INTRODUCCIÓN AL LENGUAJE NETLOGO Y LA PROGRAMACIÓN BASADA EN AGENTES.

Francisco Quesada Chaverri

Marzo de 2018

Contenido

Introducción

¿Qué es NetLogo?

El Manual del Usuario de NetLogo comienza diciendo:
"NetLogo es un ambiente para la programación de modelos en el cual se pueden simular fenómenos sociales y naturales". Más adelante se agrega *"NetLogo es particularmente apto para modelar sistemas complejos que evolucionan con el tiempo. Los modeladores pueden dar instrucciones a cientos o miles de agentes que operan independientemente. Esto hace posible explorar la conexión entre la conducta de los individuos a nivel micro y los patrones que emergen a nivel macro como resultado de las interacciones entre dichos agentes".*
Como se desprende de la anterior descripción, NetLogo no es sólo un lenguaje de programación, en realidad es un ambiente computacional que incluye como núcleo central un lenguaje de programación para la creación de modelos basados en una multitud de agentes, disciplina que se conoce con el nombre de Modelado Basado en Agentes o MBA (en inglés Agent Based Modeling o ABM). NetLogo fue creado en 1999 por Uri Wilensky y desde entonces, ha estado en continuo desarrollo, siempre bajo su dirección, en el Center for Connected Learning and Computer Based Modeling de Northwestern University. NetLogo es un producto completamente gratuito, con excelente documentación -aunque la mayoría en inglés- y una comunidad de usuarios muy activa y participativa, que incluye a educadores, investigadores de distintas áreas de las ciencias y la ingeniería y simples amantes del arte-ciencia de la programación. NetLogo se encuentra disponible para las tres plataformas: Windows, OS y Linux. Una de las ventajas de NetLogo es la gran cantidad de modelos disponibles, muchos de ellos incluidos dentro del software mismo o en la Web, los cuales son de gran utilidad para aprender el lenguaje. El ambiente NetLogo incluye varios módulos o extensiones que amplían sus posibilidades en varias direcciones. Basta echar un breve vistazo a las diversas secciones de la página principal de Netlogo (NetLogo Home Page) en: https://ccl.northwestern.edu/netlogo para darnos cuenta de la inmensa variedad de artículos, libros, modelos e incluso grupos de interés que cubre la sombrilla de este ambiente de programación.

Cualquier persona que aspire a incursionar en el campo del modelado basado en agentes, ya sea con el fin de construir sus propios modelos o de comprender el código de modelos construidos por otros, debe comenzar por aprender los fundamentos del lenguaje de programación en que dichos modelos están escritos. En este libro de introducción al software de modelado basado en agentes NetLogo, el lenguaje ocupa el lugar primordial. En el libro no se presupone que los lectores hayan estado expuestos previamente a otros lenguajes de programación, por lo que se han incluido explicaciones generales sobre los principales conceptos comunes no sólo a NetLogo sino a todo lenguaje de programación: empleo de variables, expresiones condicionales, recursión e iteración de bloques de órdenes, manejo de listas y cadenas, entre otros conceptos importantes. Al autor le gustaría pensar que quienes completen la lectura del libro habrán adquirido un conocimiento suficientemente amplio del lenguaje y de los principales recursos del ambiente NetLogo, que les permitiese incursionar en la creación

de sus propios programas y modelos. De ahí en adelante, la labor de extender y profundizar el conocimiento de este rico ambiente sería cosa relativamente fácil, con la ayuda de los recursos disponibles en el sitio Web de NetLogo y los numerosos ejemplos que el programa incluye. El código de los ejemplos y modelos del libro, se discute con detalle y se puede bajar del sitio del autor:
www.franciscoquesada.com/netlogo.php.

El libro está estructurado de la siguiente manera: Los tres primeros capítulos presentan, por medio de ejemplos, un conjunto de primitivas de NetLogo y de conceptos básicos sobre programación, con los cuales es posible crear los primeros programas y modelos. Los capítulos 4 y 5 presentan una colección de modelos, en cuyo código se hace uso de las primitivas y conceptos introducidos en los capítulos anteriores o bien de primitivas o conceptos nuevos que el modelo requiere para su implementación. El capítulo 6 presenta algunos temas complementarios de NetLogo. En el Apéndice A se describen brevemente algunos de los aspectos de NetLogo no incluidos en el libro.

NetLogo en español.

Al momento de publicarse este libro no existe una versión del software NetLogo en español. Existe, no obstante, una versión cuya interfaz muestra los nombres de menús, ventanas y botones en español[1], pero no una versión en español del lenguaje NetLogo propiamente dicho, es decir, en la cual se pudiera escribir, por ejemplo, “mostrar contar tortugas con [color = rojo]” en vez de “show count turtles with [color = red]”. No sabemos si alguna vez existirá una versión en español como la descrita, pero a decir verdad y en vista de que para manejarse en la versión inglesa se requiere un nivel muy básico de este idioma, tampoco es cosa que debería preocuparnos demasiado. El hecho de que no exista una versión del lenguaje en español también tiene sus ventajas: promueve un mayor contacto con la comunidad internacional de usuarios de NetLogo, la cual programa mayoritariamente en inglés. Lo que sí es importante para nuestra comunidad hispano-hablante es que exista documentación en español: manuales, tutoriales, libros y ejemplos con comentarios en nuestra lengua. Con este modesto tutorial, el autor espera estar haciendo una pequeña contribución en esta dirección.

Algunas observaciones sobre la traducción al español.

En la jerga de la tecnología, algunas palabras se traducen y usan de modo algo diferente en España que en Latinoamérica. Hemos optado por “computador” en vez de “ordenador” y “archivo” en vez de “fichero”. La elección entre “raza” o “familia” para la palabra “breed” se ha decidido en favor de la segunda opción. La expresión “agent-set” tiene un significado muy preciso en NetLogo y nos ha parecido más apropiado traducirla como “conjunto-agentes” en vez de “conjunto de agentes”, cuyo significado podría depender más del contexto. En el caso de las primitivas llamadas “reporters” en inglés hemos decidido usar el participio activo del verbo reportar y traducirlas como “reportadoras” en vez de “reporteras” o “reportes”. La expresión inglesa “by default” se ha traducido como “por omisión”. Siguiendo la documentación en inglés, hemos preferido usar “Ventana del Observador” en vez de “Consola”.

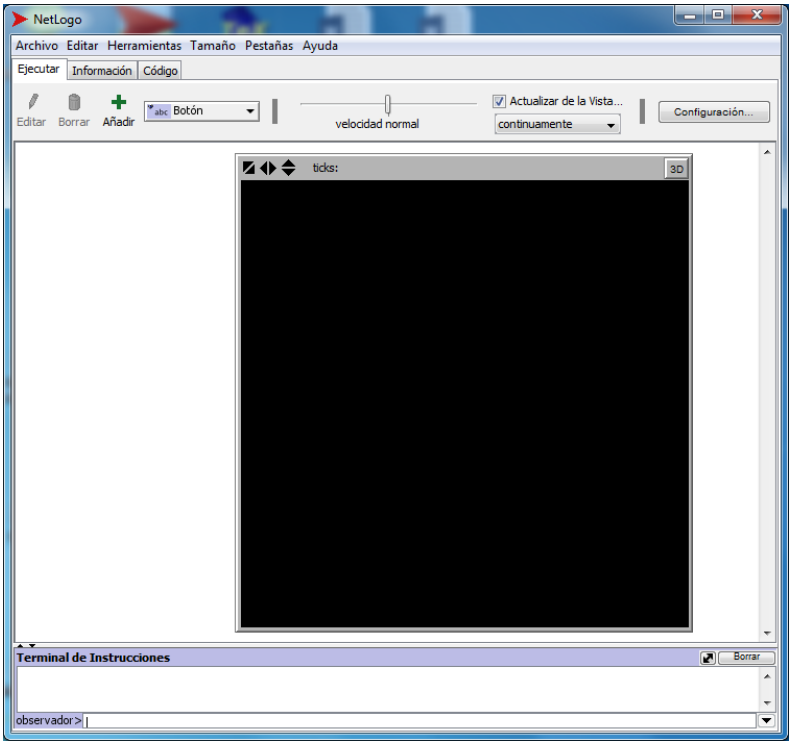
[1] Esta versión aún muestra algunos elementos en inglés, como por ejemplo los mensajes de error.

Fuentes adicionales de información. En este libro haremos frecuente referencia a dos fuentes de información de gran importancia en relación con el aprendizaje de NetLogo. La primera de ellas, e indiscutiblemente la más importante, es el manual del usuario "NetLogo 5.2 User Manual" [16], el cual se puede bajar gratuitamente del sitio Web de Netlogo antes mencionado y del cual, lamentablemente, no existe aún traducción al español. Recomendamos bajar la versión en pdf imprimible (printable) e imprimirla para tenerla a mano como referencia. Esta guía, a la cual nos referiremos como "el Manual", contiene no sólo la lista completa de primitivas y muchos ejemplos importantes, sino que incluye una descripción de todos los aspectos de NetLogo, incluyendo sus extensiones. La segunda fuente de información a que haremos referencia es el excelente libro en inglés de Wilenski y Rand [15]. Además de enseñar cómo programar modelos de una manera gradual con NetLogo, empezando con lo más básico, el libro es una fuente de invaluable información sobre muchos otros aspectos de NetLogo y del modelado basado en agentes (MBA) en general. Y desde luego, no podemos dejar de mencionar como fuente muy valiosa de información y de aprendizaje, los numerosos ejemplos que el software incluye en su Biblioteca de Modelos (disponible en el menú Archivos del programa). Al público hispano-hablante también queremos recomendar el excelente y muy completo libro de los autores García y Sancho [5] del cual existen versiones tanto en español como en inglés. Como apoyo adicional, existe una traducción al español del Diccionario de Primitivas de NetLogo, aunque de momento sólo en formato pdf. Esta traducción se puede consultar o bajar del sitio Web de Netlogo o del sitio personal del autor: www.franciscoquesada.como/netlogo.php.

Recomendamos vehementemente a las personas que desean aprender a programar en NetLogo que bajen e instalen el programa NetLogo en sus computadores y carguen o digiten los ejemplos del libro. También les instamos a experimentar con sus propios ejemplos de órdenes o procedimientos. Netlogo se puede bajar de la página oficial (Home Page): https://ccl.northwestern.edu/netlogo/

Capítulo 1: Ambiente y lenguaje I.

Cuando se inicia el software NetLogo, lo primero que aparece en la pantalla del computador es una ventana, la cual constituye la interfaz con el usuario y que llamaremos *la interfaz* de NetLogo. Su aspecto es el que se muestra en la figura de la izquierda. En la interfaz se puede apreciar una serie de elementos, como menús, ventanas y botones, los cuales tienen por objeto facilitar la creación de los modelos. La interfaz junto con el lenguaje de programación subyacente conforma lo que se conoce como un *ambiente de programación*. Los tres elementos principales del ambiente NetLogo son:



- La interfaz: es la ventana con que el programador se comunica con NetLogo para construir los modelos y ejecutarlos.
- El lenguaje: está constituido por las palabras y construcciones gramaticales con las que se construyen los programas de NetLogo.
- Los agentes: Son los entes que ejecutan las acciones del programa o modelo.

El ambiente NetLogo fue diseñado específicamente para facilitar la creación de modelos basados en muchos agentes, lo cual explica la presencia de los elementos que conforman su interfaz, así como las características del lenguaje. No obstante, en el ambiente NetLogo también es posible crear programas de la más diversa índole y no necesariamente basados en agentes. El elemento más notable de la interfaz es el cuadrado negro ubicado en su parte superior derecha, conocido como *el mundo*. Este es el escenario donde los agentes ejecutan la mayoría de sus acciones, en forma similar a como lo hacen los actores en el escenario de un teatro. Los modelos construidos con NetLogo no son programas autónomos o de la modalidad conocida como *stand alone*, es decir, los programas no corren si no está presente también el software NetLogo en el computador. Los autores de NetLogo han creado recientemente una versión en-línea, que permite correr los modelos de NetLogo sin necesidad de instalar el software, conectándose al sitio oficial de NetLogo o NetLogo Home page.

El lenguaje.

El lenguaje NetLogo es el núcleo central del ambiente NetLogo y es sin duda su parte más importante. Un programa de cómputo no es otra cosa que una secuencia organizada de órdenes, formuladas en un lenguaje técnico, las cuales hacen posible que el computador realice una serie de tareas propuestas por el programador. Para no tener que escribir las órdenes en el primitivo lenguaje del computador -un lenguaje a base de ceros y unos- se han creado los llamados lenguajes de programación de alto nivel. El lenguaje NetLogo –como todo lenguaje de programación- posee sus propias reglas de sintáxis[2], las cuales se deben respetar rigurosamente. Afortunadamente la sintaxis de NetLogo es bastante sencilla. El lenguaje Netlogo es del tipo conocido como *lenguaje interpretado*, lo cual significa que cada orden de un programa es tomada por separado, traducida a lenguaje de máquina y ejecutada en forma inmediata por el computador. El programa que realiza este proceso se conoce con el nombre de "el intérprete de NetLogo" y al cual nos referiremos simplemente como "el intérprete". Esto difiere del modo como funcionan los lenguajes conocidos como lenguajes compilados. En estos lenguajes el texto que conforma el programa, llamado "código fuente", es tomado como un todo y convertido en un programa autónomo llamado "código objeto". Dicho código objeto se encuentra traducido a lenguaje de máquina, por lo cual podría ser ejecutado por cualquier máquina[3], sin necesidad de la presencia del lenguaje de programación en que se escribió y se compiló el código fuente. Los lenguajes interpretados ofrecen dos ventajas muy importantes sobre los compilados: 1) es posible probar órdenes, sin que las mismas formen parte de un programa. Esto permite conocer el efecto de órdenes aisladas, lo mismo que verificar la corrección de su sintaxis y 2) En un lenguaje interpretado es posible escribir y probar los programas en forma incremental, observando como se desempeña el programa conforme le vamos agregando nuevas órdenes, modalidad esta que facilita la detección temprana de errores en la programación. Los lenguajes compilados, por su parte, también tienen sus ventajas sobre los interpretados: 1) Se ejecutan con mayor rapidez en la máquina. 2) Los programas creados en los lenguajes compilados son más portables pues no necesitan de la presencia del software compilador en el computador que corre el código objeto. En este libro tratamos de mantener cierto nivel de consistencia en el uso de las palabras "modelo", "programa" y "procedimiento". Llamaremos "modelo" a un programa cuyo propósito es modelar algún fenómeno de la vida real o incluso un fenómeno ficticio. A los programas que no pretenden modelar fenómenos les llamamos simplemente "programas". La palabra "procedimiento" se reserva, la mayoría de las veces, para denotar las pequeñas unidades de código, los "pequeños programas" que conforman un programa o modelo de mayor tamaño. Finalmente usamos la palabrá "código" (code), palabra usada con mucha frecuencia en programación, para designar ya sea la totalidad del texto de un procedimiento, programa o modelo o incluso un trozo de dicho texto. La sintaxis del lenguaje NetLogo guarda gran similitud con la del lenguaje Logo, del cual se puede decir que es un pariente cercano. Ambos lenguajes se han inspirado en el lenguaje Lisp, uno de los lenguajes más antiguos y poderosos que existen. El aprendizaje del vocabulario básico y las reglas más importantes de sintaxis de un lenguaje de programación se adquieren con la práctica, mediante la construcción de

[2] No distinguiremos entre los conceptos de sintaxis y gramática de un lenguaje.
[3] Con el mismo sistema operativo, p. ej. Windows, OS o Linux.

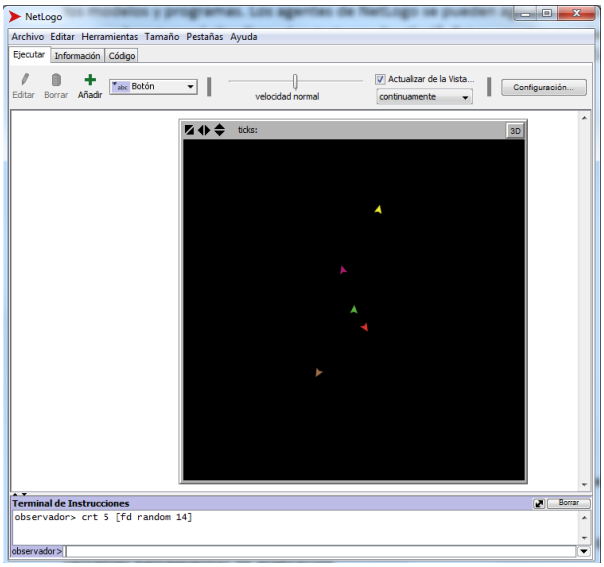
programas. El intérprete de NetLogo posee la capacidad de detectar la mayoría de los errores de ortografía o gramática del lenguaje, a los que para simplificar llamaremos "errores de sintaxis", e informar sobre la naturaleza de dichos errores por medio de mensajes llamados "mensajes de error".

Primitivas: el vocabulario básico. Cada lenguaje de programación posee un vocabulario básico propio, con el cual se escriben las órdenes o instrucciones que conforman los programas. En este libro usaremos las palabras órden e instrucción con el mismo significado. A las palabras básicas propias de un lenguaje de programación se las refiere con el nombre de "palabras primitivas" o simplemente "primitivas" del lenguaje. NetLogo posee un conjunto numeroso de primitivas, que sirven para atender una apreciable variedad de situaciones. El Diccionario de Primitivas de NetLogo, disponible en el sitio de NetLogo, contiene una descripción del funcionamiento de cada primitiva del lenguaje.

Los agentes de NetLogo.

NetLogo posee un conjunto de agentes presinstalados en el sistema, listos para ser usados en los modelos y programas. Los agentes de NetLogo se pueden agrupar en cuatro clases:

- Agentes móviles llamados *tortugas* (turtles)[4]. Estos agentes pueden deplazarse por el mundo (el cuadrado negro de la interfaz que muestra la figura). Su aspecto inicial -el cual puede ser modificado- es el de pequeños triángulos:



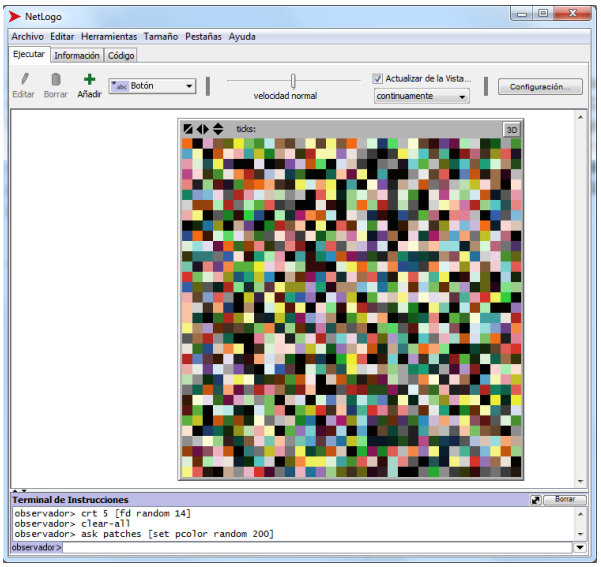
La figura muestra la interfaz de NetLogo con cinco tortugas de distintos colores sobre el mundo (el cuadrado negro).

Opcionalmente, las tortugas pueden dejar un trazo conforme avanzan, lo que las convierte en versátiles herramientas de graficación.

- Agentes estáticos llamados *parcelas* (patches). Las parcelas se encuentran formando un cuadriculado que cubre el mundo, como si éste estuviera

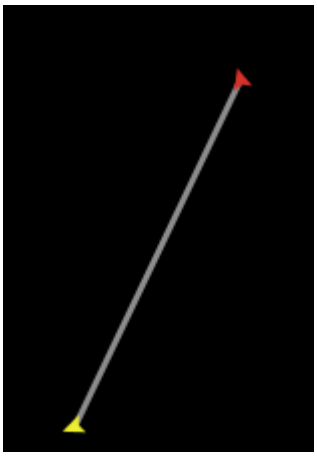
[4] La costumbre de llamar tortugas a los agentes móviles proviene del lenguaje Logo, que como se ha señalado, es un pariente cercano de NetLogo.

recubierto por baldosas. Al abrir el programa las parcelas no se distinguen unas de otras porque, por omisión (by default), inicialmente todas tienen color negro. El número de parcelas, su tamaño y color se puede modificar. Cuando el programa se abre, el mundo se encuentra cubierto por un entramado de 33 x 33 = 1089 parcelas de forma cuadrada:



La figura muestra el mundo recubierto por parcelas con colores generados al azar.

- Agentes llamados *enlaces* (links). Estos agentes sirven para relacionar o "enlazar" a dos o más tortugas. Cuando se establece un enlace, el mismo se vuelve visible en forma de una línea que une a ambas tortugas. El aspecto de esta línea se puede modificar o incluso ocultar. Los enlaces pueden ser *direccionales* o *no direccionales*. Cuando un enlace es direccional, a la tortuga donde nace el enlace se le conoce como "origen" o "raíz" del enlace mientras que a la tortuga hacia donde el enlace se dirige se la llama "destino", "meta" u "hoja" del enlace. La figura de la izquierda muestra un enlace no direccional entre dos tortugas.



- El cuarto agente es el *Observador*. Es el agente de mayor jerarquía y este agente no es visible. El Observador puede dar órdenes a los otros agentes. Nos comunicamos con el Observador a través de la "Ventana del Observador", ubicada en la parte inferior de la interfaz.

Todos los agentes son programables: la esencia de la programación basada en agentes consiste precisamente en el hecho que los agentes son entes autónomos, capaces de recibir y ejecutar órdenes individualmente.

Conjuntos de agentes (conjunto-agentes).

NetLogo permite agrupar los agentes en conjuntos y tratarlos como entidades a las que se puede girar instrucciones. A estos conjuntos se les llama “conjunto-agentes” (agent-sets). Es posible definir conjunto-agentes que están constituidos por tortugas, parcelas o enlaces, pero siempre debe tratarse de agentes de un mismo tipo. Se pueden crear conjunto-agentes imponiendo restricciones a un conjunto-agentes previamente creado. Por ejemplo, luego de haber creado un conjunto-agentes formado por muchas tortugas, podemos definir un nuevo conjunto-agentes integrado solamente por aquellas tortugas cuyo color es rojo. Como veremos en los ejemplos adelante, los conjunto-agentes no son solamente una abstracción del lenguaje ordinario, sino que son entidades con existencia computacional propia dentro del ambiente NetLogo, de los cuales podemos sacar provecho en la construcción de modelos.

El mundo.

Examinamos a contínuación las características más importantes del mundo:

- El mundo originalmente está constituido por un conjunto de pequeñas parcelas cuadradas formando un entramado del tipo conocido como un cuadriculado.
- Cada parcela de este entramado es un agente programable.
- Cada parcela ocupa una posición dentro del cuadriculado dada por un par de coordenadas cartesianas (x, y). El origen de coordenadas (0, 0) se sitúa en el centro del mundo, pero es posible cambiar su ubicación.
- Las coordenadas de una parcela se refieren a su punto central y son siempre números enteros (las tortugas, en cambio, pueden ocupar posiciones cuyas coordenadas no sean necesariamente números enteros).
- Las distancias recorridas por las tortugas se miden en pasos y existe una relación entre las longitudes de los pasos de una tortuga y las dimensiones de las parcelas: un paso de tortuga equivale a la medida del lado de una parcela, de modo que, si una tortuga se encuentra en el centro de una parcela y camina un paso hacia arriba, llegará exactamente al centro de la parcela inmediatamente superior.
- El número de parcelas que tiene el mundo se puede variar, lo mismo que el tamaño y su forma.
- En su configuración inicial, el mundo se comporta como si sus lados opuestos se encontraran unidos: si una tortuga sale del mundo por el lado derecho del cuadrado, la veremos reaparecer a la misma altura por el lado izquierdo y viceversa, fenómeno que se aplica también a los lados superior e inferior. Esta condición topológica del mundo puede ser cambiada, como se discutirá cuando se traten las distintas topologías del mundo.

La vista (the view).

El aspecto gráfico del mundo está constituido por lo que en NetLogo se denominan “vistas” (views). Mientras un modelo corre, la apariencia del mundo varía, ofreciendo secuencias de vistas que transmiten la sensación de movimiento y la evolución de los

fenómenos que se modelan, igual que lo hacen los cuadros (frames) en una película. Cada uno de estos cuadros constituye una vista: una imagen congelada del mundo en un momento dado. Esta imagen está constituida sólo por el cuadrado negro del mundo (o con el color que tenga en ese momento), más lo que hay en su interior (parcelas, tortugas y trazos hechos por ellas), pero la vista no incluye otros elementos de la interfaz que se encuentran afuera del mundo, como botones, deslizadores, gráficos menús o ventanas.

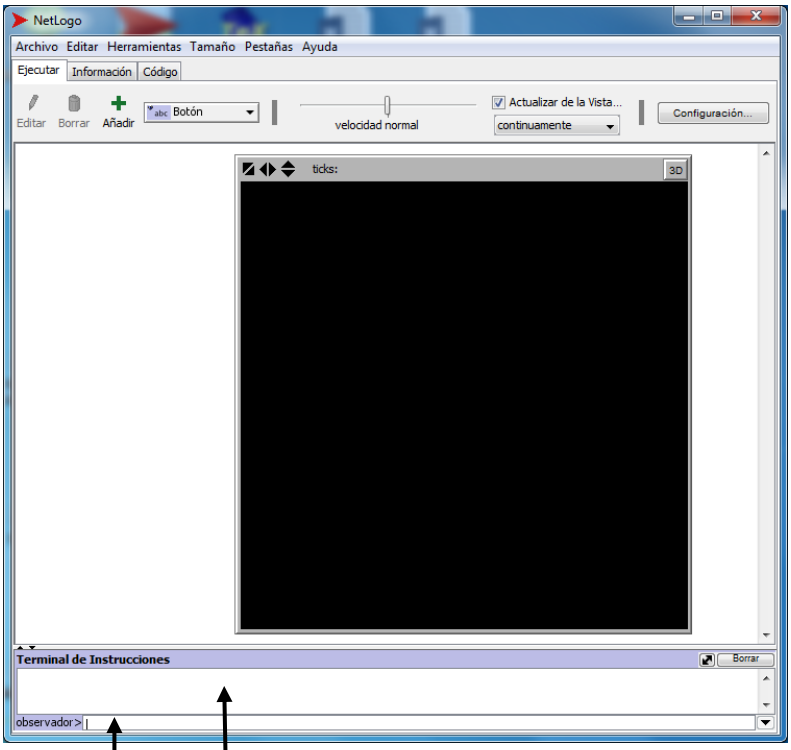
Dónde escribir el código de NetLogo.

