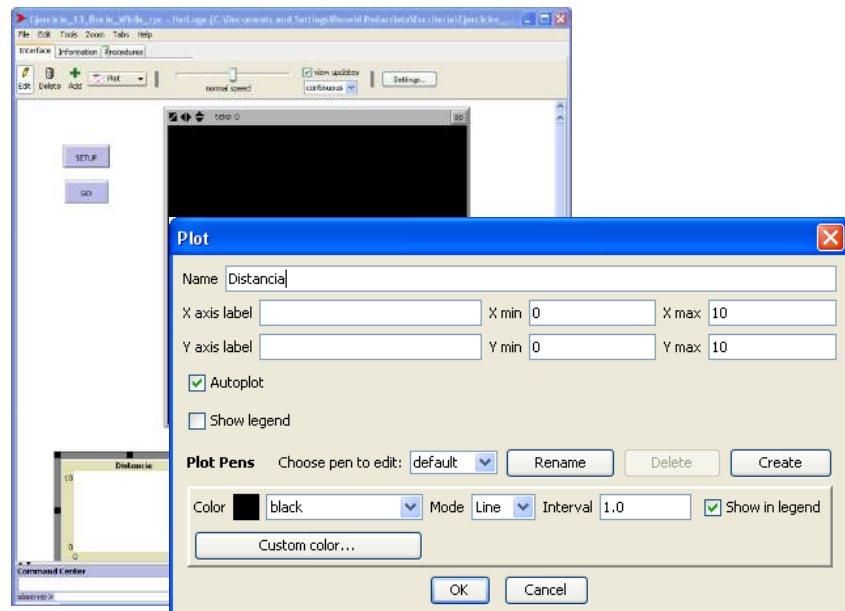
Para realizar este ejercicio partiremos del código del ejercicio anterior (ejercicio 13). Lo que pretendemos hacer es representar gráficamente la distancia que avanza cada uno de los coches del ejercicio anterior en cada iteración.



Para ello, lo primero será colocar un gráfico en la vista interfaz. Las gráficas se introducen de la misma manera que los botones, barras desplazadoras (sliders), etc. Selecciona "plot" tal como se muestra en la siguiente captura de pantalla:



Coloca la gráfica en un hueco libre de la interfaz. Automáticamente se abrirá una ventana con las propiedades de la gráfica, que deberemos rellenar:



En el campo name introduciremos un nombre para esta gráfica. Escribe “Distancia” (sin las comillas). Debemos recordar el nombre que introducimos aquí, ya que lo utilizaremos en el código del programa para indicarle a Netlogo dónde debe dibujar.

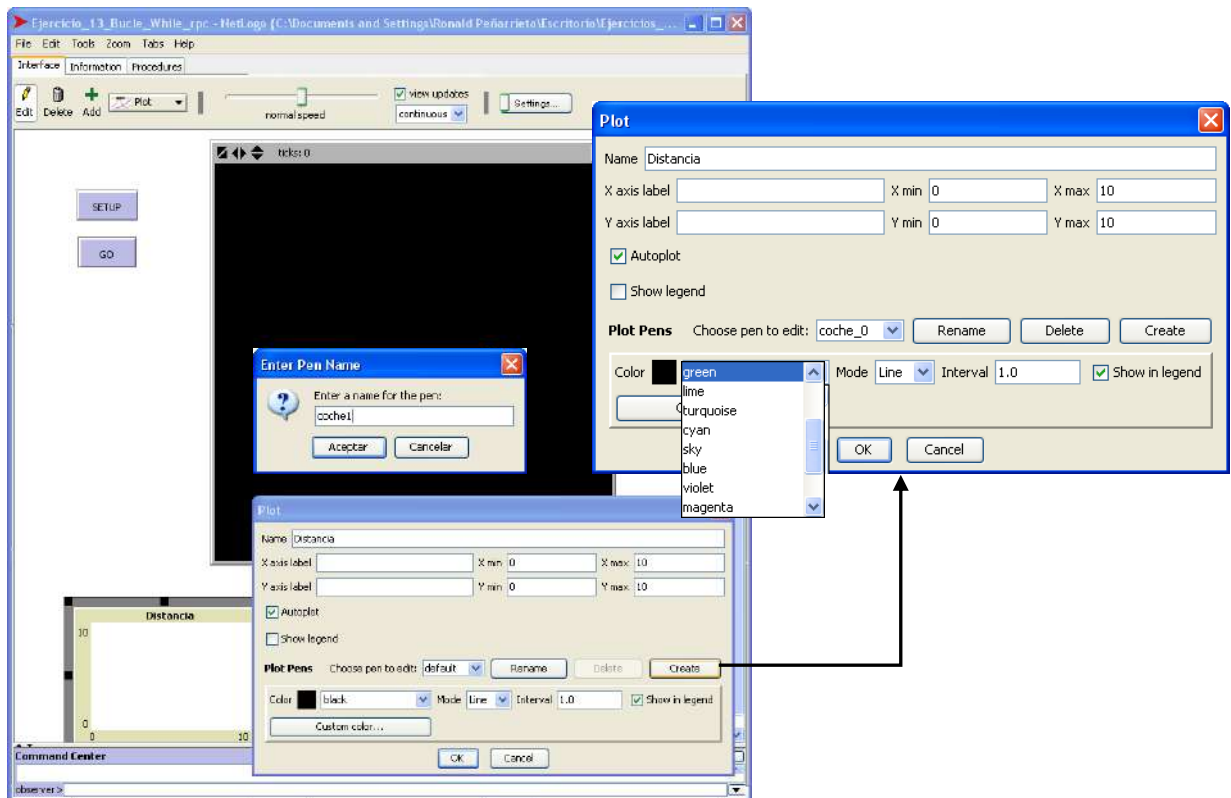
A continuación, las líneas X axis label y Y axis label nos permiten introducir una etiqueta para los ejes de coordenadas, así como definir el rango en el que éstos estarán definidos. En un primer momento no modificaremos estos parámetros.