En los lenguajes de tipo interpretado, como es el caso de NetLogo, además de la escritura de programas completos, también es posible probar instrucciones aisladas o secuencias de instrucciones que no tienen la estructura de un programa completo. En NetLogo estas dos situaciones se manejan en áreas distintas de la interfaz. Los programas de NetLogo se deben escribir en el área llamada “Editor de Programas”. Las secuencias de órdenes que no constituyen un programa, se deben escribir en la “Ventana del Observador”, un área a veces referida en otros lenguajes como “la consola”.

La ventana del Observador.

La Ventana del Observador, área alargada y angosta ubicada en la parte inferior de la interfaz y a la derecha de la palabra “Observador”, es el lugar donde se deben ingresar las instrucciones u órdenes aisladas. Una vez escrita una orden o una secuencia de ellas en este lugar, es necesario pulsar la tecla <Intro>[5] para que la orden se ejecute. En caso de que la orden se haya digitado erróneamente, el intérprete de NetLogo desplegará un “mensaje de error” en la Terminal de Instrucciones, la ventana inmediatamente encima de la Ventana del Observador. Suponiendo que usted ya ha instalado NetLogo en su computador, le invitamos a probar las primeras órdenes de NetLogo en esta ventana. Por razones pedagógicas, los primeros ejemplos de órdenes y de procedimientos estarán dirigidos a los agentes móviles (las tortugas), para que realicen trazos en la pantalla.

[5] Conocida también como “tecla Enter” o “tecla de Entrada”.



Ventana del Observador

Terminal de Instrucciones

Ejemplo 1: Primeras órdenes en la Ventana del Observador.

Primitivas: create-turtles (crear-tortugas), pendown (pluma-abajo), forward (adelante), right (derecha), ask (pedir, solicitar), turtles (tortugas), + (operador de adición), show (mostrar), clear-all (limpiar-todo).

En la Ventana del Observador digite textualmente las órdenes que se muestran a continuación. Para que la orden se ejecute oprima la tecla <Intro> después de haberla escrito.

1- Digite:

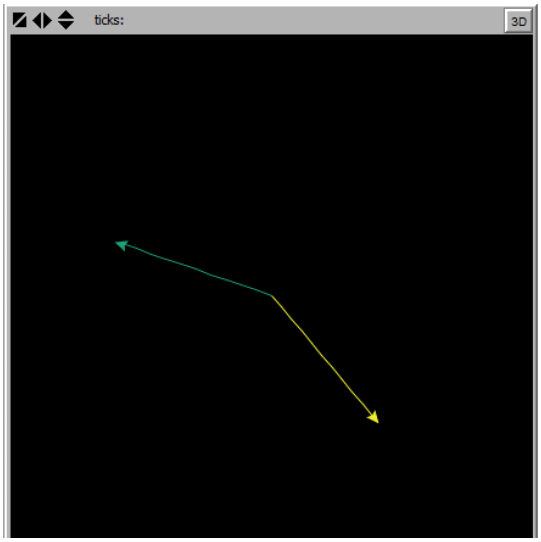
create-turtles 2

Esta orden crea 2 tortugas, ambas en la parcela (0 0) (el centro del mundo) que es el lugar donde -por omisión- nacen las tortugas. Debido a que las tortugas se crean en el mismo punto, podría parecer que sólo hay una. Si ha cometido un error al escribir la orden, verá un "mensaje de error"'. Si ese es el caso, lea el mensaje con atención e inténtelo nuevamente escribiendo la orden correctamente.

2. Digite:

ask turtles [pendown forward 10]

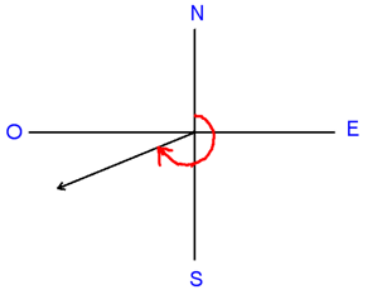
Esta orden se traduce como "pedir-a-tortugas [bajar-pluma avanzar-adelante 10]" y cuyo resultado posible se muestra en la figura.



Cuando las tortugas bajan la pluma van dejando un trazo de su mismo color al avanzar. Las tortugas deben ejecutan (acatar) todas las órdenes contenidas dentro de los corchetes de la expresión “ask turtles”. Las órdenes se separan por medio de espacios en blanco, nunca por medio de comas u otros caracteres. Las tortugas nacen con colores y orientaciones asignados al azar por el sistema NetLogo. Podemos pensar que el intérprete es quien los asigna.

3. Digite:
ask turtles [show 3 + 3]
==> (turtle 0): 6
==> (turtle 1): 6
Esta orden pide a las tortugas creadas mostrar el resultado de la suma 3 + 3. Usaremos el símbolo “==>” para indicar los resultados que muestra NetLogo en la Terminal de Instrucciones, el lugar de salida (output area) por omisión del programa. El resultado “6” se muestra dos veces porque la orden se giró a todas las tortugas y en el momento había dos de ellas. Note que cuando se escribe una orden en la Ventana del Observador, una vez ejectuada la misma, el texto de la orden aparece replicado a modo de “eco” a la derecha de la palabra “observador>”, en la Terminal de Instrucciones. La primitiva “show” muestra el resultado de la suma precedido del nombre del agente que ejecutó la orden. Es posible redirigir la salida a otras áreas, por ejemplo a una ventana de salida o a un archivo externo, como se verá más adelante. Nótese que las tortugas se numeran conforme se van creando, comenzando a partir del número 0. Otro detalle a tomar en cuenta es que en NetLogo - igual que en el lenguaje Logo- se debe dejar espacios a ambos lados de los operadores aritméticos binarios como “+” o el signo “=” (no hacerlo genera un mensaje de error).
4. La siguiente orden va dirigida solamente a la tortuga 0:
ask turtle 0 [right 90 forward 10]
Esta orden pide a la tortuga 0 que gire 90 grados a la derecha sobre su propio eje y luego avance 10 pasos hacia adelante. Las órdenes dirigidas a un agente o a un conjunto-agentes se deben encerrar entre corchetes aún cuando se trate de una sola orden o cuando el conjunto-agentes esté constituido por un solo miembro, como en el ejemplo presente.
5. Digite la orden
clear-all
La orden “clear-all” realiza una limpieza del mundo: borra las tortugas y los trazos que han quedado de órdenes previas. Su forma abreviada es “ca”.
6. Digite la orden
show “Hola todo el mundo”
==> observer: “Hola todo el mundo”

Se considera que las órdenes no dirigidas a tortugas, parcelas o enlaces, están dirigidas al Observador. El texto a la derecha de la primitiva “show” se debe encerrar entre comillas. De no hacerse, el intérprete pensaría que las palabras que lo componen son primitivas de NetLogo o nombres de procedimientos y tratará de evaluarlas como tales y, al no ser el caso, se generaría un mensaje de error. Cuando se crean tortugas -y a menos que se indique lo contrario- las mismas aparecen dotadas de un color y una orientación asignados al azar. La *orientación* (heading) de una tortuga es la dirección en que la tortuga mira, es decir, es la dirección en que avanzaría si tuviera que hacerlo. La orientación se determina por medio de un ángulo cuyo valor varía entre 0 y 360 grados.

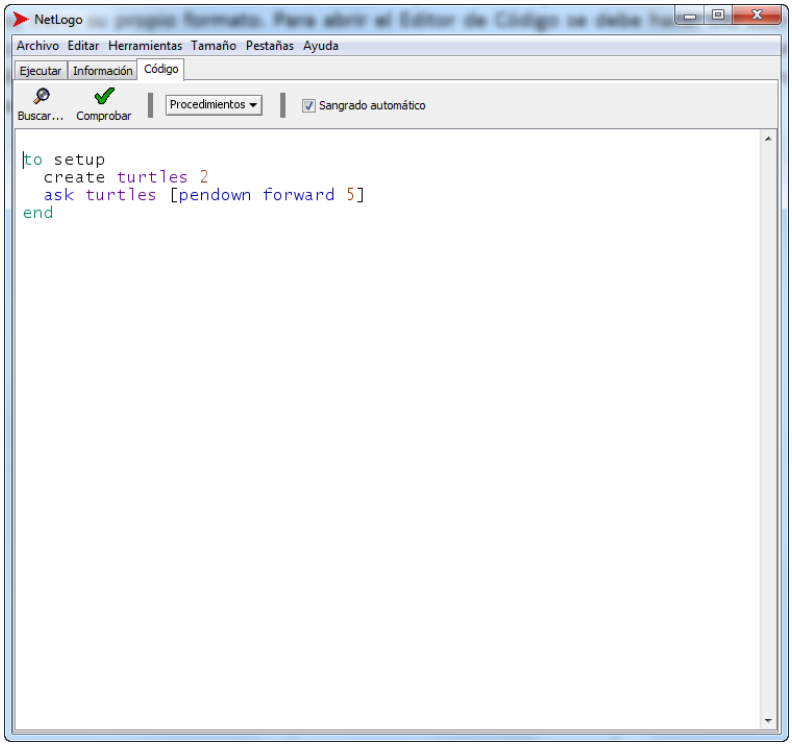


El ángulo de orientación de las tortugas se mide a partir del norte en sentido horario. La figura anterior muestra un ángulo de orientación en el tercer cuadrante, comprendido entre 180º y 270º. Una orientación con valor 0 corresponde a la dirección que apunta hacia la parte superior de la pantalla, conocida como la dirección norte. Nótese que esto difiere del criterio comúnmente usado en matemática, en donde una orientación de 0 grados suele apuntar hacia la derecha, en la dirección este. Algunas primitivas tienen versiones abreviadas, como por ejemplo forward, que se puede abreviar como fd y create-turtles como crt. Las órdenes escritas en la Ventana del Observador quedan almacenadas en dicha ventana y pueden ser reutilizadas posteriormente. Se las puede encontrar activando el cursor de la ventana y buscándolas mediante las teclas de movimiento del cursor.

El editor de programas.

El editor programas es el lugar donde se escribe el texto o código de los programas de NetLogo. Es común organizar los modelos y programas fraccionándolos en programas de menor tamaño llamados *procedimientos*. En el Editor de Código no se deben escribir

órdenes o secuencias de órdenes aisladas que no sean parte de algún procedimiento. Se abre el Editor de Código haciendo clic sobre la solapa con el nombre “Código”, ubicada en la parte superior de la interfaz. Al hacerlo veremos un cambio en el aspecto de la interfaz: el mundo es sustituido por una zona en blanco, en la cual se escribe el código de los procedimientos, como se muestra en la figura. Cada procedimiento de NetLogo es una secuencia de órdenes “empaquetadas” bajo un nombre, a saber, el nombre asignado al procedimiento. El formato de todo procedimiento de NetLogo es el siguiente:



- La primera línea del código de un procedimiento debe ser ocupada únicamente por el nombre del procedimiento, precedido de la primitiva “to” (para).
- En las líneas siguientes al nombre se escribe “el cuerpo” del procedimiento, es decir, las órdenes que lo conforman, una después de la otra y no se debe usar comas u otros caracteres para separarlas, solamente espacios en blanco. Es posible distribuir las órdenes en tantas líneas como se desee, pues el intérprete es insensible a los espacios en blanco y a los cambios blandos o forzados de línea y tampoco distingue entre minúsculas y mayúsculas. Se puede indentar el texto libremente empleando la tecla de tabulación. Existen recomendaciones sobre cómo indentar y separar en líneas el código, de manera que resulte más comprensible a las personas que lo leen. Los ejemplos en la Biblioteca de Modelos pueden servir de guía.
- Todo procedimiento debe terminar con la primitiva “end” (fin), la cual se escribe en línea aparte.

Para escribir un procedimiento debemos comenzar por abrir el Editor de Código, haciendo clic sobre la solapa llamada “Código”. Inmediatamente veremos el cursor parpadeando en la zona en blanco, en espera de que comencemos a introducir código. Nuestro primer ejemplo será muy sencillo: limpiaremos la pantalla y crearemos 10 tortugas que avanzarán 12 pasos hacia adelante dejando trazo. Llamaremos al procedimiento “diez-tortugas” (el nombre debe constar de una sola palabra).

Ejemplo 2: Mi primer procedimiento.

Primitivas: to (para), end (fin), create-turtles (crear-tortugas), pendown (pluma-abajo) forward (adelante), ask (pedir, solicitar), turtles (tortugas), clear-all (limpiar-todo).

He aquí el código:

```
to diez-tortugas
clear-all
create-turtles 10
ask turtles [pendown forward 12]
end
```

Explicaciones y comentarios adicionales. Para correr el procedimiento se debe regresar a la vista frontal de la interfaz haciendo clic sobre la pestaña "Ejecutar" y escribir el nombre del procedimiento "diez-tortugas" en la Ventan del Observador, seguido de la tecla "Intro". La primera orden "clear-all" tiene por objeto borrar las tortugas y los trazos que pueden haber quedado en el mundo de anteriores órdenes aisladas o procedimientos que se hayan corrido. Si no incluyéramos esta orden, los trazos de cada nueva corrida se mezclarían con los de corridas anteriores y en cada ocasión se agregarían diez nuevas tortugas al mundo (sugerencia: comprobarlo borrando la primitiva clear-all). Si usted cometió algún error al digitar el código, cuando intente regresar a la vista frontal de la interfaz, verá que la pestaña "Código" muestra su nombre en rojo y un texto con fondo amarillo le señala el tipo de error que el intérprete ha encontrado. Lea con atención este texto y trate de reparar el problema. No obstante, si usted insiste en volver a la vista frontal de la interfaz sin reparar el error, puede hacerlo volviendo a hacer clic sobre la pestaña "Ejecutar", pero el procedimiento no correrá. Mientras subsistan errores en el código escrito en el Editor de Código, la pestaña "Código" permanecerá en letras rojas.

Las tortugas como herramientas de graficación.

La idea de la tortuga se debe a Seymour Papert, uno de los creadores del lenguaje Logo. Papert pensó en crear un ente virtual programable capaz -entre otras cosas- de graficar. La metáfora que Papert empleó fue la de representar este ente o robot programable por un animalito -escogió una tortuga- que pudiera ser dirigido por medio de movimientos básicos y que pudiera dejar trazo al moverse por la pantalla del computador [11]. Papert pensó que la manera más natural en que este ente debía desplazarse por la pantalla era utilizando cuatro movimientos básicos, dos de traslación y dos de giro, como lo hacen muchos animales al movilizarse por el terreno. Los movimientos de traslación permitirían deplazar la tortuga hacia adelante o hacia atrás y los de giro cambiar de dirección, girando sobre el propio eje a derecha o izquierda. A cada uno de estos cuatro movimientos se asignó una primitiva: para trasladarse las tortugas emplean las primitivas "forward num" (adelante num) y "back num" (atrás num) y para girar utilizan las primitivas "right num" (derecha num) y "left num" (izquierda num). En las traslaciones el argumento num representa el número de pasos que la tortuga debe dar, mientras que en los giros hacia la derecha o izquierda el

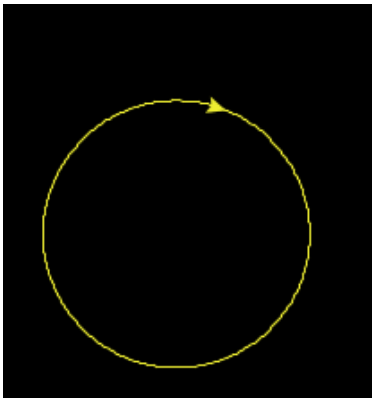
argumento num expresa el ángulo de giro expresado en grados. Las tortugas tienen la opción de dejar trazo conforme avanzan, ocultarse o hacerse visibles o incluso cambiar su apariencia (ver la Shapes Editor Guide en la sección Shapes del Manual del Usuario [16]). Las tortugas también son capaces de borrar los trazos que han dejado o estampar sus figuras. Sin embargo, hay una diferencia que es importante señalar con respecto a algunas implementaciones existentes del lenguaje Logo. En NetLogo las tortugas no son capaces de interactuar con sus propios trazos –salvo la acción de borrarlos- pues éstos se encuentran en una capa diferente a aquella en que las tortugas se movilizan, y por ello no los pueden detectar. Como consecuencia de esto, en NetLogo no existe una primitiva que permita rellenar con un color el interior de un contorno cerrado, como lo hace la primitiva "pintar" (fill) en varias implementaciones del lenguaje Logo.
En el siguiente ejemplo, se crea una tortuga y se le ordena que dibuje un círculo. Puesto que las tortugas sólo pueden avanzar en línea recta, no se puede lograr que una tortuga trace una curva en el verdadero sentido de la palabra. Pero es posible aproximar una curva mediante una línea quebrada con trazos rectos de pequeña longitud.

Ejemplo 3: Una tortuga dibuja un círculo.

Primitivas: pd (abreviatura de pendown), forward (adelante), right (derecha), set (asignar), color, yellow (amarillo), repeat (repetir).

Construiremos un polígono con muchos lados de tamaño muy pequeño, el cual tendrá la apariencia de un círculo. Abrimos el Editor de Código, haciendo clic sobre la solapa Código, y escribimos el siguiente código:

```
to círculo
clear-all
create-turtles 1
ask turtle 0 [pd set color yellow repeat 360 [forward 0.1 right 1 ] ]
end
```



Explicaciones y comentarios adicionales. Las tortugas se numeran conforme se van creando, a partir de 0. Si en vez de "ask turtle 0 [...órdenes...]", hubiésemos escrito "ask turtles [...órdenes...]", la orden estaría dirigida a todas las tortugas y el efecto habría sido el mismo pues, en ese momento, la única tortuga que hay en el mundo es la tortuga 0. La primitiva "pd" es una abreviatura de "pendown" (pluma-abajo). La primitiva "set" (asignar o fijar) se usa para asignar un nuevo valor a una propiedad de un agente o de una variable, como es el caso del color de la tortuga. La orden "set color yellow" se traduce como "asignar color amarillo". Esta orden está dirigida a la tortuga 0 por estar dentro de los corchetes de "ask turtle 0 […]". La tortuga adopta el color amarillo y este será también el color de su trazo. Los colores en NetLogo poseen un código numérico, pero aquellos más comunes se pueden referir también por

el nombre (para mayor información sobre colores ver la opción Colors en la Programing Guide del Manual [16]). La primitiva “repeat num [..órdenes..]” se usa para repetir las órdenes encerradas entre corchetes. Las órdenes se repiten tantas veces como lo indique la entrada “num”. En nuestro ejemplo “repeat 360 [forward 0.1 right 1]” se ordena a la tortuga repetir 360 veces la secuencia “forward 0.1 right 1” que consiste en avanzar una décima de paso hacia adelante y girar 1 grado a la derecha. Esto genera un polígono de 360 lados y de tamaño muy pequeño (una décima de paso), el cual tiene la apariencia de un círculo, como lo muestra la figura anterior. El procedimiento se pone en marcha regresando a la vista normal de la interfaz (haciendo clic sobre la solapa “Ejecutar”) y digitando la palabra “círculo” en la Ventana del Observador. La acción de escribir el nombre de un procedimiento se conoce como “llamar” “correr” o “invocar” el procedimiento. Los procedimientos también pueden ser invocados desde otros procedimientos.

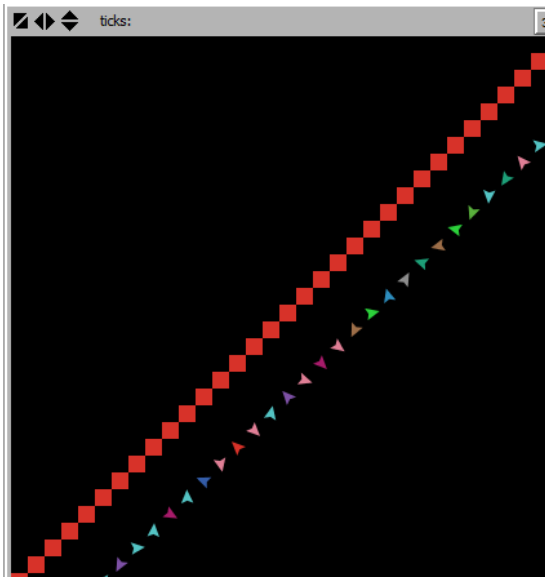
Las parcelas.

Las parcelas (patches) son agentes estáticos que pueden ser programados para desempeñar roles muy variados en la creación de modelos. Lo mismo que las tortugas, cada parcela posee características propias como las coordenadas de su posición, su color y su tamaño. Estas características pueden ser modificadas por los usuarios. Por omisión el mundo de NetLogo está constituido por un cuadriculado de 1089 parcelas de color negro. En esta configuración, a la parcela que se encuentra en la esquina superior derecha del mundo le corresponden las coordenadas (16, 16). En NetLogo las coordenadas de una parcela no se encierran entre paréntesis ni se separan por comas. La parcela de coordenadas (2, 4), por ejemplo, debe ser referida como “patch 2 4”. El lenguaje NetLogo también tiene primitivas que permiten hacer referencia a las parcelas vecinas de una parcela dada. El acceso a las propiedades de estos vecindarios es una herramienta muy poderosa en la creación de modelos. Cuando se varía el número de parcelas del mundo a veces es necesario variar también el tamaño de las parcelas, a fin de que el entramado se mantenga visible. Por ejemplo, una duplicación del número de las parcelas obliga a una reducción del tamaño de las mismas, a fin de que quepan en el cuadrado del mundo. Para cambiar el número o tamaño de las parcelas se debe abrir la ventana “Configuración” de la interfaz, haciendo clic sobre el botón del mismo nombre. El tamaño de la interfaz en la pantalla puede variarse arrastrando las líneas horizontales o verticales que demarcan sus fronteras, como con la mayoría de las aplicaciones Windows. El tamaño del mundo también puede aumentarse o encogerse en las direcciones horizontal, vertical y diagonal mediante las parejas de flechas negras que se encuentran sobre la banda gris en su extremo superior y a la izquierda de la palabra “ticks”. El siguiente ejemplo tratará con parcelas y tortugas.

Ejemplo 4: Parcelas en diagonal.

Primitivas: patches (parcelas), with (con), pxcor (coordenada x de parcela), pycor (coordenada y de parcela), pcolor (color de parcela), = (signo igual), > (signo mayor que), sprout (brotar),

```
to diagonal
clear-all
ask patches with [pxcor = pycor] [set pcolor red]
ask patches with [pxcor = pycor + 5 ] [sprout 1]
end
```



Explicaciones y comentarios adicionales. Varias de las primitivas que se refieren a las parcelas tienen nombres similares a los de las tortugas, pero inician con la letra "p". Tal es el caso de pxcor, pycor o pcolor, cuyos equivalentes para tortugas son xcor, ycor y color. La primitiva "sprout num" (brotar num) pertenece sólo a las parcelas y su efecto es el de hacer brotar tantas tortugas como lo indique la entrada "num" en las parcelas que emiten la orden. Como ya se ha dicho, las tortugas nacen con orientaciones y colores asignados al azar. La primera orden "ask patches with [pxcor = pycor] [set pcolor red]" la podríamos traducir como "pedir a las parcelas con [coordenada x = coordenada y] [fijar el color rojo]". En el ejemplo tenemos dos órdenes que hacen uso de la primitiva "with" (con). Esta primitiva permite formar órdenes compuestas, que amplían la capacidad expresiva del lenguaje. Con esta primitiva es posible construir el conjunto-agentes que satisface una determinada condición:

conjunto-agentes with [condición].

La condición que sigue a la primitiva "with" debe encerrarse entre corchetes. Por ejemplo, la condición [pxcor = pycor] permite construir el conjunto-agentes de las parcelas cuya coordenada x es igual a su coordenada y (son las parcelas en la diagonal del mundo).

Los enlaces (links).

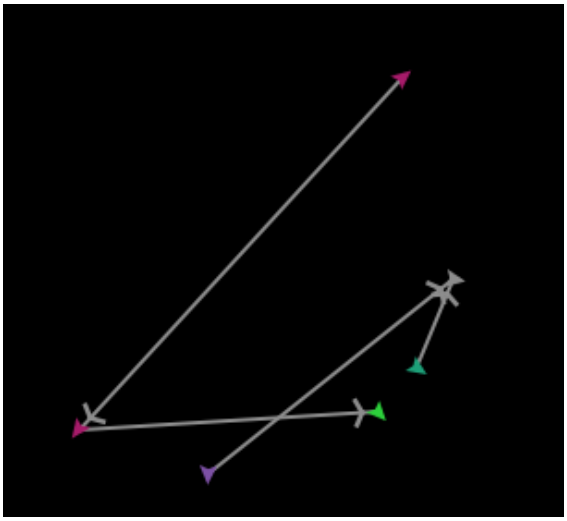
Los enlaces son agentes que conectan a dos tortugas. La existencia de un enlace está supeditada a la existencia de las dos tortugas que le han de servir de extremos. NetLogo tiene dos tipos de enlaces: *enlaces dirigidos* y *enlaces no dirigidos*. Un enlace dirigido tiene un "origen" y un "destino", también llamados "raíz" y "hoja" respectivamente. Los enlaces pueden ser de gran utilidad para modelar relaciones entre agentes. Las redes o

grafos son una de las aplicaciones más importantes de los enlaces. En un grafo las tortugas pueden representan los nodos y los enlaces las aristas. No se pueden crear enlaces entre las parcelas. Los enlaces entre tortugas deben ser de un mismo tipo: dirigidos o no dirigidos. Sin embargo, si se crean familias de tortugas -concepto que cubriremos más adelante- es posible que cada familia tenga su propio tipo de enlace. No es posible crear un enlace entre una tortuga y ella misma (enlaces llamados lazos o bucles). Sin embargo, una misma tortuga puede servir como extremo de enlaces hacia o desde varias otras tortugas. NetLogo tiene muchas primitivas relacionadas con sus agentes, ya sean éstos tortugas, parcelas o enlaces. En el caso de los enlaces el sufijo "with" (con) es característico de las primitivas relacionadas con enlaces no dirigidos mientras que los sufijos "from" (desde) y "to" (hacia) se usan para enlaces dirigidos. Ejemplos de esto son las primitivas create-link-to (crear-enlace-hacia), create-link-from (crear-enlace-desde) o create-link-with (crear-enlace-con). Cada una de estas tres primitivas tiene una versión en plural, por ejemplo, create-links-with (crear-enlaces-con). Los enlaces, igual que las tortugas, pueden ser visibles (siuación por omisión) o no visibles. En el siguiente ejemplo crearemos 6 tortugas, se les pedirá que avancen 10 pasos (para distinguirlas visualmente unas de otras en la pantalla) y mediante la primitiva create-link-to o create-link-from crearemos enlaces dirigidos entre algunas de ellas. Las tortugas podrían, por ejemplo, representar un grupo de personas en una oficina, en donde hay que elegir a la coordinadora de un proyecto. Si el enlace va de la tortuga A a la B, esto podría significar que la tortuga A votaría por la tortuga B para coordinadora.