Ahora nos centraremos en el campo “Plot Pens”, que nos permitirá seleccionar lápices de distintos colores para los distintos datos que podemos representar dentro de una misma gráfica.

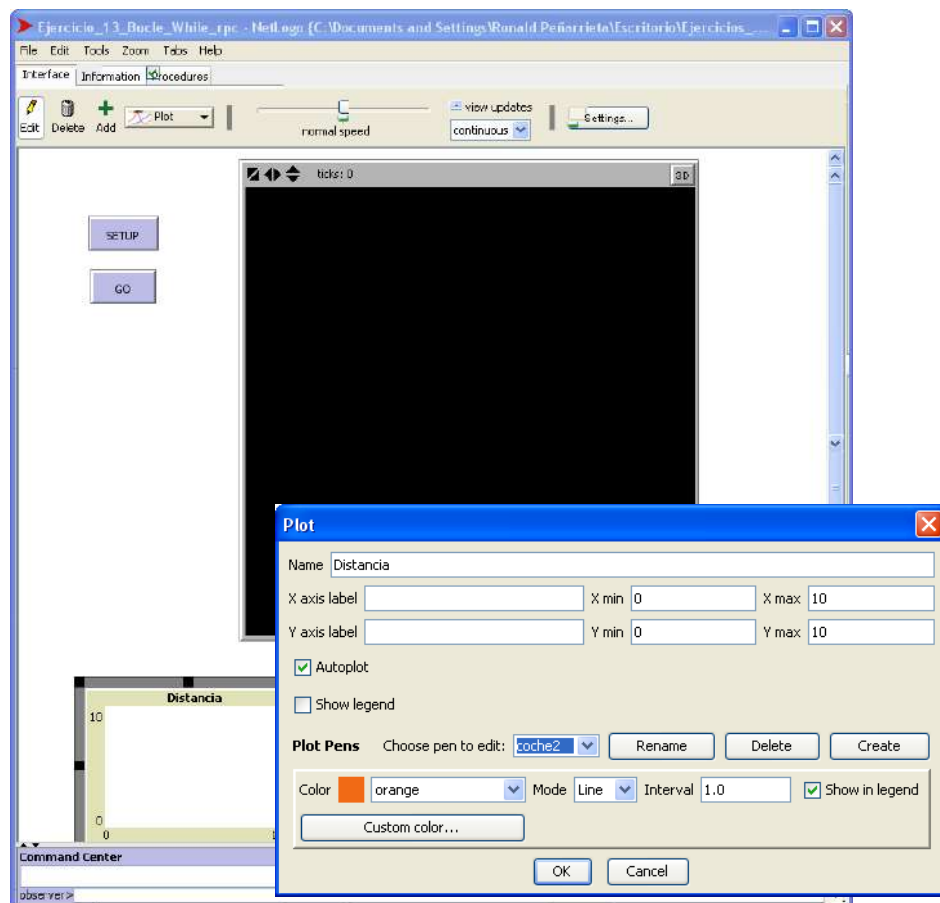
Como el primero de los coches (coche 0) tiene color verde, representaremos su distancia avanzada en cada iteración mediante trazos verdes en el gráfico que hemos creado.

Para ello, pulsamos el botón create. Aparecerá una ventana en la que deberemos introducir un nombre para el lápiz que estamos creando. Introduce “coche1” en este campo, tal como se indica en la siguiente captura de pantalla:

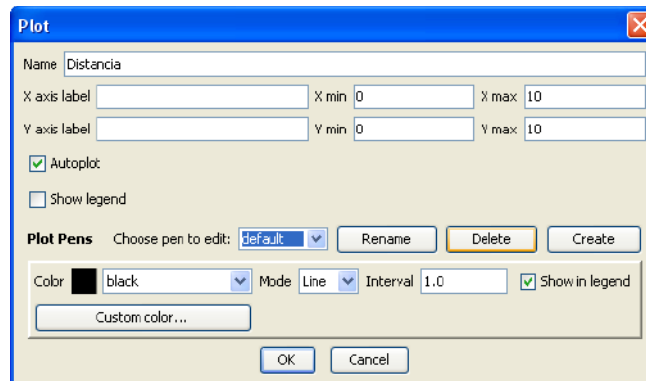
A continuación elige el color verde para este lápiz:



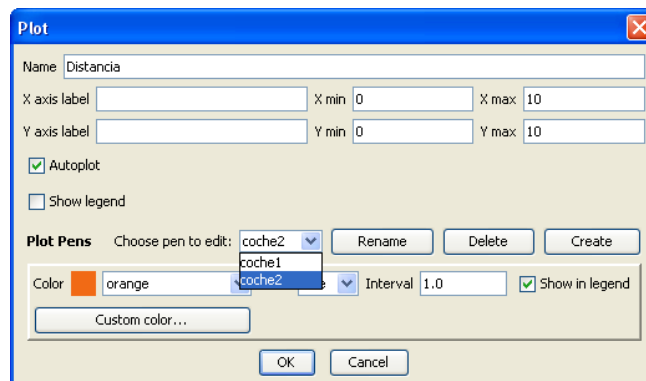
Repite los mismos pasos para crear un lápiz para el segundo coche (coche1). Haz que este lápiz sea de color naranja, para que coincida con el color del coche.