Ejemplo 5: Elección de una coordinadora.

Primitivas: create-link-to (crear-enlace-hacia), create-link-from (crear-enlace-desde).

```
to elección
clear-all
crt 6 [fd 10]
ask turtle 0 [create-link-to turtle 2]
ask turtle 0 [create-link-from turtle 5]
ask turtle 3 [create-link-to turtle 4]
ask turtle 1 [create-link-to turtle 4]
end
```



La figura muestra los enlaces prodcidos por el procedimiento "elección".

Explicaciones y comentarios adicionales. Los sufijos "to" (hacia) y "from" (desde) indican la dirección de un enlace. La instrucción "ask turtle 0 [create-link-to turtle 2]" crea un enlace dirigido que va de la tortuga 0 hacia la tortuga 2, mientras que la instrucción "ask turtle 0 [create-link-from turtle 5] crea un enlace dirigido desde la tortuga 5 hacia la tortuga 0. El ejemplo muestra cómo de una misma tortuga puede entrar y salir un enlace (tortuga 0). A la tortuga 4, en cambio, ingresan dos enlaces, lo que la haría ganar la elección como coordinadora. Si quiséramos enlaces no dirigidos habríamos empleado la primitiva create-link-with. Los enlaces no dirigidos pueden servir para modelar relaciones simétricas, como por ejemplo la relación de amistad.

Tres principios importantes.

El principio de concatenación.

Cuando un procedimiento es llamado por otro procedimiento decimos que ambos procedimientos se encuentran concatenados. El procedimiento que hace la llamada es el eslabón superior de la cadena y el procedimiento llamado es el eslabón inferior. Los grandes programas suelen estar constituidos por un enjambre de procedimientos de menor tamaño que se encuentran concatenados entre sí formando redes. Un programa puede incluso concatenarse consigo mismo, como ocurre en el esquema de recursividad, el cual será tratado más adelante. La concatenación de procedimientos permite la aplicación de otros dos principios o disciplinas de gran importancia en las ciencias y la ingeniería, los cuales aplicamos con frecuencia incluso en la vida cotidiana. Estos principios adquieren particular importancia en programación de computadoras.

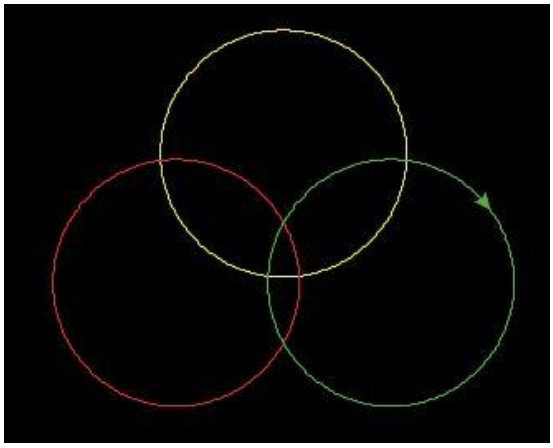
El principio de subdivisión: Dicho en pocas palabras este principio consiste en *la subdivisión adecuada de una tarea compleja en subtareas de menor complejidad y tamaño*. Sin embargo, la eficacia del principio reside en la buena elección de las subtareas y su mutua relación. El escritor de una novela podría solicitar la revisión de la ortografía y redacción de su novela asignando distintos capítulos a distintos revisores. Sería absurdo que decidiera subdividir la revisión en dos partes: la primera parte consistiría en revisar todas las palabras de la novela que comienzan con consonante y la

segunda parte aquellas que que comienzan con vocal. El sentido común nos previene de realizar ciertas subdivisiones que podríamoso calificar de absurdas, pero en proyectos de elevado grado de complejidad la elección de las subtareas no siempre suele ser obvia.
El principio de reutilización: Si usted necesita colgar diez cuadros en las paredes de su casa, necesita hacerse con al menos diez clavos, pero no necesita diez martillos, con uno es suficiente. Mediante el principio de reutilización es posible hacer repetido uso de herramientas y recursos de que ya se dispone, toda vez que esto sea posible y tenga sentido. Un buen ejemplo de este principio en computación nos lo dan los archivos de tipo CSS en un sitio Web, los cuales almacenan los detalles de estilo de las páginas web. Si, por ejemplo, quisiéramos cambiar el tipo de letra o el color de los párrafos de las 100 páginas de que consta un sitio, bastaría hacer un pequeño cambio en el archivo CSS en donde se define el estilo, en vez de tener que modificar una a una las 100 páginas.

Ejemplo 6: Concatenación, subdivisión y reutilización.

Primitivas: setxy (asignarxy), set (asignar), repeat (repetir), pd (abrev. de pendown, pluma-abajo), pu (abreviatura de penup, pluma-arriba), fd (abrev. de forward), rt (abrev. de right, derecha).

En este ejemplo una tortuga dibuja tres círculos de colores diferentes entrelazados entre sí, como lo muestra la figura. El procedimiento se pone en marcha digitando “tres-círculos” en la Ventana del Observador.



He aquí el código:

```
to tres-círculos
crt 1
ask turtle 0 [setxy 3 2 pd set color yellow círculo
pu setxy 7 -2 set color red pd círculo
pu setxy -2 -2 set color green pd círculo ]
end

to círculo
repeat 360 [fd 0.1 rt 1]
end
```

Explicaciones y comentarios adicionales. Pese a su pequeñez, el procedimiento ilustra la aplicación de los tres principios: subdivisión, concatenación y reutilización. El procedimiento “círculo” se ha sacado del cuerpo de “tres-círculos” y es invocado desde este procedimiento (principios de concatenación y subdivisión). El procedimiento “círculo” es utilizado en tres ocasiones (principio de reutilización). La primitiva “setxy num1 num2”, traslada la tortuga al punto de coordenadas (num1, num2). Como no se quiere que la tortuga deje un trazo cuando se traslada al punto donde debe comenzar el siguiente círculo, es necesario ordenarle antes que levanta la pluma (pu), pero una vez llegado al punto deberá bajarla (pd). Otra ventaja que brinda el subdividir un programa en procedimientos que realizan tareas específicas tiene que ver con la introducción de cambios o la reparación de errores, es decir, con el mantenimiento del programa. Si, por ejemplo, se quisiera que los círculos fueran más grandes o más pequeños, sólo se tendría que hacer el cambio una vez en el procedimiento círculo. Finalmente, los tres principios anteriores brindan una ventaja más: el código que resulta de su aplicación es más claro y fácil de entender.

Tipos de primitivas de NetLogo.

Las primitivas de NetLogo poseen algunas características importantes que permiten agruparlas en clases. Mencionaremos dos criterios de clasificación de las primitivas, los cuales son independientes el uno del otro. La primera clasificación se basa en el hecho que las primitivas requieran o no de datos de entrada.

Primitivas que requieren de entradas. Consideremos como ejemplo la primitiva “forward num”, que ordena a una tortuga avanzar hacia adelante un número de pasos dado por la cantidad “num”. En la jerga de los lenguajes de computación se dice que “num” es una “entrada” de la primitiva forward. A veces se emplean también las palabras “parámetro” o “argumento” como sinónimos de “entrada”. Algunas de las primitivas que requieren de entradas son: right, create-turtles y set. Las órdenes “setxy 10 3” o “repite 4 [forward 10 left 90]” nos muestran que las primitivas “setxy” y “repite” operan con dos entradas. En el caso de “repite” su segunda entrada no es un número sino una lista de órdenes. Con frecuencia ocurre que una primitiva que requiere entradas recibe los valores de las entradas de otra primitiva, cuya función es suministrar o, como suele decirse, “reportar” esos valores. Esto nos lleva a la segunda clasificación de las primitivas.

Primitivas actoras y primitivas reportadoras. Se dice que una primitiva es *actora* si realiza algún tipo de acción. Ejemplos de primitivas actoras vistas hasta el momento son: forward, clear-all y create-turtles, entre otras. Estas primitivas generan las acciones de avanzar hacia adelante, limpiar la pantalla y crear nuevas tortugas, respectivamente. A las primitivas actoras se las conoce también con el nombre de “comandos”. Las primitivas que no son actoras las llamaremos primitivas *reportadoras (reporters).* Estas primitivas se caracterizan por suministrar o reportar un dato, el cual ponen a disposición de otra primitiva o procedimiento que lo requiera. Un ejemplo de primitiva reportadora es “color”, la cual en vez de generar una acción reporta el número del color de la tortuga al agente que lo solicita. Como ejemplo examinemos las dos siguientes órdenes (en la Ventana del Observador):

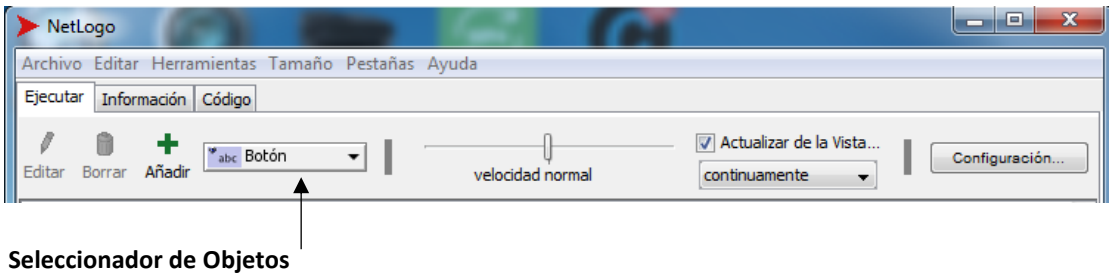
crt 1
ask turtle 0 [show color]
==>(turtle 0): 75

La primera orden crea una tortuga (supongamos que es la primera que se crea y por tanto le corresponde el número de identidad 0). En la segunda orden se le pide a la tortuga 0 que muestre su color. Aquí la primitiva reportadora "color" reporta su valor (75) a la primitiva "show", la cual es una primitiva actora que toma este valor y lo muestra en la Terminal de Instrucciones. Si los lectores no obtienen el número 75 como el color de la tortuga, esto es perfectamente normal. Recuérdese que las tortugas se crean con colores asignados al azar. Este es un ejemplo de una construcción muy frecuente en que se combinan dos primitivas, una que reporta un dato y otra que lo recibe como entrada. Como veremos adelante, esto mismo se puede implementar con los procedimientos, uno que reporta y otro que recibe.

Capítulo 2: Ambiente y lenguaje II.

Botones para llamar procedimientos.

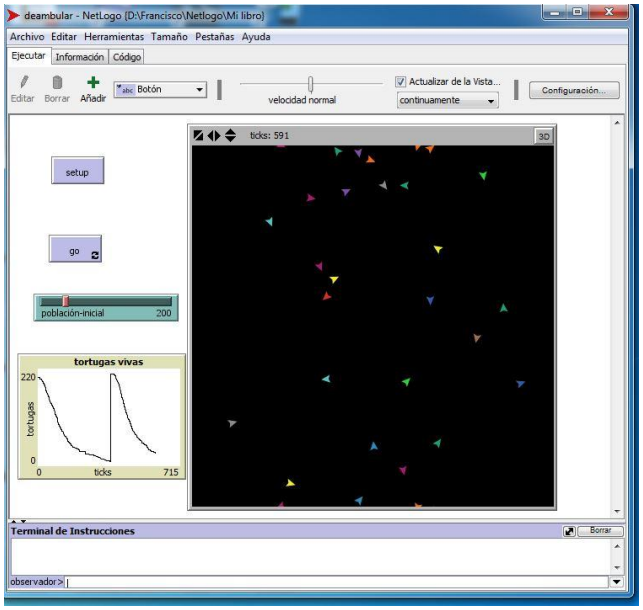
Si abrimos algunos de los modelos de la Biblioteca de Modelos del Menú Archivos, notaremos que la gran mayoría muestra la presencia de elementos como botones, deslizadores o gráficos en el área blanca a la izquierda del cuadrado del mundo. Estos objetos se plantan para desempeñar funciones específicas, por ejemplo, poner un procedimiento en marcha, acelerar o retardar un proceso o brindar información sobre la evolución del modelo, entre otras cosas. El tipo de objeto que se desea plantar se escoge pulsando la pestaña del ícono "Seleccionador de Objetos", lo cual abre un menú desplegable con las distintas opciones que se pueden plantar. Por omisión el Seleccionador de Objetos muestra la opción "Botón".



Seleccionador de Objetos

Los botones setup y go.

Una costumbre muy difundida en la comunidad NetLogo consiste en plantar dos botones llamados "setup" (configurar) y "go" (¡en marcha!). El botón "setup" suele usarse para preparar el terreno antes de que se inicien las acciones del modelo, por ejemplo, borrar los gráficos que han quedado de anteriores procedimientos o corridas, asignar los valores iniciales a las variables y crear algunos agentes. El botón "go" pone en marcha las acciones del modelo. La razón que justifica plantar estos botones es muy simple: es más cómodo correr un procedimiento pulsando un botón que escribiendo su nombre en la Ventana del Observador. Además de los botones setup y go mencionados, con frecuencia se plantan botones para desencadenar otros tipos de procesos o para escoger distintas modalidades en las que un modelo puede correr. La idea no es otra que la de hacer más ágil el manejo y la obtención de información del modelo.



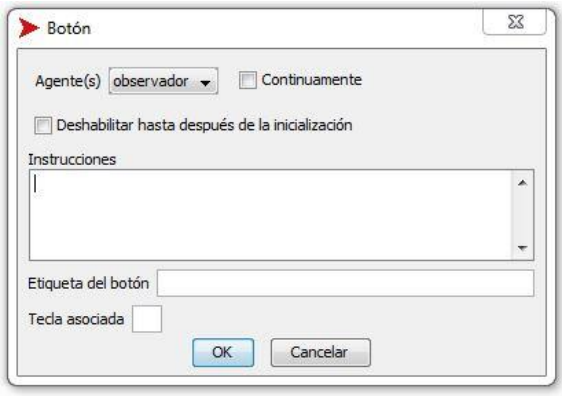
A cada botón debe corresponder un procedimiento -usualmente del mismo nombre- cuyo código se encuentra en el Editor de Código y el cual se activa haciendo clic en el botón. En la ventana del botón también es posible escribir órdenes.

Cómo plantar un botón.

Podemos describir la acción de plantar un objeto (por ejemplo, un botón) mediante tres pasos:

- Seleccionar. Si en el Seleccionador de Objetos no está escrita la palabra “Botón”, abrimos su ventana desplegable y seleccionamos la opción “Botón”.
- Plantar. Se activa la función de plantar haciendo clic sobre el ícono con el signo “+” encima de la palabra “Añadir”, en la Barra de Operaciones, lo que provoca que el puntero cambie su forma a la de un signo “+”. Se lleva el puntero del ratón al lugar donde se quiere plantar el botón -normalmente se escoge un punto en el área blanca a la izquierda del mundo- y se hace clic en ese lugar.

- Nombrar. Inmediatamente veremos abrise la ventana de edición del botón (o del objeto que estamos plantando) donde podemos escribir el nombre del procedimiento que el botón invocará (por ejemplo “setup” o “go”). Escribimos el nombre dentro del cuadro “Instrucciones”, en el lugar donde se encuentra parpadeando el cursor. Finalmente se hace clic sobre la opción OK de esta ventana.



El botón ha quedado plantado, mostrando el nombre del procedimiento asociado a él en su cara frontal. El nombre del botón permanecerá en letras rojas hasta tanto no se escriba el procedimiento asociado al botón en el Editor de Código. Si se desea cambiar el lugar donde el botón quedó plantado, se puede reposicionar haciendo clic derecho sobre el botón y escogiendo la opción “Select”, lo cual permite arrastrarlo a una nueva posición.

Cada botón posee su propia ventana editable de propiedades, la cual se abre haciendo clic derecho sobre el botón.

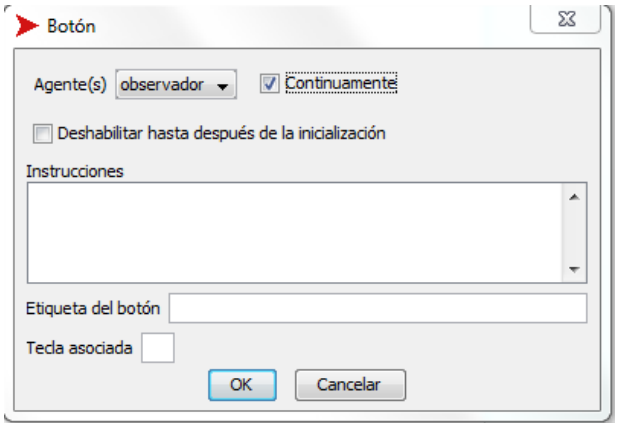
Si se desea que el nombre del botón no coincida con el nombre del procedimiento asociado a él, se escribe el nombre del botón dentro de su ventana, en el campo “Etiqueta”. Sin embargo, al hacer clic sobre el botón, el procedimiento que se pondrá en marcha será aquél cuyo nombre se ha escrito dentro del cuadro de instrucciones.

Repetición continua de un procedimiento.

Una de las facetas que más interesa estudiar de un modelo es la forma como éste evoluciona en el tiempo. Esta evolución se manifiesta de diversas maneras, como por ejemplo, el número y conducta de los agentes o los valores de las variables del modelo. En la mayoría de los casos, la evolución temporal se suele modelar a través de la repetición continua de uno o varios procesos, los cuales están representados por procedimientos cuyos parámetros por lo general varían con cada repetición. Surge entonces la cuestión de si es posible que un procedimiento se repita un número ilimitado de veces. En NetLogo, como en otros lenguajes de programación, existen mecanismos que permiten programar procesos iterativos, incluyendo procesos que se repiten un número ilimitado de veces, o como se dice en NetLogo, que se repiten "continuamente". Esto se logra mediante el uso de primitivas o esquemas de programación que se incluyen en el código del programa y a los cuales nos referiremos más adelante. Ahora presentaremos un recurso muy sencillo que posee NetLogo, el cual permite la iteración continua sin necesidad de tener que programarla en el código. Esto se consigue habilitando una propiedad de los botones.

La casilla "Continuamente" tiene la clave. Para que un procedimiento se repita Continuamente (un número ilimitado de veces), se debe hacer lo siguiente:

- Crear un botón asignado al procedimiento que deseamos que se repita Continuamente.
- Abrir la ventana de edición del botón y marcar la casilla "Continuamente", la cual por omisión se encuentra desmarcada.
- Hacer clic sobre el botón OK para cerrar la ventana.



Ventana del botón con la casilla "Continuamente" marcada.

Realizadas estas acciones, cada vez que el procedimiento es llamado pulsando el botón, el procedimiento correrá repitiéndose una y otra vez y sólo se detendrá si ocurre alguna de tres cosas: 1) se cumple alguna condición de parada impuesta en el código, 2) el usuario fuerza la terminación de las repeticiones pulsando de nuevo el botón 3) el usuario detiene las acciones abriendo la solapa "Herramientas" de la Barra de Menús y seleccionado la opción "Detener". Sin embargo, tómese nota de que cuando el procedimiento es llamado desde la Ventana del Observador, el procedimiento sólo corre una vez, aunque tenga la casilla "Continuamente" marcada.

Cuando se marca la casilla "Continuamente" en la ventan de un botón, su procedimiento asociado se repite ininterrumpidamente y se puede detener haciendo clic nuevamente sobre el botón.

Comentarios dentro del código.

Hasta el momento los comentarios y explicaciones sobre el código de los ejemplos se han agregado al final del código y debido a la naturaleza pedagógica de este libro seguiremos empleando esta práctica en la mayoría de los ejemplos. Sin embargo, todos los lenguajes de computación ofrecen la posibilidad de intercalar comentarios explicativos dentro del código mismo de un modelo o programa. Estos comentarios explicativos están dirigidos exclusivamente a los humanos y su finalidad es facilitar la comprensión del código a las personas que lo leen, incluyendo a quienes lo han escrito. La construcción de grandes programas suele ser un trabajo en equipo en el cual hay grupos de personas que se ocupan de distintos módulos del programa. Los comentarios juegan un papel de gran ayuda para el buen entendimiento entre estos equipos. Más aún, los comentarios también pueden ser de gran utilidad a las mismas personas que han escrito el código o partes del mismo, pues con mucha frecuencia se debe dar mantenimiento o introducir mejoras a programas escritos tiempo atrás. En NetLogo los comentarios se inician con el par de caracteres ";;" de punto y coma y terminar al final de la línea. Esto le advierte al intérprete que el texto que sigue a estos caracteres y concluye en el final de la línea no es parte del código y debe ser ignorado. Un comentario puede iniciarse en cualquier parte de una línea y no necesariamente al inicio de la misma. Si un comentario ocupa más de una línea, cada nueva línea debe empezar con los dos caracteres mencionados. Si se examinan algunos ejemplos de modelos de NetLogo en la Biblioteca de Modelos, se podrá ver la frecuencia con que aparecen comentarios intercalados dentro del código.

Ejemplo 7: Atadura entre dos tortugas.

Primitivas: setxy (fijarxy), create-link-with (crear-enlace-con), tie (atar), wait (esperar).
Otros detalles: Se plantan los botones "setup" y "go", se controla la velocidad del proceso mediante la primitiva "wait", se introducen comentarios dentro del código.

En el ejemplo se crea una atadura entre dos tortugas, lo cual requiere que primero se establezca un enlace entre las dos. Seguidamente se le pide a una de las tortugas que dibuje un círculo. Como el enlace sobre la que se define la atadura es de tipo no direccional, ambas tortugas se pueden considerar como origen y destino de la atadura. En consecuencia, si la tortuga a la que se da la orden gira o avanza, la otra debe hacer lo mismo en la misma cantidad de modo que la distancia entre ambas tortugas permanece constante. No obstante, la atadura no obliga a la tortuga que es tirada a adoptar la misma orientación (heading) inicial que aquella que la tira.

```
to setup
clear-all ;; este es un ejemplo de comentario que será ignorado por el intérprete
crt 2
ask turtle 0 [pd setxy 0 0]  ;; setxy sirve para asignar una posición a una tortuga
ask turtle 1 [pd setxy -2 -2]
end

to go
ask turtle 0 [ create-link-with turtle 1 [tie] ]  ;; se crea la atadura
ask turtle 1 [ repeat 360 [fd 0.1 rt 1 wait 0.1 ]]
end
```

Explicaciones y comentarios adicionales. Para correr el procedimiento primero se debe hacer clic sobre el botón "setup" y luego sobre el botón "go". A la tortuga 1 se le pide trazar un círculo, por lo que su orientación será en todo momento tangente al círculo. La tortuga 0, en cambio, es "arrastrada" por la tortuga 0 y su orientación no es necesariamente tangente al círculo que traza. Es posible definir ataduras asimétricas mediante enlaces dirigidos. En tal caso, cuando la tortuga origen (raíz) se mueve, la tortuga destino (hoja) debe seguirla, pero no a la inversa. La primitiva setxy asigna a la tortuga las coordenadas indicadas como entradas. Recuérdese que las coordenadas de las parcelas no se escriben dentro de paréntesis, como es costumbre en matemática. La orden "wait 0.1" (esperar 0.1) produce una espera de aproximadamente una décima de segundo. Se ha introducido para poder apreciar el movimiento de las tortugas, pues de lo contrario todo ocurriría demasiado rápido (sugerencia: suprima la orden "wait 0.1" y observe lo que sucede).

Ejemplo 8: Tortuga recorre un círculo una y otra vez.

Primitivas: heading (orientación), wait (esperar).
Otros detalles: Se plantan los botones "go" y "setup". Se marca la casilla "Continuamente" del botón "go", para que el procedimiento se repita incesantemente.

En este ejemplo una tortuga dibuja un círculo y lo recorre lenta pero incansablemente, hasta ser detenida por el usuario. Se utiliza la primitiva "heading" (orientación) para asignar la orientación inicial de la tortuga. La primitiva "wait" (esperar) se usa aquí como en ejemplo anterior, para bajar la velocidad de la tortuga y así poder observar su movimiento. El control de la velocidad de un procedimiento también se puede realizar desde la interfaz, utilizando el deslizador en la parte superior del mundo, cuyo valor por defecto es "velocidad normal".
Para correr el programa se hace clic sobre el botón "setup" y luego sobre el botón "go". Para detenerlo se debe hacer clic nuevamente sobre el botón "go". El código es el siguiente:

```
to setup
clear-all
crt 1 [set heading 0 pd]
end
```

```
to go
ask turtle 0 [fd 0.3 right 3]
wait 0.1
end
```

El papel del azar en la construcción de modelos.

Son muchos los fenómenos de la vida real que se pueden modelar eficientemente utilizando mecanismos basados en el azar. Para generar eventos aleatorios NetLogo cuenta con la primitiva "random num" (azar num), la cual funciona con un valor numérico de entrada. Su funcionamiento es el siguiente: "random n" reporta un número entero al azar entre 0 y n - 1. Por ejemplo, la orden "random 5" reportaría un número escogido al azar del conjunto de 5 elementos {0, 1, 2, 3, 4}. Con un poco de ingenio se puede hacer que esta primitiva escoja números al azar en otros rangos numéricos, como se muestra en los ejemplos de órdenes que siguen. Otro mecanismo muy útil para generar condiciones aleatorias lo brinda la primitiva "one-of" (una-de), la cual tiene como entrada una lista de opciones. Esta primitiva reporta una de las opciones de la lista suministrada escogiéndola al azar. En NetLogo una lista se define encerrando sus miembros entre corchetes y separándolos por espacios en blanco. Por ejemplo, [1 3 5 7] es una lista que contiene cuatro enteros. Sin haberlo mencionado explícitamente, hemos ya estado en contacto con las listas, por ejemplo, cuando hemos usado la primitiva "repite", la cual opera con dos entradas, una de las cuales es la lista de órdenes. Veremos más sobre listas adelante.

Ejemplo 9: Órdenes con la primitiva "random".

random 50, reporta un entero al azar entre 0 y 49.
random 50.3, reporta un entero al azar entre 0 y 49.
(random 50) + 50, reporta un entero al azar entre 50 y 99. Si, por ejemplo, random 50 genera el número 17, **(random 50) + 50** tomaría el valor 17 + 50 = 67.
random 100) / 100, reporta un número entre 0 y 1 escogido al azar del conjunto {0.00, 0.01, 0.02, 0.03, ...0.98, 0.99. Si, por ejemplo, random 100 reporta el número 17, **random 100 / 100** tomaría el valor 17 / 100 = 0.17.
(random 1000) / 1000, reporta un número entre 0 y 1 escogido al azar del conjunto {0.000, 0.001, 0.002, 0.003, ...0.998, 0.999}

Ejemplo 10: Órdenes con la primitiva "one-of".

show one-of [1 2 3 4]
==>2, se muestra un número de la lista [1 2 3 4] elegido al azar.
show one-of {"A" "B" "C" "A" "A"}
==>A
Variando la frecuencia de aparición de los miembros que componen una lista se pueden modificar las probabilidades de que salga cada miembro. En la orden anterior la letra "A" tiene el triple de probabilidades de salir que las letras "B" o "C". Internamente la

generación de números aleatorios está basada en un número o semilla (seed), cuyos dígitos satisfacen criterios estadísticos de aleatoreidad, como por ejemplo es el caso de los dígitos de la expansión decimal del número pi. Los usuarios podemos cambiar el valor de la semilla utilizada por NetLogo mediante la primitiva "random-seed" (para mayor información sobre este tema consultar la sección Random Numbers en la Programming Guide del Manual del Usuario [16]).

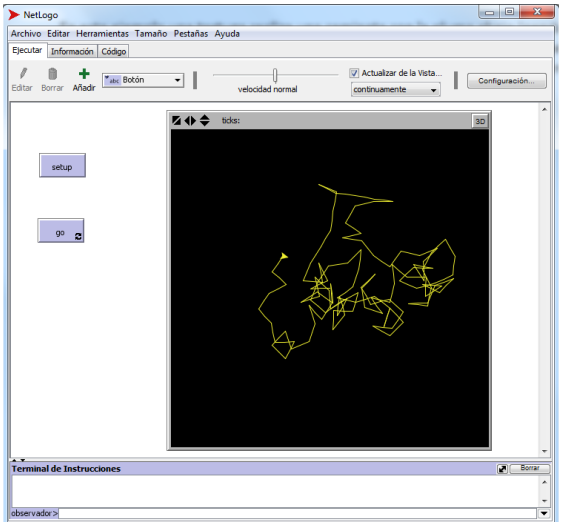
Ejemplo 11: Una caminata aleatoria.

Primitivas: random (azar), wait (esperar).
Otros detalles: Se regula la velocidad del proceso con la primitiva "wait".

En este ejemplo una tortuga realiza una caminata con la pluma abajo (pendown o pd), de modo que va dejando un trazo conforma avanza. La caminata consiste en repetir incesamente las órdenes de avanzar dos pasos hacia adelante y fijar la orientación al azar (random 360), lo cual resulta en una caminata aleatoria.

Preparativos: plantar los botones "setup" y "go" (marcar la casilla "Continuamente" de este último procedimiento).

```
to setup
clear-all
crt 1 [pd set color yellow]
end

to go
ask turtle 0 [fd 2 set heading random 360]
wait 0.2
end
```



Aspecto de la trayectoria de la tortuga luego de algunas pasadas por "go".

Variables: primer encuentro.

Las variables constituyen uno de los recursos más poderosos e importantes de los lenguajes de programación. En NetLogo, todo atributo de los agentes o del ambiente cuyo valor se puede modificar está representado por una variable. Es útil imaginar una variable como una caja en donde se almacena un objeto cuyo valor puede ser cambiado ("variado") en cualquier momento. Los objetos pueden ser de muy distinta naturaleza: números, trozos de texto, listas o incluso conjunto-agentes. En toda variable podemos distinguir dos características básicas: el nombre de la variable y su valor, también llamado el contenido. Volviendo a la comparación de una variable con una caja, el nombre de la variable representaría el nombre de la caja y el valor sería su contenido. Hay variables que son creadas por los usuarios y otras que son creadas por el sistema. A estas últimas las llamamos "variables preinstaladas" (en el sistema). Esto sugiere una primera división de las variables en dos grandes grupos:
I. Variables presinstaladas en el sistema.
II. Variables definidas por los usuarios.

Variables preinstaladas en el sistema.
Los agentes de NetLogo poseen varias características propias, las cuales pueden ser cambiadas por los usuarios. Las tortugas, por ejemplo, poseen color, posición (dada por sus coordenadas xcor y ycor), orientación (heading) y el estado de su pluma (penup o pendown), entre otras cosas. Las parcelas por ejemplo poseen coordenadas de posición (pxcor, pycor) y color (pcolor). Los enlaces, por su parte, también poseen características propias como son las tortugas que forman sus extremos, el grosor del enlace y la propiedad de ser visible o no. Todas estas características de los agentes se representan por medio de variables. Algunas veces la variación es producida por los usuarios, por ejemplo, cuando se fija el color de una tortuga, otras veces el sistema NetLogo lo hace de manera automática, por ejemplo, si pedimos a una tortuga que avance una cierta cantidad, el sistema ajusta automáticamente los valores de sus coordenadas de posición. En el ejemplo anterior se asignó el color amarilo a una tortuga mediante la orden "set color yellow". El valor de las variables presinstaladas se cambia con la primitiva "set". El formato de la orden es:

set nombre-de-la-variable nuevo-valor

Ejemplo 12: Órdenes con la primitiva set.

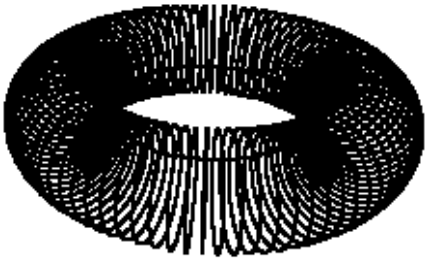
crt 2, se crean dos tortugas cuyos números de identidad son 0 y 1.
ask turtle 1 [print heading]
==> 225, la tortuga 1 imprime su orientación.
ask turtle 1 [set heading 45 print heading]
==> 45, la tortuga 1 asigna el nuevo valor 45 a su orientación y la muestra en la Terminal de Instrucciones.
ask turtle 0 [print who]
==> 0, la tortuga 0 imprime su número de indentidad.
ask patch 3 6 [print pcolor]
==> 0, la parcela 3 6 imprime su color (0 = negro).
ask patch 3 6 [set pcolor red], la parcela 3 6 cambia su color a rojo.

La primitiva “print” envía el valor de la entrada a la Terminal de Instrucciones, pero a diferencia de la primitiva “show”, el valor no va precedido del nombre del agente que ejecuta la orden. Algunas variables pueden ser consultadas por los usuarios, pero no pueden ser modificadas. Tal es el caso de la variable “who”, la cual reporta el número de identificación de una tortuga. Esta variable es creada por el sistema en el momento en que una tortuga nace. Los números who se van asignando conforme las tortugas se crean, comenzando a partr de 0. Cuando una tortuga muere su número who no es reasignado a ninguna de las tortugas que quedan vivas, es decir, no se produce una renumeración del conjunto de las tortugas vivas, pero el sistema podría reasignar el número eliminado a alguna tortuga que naciera después.

Las estructuras topológicas del mundo.

El área de la matemática que estudia las propiedades fundamentales de los conjuntos, las figuras y los espacios se conoce con el nombre de *topología*. A diferencia de la geometría, la cual toma en cuenta la medida de las longitudes, áreas y ángulos, en el caso de las superficies la topología ignora estas cantidades y estudia propiedades aún más básicas como, por ejemplo, el número de agujeros que posee la superficie, la forma en que se conectan unas partes con otras, o el número de vecinos que tienen las parcelas cuando se cuadricula la superficie. He aquí las tres estructuras topológicas que puede adoptar el mundo de NetLogo.

- Topología del toro o de la rosquilla. Es la topología del mundo por omisión. Como ya hemos visto, cuando una tortuga abandona el cuadrado del mundo por el extremo derecho, se la ve aparecer por el extremo izquierdo y viceversa. Similarmente, si abandona el mundo por el extremo superior se la verá emerger por el extremo inferior y viceversa. Esta no es una propiedad geométrica del mundo sino una propiedad topológica. Una superficie que tenga esta propiedad recibe el nombre de “toro” y tiene la forma de una rosquilla, como explicamos a continuación. Cuando hablamos del toro o rosquilla nos referimos al objeto bidimensional constituido por su superficie, no al cuerpo sólido que además de la superficie incluye su interior. La mejor manera de representar la superficie de un toro en la pantalla plana del computador es haciéndole un par de cortes y extendiendo la superficie resultante sobre la pantalla. Esto se comprende mejor si realizamos el proceso inverso, que consiste en convertir un trozo plano de papel en un toro por medio de pegar sus bordes opuestos. Imaginemos que tomamos una hoja rectangular o cuadrada de papel y que unimos el extremo superior de la hoja con su extremo inferior. El resultado sería una superficie en forma de tubo horizontal. Si a continuación -y suponiendo que la hoja estuviera hecha de un material flexible- dobláramos el tubo hasta unir sus dos extremos, el resultado sería una rosquilla hueca (un toro). Recomendamos ver una bella animación de esta construcción en la página Web: http://mathematica.stackexchange.com/questions/42493/morphing-a-sheet-of-paper-into-a-torus.

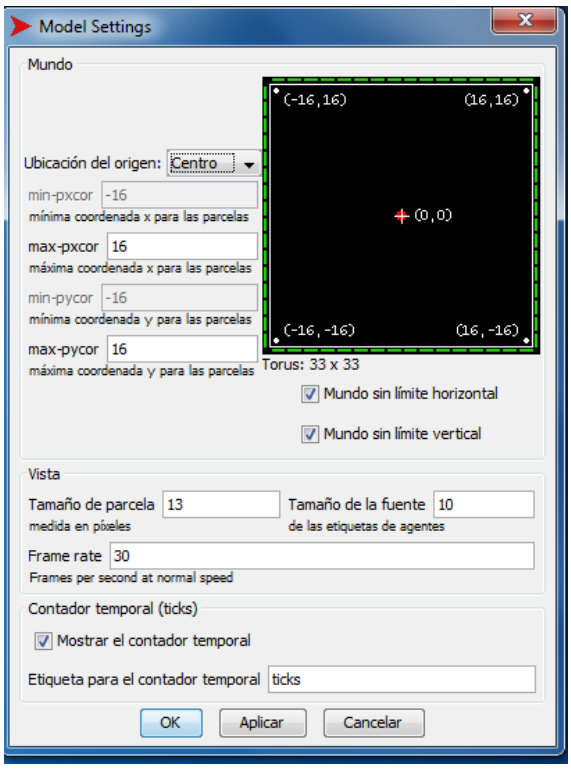


Cuando el mundo está configurado con la topología del toro (la topología por omisión), debemos tener presente que el cuadrado que miramos en la pantalla es un toro que ha sido abierto mediante dos cortes: un primer corte transversal, que lo convierte en un tubo y un segundo corte longitudinal sobre el tubo, que lo convertiría en un rectángulo o un cuadrado, según las dimensiones del tubo. Hay que tener en cuenta que la superficie de un toro no tiene bordes ni esquinas, mientras que el cuadrado que lo representa en la pantalla sí los tiene, sin embargo no se trata de bordes reales sino de bordes aparentes. Si cuadriculáramos un toro, toda parcela del cuadriculado poseería ocho parcelas vecinas, cuatro por los lados y otras cuatro en las esquinas porque la superficie de un toro no tiene bordes ni esquinas. Cuando se cuadricula una región cuadrada, como una hoja de papel, observamos que las parcelas de los bordes y de las esquinas tienen menos vecinas que las parcelas del interior.

- Topología del cilindro o tubo. El mundo adquiere la forma de un cilindro o tubo si se permite que dos de los bordes opuestos del cuadrado que representa el mundo se unan, pero los otros dos bordes no. Por ejemplo, si se unen solamente los bordes superior e inferior, se obtendría un tubo o cilindro truncado en los extremos derecho e izquierdo, el cual estaría orientado horizontalmente. En este caso los bordes izquierdo y derecho del cuadrado serían también los bordes izquierdo y derecho del mundo y estas parcelas tendrían 5 vecinas mientras que las restantes tendrían 8 parcelas vecinas. En la tolopogía del cilindro no existen parcelas esquineras. Si se permite sólo la unión de los bordes derecho e izquierdo el cilindro estaría verticalmente orientado.

- Topología de la caja. Con esta estructura topológica el mundo es como una hoja de papel real, o sea una caja bidimensional cuadrada, limitada por sus cuatro lados. En este mundo los bordes y las esquinas son reales. Cuando una tortuga llega a uno de los cuatro bordes, queda imposibilitada de continuar avanzando pues habría llegado a la frontera del mundo. En esta estructura topológica las parcelas de los bordes tienen 5 vecinas, las esquineras sólo 3 vecinas y las parcelas interiores tienen 8 parcelas vecinas.

Se cambia de una topología a otra haciendo clic sobre el botón “Configuración” de la Barra de Operaciones y marcando o desmarcando las casillas “Mundo sin límite horizontal” o “Mundo sin límite vertical”. Al iniciar NetLogo ambas casillas se encuentran marcadas, lo que confiere al mundo la topología de la rosquilla. La topología del cilindro se obtiene desmarcando una de las casillas y la de la caja desmarcando ambas casillas. La figura de la izquierda muestra la ventana de configuración del mundo con la topología de la rosquilla o toro. Hay que tener presente que la topología que tenga el mundo puede afectar la medición de distancias y de ángulos. La distancia entre dos agentes -parcelas o tortugas- se medirá siempre según la trayectoria más corta. Con la topología de la rosquilla, por ejemplo, si una tortuga se encuentra muy cerca del “borde aparente” derecho y otra muy cerca del “borde aparente” izquierdo, la trayectoria más corta entre las dos sería aquella que sale del “borde aparente” derecho y entra por el izquierdo. Las orientaciones de las tortugas también pueden verse afectadas por la estructura topológica del mundo. Si se pide a una tortuga que mire (se oriente) en dirección hacia otra tortuga o parcela, lo hará en la dirección de la trayectoria más corta. Son los usuarios modeladores quienes deben escoger el tipo de topología que más conviene al modelo o programa que están desarrollando.



Ejemplo 13: Una mosca atrapada en una caja.

Primitivas: random (azar), lt (abrev. de left), wait (esperar).
Otros detalles: Se cambia la topología por omisión del mundo por la topología de la caja. Se plantan los botones “go” y “setup” y se marca la casilla “Continuamente” del botón “go”.

En este ejemplo una tortuga adopta el comportamiento de una mosca confinada dentro de una caja cuadrada bidimensional (el mundo). La mosca vuela siguiendo una trayectoria aleatoria y no puede atravesar ninguno de los bordes del mundo, debido a que su topología es la de la caja. El código es el siguiente:

```
to setup
clear-all
crt 1
```

```
end

to go
ask turtle 0 [fd 5 lt random 360]
wait 0.1
end
```

Comentarios y explicaciones. El programa se pone en marcha pulsando primero el botón "setup" y luego el botón "go" y se detiene pulsando de nuevo este último botón. La primitiva "wait num" ya la usamos en ejemplos anteriores con el mismo propósito que aquí: producir una espera para moderar la velocidad de la mosca.

El tipo de topología del mundo puede influir sobre el valor de la distancia entre dos agentes o sobre su orientación cuando a un agente se le pide mirar en dirección hacia otro agente.

Variables: segundo encuentro.

Variables creadas por los usuarios. Muchos modelos requieren que los usuarios puedan definir sus propias variables para almacenar valores de atributos que el modelo necesita. Por ejemplo, en un modelo en donde la supervivencia de un conjunto de ovejas depende de la cantidad de pasto que encuentran al deambular por un terreno árido, sería conveniente asignar a cada oveja una variable, cuyo valor fuera la energía vital que la oveja posee en cada momento. A esta variable se le asignaría un valor inicial cuando la oveja nace, valor que aumentaría cada vez que la oveja come un poco de pasto y disminuiría cuando la oveja deambula sin poder encontrarlo. Una vez que el usuario ha creado una variable puede asignarle valores, consultar dichos valores o incluso eliminar la variable. El nombre y el valor inicial de una variable se asignan en el momento de su definición. NetLogo asigna a toda variable creada por el/la usuaria un valor inicial de cero, indistintamente del tipo de dato que se piensa alojar en la variable. Los nombres que se puede dar a las variables deben constar de una sola palabra. Una buena práctica en programación consiste en asignar a cada variable un nombre que evoque el uso que se piensa dar al dato que se almacenará en ella. Para facilitar este propósito, se permite formar nombres compuestos pegando varias palabras mediante puntos, guiones y otros caracteres. Si por ejemplo el propósito de la variable es almacenar los números de matrícula de automóviles, su nombre bien podría ser "MatriculaDeAutos" o "matricula-de-autos".

Variables globales y variables-de-agentes.
Se llaman "variables globales" las variables que pueden ser consultadas o su valor modificado por cualquier agente en cualquier momento. Se puede pensar que las variables globales pertenecen al Observador, quien permite que cualquier agente consulte su valor o lo modifique. Las variables globales se deben definir al inicio del código, antes de los procedimientos y mediante la primitiva "globals":

```
globals[variable1 variable2….]
```

Cuando hay varias variables globales, todas se incluyen en los corchetes de la primitiva "globals" separadas por espacios en blanco.

Variables-de-agentes.
Los conjunto-agentes también pueden tener sus propias variables, las cuales llamaremos "variables-de-agentes"[6]. Sólo los miembros del conjunto-agentes que es dueño de las variables tienen derecho irrestricto a manipular esas variables. Las variables de conjunto-agentes también se declaran al inicio del código, antes de los procedimientos citando al conjunto-agentes seguido de la palabra "own" (poseen). Por ejemplo, las variables que pertenecen al conjunto-agentes de las tortugas se declararían mediante la siguiente expresión: turtles-own[variable1 variable2…]. A diferencia de las variables globales, cuyo valor es el mismo para todos los agentes, una variable de tipo conjunto-agentes puede tener un valor diferente para cada agente del conjunto-agentes al que la variable pertenece. Es como si bajo el mismo nombre de la variable, cada agente tuviera una versión personal de la misma.

Ejemplo 14: Creación, consulta y modificación de variables globales.

Primitivas: globals, set, print.

En este ejemplo, en el editor de código primero se crean dos variables globales llamadas "ahorro-Lidia" y "ahorro-Pedro", que representan el balance en las cuentas de ahorros de Lidia y de Pedro. Con sólo la definición de las variables y sin escribir ningún procedimiento adicional es posible consultar o modificar sus valores desde la Ventana del Observador, desde donde realizaremos consultas y cambiamos sus valores. En el Editor de Código escribamos:

globals[ahorro-Lidia ahorro-Pedro]

Seguidamente, desde la Ventana del Observador escribimos las siguientes órdenes:
print Ahorro-Lidia
==> 0, es valor inicial asignado por el sistema a la variable Ahorro-Lidia.
set ahorro-Lidia 5000, ahora asignamos un nuevo valor de 5000 a Ahorro-Lidia.
print ahorro-Lidia
==> 5000
print ahorro-Pedro
==> 0
set ahorro-Pedro 300
print ahorro Pedro
==> 300
set ahorro-Pedro ahorro-Lidia, asignar a ahorro-Pedro el valor de ahorro-Lidia.
print ahorro-Pedro
==> 5000

[6] Este nombre no está estandarizado pero lo hemos introducido para mayor claridad.

set ahorro-Lidia ahorro-Lidia + ahorro-Pedro / 2, asignar a ahorro-Lidia el valor 5000 + 2500

Ejemplo 15: Diferencia entre variables globales y variables-de-agentes.

Primitivas: globals, turtles-own (tortugas-poseen), set, one-of (una-de).
Otros detalles: Se crean variables cuyo valor es una lista. Otras variables toman valores al azar de una lista. Plantar un botón "setup".

En un momento dado, una variable global posee un solo valor, el cual puede ser cambiado por cualquier agente. Las variables de agentes, en cambio, pueden tener valores diferentes para cada agente en un mismo momento. Aunque el nombre de una variable-de-agentes es compartido por todos los agentes, en realidad cada agente posee una copia personal de la variable. En el Editor de Código escribamos los siguientes tres procedimientos, que nos permitirán hacer consultas desde la Ventana del Observador.

```
globals[edad estatura] ;; variables globales
turtles-own[nombre] ;; variable-de-agentes

to setup
crt 6 ;; se crean 6 tortugas
set edad one-of [10 20 30 35 40 50 55 60]
ask turtles [set nombre one-of  ["Julius"  "Marie" "Carlos"  "Linda" "Emilio"
"Marcela"]]
end

to mostrar-estatura-1
crt 8
ask turtles [set estatura random 50]
ask turtles [show estatura]
end

to mostrar-estatura-2
crt 8
ask turtles [set estatura random 50 show estatura]
end
```

En la Ventana del Observador escribamos algunas órdenes. Comprobemos primero que a cada tortuga se le ha asignado un nombre, en principio, diferente[7]:

```
ask turtle 0 [show nombre]
==>(turtle 0): Linda
ask turtle 1 [show nombre]
==>(turtle 1): Carlos
```

[7] Sin excluir la posibilidad de que algunos valores se puedan repetir.

En cambio, cualquier tortuga, que consulte el valor de la variable “edad” encontrará el mismo valor escogido al azar de la lista [10 20 30 35 40 50 55 60] (en este ejemplo es 30).

ask turtle 0 [show edad]
==>30
ask turtle 4 [show edad]
==>30

Observemos los diferentes resultados que se obtienen cuando a las tortugas se les pide mostrar su “estatura” mediante los procedimientos “mostrar-estatura1” y “mostrar-estatura2”. En el primer procedimiento “mostrar-estatura1”, en la instrucción “ask turtles [set estatura random 50]” cada tortuga fija el valor de la variable estatura sin mostrar su valor. Cuando la última tortuga ejecuta esta orden, el valor de la variable “estatura” queda fijado en una cantidad y no vuelve a ser cambiado. Debido a ello, cuando en la siguiente orden “ask turtles [show estatura”] se le pide a las tortugas que muestren el valor de la variable “estatura”, todas muestran el valor fijado por la última tortuga. Con el segundo procedimiento “mostrar-estatura2” las tortugas muestran diferentes valores de la variable “estatura” porque cada una muestra el valor de la variable justo después de cambiarlo. Sin embargo, hay que notar que se trataba de un mismo valor para todas las tortugas, el cual es cambiado varias veces.

Una variable global almacena un solo valor, mientras que una variable-de-agentes almacena un valor para cada agente. Esto permite asignar valores diferentes a los agentes.

El tipo de una variable.

En programación se denomina “tipo” de una variable al tipo de datos que se almacenan en la variable. Los tipos más comunes son: entero, flotante (número con decimales), booleano (verdadero o falso), lista y cadena. En NetLogo se debe agregar el tipo conjunto-agentes. En muchos lenguajes de programación –en especial los de uso general- cada vez que se define una variable el sistema obliga a indicar el tipo al que la variable pertenece. El propósito de este requisito es que el sistema reserve la cantidad de memoria necesaria para el manejo de la variable y así poder realizar una mejor administración de los recursos de la máquina (por ejemplo, se requiere más espacio para almacenar un número con decimales que un número entero). Como hemos visto, en NetLogo no es necesario declarar el tipo de dato que se piensa almacenar en las variables. Una excepción a esta regla se presenta cuando se usan algunas ventanas creadas para ingreso de datos, como se explicará más adelante.

Expresiones condicionales.

En el lenguaje ordinario nos encontramos con frecuencia con expresiones del tipo “si se da esto entonces debe ocurrir esto otro”, expresiones conocidas con el nombre de *expresiones condicionales*. Un lenguaje de programación que no tuviera la capacidad de manejar expresiones condicionales sería un lenguaje de uso muy limitado. Las expresiones condicionales permiten a un lenguaje manejar alternativas, en donde el flujo del programa se bifurca, siguiendo uno u otro camino según que una condición se

cumpla o no. En NetLogo existen dos primitivas para construir expresiones condicionales: "if" e "ifelse". La palabra inglesa "if" se traduce como el "si" condicional que usamos en la oración "si vienes a la fiesta entonces te podré presentar a mi novia". La palabra "ifelse" no es realmente una palabra inglesa. Es una fusión de las palabras "if" (si) y "else" (en otro caso), que en algunas implementaciones del Lenguaje Logo se ha traducido al español como "siotro". La sintaxis del condicional "if" es la siguiente:

if (condición) [órdenes si condición se cumple] órdenes restantes...

El intérprete examina si la condición se cumple, en cuyo caso la condición reporta el valor "true" ("verdadero"). De no cumplirse reporta el valor "false". Cuando la condición se cumple el intérprete procede a ejecutar las órdenes dentro de los corchetes, después de lo cual continúa con las restantes órdenes del procedimiento después de los corchetes. Si la condición no se cumple el intérprete ignora las órdenes dentro de los corchetes y procede a ejecutar las órdenes después de los corchetes. Por ejemplo, examinemos la orden: "if longitud > 10 [set color red] fd 4…" que se traduciría como "si el valor de la variable longitud es mayor que 10 entonces fijar el color de la tortuga en rojo y luego avanzar 4 pasos…". Si la condición no se cumple el intérprete se salta los corchetes y pasa directamente a la orden fd 4 y restantes. Una expresión o variable que reporta los valores "true" o "false" (verdadero o falso) se conoce como expresión o variable "booleana", en honor del matemático y filósofo inglés George Boole (10815-1864), quien hizo importantes contribuciones a la lógica y a la matemática. La primitiva "ifelse" tiene un formato muy parecido al de "if", con la diferencia de que hay dos corchetes con órdenes, uno para cuando la condición se cumple y otro para cuando no se cumple. Dado que una condición forzosamente se cumple o no se cumple[8], esto significa que el intérprete ejecutará siempre las órdenes dentro de alguno de los dos corchetes, pero nunca las de ambos. La sintaxis de "ifelse" es la siguiente:

ifelse (condición) [órdenes si condición se cumple] [órdenes si condición no se cumple] órdenes que siguen después del condicional…

Ejemplo 16: Plantando señuelos para escapar (uso de condicionales).