Finalmente escoge el lápiz default y elimínalo del conjunto de lápices. Para ello pulsa el botón delete.



En la lista desplegable de lápices sólo deberían quedar los lápices correspondientes a coche0 y coche1. Marca la casilla show legend y pulsa el botón ok para finalizar.



Ahora introduciremos un nuevo procedimiento llamado “grafica” en el código del ejercicio 13. El código correspondiente al ejercicio 14 se muestra a continuación. Puedes descargarlo en la sección de 'Archivos adjuntos' al final de esta página.

```

;; DECLARACIÓN DE VARIABLES GLOBALES
globals
[
  distancia_coche_1          ;; dist. que avanzará el coche 1 en cada iteración
  distancia_coche_2          ;; dist. que avanzará el coche 2 en cada iteración
]

;;DECLARACIÓN DE RAZAS
breed [coches coche]

;; PROCEDIMIENTO SETUP (preparación)
to setup
  ca                          ;; clear-all
  set-default-shape coches "car"  ;; la raza de coches tendrá forma de coche (car) por defecto
  ask patches [set pcolor white]  ;; el mundo será de color blanco
  create-coches 2              ;; creamos 2 coches
  [
    set xcor min-pxcor + 1      ;; los colocamos en la parte izquierda del mundo
    set heading 90              ;; definimos su orientación "mirando"
  ]
  ask coche 0 [set ycor ( - world-height / 4) set color green]      ;; definimos el color y la coordenada 'y' inicial del primer coche
  ask coche 1 [ set ycor world-height / 4 set color orange]          ;; definimos el color y la coordenada 'y' inicial del segundo coche
end

;; PROCEDIMIENTO GO (ejecución)
to go
  while [all? coches [xcor < max-pxcor]]  ;; (*) mientras la coordenada 'x' de todas las tortugas sean inferiores a la coordenada 'x'
                                           ;; del patch situado más a la derecha se entra en el bucle (while) y los
                                           ;; coches seguirán desplazándose una cantidad aleatoria
  [
    set distancia_coche_1 random 3          ;; en esta iteración, el coche 1 avanzará una dist. aleatoria entre 0 y 2
    set distancia_coche_2 random 3          ;; en esta iteración, el coche 2 avanzará una dist. aleatoria entre 0 y 2
    ask coche 0 [fd distancia_coche_1]      ;; pedimos al primer coche avance la cantidad aleatoria obtenida
    ask coche 1 [fd distancia_coche_2]      ;; pedimos al segundo coche avance la cantidad aleatoria obtenida
    grafica                                ;; llamamos al procedimiento gráfica
    tick                                   ;; incrementamos el contador del número de iteraciones
  ]
end

to grafica
  set-plot-y-range 0 3                    ;; definimos el rango del eje 'y' entre 0 y 3
  set-current-plot "Distancia"            ;; seleccionamos el gráfico "Distancia" que hemos creado
  set-current-plot-pen "coche1"           ;; seleccionamos el lápiz "coche_1"
  plot distancia_coche_1                   ;; dibujamos la distancia recorrida por el coche 1 en la iteración actual
  set-current-plot-pen "coche2"           ;; seleccionamos el lápiz "coche_2"
  plot distancia_coche_2                   ;; dibujamos la distancia recorrida por el coche_1 en la iteración actual
end

```

Observaciones:

- Hemos añadido el código del procedimiento grafica al final del código del ejercicio 13.
- Para llamar a este procedimiento dentro del procedimiento go, simplemente intercalamos el nombre del procedimiento grafica en el código del procedimiento go.
- El procedimiento grafica necesita conocer la distancia avanzada por los coches (variables distancia_coche_1 y distancia_coche_2) para poder realizar el gráfico. En el ejercicio 13, estas dos variables estaban definidas como variables locales, dentro del procedimiento go, ya que solamente se iban a emplear dentro de este procedimiento. Sin embargo, en este ejercicio, el procedimiento tanto el procedimiento go como el procedimiento grafica necesitan acceder al contenido de estas variables. Por ello, tendremos que definir estas dos variables como variables globales. Esto lo haremos al principio del programa empleando la palabra clave globals:

```
globals
[
  distancia_coche_1
  distancia_coche_2
]
```

- Como ya dijimos anteriormente en este tutorial, para asignar un valor a una variable global empleamos la primitiva set (observa la diferencia con las variables locales, para las cuales se emplea la primitiva let). Fijémonos en la diferencia en la forma de asignar valores a las variables distancia_coche_1 y distancia_coche_2 en el ejercicio 13 (variables locales) y en el ejercicio 14 (variables globales):

EJERCICIO 13 (variables locales)	EJERCICIO14 (variables globales)
<pre>let distancia_coche_1 random 3 let distancia_coche_2 random 3</pre>	<pre>globals [distancia_coche_1 distancia_coche_2] ;; [...] set distancia_coche_1 random 3 set distancia_coche_2 random 3</pre>

Haz clic en setup, después en go y observa cómo se va trazando la gráfica a medida que los coches avanzan.

Ejercicio 15. Listas (1)

Concepto de lista

Una lista es una estructura de datos que nos permite almacenar varios elementos de información como un único objeto.

El concepto de lista en Netlogo es equivalente al concepto de vector o array en otros lenguajes de programación.

En Netlogo, las listas se representan con dos corchetes entre los cuales se introducen los elementos que componen dicha lista. Ejemplos de lista son [3 6 83 4 2] ["a" "f" "h" "o"] [3 1 "hola" 4 1]. Un factor a tener en cuenta es que Netlogo comienza a numerar las listas a partir del elemento 0. Es decir, que, por ejemplo, para la lista [12 4 5], 12 sería el elemento 0; 4 el elemento 1 y 5 el elemento 2.

¿Cómo construir una lista?

Depende de si los datos que componen la lista son conocidos a priori o no.

- Si los datos son conocidos:

Para crear una lista, simplemente escribiremos entre corchetes los datos individuales que queremos que formen parte de la lista. Por ejemplo, para crear una lista llamada "a" con los elementos 1 2 3 4 5 6 7 simplemente escribiremos:

```
let a [1 2 3 4 5 6 7]
```

- Si los datos que forman la lista no son conocidos:

Si los datos que van a formar parte de la lista no son conocidos a priori, sino que son variables del programa (caso más habitual), deberemos emplear la primitiva `list`: `list` recibirá una serie de argumentos y nos devolverá una lista con los estos argumentos en el mismo orden en que se los hemos pasado.

La sintaxis de `list` depende del número de argumentos que recibe:

- Si recibe 2 argumentos, simplemente escribiremos los argumentos entre corchetes:

```
list [a b] ;; nota: a y b son números o variables
```

- Si recibe 3 o más argumentos, escribiremos `list` seguido de los argumentos que compondrán la lista, sin corchetes y con paréntesis englobando toda la sentencia (incluida la primitiva `list`):

```
(list a b c d e f g h i j k) ;; nota: a, b, c, d, e, f, g, h, i, j, k son números o variables
```

¿Cómo acceder a los elementos de una lista?

- Para acceder al **primer elemento de una lista** usamos la primitiva `first`. Ejemplo

first [a b c d e f g h i j k] --> a ;; nota: a, b, ..., k son números o variables.

- Para **obtener el último elemento de la lista** utilizamos el primitiva last:

last [a b c d e f g h i j k] --> k ;; nota: a, b, ..., k son números o variables.

- Para **obtener todos los elementos de la lista excepto el primero** se emplea la primitiva but-first. Ejemplo:

but-first [a b c d e f g h i j k] --> [b c d e f g h i j k] ;; nota: a, b, ..., k son números o variables.

- Para **obtener todos los elementos de la lista excepto el último** se emplea la primitiva but-last:

but-last [a b c d e f g h i j k] --> [a b c d e f g h i j] ;; nota: a, b, ..., k son números o variables.

- Para **obtener el elemento n de la lista**, empleamos la primitiva item. Ejemplo:

item 5 [a b c d e f g h i j k] --> f ;; nota: a, b, ..., k son números o variables.

Nota: Recuerda que Netlogo numera las listas a partir del elemento 0.

- Para **tomar un elemento al azar de una lista** se emplea la primitiva one-of:

one-of [a b c d e f g h i j k] --> cualquier elemento de la lista tomado al azar

- Y para **tomar varios elementos al azar de una lista** empleamos n-of:

n-of 3 [a b c d e f g h i j k] --> nos devuelve otra lista con 3 elementos de la lista inicial tomados al azar.

¿Cómo modificar una lista?

- Para introducir un nuevo elemento al comienzo de una lista, empleamos la primitiva fput:

fput z [a b c d e f g h i j k] --> [z a b c d e f g h i j k] ;; nota: a, b, ..., k son números o variables.

- Y para introducirlo al final de la lista, utilizamos la primitiva lput:

lput z [a b c d e f g h i j k] --> [a b c d e f g h i j k z] ;; nota: a, b, ..., k son números o variables.

- Para eliminar un elemento de una lista, usamos la primitiva remove:

remove d [a b c d e f g h i j k] --> [a b c e f g h i j k] ;; nota: a, b, ..., k son números o variables.

Nota: remove elimina todas las paraciones del elemento en la lista.

- Si conocemos la posición que ocupa un elemento dentro de una lista, podemos eliminarlo con la primitiva remove-item:

`remove-item 4 [a b c d e f g h i j k] --> [a b c d f g h i j k]` ;; nota: a, b, ..., k son números o variables.

Nota: Recuerda que Netlogo numera las listas a partir del elemento 0.

- Una variación de la primitiva remove es la primitiva remove-duplicates, que elimina todos los elementos repetidos en la lista, dejando únicamente la primera aparición de los mismos. Ejemplo:

`remove-duplicates [a a b b b c c d e e e e f f g g h h i i j j j k k] --> [a b c d e f g h i j k]`

- Para reemplazar un elemento de la lista por otro, empleamos la primitiva replace-item:

`replace-item 4 [a b c d e f g h i j k] z --> [a b c d z f g h i j k]` ;; nota: a, b, ..., k son números o variables.

Otras primitivas útiles relacionadas con listas:

- Para conocer la longitud de una lista (esto es, su número de elementos) empleamos la primitiva length:

`length [a b c d e f g h i j k] --> 11` ;; nota: a, b, ..., k son números o variables.