Primitivas: if (si condicional), distance (distancia), sprout (brotar), stamp (estampar), set, heading.
Otros detalles: se construye una expresión condicional con la primitiva "if".

Aunque el propósito de un programa sea el de ilustrar un ejemplo sobre el uso de una primitiva o un concepto, siempre resulta más ameno si relacionamos el ejemplo con alguna historia, aunque sea ficticia. La historia de este ejemplo es la siguiente: dos tortugas cuyor colores son rojo y azul siguen caminatas aleatorias por el mundo. La tortuga roja piensa que la azul la persigue y para confundirla diseña una estrategia que consiste en estampar imágenes inertes de sí misma, a modo de señuelos. Cada vez que la distancia entre ambas es menor que una cierta cantidad la tortuga roja estampa un

[8] Principio *Tertium non datur* o del tecero excluso de la lógica clásica.

señuelo. Este hecho se modela con una expresión condicional: “si te acercas a menos de 5 pasos entonces [estampo mi figura]”, en código: “if distancia < 5 [stamp]”. Las imágenes producidas con “stamp” (estampar) tienen la misma apariencia que la tortuga que las genera, pero no son agentes sino sólo imágenes inertes. Conforme las tortugas deambulan, es de esperar que el mundo se irá llenando de estampas inmóviles, a la vez que las tortugas siguen deambulando. El programa se detendrá oprimiendo nuevamente el botón “go”.

Preparativos: plantar los botones setup y go y marcar la casilla “Continuamente” de este último botón. He aquí el código:

```
globals[distancia]
to setup
clear-all
ask patch 10 0 [sprout 1] ;; de la parcela 10 0 brota una tortuga
ask turtle 0 [set color blue]
ask patch -10 0 [sprout 1 ] ;; de la parcela -10 0 brota una tortuga
ask turtle 1 [set color red]
end

to go
ask turtle 0 [set heading random 360 fd 2]
ask turtle 1 [set heading random 360 fd 2
set distancia distance turtle 0
if distancia < 5 [stamp]
        ]
wait 0.1
end
```

Comentarios y explicaciones. La orden “set heading random 360” se traduce como “asignar (fijar) la orientación igual al número azar 360”, es decir, seleccionando un ángulo al azar entre 0 y 259. Note bien que las palabras “distancia” y “distance” tienen significados diferentes: “distancia” es el nombre una variabla global que hemos definido mientras que “distance” es el nombre de una primitiva, la cual reporta la distancia entre dos agentes. Esta primitiva reporta la distancia que hay del agente que da la orden al agente que se suministra como entrada. En este ejemplo se le solicita a la tortuga 1 “ask turtle 1 [... set distancia distance turtle 0]” fijar el valor de la variable “distancia” igual a la distancia que la separa de la tortuga 0. Como la orden se encuentra dentro de los corchetes de la tortuga 1, es esta tortuga la que estampa su figura. Es la tortuga 1 la que huye de la tortuga 0.

Capítulo 3: Ambiente y Lenguaje III

Variables: tercer encuentro.

En este tercer encuentro con las variables introduciremos las llamadas *variables locales*. La idea básica detrás de una variable local es la de limitar su existencia a un cierto contexto, tanto en espacio como en tiempo. Fuera de este contexto la variable deja de existir, liberando el espacio de memoria que ocupaba. Las variables locales no se declaran al inicio del código, como las variables globales y se crean dentro del código con la primitiva "let" (permitir), seguido del nombre de la variable y de su valor. Por ejemplo, si quisiéramos crear una variable local llamada "energía" con un valor inicial de 100 lo haríamos con la instrucción: "let energía 100" (permitir a la variable local energía tomar el valor de 100). Con esa orden la variable "energía" es creada y le es asignado un valor. En el ejemplo anterior "Plantando señuelos para escapar", en el cual definimos la variable global "distancia" habría sido más económico cambiar la naturaleza de la variable "distancia" de global a local. Para ello, bastaría con eliminar la expresión "globals[distancia]" y en el procedimiento "go" reemplazar la primitiva "set" por "let":

```
ask turtle 1 [set heading random 360 fd 2
let distancia distance turtle 0 ;; se cambió "set" por "let"
if distancia < 5 [stamp] ]
```

Con este cambio la variable "distancia" se convierte en local, y sólo existe dentro de los corchetes de la instrucción "ask turtle 1 [...]". Si después de corrido el procedimiento, en la Ventana del Observador digitáramos la orden "ask turtle 1 [show distancia]", obtendríamos el mensaje de error "No existe nada llamado "distancia", lo cual indica que la variable ya ha sido eliminada. Si la variable fuera global, se podría seguir manipulando desde la Ventana del Observador aún después de corrido el procedimiento.

Familias de agentes (breeds).

NetLogo permite crear ciertos conjunto-agentes que llamaremos familias (breeds[9]). Es posible crear familias de tortugas o de enlaces, pero no de parcelas. Las familias deben estar constituidas por agentes del mismo tipo. Por ejemplo, una familia de tortugas llamadas "lobos" sería una subclase del conjunto-agentes de las tortugas, las cuales responderían a órdenes cuando las mismas estuvieran dirigidas a los lobos o a un lobo en particular. La creación de familias facilita el manejo de agentes en grupos de los que se esperan comportamientos diferentes. Las familias se deben declarar al inicio del código, antes de los procedimientos. Las familias de tortugas se declaran con la primitiva "breed" y las de enlaces con las primitivas "directed-link-breed" o "undirected-link-breed" según sea el caso. En la declaración de una familia hay que incluir el nombre de los miembros de la familia en plural y en singular, encerrados entre corchetes. Por ejemplo, si deseamos crear una familia de tortugas llamadas "lobos", esto se debe declarar en la forma:

[9] Tal vez la traducción al español que más se acerca a la palabra "breed" sea "camada" (crías de un mismo parto). No obstante, nos parece que "camada" se queda algo corta mientras que "raza" quizás peca por exceso. Finalmente hemos optado por "familia", que se encuentra en una posición intermedia.

breed [lobos lobo]

Dentro de los corchetes de una primitiva “breed” no se puede incluir más de una familia. Si se desea crear dos o más familias, cada una debe ser declarada por separado. Tendremos oportunidad de ver cómo utilizar familias en los ejemplos que siguen. La razón por la cual se requiere la inclusión del nombre en singular y plural se debe a que NetLogo habilita la posibilidad de utilizar expresiones con la versión singular o la plural del nombre, según sea necesario. Por ejemplo, una vez creada la familia “lobos” se pueden usar expresiones como “ask lobos [..órdenes..]” o bien “ask lobo 3 [..órdenes..”]. Aunque NetLogo no contempla la creación de familias de parcelas, es posible crear conjunto-agentes de parcelas mediante primitivas auxiliare como “with” o “neighbors”, lo cual compensa este faltante.

Variables propias de una familia. Una característica muy ventajosa de las familias es el hecho que pueden tener sus propias variables. Por ejemplo, se pueden crear variables de la familia lobos con la expresión “lobos-own[...variables...]”. Sin embargo, es necesario aclarar que, la familia de los lobos es en realidad una subclase de las tortugas y los lobos no pierden la cualidad de pertenecer al conjunto-agentes de las tortugas. En cuanto a la asignación de números who, el intérprete asigna número de corrido a todas las tortugas según se van creando, independientemente de la familia a que pertenecen. Debido a esto, no siempre los números who de los miembros de una familia serán números consecutivos, esto dependería del momento en que sus miembros han sido creados. El ejemplo siguiente ayudará a aclarar las cosas.

Ejemplo 17: Dos familias.

Primitivas: breed (familia), create-lobos, lobos-own (lobos-poseen), create-ovejas, ovejas-own.
Otros detalles: Creación de familias (breeds) con variables pertenecientes a cada familia.

En este ejemplo se crea una familia de lobos y otra de ovejas y a cada familia se le asignan dos variables. Se invoca el procedimiento “dos-familias” desde la Ventana del Observador y luego se prueban algunas órdenes en esta misma ventana. En este ejemplo no es necesario plantar botones setup ni go en la interfaz. He aquí el código:

```
breed[lobos lobo]
breed[ovejas oveja]
lobos-own[energía velocidad]
ovejas-own[resistencia peso]

to dos-familias
clear-all
crt 10
create-lobos 10
create-ovejas 10
ask lobos [set energía random 100 set velocidad 40]
ask ovejas [set resistencia 50 set peso (random 10) + 20] ;; se asigna el peso
;; de las ovejas entre 20 y 29
```

ask lobo 12 [show energía]
ask oveja 20 [show resistencia]
end

Llamamos al procedimiento "dos-familias" desde la Ventana del Observador:
dos-especies
==> (lobo 12): 27, lobo 12 muestra su energía.
==> (oveja 20): 50, oveja 20 muestra su resistencia.

Ahora digitamos órdenes aisladas en la Ventana del Observador:
ask turtle 1 [show energía]
==> Turtle breed no tiene variable energía

ask lobo 8 [show energía]
==> turtle 8 is not a lobo, la tortuga 8 no es un lobo, es una tortuga.
ask oveja 23 [show resistencia]
==> (oveja 23): 50

ask turtle 23 [show energía]
==> (oveja 23): 50, la tortuga 23 responde, pero identificándose como oveja pues en realidad es una tortuga "disfrazada" de oveja.
ask lobo 15 [ask peso]
==> lobos breed no tiene variable peso

ask ovejas [forward 5], las 10 ovejas avanzan 5 pasos.
ask turtles [set color yellow], las 30 tortugas se vuelven amarillas.

Explicaciones y comentarios adicionales. Por ser las tortugas las primeras en ser creadas reciben los números who de 0 a 9. Los lobos se numeran de 10 a 19 y las ovejas de 20 a 29. Las tortugas 0 a 9 no tienen acceso directo a ninguna de las variables de los lobos o las ovejas. Debido a que los miembros de una familia, por ejemplo, la familia "lobos" son tortugas que "se han disfrazado" de lobos, se puede hacer referencia a cada lobo de dos maneras posibles: en tanto que lobo o en tanto que tortuga, siempre y cuando en ambos casos se utilice el mismo número who. Esto explica por qué la instrucción "ask turtle 23 [show resistencia]" es equivalente a la instrucción "ask oveja 23 [show resistencia]". Los agentes "turtle 23" y "oveja 23" son el mismo agente. En cambio, la orden "ask turtle 23 [show velocidad]" produce el mensaje de error "Ovejas breed no tiene variable velocidad", pues el intérprete sabe que la tortuga 23 no es un lobo sino una oveja. Otro modo de pensar esto es el siguiente: las tortugas 0 a 9 son "tortugas puras", las del 10 a1 19 son "tortugas-lobos" y las del 20 al 29 "tortugas-ovejas". Pero desde el punto de vista de los conjunto-agentes solamente hay tres de ellos:

- turtles (tiene 30 miembros): las tortugas del 0 al 29.
- lobos (tiene 10 miembros): las tortugas del 10 al 19.
- ovejas (tiene 10 miembros): las tortugas del 20 al 29.

La variable ticks.

En NetLogo existe una variable global preinstalada cuyo nombre es “ticks”. El uso más frecuente que se da a esta variable es el de marcar el paso del tiempo contando las veces que se repite un procedimiento. La orden “tick” aumenta en una unidad el valor de la variable “ticks”, por lo que es muy común incluirla en el procedimiento “go” cuando tiene la casilla “Continuamente” marcada e interesa contar el número de veces que este procedimiento se repite. La variable se activa mediante la orden “reset-ticks” con un valor inicial de 0. En relación al manejo de la variable “ticks” se tienen las siguientes tres primitivas:

- “reset-ticks” inicializa la variable ticks con un valor de 0.
- “ticks” reporta el valor actual de la variable ticks.
- “tick” aumenta el valor de la variable ticks en una unidad.

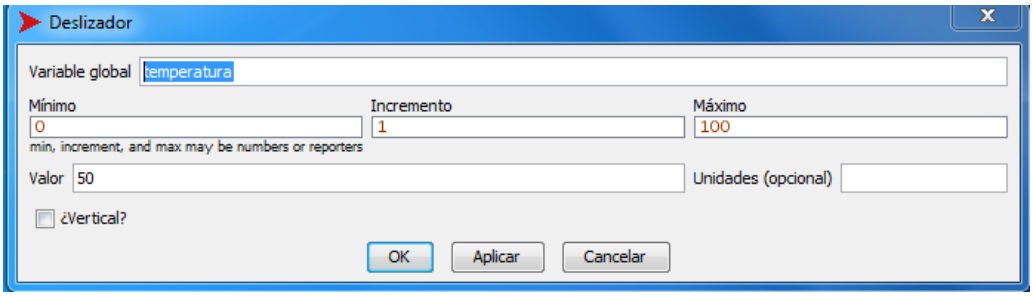
Deslizadores y gráficos.

Además de los botones, otros dos objetos de gran utilidad en la construcción de modelos son los deslizadores y los gráficos. Para plantar cualquiera de ambos objetos se procede del mismo modo que con los botones: en la ventana del Seleccionador de Objetos se selecciona “Deslizador” o “Gráfico”, según sea el caso y se planta el objeto en un lugar conveniente. Tanto deslizadores como gráficos poseen su propia ventana de edición para asignar las propiedades que determinan su comportamiento y apariencia. Esta ventana se abre posando el puntero del ratón sobre el objeto y haciendo clic derecho con el ratón.

Deslizadores.

En la construcción de modelos se presenta con mucha frecuencia la necesidad de definir variables y parámetros de entrada cuyos valores influyen directamente en la conducta del modelo. Interesa poder variar estos parámetros para estudiar la respuesta del modelo a estas variaciones. Una de las maneras más ágiles de introducir estos cambios de valores es por medio de los objetos llamados deslizadores. Estos objetos poseen una manija deslizante que sirve para modificar el valor de la variable que representan. Las variables definidas de este modo son de tipo global pero no se deben declarar en el Editor de Código mediante la primitiva “globals[...]”. Consideremos como ejemplo un delizador que define la variable global “temperatura”. Una vez plantado su aspecto es como el que muestra la figura:

En la ventana del deslizador se debe indicar: el nombre de la variable global, los valores mínimo y máximo que toma la variable, el tamaño mínimo de los incrementos de su valor, el valor inicial, las unidades (opcional) y la posición vertical u horizontal del deslizador, como lo muestra la figura siguiente:



Ventana del deslizador para una variable llamada "temperatura".

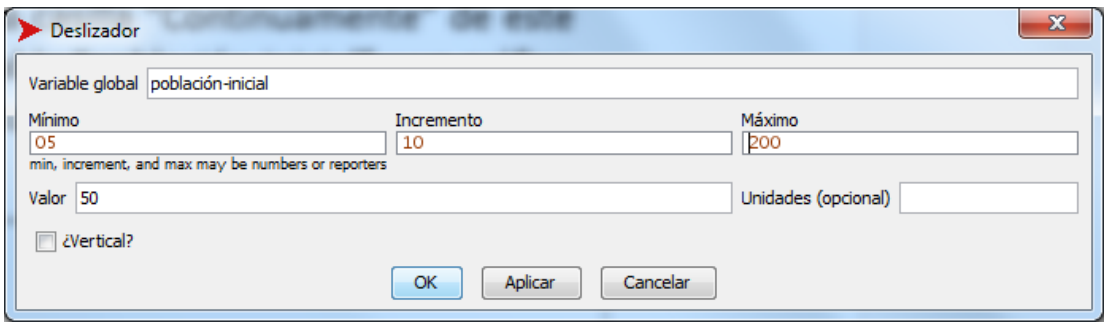
Gráficos.
Igual que los botones y los deslizadores, los gráficos son una de las opciones disponibles en la ventana del Seleccionador de Objetos. En el ejemplo que sigue se muestran las características de su ventana y se plantará un gráfico que mostrará la variación de la población de tortugas en el tiempo.

Ejemplo 18: Una población fluctuante.

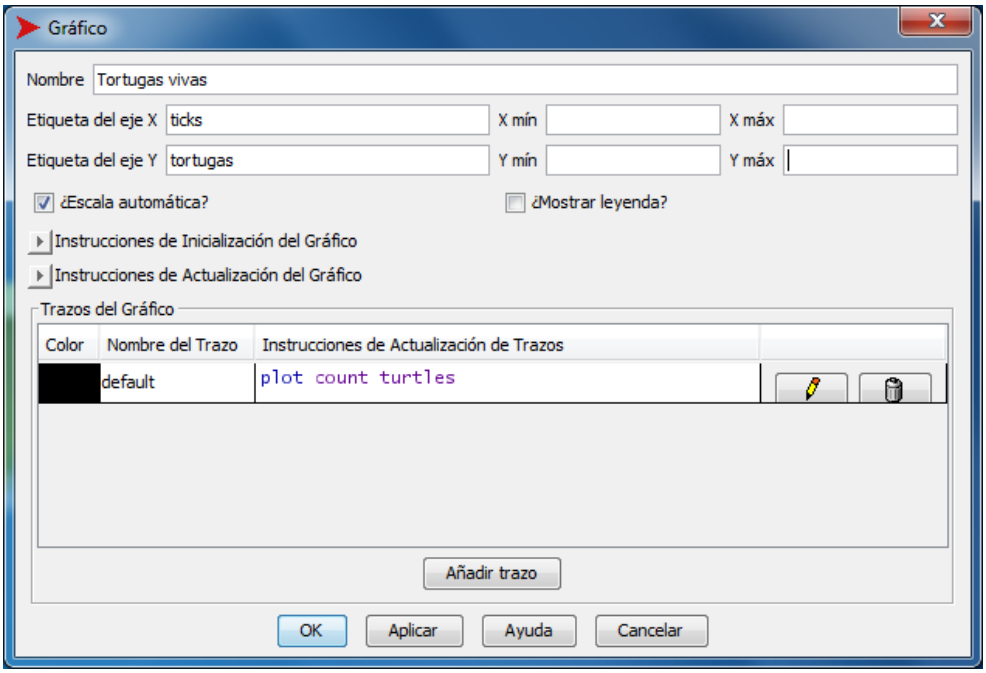
Primitivas: reset-ticks (reiniciar ticks), tick (tic), die (morir), sprout, count (contar).
Otros detalles: se usan expresiones condicionales, se planta un deslizador y un gráfico.

En este ejemplo tenemos una población de tortugas cuyo número inicial de individuos se representa mediante la variable "población-inicial". El valor de esta variable se asigna mediante un deslizador. En el ejemplo, las tortugas deambulan siguiendo trayectorias aleatorias y mueren cuando se acercan mucho al borde derecho del mundo:
"if xcor > 14 [die]", lo cual hace que la población disminuya con el tiempo. Cuando la población tiene menos de 10 individuos, se hace brotar de una parcela (patch 0 0) una cantidad de tortugas igual a la población inicial: if count turtles < 10 [sprout población-inicial]" y el ciclo se repite. Se grafica la población fluctuante de tortugas a lo largo del tiempo reportando el número de tortugas vivas (count turtles) en el eje Y con la variable ticks en el eje X.

Preparativos: Plantar botones setup y go y marcar la casilla "Continuamente" de este último. Plantar un deslizador para representar la variable "población-inicial" y un gráfico para graficar el valor de la población en todo momento. Llenar los campos del deslizador con los siguientes valores en su ventana de edición: Variable global: población-inicial, Mínimo: 05, Incremento: 10, Máximo: 200, Valor: 50.



Abrir la ventana del gráfico e introducir los siguientes valores: 1. Nombre: Tortugas vivas, 2. Etiqueta del eje X: ticks, 3. Etiqueta del eje Y: tortugas, 4. Marcar la casilla "¿Escala Automática?", 5. Instrucciones de Actualización de Trazos: plot count turtles.



Algunos de los restantes campos de la ventana tienen valores que no modificaremos. Cuando se marca la casilla "¿Escala Automática?" Netlogo ignora los valores en los campos X min, Y min, X max, Y max y ajusta continuamente la escala para que el gráfico no se salga del rango visible. He aquí el código:

```
to setup
clear-all
crt población-inicial
;; se crean tantas tortugas como lo indica la variable población-inicial, definida por
;;el deslizador
reset-ticks ;; reinicia la variable ticks en 0
end

to go
ask turtles [fd 2 set heading random 360]
ask turtles [if xcor > 14 [die]] ;; tortugas que se acercan al borde derecho mueren
ask patch 0 0 [if count turtles < 10  [sprout población-inicial]] ;; si la población inicial
;; es menor a 10 la población inicial renace
wait 0.04
tick ;;  aumentar el valor de tick en una unidad
end
```

Explicaciones y comentarios adicionales. La primitiva “count” reporta el número de miembros de un conjunto-agentes. La primitiva “sprout” hace brotar una cantidad de tortugas igual al valor de la variable población-inical, en la parcela que emite la orden (en este caso la parcela 0 0). Las tortugas brotan de esta parcela con orientaciones al azar.

Procedimientos con datos de entrada.

En el ejemplo anterior hemos visto cómo se puede asignar el valor de una variable a un procedimiento por medio de un deslizador en la interfaz. Ahora veremos cómo se pueden suministrar valores a las variables de un procedimiento digitando estos valores como entradas, a la derecha del nombre del procedimiento. Las entradas de los procedimientos se comportan como variables locales. En el código, las variables de entrada de un procedimiento –también llamadas parámetros- se declaran a la derecha de su nombre encerradas entre corchetes (quienes hayan tenido algún contacto previo con el lenguaje Logo, recordarán que en este lenguaje los parámetros de entrada de los procedimientos también se declaran a la derecha de su nombre, pero precedidos del caracter “:” dos puntos). En NetLogo, si el procedimiento posee varias entradas, todas ellas se declaran dentro del mismo par de corchetes y separadas por espacios en blanco. Cuando se llama al procedimiento, el valor asignado a la entrada se suministra a la derecha del nombre, aunque ahora sin incluir los corchetes.

Ejemplo 19: Tres pequeños procedimientos con entradas.

Primitivas: setxy (asignarxy), type (escribir).
Otros detalles: se crean tres procedimientos con parámetros de entrada.

A continación se presentan tres pequeños ejemplos de procedimientos con una, dos y tres entradas respectivamente. Los procedimientos se llaman desde la Ventana del Observador.

1- Procedimiento con una entrada.
El siguiente procedimiento dibuja un círculo cuyo color depende de la entrada “coloración”:

```
to círculo [coloración]
clear-all
crt 1 [setxy -8 8 pd set color coloración]
ask turtle 0 [repeat 360 [fd 0.1 rt 1]]
end
```

Llamamos al procedimiento en la Ventana del Observador dando a la entrada un valor numérico, por ejemplo, con el color 15:
círculo 15, la variable coloración toma el valor 15 y la tortuga dibuja un círculo con color 15 (rojo).
círculo blue, la tortuga dibuja un círculo azul. La primitiva blue reporta el número 105 (azul).

2- Procedimiento con dos entradas.

```
to casamiento [novio novia]
type novio type " y " type novia type ", os declaro marido y mujer"
end
```

Llamamos al procedimiento en la Ventana del Observador con “Carlos” y “Cecilia” como valores de las entradas:

```
casamiento “Carlos” “Cecilia”
==> Carlos y Cecilia, os declaro marido y mujer
```

3- Procedimiento con tres entradas.

El siguente ejemplo evalúa el polinomio $5x^2 + 2xy – xyz$ en las tres variables, x, y, z.

```
to polinomio [x y z]
show 5 * x * x + 2 * x * y - x * y * z
end
```

Llamamos al procedimiento en la Ventana del Observador con las entradas x = 3, y = 2, z = 7 en la manera:

```
polinomio 3 2 7
==> 15
```

Explicaciones y comentarios adicionales: La primitiva “setxy num1 num2” fija la posición de la tortuga en la parcela de coordenadas (num1 num2). La primitiva “type” es similar a “show”, con la diferencia de que al final del texto no fuerza un cambio de línea (retorno de carro). Esto permite que el resultado de varias expresiones “type” quede en una misma línea. Nótese que el espacio en blanco también es un carácter y es respetado cuando se encuentra dentro de las comillas de “type” o “show”. Hemos incluido un espacio en blanco antes y después de la palabra “y”. En NetLogo, como en la mayoría de los lenguajes de computación, se respetan ciertos convenios establecidos sobre la jerarquía de las operaciones matemáticas, lo cual ahorra la escritura de paréntesis y mejora la comprensión de las expresiones. Debido a estos convenios no ha sido necesario escribir el polinomio del ejemplo en la forma (5 * x * x) + (2 * x * y) - (x * y * z). Cuando se necesita desviarse de estos convenios, entonces se requieren los paréntesis, por ejemplo, la expresión (5 * x * x + 2) * ((x * y) - x) * y * z evaluada con las mismas entradas 3, 2 y 7 tendría un valor de 1974.

Los parámetros de entrada de un procedimiento se declaran a la derecha de su nombre y entre corchetes. Cuando son varios se separan mediante espacios en blanco.

Listas y cadenas (strings).