- Para obtener el elemento que más se repite dentro de una lista (es decir, la moda), se emplea la primitiva modes:

`modes [a a b c d e f g h i j k] --> a` ;; nota: a, b, ..., k son números o variables.

- Para conocer la posición de un elemento dentro de una lista se utiliza la primitiva position:

`position a [a b c d e f g h i j k] --> 0` ;; nota: a, b, ..., k son números o variables.

- Para desordenar aleatoriamente los elementos de una lista utilizamos la primitiva shuffle:

`shuffle [a b c d e f g h i j k] --> [d k a c e i j h f b g]` ;; nota: a, b, ..., k son números o variables.

- Para obtener una lista ordenada de manera inversa a la original, empleamos la primitiva reverse:

`reverse [a b c d e f g h i j k] --> [k j i h g f e d c b a]` ;; nota: a, b, ..., k son números o variables.

- Para obtener una lista con los elementos ordenados en orden creciente, empleamos la primitiva sort:

`sort [4 6 2 8 4] --> [2 4 4 6 8]`

Nota 1: Recuerda siempre que, para Netlogo, el primer elemento de una lista es el elemento 0.

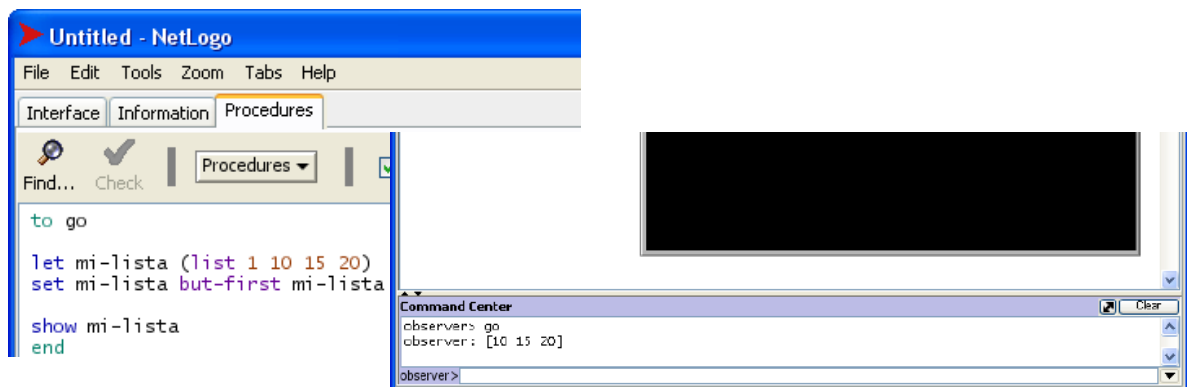
Nota 2: Recuerda que si los elementos de una lista son caracteres, éstos deben escribirse entre comillas: ["a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"]

Ejercicio 16. Listas (2)

En este ejercicio veremos algunas cuestiones útiles relacionados con las listas:

Técnicamente, las listas no pueden ser modificadas, pero sí que se pueden construir nuevas listas a partir de listas antiguas... incluso con el mismo nombre. Para ello se utiliza la orden `set`. Por ejemplo, si tenemos la lista 'mi-lista' con el siguiente contenido: [1 10 15 20] y queremos que pase a tener el contenido [10 15 20], es decir, queremos eliminar el primer elemento de la lista, lo que haremos será emplear conjuntamente las primitivas `set` y `but-first` (para más información sobre `but-first`, consulta el ejercicio anterior)

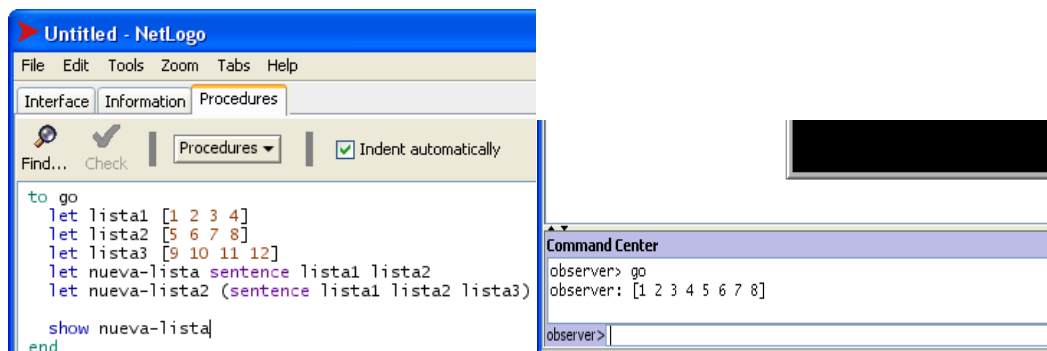
```
let mi-lista (list 1 10 15 20)      ;; creamos la lista inicial
set mi-lista but-first mi-lista    ;; asignamos a la variable mi-lista el resultado
                                   ;; de crear una nueva lista con todos los elementos
                                   ;; de la lista original excepto el primero
show mi-lista                      ;; muestra [10 15 20]
```



Esta manera de contruir nuevas listas a partir de otras existentes es aplicable, también, a los órdenes `but-last`, `remove-item`, `replace-item` que introdujimos en el ejercicio anterior.