Presentamos a contínuación dos recursos de enorme importancia del lenguaje NetLogo: las listas y las cadenas (“strings”). Las listas, junto con las cadenas constituyen las estructuras de datos básicas del lenguaje NetLogo y de los lenguajes derivados del

lenguaje Lisp[10]. Vamos a introducir primero el tema de las listas y con ello habremos hecho también un importante avance en el tema de las cadenas, pues muchas de los conceptos y primitivas para el manejo de las listas se aplican también a las cadenas. Una lista es un conjunto de objetos colocados en un cierto orden. En NetLogo los miembros o ítemes de una lista se incluyen entre corchetes o se escriben precedidos de la primitiva "list". Una cadena es una secuencia de caracteres colocados en cierto orden y encerrados entre comillas, como si formaran una palabra. Los espacios en blanco se consideran un carácter más y pueden formar parte de una cadena. Las secuencias de caracteres "soyuna-cadena#2" y "también soy una cadena" son ejemplos de cadenas con 15 y 22 caracteres respectivamente. El hecho de que los miembros de una lista y una cadena se encuentren ordenados permite referirse a cada miembro por la posición que ocupa dentro de la lista o la cadena. La lista [1 2] y la lista [2 1] se consideran distintas, lo mismo que las cadenas "a2" y "2a". Los miembros de una lista o una cadena se enumeran comenzando a partir de cero. La característica más importante de las listas es el hecho de que los miembros de una lista pueden ser, a su vez, listas. NetLogo tiene un repertorio amplio de primitivas para manipular listas y cadenas. Mencionamos a contínuación las operaciones básicas más importantes que se pueden realizar con las listas:

1) Crear una lista.
2) Reportar verdadero o falso si un objeto es o no una lista.
3) Reportar verdadero o falso si un objeto es o no miembro de una lista.
4) Reportar un miembro (ítem) de una lista según la posición que ocupa.
5) Agregar miembros (ítemes) a una lista en primer o último lugar.
6) Remover el primer o último miembro de una lista.
7) Contar el número de miembros de una lista.
8) Fusionar dos listas en una sola.

Los ejemplos de órdenes siguientes servirán para ilustrar el uso de algunas de estas primitivas.

Ejemplo 20: Órdenes con listas.

Las listas se construyen con las primitivas "list" o con los corchetes "[ ]".
show (list 1 2 "whisky"), aquí "whisky" es una cadena.
==> [1 2 "whisky"], muestra la lista recién construida.
show [1 2 "whisky"], otra manera de construir la misma lista.
==> [1 2 "whiksy"]

Los ítemes dados por palabras o trozos de texto (cadenas) se deben encerrar entre comillas, para que el intérprete no intente evaluarlos como si fueran expresiones "vivas" del lenguaje, por ejemplo, nombres de procedimientos, de primitivas o nombres de variables.

Se usa "member?" (miembro?) para saber si un objeto es miembro de una lista:
show member? "e" ["a" "b" "c" "d"]

[10] El nombre Lisp es una contración de las palabras "list processing".

==> false, reporta falso porque "e" no es miembro de la lista dada.

"fput" (poner primero) y "lput" (poner-último) colocan el item dado en el primer o último lugar en la lista dada. Los nombres provienen de contraer las expresiones first-put y last-put.
fput 18 ["es mi número predilecto"]
==> [18 "es mi número predilecto"]

La primitiva "ítem" reporta el item o miembro que ocupa la posición indicada:
show item 2 ["árbol" "botella" [6 7] 3]
==> [6 7], la lista [6, 7] es el item número 2 de la lista ["árbol" "botella" [6 7] 3], pues los ítemes se cuentan a partir de cero.

Para remover un item se usa "remove" y se indica la posición del ítem que se quiere remover.
show remove 1 ["esto" "no" "es" "una" "lista"]
==> ["esto" "es" "una" "lista"]

"but-last" (menos el último) y "but-first" (menos el primero) reportan una lista menos su último o primer miembro.
show but-last ["gato" 1 4 "perro"]
==> ["gato" 1 4].
show but-first ["gato" 1 4 "perro"]
==> [1 4 "perro"]

first y last reportan el primero y el último miembro de la lista dada.
show first [10 20 ["Hoy" "es" "lunes"]]
==> 10
show first last [10 20 ["Hoy" "es" "lunes"]]
==> "Hoy", "Hoy" es el primer miembro de la lista ["Hoy" "es" "lunes"] y esta lista es es el último miembro de la lista [10 20 ["Hoy" "es" "lunes"]]

"length" reporta el número de miembros de una lista.
show length [1 2 [3 4]]
==> 3, porque la lista tiene 3 miembros. No confundir "length con "count" que cuenta el número de integrantes de un conjunto-agentes.

Con "sentence" (frase) podemos fusionar dos listas.
show sentence [1 2][3 4]
==> [1 2 3 4]

Estos ejemplos no agotan el repertorio de primitivas para el manejo de listas. En lo que a cadenas se refiere, la mayoría de las primitivas que se aplican a las listas también se pueden aplicar a cadenas sin ninguna modificación. Recomendamos ver las listas de primitivas de listas y de cadenas, con una explicación de cada primitiva, en el Diccionario de NetLogo. Veremos el uso de las cadenas en algunos de los ejemplos más adelante.

Ejemplo 21: Asignación de asientos en un avión I.

Primitivas: let (asignar), one-of (uno-de), type.
Otros detalles: se definen dos listas.

En este ejemplo suponemos que un programa escoge al azar la posición del asiento de las personas en un vuelo, dada por un número que indica la fila en que se encuentra el asiento y una letra que indica la columna. Las filas se toman de la lista de números del 1 al 30 y las columnas se toman de la lista [“A” “B” “C” “D”]. El programa “tu-asiento” genera al azar la posición de un asiento y la escribe en la Terminal de Instrucciones. El ejemplo muestra nuevamente el uso de la primitiva “let” para definir una variable local.

```
to tu-asiento
let fila one-of [1 2 3 4 5 6 7 8 9 10 1 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30]
let columna one-of [”A” “B” “C” “D” “E” “F” “G”]
type “Tu asiento es el “ type fila type columna
end
```

Ejemplo de un posible resultado al correr el procedimiento en la Ventana del Observador:

```
tu-asiento
==> 27 C, vaya suerte, ni ventana ni pasillo.
```

Explicaciones y comentarios adicionales. La primitiva one-of, (uno-de), que ya hemos usado, selecciona al azar un ítem de la lista dada. Si se llama al procedimiento tu-asiento desde la Ventana del Observador se obtendrá la posición de un asiento. Las variables locales “fila” y “columna” definidas con “let” sólo existen mientras se ejecuta el procedimiento “tu-asiento”. Una vez ejecutado este procedimiento las variables desaparecen, lo cual podemos comprobrar si intentamos consultar su valor en la Ventana del Observador escribiendo la instrucción “show fila” o “show columna”.

Procedimientos reportadores.

Así como existen primitivas actoras (comandos) y reportadoras, lo mismo ocurre con los procedimientos. Con frecuencia se necesita que un procedimiento reporte datos a otros procedimientos. Los procedimientos reportadores deben comenzar con la primitiva “to-report” en vez del usual “to”. Además, el procedimiento reportador debe finalizar con la expresión “report dato”, donde “dato” es el valor que el procedimiento reporta. Por ejemplo, dado el procedimiento:

```
to-report columna
report one-of [”A” “B” “C” “D” “E” “F” “G”]
end
```

el mismo puede ser llamado desde la Ventana del Observador usando una primitiva que utilice una entrada, como por ejemplo “type”:

type columna
==> “D”

Ejemplo 22: asignación de asientos en un avión II.

Primitivas: to-report, report.
Otros detalles: se crean dos procedimientos reportadores, se emplean listas y se aplica el principio de reutilización.

En el ejemplo se utilizan dos procedimientos reportadores para generar los números de fila y de columna de los asientos y también generar una clave para el boleto del pasajero. Con el uso repetido de estos dos procedimientos (Principio de Reutilización) en los procedimientos “asiento” y “boleto” se generan el asiento y la clave del pasajero. He aquí el código:

```
to asiento
type “Tu lugar es el “ type fila type columna
end

to boleto
type “Tu clave del boleto es “
type columna type columna type fila type fila type columna type fila
end

to-report columna
report one-of [”A” “B” “C” “D” “E” “F” “G”]
end

to-report fila
report one-of [1 2 3 4 5 6 7 8 9 10 1 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30]
end
```

Explicaciones y comentarios adicionales. Aquí tenemos dos procedimientos reportadores “fila” y “columna” los cuales seleccionan un número y una letra respectivamente y los reportan a los procedimientos o primitivas que los soliciten. En el ejemplo son solicitados por la primitiva “type’ desde los procedimientos “asiento” y “boleto”. Si se llama a cualquiera de los procedimientos “asiento” o “boleto” desde la Ventana del Observador, se obtendrá un asiento y una clave para el boleto. Por ejemplo:

asiento
==>17F

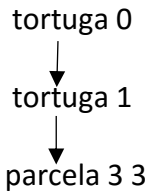
boleto
==>ED16C9

Cómo detener un procedimiento con “stop”.

No siempre se desea que el usuario sea quien detenga un modelo. En muchos casos se quiere que el modelo se detenga cuando se dan ciertas condiciones. En un programa donde una población de depredadores se alimenta de una población de presas, si ambas poblaciones desaparecieran, no nos interesaría que el programa siga corriendo, pues ya no podría ocurrir nada interesante. En NetLogo disponemos de la primitiva “stop” para detener un programa en marcha. La primitiva “stop” no funciona como el “botón de pánico” que tienen algunas aplicaciones, el cual detiene la aplicación de forma inmediata, independientemente del estado en que se encuentra. La primitiva “stop” puede detener un modelo por completo o sólo algunas de sus acciones dependiendo del nivel donde se encuentre la orden. Para entender el funcionamiento de esta primitiva debemos considerar los distintos niveles en que se puede encontrar un agente cuando emite la orden “stop”. Si un agente se encuentra encapsulado dentro de las órdenes de otro agente, decimos que el nivel del que se encuentra encapsulado es inferior al nivel del agente que lo encapsula. Por ejemplo en la orden:

```
ask turtle 0 [fd 5 ask turtle 1 [set color blue ask patch 3 3 [sprout 2]]]
```

hay tres niveles de encapsulamiento en que hay tres agentes involucrados: la tortuga 0, la tortuga 1 y la parcela 3 3. Se observa que la orden “ask patch 3 3 [sprout 2]” se encuentra encapsulada dentro de los corchetes de la tortuga 1 y esta última se encuentra encapsulada dentro de los corchetes de la tortuga 0. Esto significa que el agente parcela 3 3 se encuentra un peldaño más abajo que el agente tortuga 1 y dos peldaños más abajo que la tortuga 0.



La tortuga 0 ocupa el nivel más elevado y tiene potestad de detener sus propias acciones, que incluyen las de la tortuga 1 y la parcela 3 3. La tortuga 1, en cambio podría detener las acciones suyas y las de la parcela 3 3, pero no tiene potestad para detener las acciones de la tortuga 0. Finalmente, la parcela 3 3 sólo podría detener las acciones dentro de sus propios corchetes. Ilustramos esto mostrando algunas órdenes y sus efectos.

Un agente tiene potestad de detener las acciones dentro de su propio nivel y las de aquellos agentes encapsulados en niveles inferiores al suyo, pero no las de aquellos agentes que se encuentran en niveles superiores.

Ejemplo 23: Órdenes con la primitiva “stop”.

En la Ventana del Observador escribamos las siguientes órdenes:

crt 2 [set color red], primero se crean dos tortugas rojas.
ask turtle 0 [ask turtle 1 [fd 10] set color white], la tortuga 1 avanza 10 pasos y la tortuga 0 se vuelve blanca. Nótese que la tortuga 0 ocupa un nivel un peldaño más elevado que la tortuga 1
ask turtle 0 [ask turtle 1 [stop fd 10] set color white], la tortuga 0 se vuelve blanca, la tortuga 1 no avanza pues ella ha detenido sus propias acciones antes de la orden fd 10. Sin embargo, no tiene potestad para detener la orden set color white dada por la tortuga 0, la cual se encuentra a un nivel más alto que el suyo.
ask turtle 0 [ask turtle 1 [fd 10] stop set color white], la tortuga 1 avanza 10 pasos y la tortuga 0 no se vuelve blanca.
ask turtle 0 [stop ask turtle 1 [fd 10] set color white], la tortuga 1 no avanza y la tortuga 0 no se vuelve blanca. Aquí la tortuga 0 ha detenido sus acciones y también las de la tortuga 1, la cual se encuentran en un nivel inferior a ella.

Los mismos principios que se aplican al encapsulamiento entre agentes se aplican al encapsulamiento entre procedimientos. Un procedimiento que llama a otro procedimiento se encuentra en un nivel superior al procedimiento llamado. Una orden "stop" en el procedimiento llamado no detiene las acciones en el procedimiento que hace la llamada.

Ejemplo 24: Una lista de mil enteros.

Primitivas: ticks, reset-ticks, length (longitud de una lista), stop, lput (poner de último).
Otros detalles: se usa una expresión condicional para detener el procedimiento. Una lista se va llenando con la primitiva "lput".

En este ejemplo se crea una lista llamada "enteros" que comienza vacía y se va llenando con números enteros. Estos enteros son los valores que adquiere la variable ticks. Comenzando con el valor 0, la primitiva tick hace que el valor de la variable ticks aumente en una unidad. Debido a que la casilla "Continuamente" del procedimiento go se encuentra marcada, en cada pasada por go, la variable ticks aumentará en una unidad y la primitiva "lput" (abreviatura de "last-put") hará que el valor de dicha variable sea agregado en último lugar a la lista "enteros". Ese es el sentido de la orden:

set enteros lput ticks enteros

la cual se podría traducir al español como: "asignar a la lista enteros el mismo valor que tiene, pero agregándole como último miembro el valor de ticks". El procedimiento se detiene cuando la lista "enteros" alcanza los 1000 miembros.

Preparativos: 1) Plantar los botones usuales "setup" y "go" (con la casilla "Continuamente" marcada). He aquí el código:

```
globals[enteros]

to setup
```

```
clear-all
set enteros [ ] ;; se crea una lista de enteros inicialmente vacía
reset-ticks ;; se asigna el valor inicial 0 a la variable ticks
end

to go
set enteros lput ticks enteros ;; el valor de ticks se coloca de último en la lista
if length enteros = 1000 [show enteros stop] ;; cuando la lista alcanza mil miembros
;; la lista es enviada a la Terminal de Instrucciones y el procedimiento se detiene
tick ;; la variable ticks se aumenta en una unidad
end
```

Envío de la salida (output) a un archivo.

La zona de salida (output area) de NetLogo es por omisión la Terminal de Instrucciones. El anterior ejemplo muestra lo poco apropiada que resulta esta terminal para almacenar volúmenes grandes de información. Afortunadamente hay otras opciones, como por ejemplo desviar la información hacia un archivo de texto. Pero para ello es preciso crear previamente el archivo. Si se envía la salida a un archivo inexistente, NetLogo no se encargará de crearlo y debe ser un archivo de texto sin formato[11]. Si el archivo está ubicado en la misma carpeta donde se encuentra el ejemplo o modelo, basta con dar el nombre del archivo, por ejemplo, mi-archivo.txt. Si se encuentra en otra carpeta es necesario suministrar la ruta completa. Una vez que el archivo ha sido creado, antes de dirigir la salida (ouput) al mismo, es necesario abrirlo, y cuando se ha finalizado el envío, el archivo se debe cerrar. Resumimos estas operaciones en cuatro pasos:

1- Crear el archivo (de texto sin formato).
2- Abrir el archivo.
3- Enviar la salida al archivo.
4- Cerrar el archivo.

Ejemplo 25: Envío de una lista de números a un archivo.

En este ejemplo se construye una lista de mil enteros, igual que se hizo en el ejemplo anterior, pero esta vez la salida se envía a un archivo y no a la Terminal de Instrucciones.

Primitivas: reset-ticks, tick, type (escribir), stop, lput (poner-de-último), file-print (imprimir-archivo), length (longitud), file-open (abrir-archivo), file-close (cerrar-archivo).
Otros detalles: Se detiene el procedimiento con la primitiva stop y se envía la salida a un archivo externo.

En este ejemplo se almacenan los primeros 1000 enteros positivos en una variable global en forma de una lista. Un procedimiento aparte llamado “enviar-resultados” se encarga de enviar la lista a un archivo de texto.

[11] Los usuarios de Windows podrían utilizar, por ejemplo, el editor de texto Notepad.

Preparativos: 1) Plantar los botones usuales “setup” y “go” (marcar casilla “Continuamente”). 2) Plantar un tercer botón llamado “enviar-resultados”. 3) Crear un archivo de texto sin formato llamado “mil-enteros.txt” y ubicarlo en el mismo directorio donde se guardar el código del ejemplo. He aquí el código:

```
globals[enteros]
to setup
clear-all
set enteros [ ] ;; se crea una lista de enteros inicialmente vacía
reset-ticks
end

to go
set enteros lput ticks enteros ;; cada nuevo entero se coloca de último en la lista
if length enteros = 1000 [enviar-resultados stop]
tick
end

to enviar-resultados
file-open “mil-enteros.txt”
file-print “Estos son los primeros mil enteros”
file-print “\r\n”
file-print enteros
file-close
end
```

Explicaciones y comentarios adicionales. Algunos editores de texto, por ejemplo Notepad de Windows, no reconocen el retorno de carro enviado con las primitivas “print” y “show”. La presencia de la instrucción “\r\n” tiene por objeto asegurar que se forzará el cambio de línea. Es recomendable comprobar si el editor que se está usando reconoce el retorno de carro para determinar si hay necesidad de incluir la orden mencionada. Se recuerda que las primitiva “type” y “write” no incluyen el retorno de carro.

Antes de enviar la salida a un archivo hay que asegurarse de crearlo previamente. Si el archivo no exite NetLogo no se encargará de crearlo.

Las primitivas “show”, “type”, “print” y “write”.

Para envío de texto o valores numéricos a un lugar de salida disponemos de las primitivas “show”, “print”, “type” o “write”. Estas primitivas difieren en los siguientes aspectos: indican o no el agente que emite la orden, fuerzan o no un cambio de línea (retorno de carro) al final del texto, imprimen o no las comillas de las cadenas, inician o no con un espacio en blanco. Para obtener el resultado deseado a veces se deben emplear combinaciones de estas primitivas. Los siguientes ejemplos muestran algunas de las diferencias entre estas primitivas:

```
type “hola mundo”
```

==> hola mundo

type "hola" type "mundo"
==> holamundo

type "hola" type " " type "mundo"
==> hola mundo

show "hola" show "mundo"
==> Observer: "hola"
==> Observer: "mundo"

write "hola" write "mundo"
==> "hola" "mundo"

print "hola" print "mundo"
==> hola
==> mundo

Conjunto-agentes: segundo encuentro.

Los conjunto-agentes siempre están formados por agentes de un mismo tipo: tortugas, parcelas, enlaces o familias de estos agentes. Los conjunto-agentes, a diferencia de las listas, no son conjuntos ordenados. En cada ocasión en que se invoca a un conjunto-agentes, por ejemplo, con la primitiva "ask", el conjunto es "barajado", de manera que sus agentes pasan a ocupar un nuevo orden formado al azar. Esto evita que algunos de sus miembros ocupen posiciones preferenciales. Los conjunto-agentes no responden a las primitivas para manipular listas. Por ejemplo, usamos la primitiva "count" para contar el número de integrantes de un conjunto-agentes mientras que para listas debemos emplear "length". También se pueden crear conjunto-agentes imponiendo alguna restricción a un conjunto-agentes previamente definido. NetLogo posee primitivas que permiten la creación de conjunto-agentes de esta manera: with, one-of, turtles-on, turtles-here, with-min, with-max son algunas de ellas. Muy frecuentemente esos conjunto-agentes se crean durante un momento de la ejecución del modelo y luego desaparecen. Pero si se desea que adquieran mayor permanencia se pueden almacenar en variables usando "let" o "set", según sea el caso. Si, por ejemplo, hemos creado una variable global llamada "rojas", podríamos almacenar el conjunto-agentes de las tortugas que poseen el color rojo con la expresión "set rojas turtles with [color = red]". Ahora el conjunto-agentes llamado "rojas" podría recibir y ejecutar órdenes como, por ejemplo "ask rojas [...órdenes...]".

Paralelismo real versus paralelismo simulado.

El concepto de la programación basada en agentes se fundamenta fuertemente en la posibilidad de que los agentes realicen tareas simultáneamente. Esto conduce inevitablemente a la pregunta siguiente: ¿es la simultaneidad de acción de los agentes de Netlogo real o simulada? La respuesta a esta pregunta es clara: en las computadoras y sistemas operativos para los cuales se ha desarrollado NetLogo y la mayoría de los

programas de MBA, no es posible el paralelismo o simultaneidad real. Las acciones que parecen ocurrir simultáneamente en realidad lo hacen secuencialmente, bajo un paralelismo simulado. La velocidad de los microprocesadores modernos permite que este paralalismo simulado parezca paralelismo real. Es posible que el progreso en la construcción de computadoras y sistemas operativos que operan con paralelismo real haya resultado ser más costoso y desarrollarse más lentamente de lo que se pensó en un principio. En la actualidad, un modo para implementar procesos en paralelo consiste en la interconexión de un conjunto de computadoras en red para que cada una se ocupe de una parte del procesamiento. La reciente aparición de computadoras con dos, tres, siete y más núcleos en los microprocesadores, también es una ayuda para ciertos tipos de procesos, pero no resuelve el problema que plantea la ejecución simultánea de tareas de cientos o miles de agentes en MBA. Las verdaderas máquinas capaces de operar con paralelismo real en gran escala son algunas super-computadoras instaladas en centros de investigación o reservadas a pocas instituciones y las cuales requieren de software muy especializado. Fuera de estos pocos ejemplos, el mundo de la computación se ha forjado y continúa operando bajo el modelo secuencial. Sobre este tema recomendamos el interesante y muy accesible libro de D. Hillis [8]. No obstante, lo anterior, los usuarios aún conservan cierto nivel de control en lo que a la eficiencia del paralelismo simulado se refiere en los modelos de MBA. Para ello es conveniente entender el funcionamiento de algunas de las instrucciones de NetLogo. Una de las primitivas más empleadas en los modelos de NetLogo es “ask”. Cuando esta primitiva se dirige a un conjunto-agentes, seguida de una lista de órdenes, como por ejemplo en la orden “ask turtles [set color red set heading random 180 fd 1]”, en la cual se pide a las tortugas ejecutar las tres órdenes entre corchetes, las cosas ocurren del siguiente modo: cada vez que la primitiva ask es invocada, produce una lista aleatoria de los agentes y luego cada agente debe ejecutar la secuencia completa de órdenes dentro de los corchetes. Esto significa que hasta que un agente no haya ejecutado todas las órdenes de la secuencia, el siguiente agente no podrá comenzar la ejecución. Sin embargo, si fuera relevante en algún sentido, se podría mejorar el paralelismo simulado partiendo esa orden en tres órdenes separadas: ask turtles [set color red] ask turtles [set heading random 180] ask turtles [fd 1] y de este modo todos los agentes de uno en uno fijarán primero el color rojo, a continuación, en un nuevo orden y siempre de uno en uno fijarán la orientación y finalmente, en un nuevo orden y de uno en uno todos avanzarán un paso. Agreguemos a esta pequeña digresión el hecho que, en muchos modelos, los resultados obtenidos no difieren o difieren en muy pequeña medida tanto si el paralelismo es real como si es simulado.

Capítulo 4: Modelos I

En Los capítulos anteriores hemos introducido un conjunto básico de primitivas de NetLogo junto con una serie de conceptos importantes de programación. También hemos mostrado cómo comunicarnos con NetLogo a través de su interfaz. Pese a que aún quedan muchas cosas por aprender acerca del lenguaje y sobre el ambiente NetLogo, hemos llegado a un punto en el que es posible empezar a construir modelos sencillos. La mayoria de los ejemplos que se han seleccionado para este capítulo pertenecen a un tipo que podríamos llamar "modelos de fantasía". Les llamamos así porque modelan situaciones imaginarias, algunas de las cuales, no obstante, podrían ocurrir en la realidad en modo muy similar al propuesto por el modelo. Consideramos que es más ameno relacionar un programa o modelo con algún tipo de historia, que mostrar un código desligado de toda conexión con nuestra experiencia. A veces es un fenómeno visto, una experiencia vivida o una noticia leída en los periódicos la que origina la creación de un modelo. Como veremos, la presencia de una historia también tiene la ventaja de inducir ideas acerca de maneras como aumentar o introducir variaciones al modelo o de conectarlo con otras áreas de conocimiento. En este sentido, los ejemplos de este capítulo comienzan a mostrar un aspecto muy valioso de la actividad de la programación, tanto si se trata de modelos como de programas en general. La creación de modelos es un caldo de cultivo para la generación de procesos mentales que se dan conforme se programa el modelo y los cuales brindan un amplio espacio para el ejercicio de la creatividad, aspecto que confiere un valor muy especial a la práctica de la programación en el ámbito educativo. Tal vez algunos lectores y lectoras se sorprendan al percibir la riqueza de ideas que encierran o que pueden emanar de algunos modelos, pese a la pequeñez de su código. Como se verá, un modelo no es necesariamente más complejo ni más exitoso por el hecho de emplear mayor número de primitivas, ni por la cantidad de líneas de código que contiene, del mismo modo que la belleza de una obra musical no aumenta por el hecho de utilizar mayor cantidad de notas del pentagrama o por el número de páginas de su partitura. Podría servirnos de ejemplo la música de Mozart o la vida misma con su inmensa variedad y complejidad, la cual está escrita con un alfabeto pequeño de cuatro letras: A, T, C y G correspondientes a los nucleótidos Adenina, Timina, Citosina y Guanina. En este capítulo se continuará con la práctica de intercalar, entre los modelos y ejemplos, nuevos conceptos y técnicas de programación. Los ejemplos, a diferencia de los modelos, son pequeñas unidades de código que pretenden mostrar el funcionamiento de una primitiva o un ilustrar un concepto de programación.

Modelo 1: Tina y Magda visitan la ciudad.

Primitivas: remainder (residuo), pu (abrev. de pluma arriba), one-of (uno-de), xcor, ycor, set (asignar), sound:play-note (sonido:tocar-nota), beep (sonido bip), stop (detener), or (disyunción "o") and (conjunción "y").
Otros detalles: Se planta un deslizador, se detiene el modelo con "stop" cuando se cumple una condición, se emplea la topología del cuadrado y se carga la extensión "sound".