Para **concatenar** varias listas, usamos la orden `sentence`. Observa el siguiente ejemplo:

```
let lista1 [1 2 3 4]                ;; creamos la lista inicial
let lista2 [5 6 7 8]                ;; asignamos a la variable mi-lista el
let lista3 [9 10 11 12]
let nueva-lista sentence lista1 lista2 ;; [1 2 3 4 5 6 7 8]
let nueva-lista2 (sentence lista1 lista2 lista3) ;; [1 2 3 4 5 6 7 8 9 10 11 12]
```



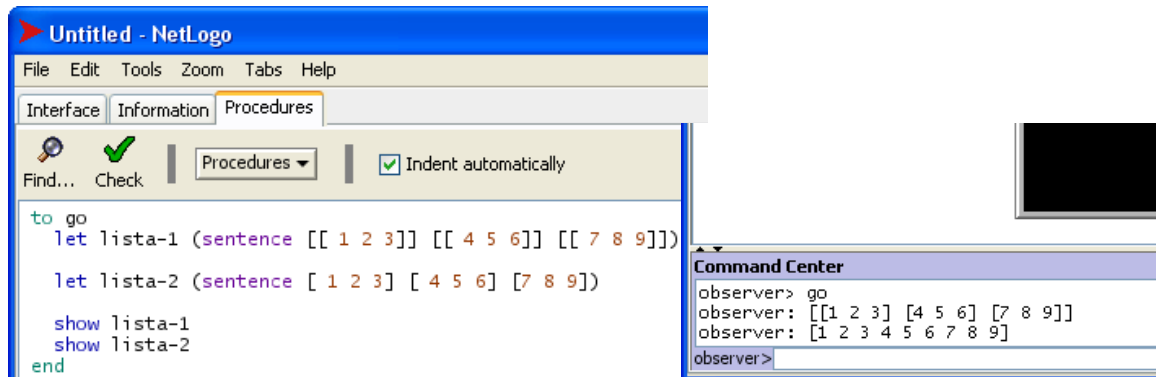
Observa que cuando vamos a concatenar tras o más listas, la sentencia `sentence` completa debe ir entre paréntesis.

Los elementos de las listas de Netlogo pueden ser, a su vez, otras listas. Para crear estas 'lista de listas' podemos utilizar nuevamente la orden sentence, pero esta vez con doble corchete. Compara las siguientes dos líneas de código:

```
let lista (sentence [[ 1 2 3 ]] [[ 4 5 6 ]] [[7 8 9]]) ;;[1 2 3] [4 5 6] [7 8 9]

;; frente a...

let lista (sentence [ 1 2 3 ] [ 4 5 6 ] [7 8 9]) ;;[1 2 3 4 5 6 7 8 9]
```

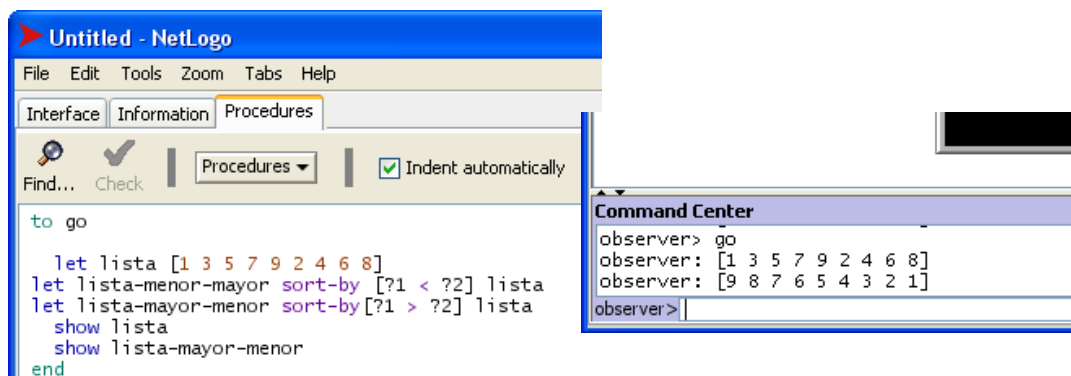


Se pueden anidar las funciones de extracción para modificar elementos en 'listas de listas'. Observa el siguiente ejemplo, en el que deseamos sustituir el '6' por un '7':

```
let lista [[1 2] [3 4] [5 6]]
set mylist (replace-item 2 mylist (replace-item 1 (item 2 mylist) 7) ) ;; [[1 2] [3 4] [5 7]]
```

Para **ordenar los elementos de una lista**, empleamos la primitiva sort-by:

```
let lista [1 3 5 7 9 2 4 6 8]
let lista-menor-mayor sort-by [?1 < ?2] lista ;; [1 2 3 4 5 6 7 8 9]
let lista-mayor-menor sort-by [?1 > ?2] lista ;; [9 8 7 6 5 4 3 2 1]
```