La historia. Dos amigas visitan una nueva ciudad en sus motocicletas. Llevadas por el entusiasmo de conocer los muchos sitios nuevos e interesantes que la ciudad ofrece, se separan y pierden contacto la una con la otra. Las chicas decidieron no llevar sus celulares, por lo que no les queda otro modo de volverse a encontrar que manejar por los cuadrantes de la ciudad hasta poder establecer contacto visual entre ellas. Para que esto ocurra es necesario que ambas chicas se encuentren a menos de una cierta distancia una de la otra y sobre la misma calle o avenida, a fin de que ninguna edificación se interponga en su línea de vista. La distancia que las debe separar para que puedan verse no debe sobrepasar una cierta cantidad, fijada mediante el deslizador "max-visible" en la interfaz. Mientras las jóvenes conducen intentando volverse a encontrar, deciden ir tocando las bocinas de sus motocicletas (este es un recurso meramente cosmético, el cual en realidad no ayuda a que las muchachas se vuelvan a encontrar). A fin de hacer la simulación algo más realista, se evitará que las amigas se devuelvan sobre un mismo cuadrante que acaban de recorrer, por lo cual no está permitido que hagan giros de 180 grados mientras conducen sus motocicletas. Los giros permitidos serán: giro de 90 a la derecha, giro de 90 a la izquierda o giro de 0 grados, cuyo significado es el de continuar avanzando en línea recta. La topología empleada para el mundo debe ser la del cuadrado. El modelo se detiene cuando las muchachas logran encontrarse de nuevo.

Preparativos: Configurar el mundo con la topología del cuadrado. Plantar los botones setup y go, éste último con la casilla "Continuamente" marcada, plantar un deslizador para la variable llamada "max-visible".

Problemas a resolver:

1. Hay que determinar cuándo las chicas se encuentran sobre la misma calle o la misma avenida y cuándo la distancia que las separa es menor que el valor fijado en el deslizador "max-visible".

Este problema se solucionará definiendo cuatro variables: Mxcor, Mycor, Txcor, Tycor, donde el par Mxcor, Mycor reporta la posición de Magda en todo momento y el par Txcor, Tycor la posición de Tina. Si llamamos "calles" a las líneas horizontales (coordenada Y constante) y "avenidas" a las líneas verticales (coordenada X constante), la condición para que Tina y Magda pueden verse es que se encuentren sobre la misma calle o la misma avenida, condición que se cumple si la coordenada X de Tina es igual a la coordenada X de Magda (misma avenida) o bien si la coordenada Y de Tina es igual a la coordenada Y de Magda (misma calle). Podemos expresar esta condición en código mediante la expresión:

Txcor = Mxcor or Tycor = Mycor

donde la primitiva "or"' equivale a la disyunción "o" en español. La otra condición es que la distancia entre las dos chicas sea menor que el valor fijado en el deslizador max-visible mediante la expresión "distancia < max-visible". La exigencia de que ambas condiciones se cumplan simultáneamente se expresa construyendo la conjunción de las condiciones mediante la primitiva "and" (y):

(distancia < max-visible) and (Txcor = Mxcor or Tycor = Mycor)

2. Las chicas deben tocar la bocina de sus motocicletas cada cierto tiempo, pero no al unísono y además las bocinas deben tener sonidos diferentes. Para ello importamos la extensión de sonido "sound", que permite emplear la primitiva "play-note", la cual genera el sonido de las bocinas de las motocicletas. Para controlar el momento en que cada chica toca la bocina se hace uso de la primitiva "remainder num1 num2". Esta primitiva reporta el residuo de la división num1 entre num2. Por ejemplo, "remainder 24 7" reporta 3 (la división 24 entre 7 arroja un residuo de 3) y "remainder 24 3" reporta 0. En el código se establece que una de las chicas toque su bocina cada vez que la división de la variable ticks dividida entre 100 reporte un residuo igual a cero:
   "if remainder ticks 100 = 0 [sound:play-note...]".
   La otra chica lo hace cuando el residuo reportado sea de 20. En ambos casos cada una toca la bocina cada 100 ticks, pero no al mismo tiempo.
3. Hay que trazar los cuadrantes de la ciudad. El procedimiento "cuadrantes-ciudad" se encarga de trazar el entramado de calles y avenidas de la ciudad y se escribe como un procedimiento aparte.

He aquí el código completo:

```
extensions [sound]
globals[distancia Txcor Tycor Mxcor Mycor]

to setup
clear-all
crt 2 [set heading 0]
cuadrantes-ciudad ;; se dibujan los cuadrantes de la ciudad
ask turtle 0 [pu set xcor -15  set ycor -15 set color blue]
ask turtle 1 [pu set color red]
reset-ticks
end

to go
ask turtles [fd 1 right one-of [90 0 -90]]
ask turtle 0 [set Txcor xcor set Tycor ycor
if remainder ticks 100 = 0 [sound:play-note "TRUMPET" 60 64 0.5]]
ask turtle 1 [set Mxcor xcor set Mycor ycor
if remainder ticks 100 = 20 [sound:play-note "TRUMPET" 70 64 0.5] ]
ask turtle 0 [set distancia distance turtle 1]
;; la siguiente orden contiene la condición de parada cuando las chicas se encuentran
if (distancia < max-visible) and (Txcor = Mxcor or Tycor = Mycor)
[sound:play-note "TRUMPET" 80 64 0.5 sound:play-note "TRUMPET" 90 64 0.5 stop]
wait 0.04
tick
end

to cuadrantes-ciudad
ask patches [set pcolor gray]
ask turtle 0 [set color white pu set heading 0 set xcor -16
set ycor -16 pd repeat 16 [fd 32 rt 90 fd 1 rt 90
fd 32 lt 90 fd 1 lt 90 fd 32] bk 32 lt 90 repeat 16
```

```
[fd 32 rt 90 fd 1 rt 90 fd 32 lt 90 fd 1 lt 90 fd 32]
  ]
end
```

Explicaciones y comentarios adicionales:
Las extensiones se declaran al inicio del código, antes de los procedimientos, con la primitiva "extensions". La primitiva "play-note" debe ser precedida de la primitiva "sound:" que es el nombre de la extensión a la que pertenece. Esta primitiva tiene cuatro entradas: nombre del instrumento, altura (pitch) de la nota, volúmen (también llamado velocidad) y duración de la nota en segundos. En la orden: sound:play-note "TRUMPET" 80 64 0.5, el instrumento es la trompeta, la nota tiene una altura de 80, un volumen de 64 y una duración de 0.5 segundos. La extensión "sound" utiliza los mismos parámetros de entrada que el conocido protocolo MIDI[12], y la colección de instrumentos coincide con la lista de instrumentos General MIDI de este protocolo. Puesto que la variable ticks aumenta en una unidad cada vez que el intérprete recorre el procedimiento go (aumento producido por la primitiva "tick"), resulta inevitable que cada 100 ticks cada uno de los 100 residuos de la división "ticks / 100" ocurrirá una vez. Se ha escogido el valor 100 para espaciar los bocinazos cada 100 ticks y no aturdir a los usuarios.

Ejercicio 4.1. En el modelo anterior las chicas tocan las bocinas de modo sincronizado. Magda siempre toca la bocina 20 ticks depués de que lo ha hecho Tina. Diseñar un mecanismo para que las chicas toquen la bocina al azar y de modo no coordinado, lo que daría un toque un poco más realista al modelo.

El anterior modelo bien podría adaptarse a varias otras situaciones, ya fuera con otros tipos de agentes o de entramado, por ejemplo, personas en un bosque, microorganismos en el tejido de un órgano, insectos en la maleza, moléculas en una solución o en una estructura cristalina y hasta personas en una inmensa tienda de departamentos. De hecho, fue una historia contada por un amigo quien por un rato perdió contacto con su esposa en una enorme tienda de departamentos de varios pisos, la que inspiró el ejemplo.

Modelo 2: Fondo para pensionados I.

Primitivas: if, write (escribir), turtles-own, setxy, random-xcor, random-ycor.
Otros detalles: Se colocan deslizadores para varias de las variables del modelo y se grafican el capital del fondo y la población (trabajadores activos + jubilados).

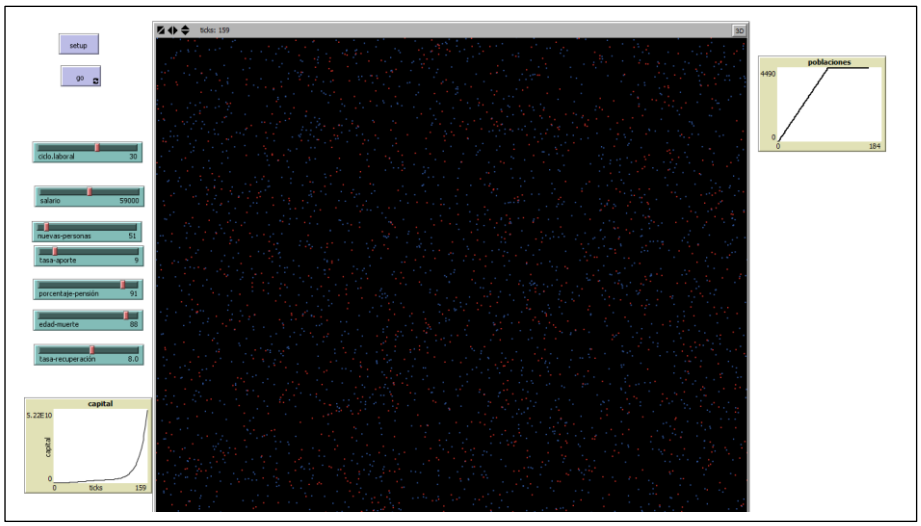
La historia. Los empleados de una institución estatal cotizan un porcentaje de sus salarios a un fondo de pensiones durante todo su ciclo laboral, cumplido el cual adquieren derecho a recibir una pensión hasta el día de su muerte. Las únicas entradas del fondo provienen del aporte que los trabajadores y las trabajadoras hacen en forma de un porcentaje de los salarios que les paga el estado, más un ingreso adicional

[12] MIDI es el acrónimo de Musical Instrument Digital Interface.

proveniente de los intereses que gana el capital acumulado del fondo, al ser colocado en el mercado financiero (en forma de bonos, acciones o en una cuenta ahorros). La finalidad del modelo es analizar la estabilidad del fondo, es decir, las condiciones bajo las cuales el fondo es sostenible y bajo las cuales no lo es. Se dice que un fondo es sostenible si puede seguir pagando las pensiones a los trabajadores por tiempo indefinido sin entrar en quiebra. En la primera versión del modelo se supone que las personas ingresan al régimen laboral a la edad de 25 años, trabajan durante la duración del ciclo laboral y luego ingresan al régimen de personas pensionadas hasta su muerte. En el modelo se utilizan unidades anuales para salarios y deducciones. La primera versión del modelo está basada en algunas hipótesis simplificadoras, no obstante, contiene el mecanismo esencial bajo el cual operan los fondos de pensiones. En la segunda versión se eliminarán algunas de las hipótesis simplificadoras y se presentará un modelo más realista. Describimos primero el significado de aquellas variables del modelo que se definen mediante deslizadores en la interfaz:

- Deslizador “ciclo-laboral”: determina la cantidad de años que deben trabajar los empleados antes de pensionarse.
- Deslizador “nuevas-personas”: determina el número de nuevos trabajadores que ingresan al régimen laboral cada año.
- Deslizador “salario”: fija el salario anual único de todos los empleados, el cual no varía con el tiempo.
- Deslizador “tasa-aporte”: fija el porcentaje del salario que aportan los trabajadores activos y pensionados al fondo anualmente.
- Deslizador “porcentaje-pensión”: fija el porcentaje del salario que el trabajador recibe como pensión.
- Deslizador “edad-muerte”: Establece la edad a que mueren todos los trabajadores.
- Deslizador “tasa-recuperación”: fija la tasa anual a que se coloca el capital del fondo para ganar intereses.

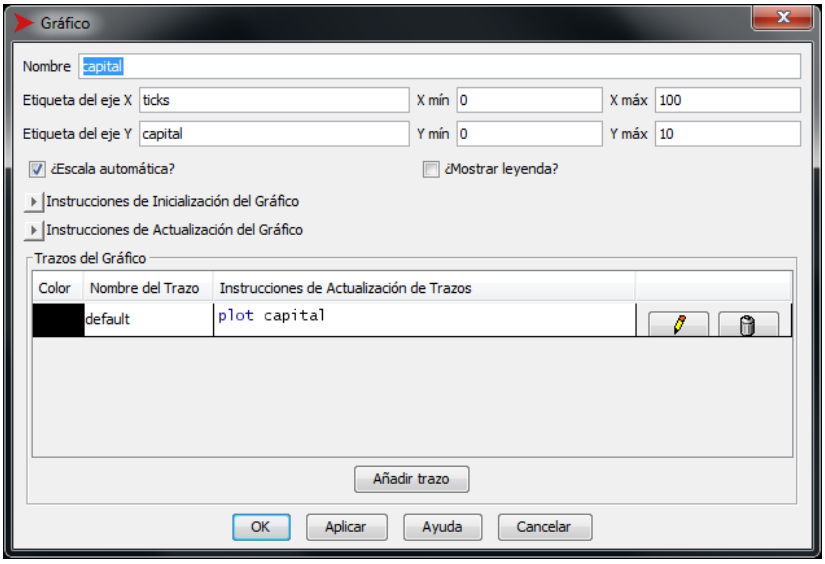
En este modelo los efectos visuales de los agentes tienen escasa importancia y bien podría prescindirse de ellos. Los trabajadores activos se representan por puntos inmóviles de color rojo mientras que los pensionados mediante puntos inmóviles de color azul. No obstante, se verá mucha actividad en la pantalla: los puntos rojos que emergen son los trabajadores que ingresan al régimen laboral, los puntos azules que desaparecen son los trabajadores pensionados que fallecen y lo puntos rojos que se cambian a color azul son los trabajadores que pasan del régimen laboral al de pensionados. En la interfaz se plantan dos gráficos para “plotear” el capital del fondo y el número de empleados vivos, ya sean estos trabajadores activos o pensionados. Si la curva del capital (eje Y) se vuelve decreciente y termina cortando al eje X (que mide el tiempo), el programa se detiene, lo cual significa que el fondo habría quebrado. Debido a que cada año un número de empleados ingresa al régimen laboral a la edad de 25 años y ese mismo número muere a la edad de muerte, la gráfica de la población siempre alcanza un momento a partir del cual permanece constante.



Vista parcial de la interfaz: la gráfica de la izquierda muestra el crecimiento del capital y la de la derecha el valor de la población, la cual después de un tiempo, como puede verse, se vuelve constante.

Problemas a resolver. El problema principial consiste en tener actualizado el capital del fondo en todo momento. Los aportes al fondo que provienen de las cuotas que pagan los empleados se calculan en el procedimiento “contribuir” y los egresos -el pago de las pensiones a los pensionados- se calculan en el procedimiento “recibir-pensión”. A fin de que a los empleados no les corresponda la misma edad, sino que puedan mostrar distintas edades, la variable “edad” no puede ser de tipo global sino de tipo “turtles-own”. En “contribuir” cada empleado activo suma a la variable “capital” un porcentaje de su salario: set capital capital + salario * tasa-aporte / 100. En “recibir-pensión” cada empleado sustrae a la variable “capital” un porcentaje de su salario, tomando en cuenta el hecho de que el pensionado sigue cotizando a su propia pensión:
set capital capital - salario * (porcentaje-pensión - tasa-aporte) / 100. Los intereses que gana el capital por estar colocado en un banco, en bonos o acciones se suman al capital en el procedimiento go: set capital capital + capital * tasa-recuperación / 100. Si el fondo no quiebra debe ser detenido por el usuario haciendo clic sobre el botón “go”.

Preparativos. Plantar los botones setup y go (casilla “Continuamente”) y plantar los deslizadores de las siete variables anteriormente mencionadas. La variable global “capital” se declara en el código y representa el capital que tiene el fondo en todo momento. El valor inicial del capital es cero. Plantar los gráficos para el capital y la población. La ventana de edición del primero es la siguiente:



En la ventana de edición del gráfico para la población ingresar:
Nombre: población
Etiqueta del eje X: ticks
Etiqueta del eje Y: población
Instrucción Actualización de Trazos: plot count turtles

El código del modelo es el siguiente:

```
globals[capital]

turtles-own[edad]

to setup
  clear-all
  reset-ticks
  ask turtles [set edad  25]

end

to go
  create-turtles nuevas-personas [ set edad 25 set color red setxy random-xcor
random-ycor ]
  ask turtles [ifelse edad < 25 + ciclo-laboral [contribuir ]
[set color blue recibir-pensión]
  if edad > edad-muerte [die]
 ]
  set capital capital + capital * tasa-recuperación / 100
 if capital < 0 [show "Terminado"
   write "ticks " write ticks  write " Capital: "
   print capital stop]
  tick
end
```

```
to contribuir
set capital capital + salario * tasa-aporte / 100
set edad edad + 1
end

to recibir-pensión
  set capital capital - salario * (porcentaje-pensión - tasa-aporte) / 100
  set edad edad + 1
end
```

Explicaciones y comentarios adicionales. Cada nueva ocasión en que se corre el procedimiento "go" representa el transcurso de un nuevo año en el modelo, es decir se crean nuevas personas (tortugas) que ingresan al régimen laboral, otras podrían pasar del régimen laboral al de pensionados y otras podrían morir. El modelo arranca cuando los empleados tienen la edad de 25 años y comienzan a cotizar para el fondo (ticks = 0). Para cada empleado, la variable edad aumenta en una unidad en cada pasada por go y empleados que ingresaron al regimen laboral en distintas pasadas por go tendrán distintas edades. Como todos los empleados comienzan a trabajar a los 25 años, la edad de retiro es 25 + ciclo-laboral. En cada pasada por "go" cada tortuga (persona) compara su edad con esta cantidad. Si su edad es mayor que 25 + ciclo-laboral, la tortuga es dirigida al procedimiento "recibir-pensión", en caso contrario es dirigida al procedimiento "contribuir" para que siga cotizando.

Estabilidad del fondo. Variando los diferentes parámetros del modelo podemos encontrar configuraciones que se encuentran en las proximidades de los llamados "puntos críticos", donde pequeñas variaciones de alguno de los parámetros, en el sentido de recargar al fondo, lo convierten en inestable. Se puede comprobar que la estabilidad del fondo no depende del valor inicial del capital, ni del número de personas que ingresan cada año al régimen laboral, mientras este número se mantenga constante, ni del hecho que todos los empleados reciban el mismo salario. Puesto que en el modelo las tortugas no interactúan entre sí, el número de personas que ingresan cada año es irrelevante: un fondo donde ingresan cien o mil personas nuevas por año posee la misma estabilidad que un fondo donde ingresa una persona al año. En lo que se refiere al monto del salario, en la segunda versión del modelo veremos que la diversidad de salarios tampoco influye en la estabilidad del fondo, mientras esta diversidad se mantenga aproximadamente constante: los salarios altos pagan las pensiones altas y los salarios bajos las pensiones bajas. A continuación, se muestra una configuración que se encuentra cerca de un punto crítico:

1. Duración del ciclo laboral: 30 años
2. Porcentaje del salario que se aporta (tasa-aporte) para la pensión: 9%.
3. Pocentaje del salario que recibe como pensión: 96% - 9% de aporte = 87%
4. Edad de muerte: 90 años.
5. Tasa porcentual de recuperación del capital (tasa-recuperación): 8%

Cualquier incremento o disminución de los anteriores valores, en el sentido en que sobrecarguen al fondo, conducen a la quiebra del mismo. Por ejemplo, disminuir la tasa de recuperación del capital a 7% o la tasa de aporte de los empleados al 8% enviaría el fondo a la quiebra. En muchos casos el patrono o el estado aportan una parte del ahorro

a los fondos de pensiones, lo que permitiría ubicar los puntos críticos en configuraciones más favorables para los empleados. Como veremos en la segunda versión del fondo, la mortalidad en la población de los pensionados no parece afectar negativamente al fondo. Los pensionados suelen morir con mayor frecuencia que los trabajadores activos: un pensionado que muere antes de la edad de muerte fijada es una pensión que el fondo no tendrá que seguir pagando. Esto puede compensar el efecto que tiene la muerte de trabajadores activos. Nuestro modelo de fondo de pensiones es completamente autosuficiente, ya que no recibe -como muchos otros fondos en la vida real- aportes solidarios del estado o de los patronos. Modelos similares al presente fondo de ahorro se pueden tratar matemáticamente mediante ecuaciones en diferencias finitas o con ecuaciones diferenciales. Este es el enfoque utilizado en la disciplina llamada *Dinámica de Sistemas* [2], en el que al programador sólo se les pide formular las relaciones básicas entre las variables y el software se encarga de traducir estas relaciones a ecuaciones diferenciales y aproximar las soluciones mediante métodos numéricos (método de Euler o de Runge-Kutta, por ejemplo). NetLogo posee una extensión que permite crear modelos según la Dinámica de Sistemas, pero no trataremos ese tema en este libro. Los lectores y lectoras interesadas en esta extensión pueden consultarla en la sección de extensiones del Manual del Usuario. A este respecto interesa destacar la sencillez del código del presente modelo basado en NetLogo y el hecho de que el mismo no realiza proyecciones aproximadas de la evolución del fondo (como en Dinámica de Sistemas) sino proyecciones exactas. Es el hecho de que cada agente posee sus propias variables con las que puede realizar sus cálculos, lo que permite programar con tanta sencillez y eficiencia algunos modelos basados en ABM, como el del presente ejemplo.

Modelo 3: Fondo para pensionados II.

Primitivas: mismas del ejemplo anterior.
Otros detalles: se fijan los salarios y la edad de muerte al azar. Se introduce deslizador para fijar la tasa de aumento salarial. Se eliminan los deslizadores de salario-único y edad-muerte.

Esta vesión del fondo de pensiones se ajustará más a la realidad en los siguientes tres puntos:
1. Se introduce mortalidad entre los trabajadores activos y los pensionados.
2. Se elimina el salario único y se cuenta con variedad de salarios.
3. Se introducen aumentos anuales a los salarios y las pensiones, debido a la inflación (deslizador tasa-aumento-salarial).

La población ahora no permanece constante y experimenta muy pequeñas fluctuaciones alrededor de un valor constante. La presencia de diversos salarios no afecta la estabilidad del fondo ni el hecho de que se haya introducido la mortalidad. El factor que introduce tensión en el fondo es el aumento salarial anual de trabajadores activos y de pensionados, producido principalmente por el factor inflacionario de las economías. Este hecho obliga a endurecer en alguna medida –la cual depende del porcentaje de aumento- las condiciones para que el fondo se sostenga. De acuerdo a la pauta observada en las economías de los distintos países, un aumento en la tasa inflacionaria

va aparejado con un aumento en las tasas de interés, lo cual permite al fondo compensar el aumento en el pago de las pensiones colocando el capital a un interés más alto en el mercado financiero. Para diversificar los salarios anuales (en dólares, por ejemplo) se emplea la primitiva “one-of” (uno-de), la cual selecciona al azar un ítem de una lista. Para asignar diferentes probabilidades a distintos ítemes se recurre al truco de repetir un ítem varias veces. Por ejemplo, en la orden “one-of [36000 60000 60000 60000 24000 24000 80000]” el ítem 60000 tiene mayor probabilidad de ser seleccionado que los demás, de hecho, la probabilidad es de 3/7, mientras que para 36000 y 80000 la probabilidad es de 1/7. Se utiliza también este truco para asignar distintas probabilidades a las edades de muerte de las personas. Estas modificaciones producen muy pocos cambios en el código.

```
globals[capital]

turtles-own[salario años edad-muerte]

to setup
  clear-all
    reset-ticks
end

to go
  create-turtles nuevas-personas
  [set años 25 set color red setxy random-xcor random-ycor
   set salario one-of [12000 36000 36000 3600 60000 60000
                       60000 100000]
    set edad-muerte one-of [35 40 60 60 70 70 70 80 80 80 80 90
                            90 90 90 90 100]
   ]
   ask turtles [set salario salario + salario * tasa-aumento-salarial / 100
    ifelse años < ciclo-laboral + 25 [contribuir ]
   [set color blue recibir-pensión]
  if años > edad-muerte [die]
 ]
  set capital capital + capital * tasa-depósito / 100
 if capital < 0 [show "Terminado"
   write "años: " write ticks  write " Capital: "
   print capital stop]
  tick
end

to contribuir
set capital capital + salario * tasa-aporte / 100
set años años + 1
end

to recibir-pensión
  set capital capital - salario * (porcentaje-pensión - tasa-aporte) / 100
  set años años + 1
end
```

Explicaciones y comentarios adicionales. Con los nuevos cambios tenemos un punto crítico con las siguientes condiciones: aporte 10 %, porcentaje-pensión 90% (menos el 9% de aporte) trabajadores reciben un 81%, ciclo laboral 31 años, mortalidad aleatoria, tasa recuperación del capital en un banco 10%, aumento salarial anual 3%.

Interacciones entre agentes I.

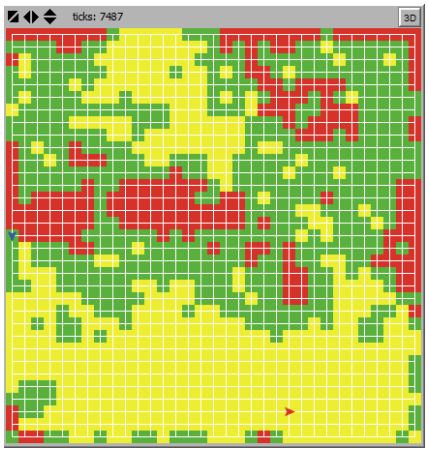
En los dos ejemplos anteriores sobre el tema de un fondo de pensiones se utilizan muchos agentes, sin embargo, hemos visto que los agentes no interactúan entre si. En la construcción de modelos basados en multitud de agentes, muchos modelos requieren que los agentes interactúen entre si de maneras muy diversas. Es de esperar que cualquier lenguaje de esta disciplina ofrezca una variedad de primitivas o mecanismos que faciliten estos tipos de interacciones. En el modelo "Tina y Magda visitan la ciudad" existe una interacción entre los dos agentes a través de la comparación de las variables que almacenan las coordenadas de las chicas y la variable "distancia", la cual reporta la distancia entre ambas. La consulta o comparación entre variables es uno de los mecanismos más frecuentemente utilizados para modelar interacciones entre agentes. En el siguiente modelo las tortugas interactúan con las parcelas modificando el valor de las variables de estas últimas.

Modelo 4: Contando visitas.