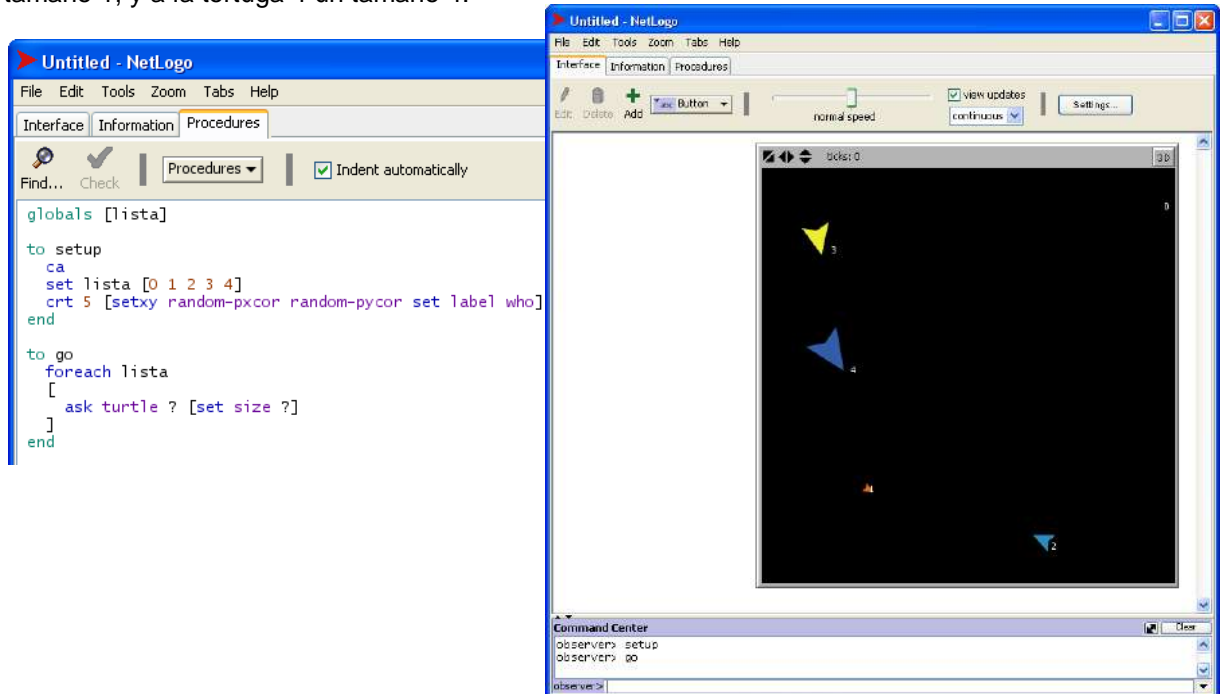


Ejercicio 17. Operaciones sobre listas

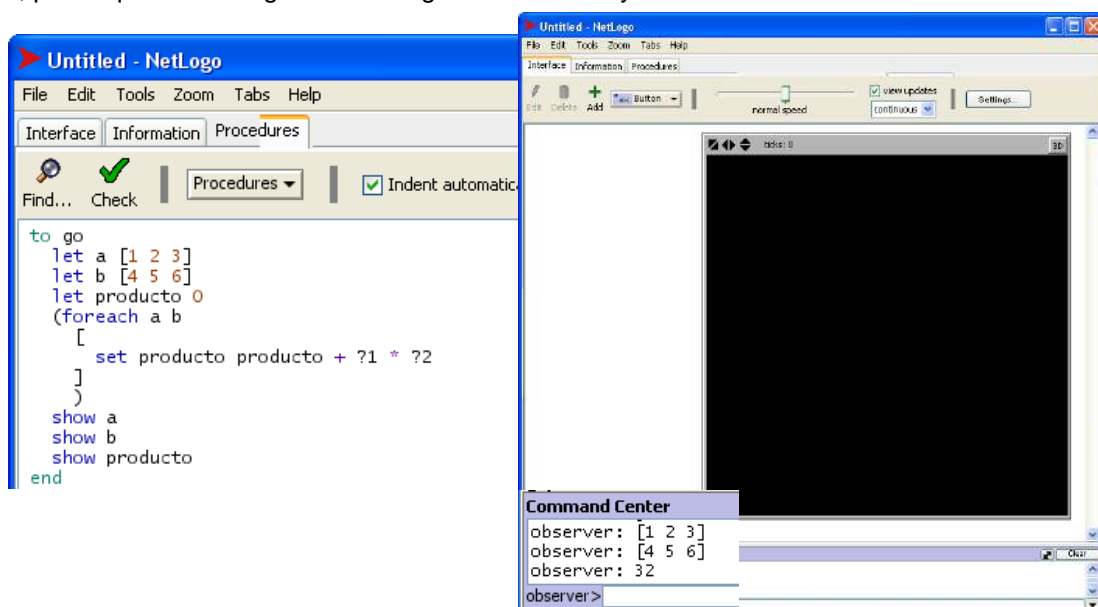
Aunque para realizar operaciones con cada elemento de una lista se puede utilizar un bucle lógico con repeat, con while o un procedimiento recursivo, existen 2 primitivas especiales: foreach y map

foreach realiza un conjunto de operaciones sobre cada elemento de una lista. Se utiliza ? para hacer referencia al valor actual en el bloque de operaciones.

En el siguiente programa, creamos 5 tortugas y asignamos a cada una de ellas un tamaño acorde a su identificador. Es decir, a la tortuga 0 le damos un tamaño 0, a la tortuga 1 un tamaño 1, y a la tortuga 4 un tamaño 4.

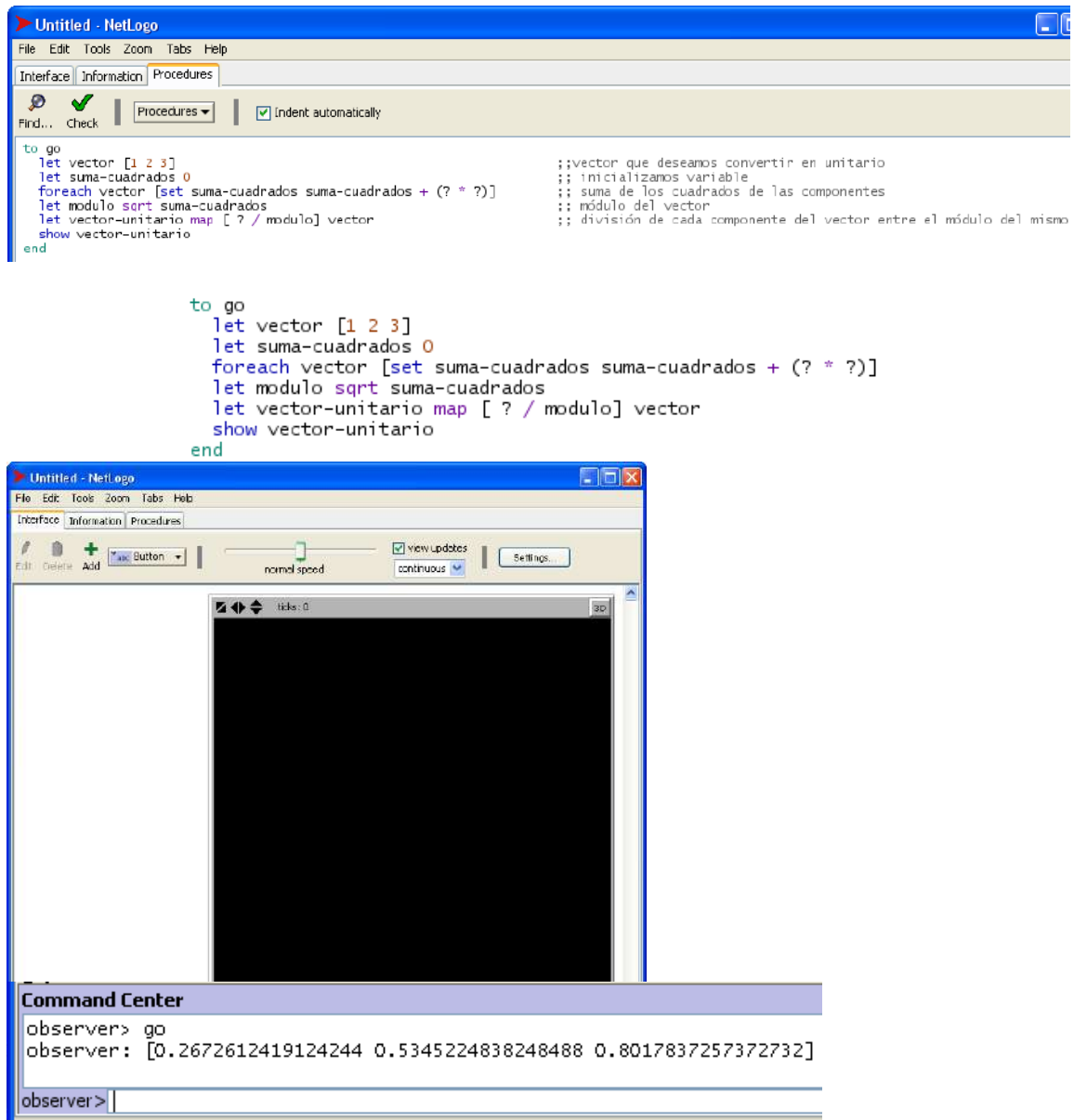


El símbolo ? va tomando valores distintos en cada repetición del bucle, según los valores de los elementos de la lista. En la primera repetición del bucle, ? vale 0 (primer elemento de la lista) y, así, se pide a la tortuga 0 que adquiera un tamaño 0; en la segunda repetición del bucle, ? vale 1, por lo que a la tortuga 1 se le asigna un valor 1... y así sucesivamente.



foreach también puede operar sobre varias listas a la vez (aunque éstas deben ser de la misma longitud). En este caso hay que colocar la operación completa entre paréntesis. El símbolo ?1 hace referencia al valor que toma el elemento actual de la lista 1; ?2 hace referencia al valor que toma el elemento actual de la lista 2, etc.

El siguiente ejemplo aclara la sintaxis de la orden foreach cuando opera sobre más de una lista. En él calcularemos el producto escalar de dos vectores: $a = [1 \ 2 \ 3]$ y $b = [4 \ 5 \ 6]$. El resultado del producto escalar será: $1*4 + 2*5 + 3*6 = 32$. Veamos cómo se programaría todo esto (puedes descargar el código de la sección 'Archivos adjuntos', Ejercicio17b.nlogo, al final de esta página):



Finalmente, aprenderemos el manejo de la primitiva **map**. Se trata de una orden que toma como entrada una lista y da como resultado otra lista que contiene el resultado de aplicar una función a cada uno de los elementos de la original. Al igual que ocurría con la primitiva foreach, con ? se hace referencia al valor actual. Su sintaxis es la siguiente:

map ["operación"] lista

Para aclarar ideas, observemos la siguiente rutina en la cual queremos convertir el vector [1 2 3] en un vector unitario. Para ello, debemos dividir las componentes del vector entre el módulo del mismo. Aprovecharemos para repasar el uso de la orden foreach y calcularemos la suma

de los cuadrados de los componentes del vector empleando esta primitiva. Después calcularemos el módulo del vector (raíz cuadrada de la suma de los cuadrados de los componentes) y, finalmente, emplearemos la orden map para obtener un nuevo vector (unitario) en el que cada componente es la correspondiente componente del vector inicial dividida entre el módulo del vector.