Primitivas: patches-own (parcelas-poseen), any? (alguna?), not (no), max, min, max-one-of (max-uno-de), with (con), of (de), count (contar), operadores <, >, >= (mayor o igual), beep (sonido bip), and (conjunción y).
Otros detalles: uso de condicionales, uso de stop para detener el programa, las tortugas hacen uso del derecho que tienen a modificar el valor de las variables de las parcelas sobre las que se encuentran.

En este modelo aprovecharemos la infraestructura de cuadrantes de la ciudad del modelo "Tina y Magda visitan la ciudad" donde las dos chicas recorrían sus calles siguiendo trayectorias aleatorias, pero esta vez no se trata de Tina y Magda sino de Pedro y Carmen, quienes trabajan para la Oficina de Salud de la ciudad.

La historia. Los agentes Pedro y Carmen, ambos de la Oficina de Salud de la ciudad, deben visitar las viviendas para atender una campaña de vacunación contra una epidemia. En cada cuadrante de la ciudad hay varios edificios de apartamentos en donde habitan las familias que Pedro y Carmen deben visitar. Después de vacunar a los moradores de una vivienda en un cuadrante, a los agentes les gustaría agotar las visitas a las restantes viviendas del edificio y del cuadrante, pero lamentablemente esto no es posible debido a los desiguales horarios de trabajo de sus moradores y entonces Pedro y Carmen proceden a visitar viviendas en los edificios vecinos, en el entendido de que



deberán regresar a los cuadrantes donde quedaron viviendas sin visitar. Debido a esto, la trayectoria de visitas de Carmen y Pedro se asemeja más una trayectoria aleatoria que a una trayectoria planificada. Otra consecuencia de este hecho y del hecho que no todos los cuadrantes tienen el mismo número de apartamentos es que al terminar la campaña de vacunación, el número de visitas que cada cuadrante recibió es bastante desigual: habrá cuadrantes que recibieron muchas visitas, mientras otros recibieron muy pocas. La Oficina de Salud exige que cada agente lleve un control del número de visitas que ha realizado a cada cuadrante. La campaña termina cuando no queda ningún cuadrante que no haya recibido al menos una visita por parte de alguno de los agentes. En ese momento los agentes llevan los resultados de las visitas hechas a la gerencia de la Oficina de Salud, en donde se confecciona un mapa en tres colores: los cuadrantes que recibieron entre 1 y 11 visitas (entre los dos agentes) se pintan de amarillo, aquellos que recibieron entre 12 y 20 visitas se pintan de verde y los que recibieron más de 20 visitas se pintan de rojo. La figura muestra el mapa que ha resultado de una corrida del modelo (la distribución de colores puede variar mucho de una corrida a otra). Cuando el modelo se corre, vemos a los agentes visitar los cuadrantes de la ciudad, los cuales cambian de colores según el número de visitas recibidas va aumentando, hasta que el proceso se detiene cuando no quedan cuadrantes sin visitar. Una vez que el modelo se ha detenido es posible realizar consultas desde la Ventana del Observador.

Preparativos: configurar el mundo con la topología del cuadrado. Plantar los botones setup y go, éste último con la casilla “Continuamente” marcada.

Plan general y problemas a resolver. Hay que llevar la contabilidad de las visitas que han hecho Pedro y Carmen a cada cuadrante de la ciudad. Para ello cada parcela representará un cuadrante de la ciudad y serán los cuadrantes (las parcelas) los que se encargarán de llevar la contabilidad de las visitas recibidas por parte de cada agente. Esto se logra asignando dos variables a cada cuadrante con la primitiva patches-own (parcelas-poseen) llamadas “visitasP” y “visitasC” que cuentan el número de visitas que han hecho a la parcela Pedro y Carmen respectivamente. La clave de la lógica del modelo está en la expresión:

si eres Pedro (si tu número who es 0) aumenta en una unidad la variable visitasP de la parcela en que te encuentras y si no eres Pedro (y por tanto eres Carmen) entonces aumenta en uno la variable visitasC,

la cual se traduce a código como: ask turtles [ifelse who = 0 [set visitasP visitasP + 1] [set visitasC visitasC + 1].

Nótese que en esta orden se le pide a cada tortuga que aumente en una unidad el valor de la variable visitasP o visitasC -según que sea su número who sea 0 o no lo sea- las

cuales son variables que pertenecen a las parcelas y no a las tortugas. Aquí estamos en presencia de la facultad que se otorga a las tortugas de tratar las variables de la parcela sobre la que se encuentran como si fueran sus propias variables. A contínuación el código:

```
patches-own[visitasP visitasC]
;; variables que almacenan el número de visitas
;; de la tortuga 0 (Pedro) y la tortuga 1 (Carmen) respectivamente

to setup
clear-all
crt 2 [set heading 0]
cuadrantes-ciudad ;; se dibujan los cuadrantes de la ciudad
ask turtle 0 [pu set xcor -15  set ycor -15 set color blue]
ask turtle 1 [pu set color red]
reset-ticks
end

to go
ask turtles [ifelse who = 0 [set visitasP visitasP + 1 ]
[set visitasC visitasC + 1 ]
fd 1 rt one-of [90 -90 0]]
ask patches [if visitasP + visitasC > 0
and visitasP + visitasC < 12 [set pcolor yellow]
if visitasP + visitasC >= 12 and visitasP + visitasC < 21
[set pcolor green]
if visitasP + visitasC >= 21 [set pcolor red]]
if not any? patches with [pcolor = gray] [beep stop]
wait 0.01 ;; para desacelerar el movimiento de las tortugas
tick
end

to cuadrantes-ciudad ;; este procedimiento dibuja los cuadrantes de la ciudad
ask patches [set pcolor gray]
ask turtle 0 [set color white pu set heading 0
set xcor -16 set ycor -16 pd
repeat 17 [fd 33 rt 90 fd 1
rt 90 fd 33 lt 90 fd 1 lt 90 fd 33] bk 33 lt 90
repeat 17 [fd 32 rt 90 fd 1
rt 90 fd 33 lt 90 fd 1 lt 90 fd 33] ]
end
```

Comentarios y explicaciones adicionales. Los agentes inician las visistas desde distintos puntos de la ciudad. Cada parcela lleva el control de las visitas de ambos agentes mediante el mecanismo explicado anteriormente. Cuando ya no quedan parcelas sin recibir al menos una visita, es decir, cuando no queden parcelas de color gris: “if not any? patches with [pcolor = gray] [beep stop]”, entonces el procedimiento produce un sonido “bip” y se detiene. La expresión “if not any?” debe enenderse como “si no alguna” o sea “si no queda alguna”, que usando la tan familiar doble negación de nuestro idioma se podría también traducir como “si no queda ninguna”. Una vez corrido el modelo, es posible hacer varios tipos de consultas sobre las visitas en la Ventana del

Observador, como la media, la desviación estándar o la variancia de las visitas hechas a cada cuadrante por cada agente, con las primitivas “mean”, “standard-deviation” y “variance”, entre otras consultas. Veamos algunas de estas consultas utilizando los datos obtenidos de una corrida del modelo:

print count patches
==> 1089, la totalidad de parcelas del mundo (por omisión).
print (count patches with [pcolor = yellow]
==> 452, número de cuadrantes de color amarillo.
print (count patches with [pcolor = yellow] / count patches) * 100
==> 41.5%, porcentaje de cuadrantes amarillos.
print mean [visitasP] of patches with [pcolor = green]
==> 8.043, muestra promedio de visitas hechas por el agente 0 (Pedro) a los cuadrantes verdes.
print standard-deviation [visitasP] of patches with [pcolor = green]
==> 3.411, es la desviación estandar de visitas hechas por Pedro a los cuadrantes verdes.
print max [visitasP + visitasC] of patches
==> 69, número máximo de visitas recibidas por algún o algunos cuadrantes, es decir, hubo al menos un cuadrante que se visitó 69 veces. Podemos averiguar cuál es uno de esos cuadrantes.
print max-one-of patches [visitasP + visitasC]
==> (patch 16 16), la parcela 16 16 recibió el máximo de visitas (69). Si había más de una parcela que recibió 69 visitas, el sistema habrá elegido una al azar.
La orden “max-one-of agentes [variable]” reporta alguno de los agentes con valor máximo de la variable encerrada entre corchetes.
print count patches with [visitasP + visitasC = 69]
==> 1, sólo hay un cuadrante con 69 visitas.
print count patches with [visitasP + visitasC = 1]
==> 3, en 3 cuadrantes se vacunó a todos sus moradores en una sola visita. (Tal vez había pocas viviendas en ese cuadrante o la visita se hizo en ocasión del final, del mundial de fútbol o el día del Super Bowl).
print ticks
==> 8806, duración en ticks de la simulación o número de visitas hechas por cada agente.

Ejercicio 4.2

1- Si se corre el modelo varias veces, ¿por qué es frecuente que los primeros cuadrantes verdes o rojos aparezcan en los bordes o las esquinas?
2- ¿Por qué el promedio de visitasP es igual al de visitasC, pero la desviación típica no?
3- Calcular la moda de visitasC. Este valor sería el número de visitas de Carmen a un cuadrante que se repitió más veces durante el proceso.
4- Los agentes Carmen y Pedro son tratados sobre una base de igualdad en el código. Cualquier diferencia en la forma en que han desempeñado su tarea es debida exclusivamente al azar y se nivelaría a lo largo de varias corridas. Sin embargo, en una sola corrida podrían detectarse algunas diferencias. Pensando como gerente de la Oficina de Salud, ¿sería posible elaborar un criterio que permita establecer cuál de los

dos agentes ha sido más eficiente en su trabajo, mirando el mapa y las consultas de una sola corrida?

Interacciones basadas en proximidad espacial.

Existe un tipo de interacción que ocurre con frecuencia entre los agentes de un modelo, en donde el espacio físico juega un papel importante en la interacción. Puede tratarse del espacio físico real (bidimensional o tridimensional) o un espacio abstracto el cual se representa por medio de las parcelas. En este tipo de interacciones la proximidad o cercanía espacial entre agentes, lo mismo que su posición u orientación relativa puede ser un factor importante de la interación. Por ejemplo, dos agentes podrían reaccionar si uno se encuentra dentro de un vecindario redondo o cónico de cierto tamaño de otro agente o si cada uno encara al otro. NetLogo tiene una variedad de primitivas que permiten modelar interacciones de proximidad espacial de muchas maneras posibles: detectando los agentes que se encuentran por delante, a la derecha o izquierda o incluso en una cierta dirección y a determinada distancia de un agente dado. Todo esto puede ser de utilidad para modelar fenómenos físicos como la conducta de las partículas en campos de atracción-repulsión o en universos virtuales, donde el espacio podría no ser homogéneo o isotrópico[13], o en donde las interacciones podrían presentar una asimetría local: por ejemplo, un agente podría reaccionar de modo distinto cuando detecta a otro agente a su izquierda que cuando lo detecta a su derecha, como es el caso en el modelo siguiente. En los próximos modelos haremos uso de las primitivas: “any?” (alguna?), “turtles-on” (tortugas-sobre), “patch-ahead” (parcela-adelante) y “patch-left-and-ahead” (parcela-adelante-a-la-izquierda) para modelar interacciones. La primitiva “turtles-on” reporta el conjunto-agentes de las tortugas que se encuentren sobre la parcela que se indique como entrada. Por ejemplo “turtles-on patch 1 4” reporta el conjunto-agentes de las tortugas que se encuentran sobre la parcela de coordenadas (1, 4). La primitiva “patch-ahead num” reporta la parcela que se encuentra “num” pasos delante del agente que emite la orden. Estas tres primitivas se pueden combinar en expresiones como, por ejemplo:

Ifelse any? turtles-on patch-ahead 2 [órdenes si verdadero] [órdenes si falso]

la cual se traduce al español como: si hay algunas tortugas sobre la parcela 2 pasos delante de mí [órdenes si hay tortugas] [órdenes si no las hay].

Antes de exponer el siguiente modelo presentamos una lista de primitivas de NetLogo que se pueden usar para modelar interacciones de proximidad espacial, las cuales se han tomado del Diccionario de Primitivas:

distance (distancia), downhill (cuesta-abajo), downhill4 (cuesta-abajo4), uphill (cuesta-arriba), uphill4 (cuesta-arriba4), face (mirar-hacia), facexy (mirar-haciaxy), in-cone (dentro-del-cono) move-to (moverse-a), patch-ahead (parcela-adelante), patch-at

[13] Es decir, un espacio vacío en donde existieran lugares o direcciones preferenciales.

(parcela-en), patch-at-heading-and-distance (parcela-hacia-dirección-y-distancia), patch-here (parcela-aquí), patch-left-and-ahead (parcela-a-izquierda-y-adelante), patch-right-and-ahead (parcela-a-derecha-y-adelante), towards (hacia), towardsxy, (haciaxy), neighbors (vecinos), neighbors4 (vecinos4).
No se han incluido en esta lista las primitivas relacionadas con enlaces, muchas de las cuales son útiles para modelar interacciones por medio de enlaces.

Modelo 5: Alfiones y betiones.

Primitivas: patch-left-and-ahead (parcela-izquierda-y-adelante), any? (alguna?), patch-ahead (parcela-adelante), <breeds>-on (familias-sobre).
Otros detalles: se crean dos familias de tortugas para representar a las partículas.

La historia. En este modelo habrá dos tipos de partículas elementales imaginarias, que llamaremos “alfiones” y “betiones”, que interactúan entre sí de acuerdo a ciertas leyes basadas en proximidad espacial. Los alfiones responden a leyes de interacción asimétrica pues reaccionan solamente cuando detectan partículas adelante y a su izquierda. Los betiones tienen una conducta mixta pues reaccionan de modo asimétrico antes los alfiones, pero de modo simétrico ante las partículas de su mismo tipo. La reacción de cada una dependerá de si la partícula detectada es de su misma especie o no y si se encuentra delante y a su izquierda o no.

Plan general y problemas a resolver. Puesto que las partículas deben reaccionar de modo diferente, definiremos dos familias de agentes, una para cada tipo de partícula. Las partículas se distinguirán visualmente por su color: los alfiones serán amarillos y los betiones rojos. Para modelar las interacciones entre las partículas se utilizarán las primitivas “turtles-on”, “patch-ahead num” y “patch-left-and-ahead”. Para las interacciones de tipo asimétrico descritas en la historia se utilizará la primitiva “patch-left-and-ahead” (parcela-izquierda-y-enfrente), la cual opera con dos entradas que son el ángulo a la izquierda y la distancia hacia adelante medida en pasos. En el ejemplo esta primitiva se usará con las entradas “patch-left-and-ahead 90 1”, la cual reportará la parcela que se encuentra a una distancia de un paso y 90 grados a la izquierda del agente que emite la orden. A cada partícula se le pregunta si hay partículas a un paso de distancia y a su izquierda. En el caso de los alfiones, por ejemplo, la pregunta toma la forma: ifelse any? alfiones-on patch-left-and-ahead 90 1, cuya traducción al español sería: si hay algunos alfiones en al parcela 90 grados a la izquierda y a un paso de distancia. La orden completa consiste entonces en dos condicionales, un condicional if encapsulado dentro de un condicional ifelse:

```
ask alfiones [ifelse any? alfiones-on patch-left-and-ahead 90 1 [left 90 fd 1]
[if any? betiones-on patch-left-and-ahead 90 1 [rt 90 fd 1] fd 1]
```

La orden para los betiones tiene una estructura similar, aunque con alguna diferencia en las acciones que se pide realizar a las partículas.

Preparativos: Plantar botones set-up y go y deslizadores para las variables número-de-alfiones y número-de-betiones. Al botón go se le marca la casilla “Continuamente”. El código es el siguiente:

```
breed [alfiones alfión]
breed [betiones betión]

to setup
clear-all
create-alfiones número-de-alfiones [setxy random-xcor random-ycor set color
yellow]
create-betiones número-de-betiones [setxy random-xcor random-ycor set color red]
reset-ticks
end

to go
ask alfiones [ifelse any? alfiones-on patch-left-and-ahead 90 2
[left 90 fd 1] [if any? betiones-on patch-left-and-ahead 90 1
[rt 90 fd 1] fd 1]
 ]
ask betiones [ifelse any? alfiones-on patch-left-and-ahead 90 2
[lt 90 fd 1] [if any? betiones-on patch-ahead 1 [bk 1] fd 1]
 ]
wait 0.1
end
```

Explicaciones y comentarios adicionales. Para girar órdenes a los alfiones que sean ignoradas por los betiones o viceversa, se han creado dos familias de partículas llamadas alfiones y betiones. Emitir órdenes que segregan a ciertos conjuntos de agentes también se podría hacer apelando a alguna característica distintiva de los agentes (por ejemplo, el color), como podría ser “ask turtles with [color = red]”. Obsérvese cómo, una vez creada una familia, automáticamente se incorporan al lenguaje primitivas que hacen referencia a las familias por su nombre como, por ejemplo, “create alfiones”, “alfiones-on” o “ask alfión 0”. Las variables globales “número-de-alfiones” y “número-de-betiones” se definen mediante sendos deslizadores en la interfaz. Como los alfiones son creados antes que los betiones, se les asignan los primeros números who de la lista.

Sugerencias para exploración. Note la diferencia en la conducta de ambos tipos de partículas. Obsérvese y trate de explicar la conducta de los betiones, los cuales se detienen y permanecen inmóviles por lapsos relativamente largos de tiempo. El modelo también se presta muy bien para observar los cambios de conducta que se producen cuando se varía alguna de las entradas de la primitiva “patch-left-and-ahead num num”. También se puede experimentar variando algunas de las órdenes dentro de los corchetes de las expresiones condicionales. Para apreciar la trayectoria típica de los alfiones, se puede pedir a un alfión que baje la pluma:

```
ask one-of alfiones [pd].
```

Modelo 6: El nucleón del diablo.

Primitivas: patch-left-and-ahead, any?, shape (forma), dot (punto grueso), distancexy.

La historia. En esta extensión del modelo sobre las partículas elementales imaginarias, la novedad consiste en la introducción de una partícula central, también conocida como "el nucléon del diablo" (muy diferente al bosón de Higgs, llamado "la partícula de Dios"). El nucléon del diablo parece ser algo impredecible, a veces atrae y otras veces repele a las demás partículas, y su color es blanco.

```
breed [alfiones alfión]
breed [betiones betión]

to setup
clear-all
crt 1 [set color white] ;; este es el nucleón
create-alfiones número-de-alfiones [setxy random-xcor
random-ycor set color yellow]
create-betiones número-de-betiones [setxy random-xcor
random-ycor set color red]
ask turtles [set shape "dot"]
ask turtle 1  [pu] ;; por si se desea ver la trayectoria
;; de un alfión, cambiar pd por pu
reset-ticks
end

to go
ask alfiones [if distancexy 0 0 < 6 [rt 180 ]
ifelse any? alfiones-on  patch-left-and-ahead 90 1
[left 90 fd 1] [if any? betiones-on patch-left-and-ahead 90 1
[rt 90 fd 1] fd 1]
  ]
ask betiones [if distancexy 0 0 < 6 [rt 180 ]
ifelse any? alfiones-on  patch-left-and-ahead 1 90
[lt 90 fd 1] [if any? betiones-on patch-ahead 1 [fd 1] fd 1]
]
wait 0.1
end
```

Explicaciones y comentarios adicionales. Este código difiere del anterior básicamente en un par de órdenes nuevas que se le han agregado. Puesto que el nucléon se encuentra en la parcela (0, 0), resulta claro que la orden que prohibe el acercamiento de las partículas al nucleón es "if distancexy 0 0 < 6 [rt 180]". Si se corre el modelo se podría notar que ocurre algo interesante: algunas partículas no son rechazadas por el nucleón y permanecen atrapadas cerca del mismo, en continuo movimiento vibratorio. Después de un tiempo algunas de estas partículas logran escapar y unirse al conjunto de las partículas externas. También ocurre que otras que se encuentran lejos del nucleón se acercan demasiado y quedan atrapadas. En principio podría parecer que esta conducta viola lo que el código indica, pero si se analiza con atención la dinámica de las partículas

se descubre el por qué de este comportamiento. Sin que hubiese sido el propósito inicial del modelo, el mismo nos ha presentado un interesante comportamiento emergente y una interesante e inesperada analogía con la conducta de los electrones de las órbitas externas de los átomos. Estos electrones normalmente se encuentran atrapados en órbitas alrededor del núcleo, pero eventualmente pueden abandonarlas cuando colisionan con electrones libres o cuando son capturados por un átomo vecino, el cual tiene preferencia por captar antes que por ceder electrones. El punto importante a resaltar aquí es el hecho que el ejemplo nos puede hacer conscientes de que pequeños cambios en la conducta de los agentes pueden producir notables cambios a nivel macro. Invitamos a lectores y lectoras a experimentar variando la distancia de rechazo del nucleón (la cual no necesariamente debe ser la misma para ambas partículas) o permitiendo que sólo las partículas de un tipo sean rechazadas. Sugerimos cambiar el ángulo de la instrucción "rt 180" por un valor de 90 y observar el cambio en la conducta de las partículas. El modelo se presta para explorar resultados variando los parámetros o cambiando las órdenes.

Ejercicio 4.3: Explicar por qué algunas partículas quedan atrapadas vibrando cerca del nucleón del diablo y por qué después de un tiempo algunas logran escapar.

Modelo 7: Concierto de rock en Whoolsock I.

Primitivas: reset-ticks, tick, ifelse, any? (alguna?), turtles-on (tortugas-sobre), patch-ahead (parcela-adelante), random-xcor (azar-xcor), random-ycor.
Otros detalles: la variable global "fans" se define mediante un deslizador en la interfaz.

La historia. En una granja llamada Whoolsock de una gran ciudad, tendrá lugar un gran concierto de rock por la famosa banda de los *Stunning Rolls*. Como el acceso al área es por el lado oeste y la tarima donde tocará la banda se encuentra en el lado este, la muchedumbre se desplaza por el descampado de oeste a este. Debido a la gran cantidad de gente, las personas caminan cambiando una y otra vez de dirección para evitar colisionar entre ellas. Cuando la multitud escucha por los altavoces a *Jick Magger*, el cantante de la banda, afinando su guitarra, el griterío y el júbilo son enormes. El número de personas se fija en el deslizador "fans" de la interfaz. Cada vez que se corre el procedimiento go, cada tortuga (persona) pregunta si en la parcela delante de ella (patch-ahead) hay alguna otra tortuga. Si la respuesta es negativa la persona avanza un paso hacia adelante, pero si la respuesta es afirmativa, para evitar la colisión, la persona toma una nueva orientación dada al azar (random 180) y vuelve a intentar avanzar formulando la misma pregunta.

Plan general y problemas a resolver. Hay que modelar la caminata de las tortugas sin colisionar con otras tortugas. Las tortugas se desplazarán de oeste a este, aunque su trayectoria difícilmente podría ser en línea recta. Para evitar las colisiones se empleará una expresión condicional en que cada tortuga pregunta si en la parcela un paso delante de ella hay otras tortugas. En caso afirmativo la tortuga elige una nueva orientación al

azar y vuelve a preguntar. Sólo cuando la respuesta sea negativa la tortuga avanzará 1 paso. Esta conducta queda resumida en la orden:
ask turtles [ifelse any? turtles-on patch-ahead 1 [set heading random 180] [fd 1]], que se traduce como: pedir a las tortugas [si hay algunas tortugas en la parcela 1 paso adelante [fijar orientación azar 180] [avanzar 1 paso]]

Preparativos: Plantar botones set-up y go y un deslizador llamado "fans" (abrev. de fanáticos o seguidores), el cual determina el número de personas que asisten al concierto. Marcar la casilla "Continuamente" del botón go. He aquí el código:

```
to setup
clear-all
crt fans [setxy random-xcor random-ycor set color yellow]
ask turtle 0 [set color red]
reset-ticks
end

to go
ask turtles [ifelse any? turtles-on patch-ahead 1
[set heading random 180]
[ fd 1]
          ]
wait 0.1
tick
end
```

Explicaciones y comentarios adicionales. Las posiciones iniciales de las tortugas se fijan al azar. La razón para haber incluido la variable ticks en el procedimiento go es para contar el número de veces que el intérprete pasa por "go" y poder tener una estimación del tiempo transcurrido, en caso de que deseáramos realizar algunas mediciones. Esto nos será más útil en la segunda versión del modelo. La tortuga 0 se pintó de rojo para poder apreciar la trayectoria de una tortuga típica.

Ejercicio 4.4 Hacer un pequeño cambio en el código para que las personas se desplacen de norte a sur.

Modelo 8: Concierto de rock en Whoolsock II.

Primitivas: No hay primitivas nuevas.
Otros detalles: Se crean dos variables globales para medir y comparar la longitud de los trayectos rectos de una tortuga típica.

En este ejemplo añadimos la siguiente novedad al ejemplo anterior: vamos a estudiar la longitud de los tramos rectos que caminan las personas conforme se desplazan por el descampado. Para ello seguiremos la caminata de una tortuga representativa del conjunto, en la suposición de que todas las tortugas muestran un comportamiento "estadísticamente equivalente". Le pediremos a la tortuga representativa (la tortuga 0)

que cada vez que termina de realizar un tramo recto, anote su longitud medida en pasos en una variable llamada “EsteTramo”. En esta variable no se anotarán todos los tramos rectos, sino sólo el último que la tortuga ha realizado, es decir, cada vez que la tortuga inicia un nuevo tramo recto borra la longitud del que tenía guardado en la variable y lo reemplaza por la longitud del tramo que acaba de terminar. Además, le pediremos que en otra variable llamada “TramoMax” anote el valor del mayor de los tramos rectos que ha realizado hasta ese momento y lo imprima en la Terminal de Instrucciones. El modelo debe ser detenido manualmente por el usuario oprimiendo el botón go.

Plan general y problemas a resolver. En esta nueva versión del modelo, el problema que hay que resolver se encuentra en la forma cómo la tortuga 0 manipula las dos variables EsteTramo y TramoMax. Describimos la solución mostrando lo que la tortuga 0 debe hacer con estas dos variables cuando se encuentra en medio de la caminata, después de haber realizado varios tramos rectos. Mientras la tortuga 0 se encuentra realizando un tramo recto, la variable EsteTramo debe tener un valor igual a la longitud del tramo que la tortuga 0 se encuentra construyendo y ella -la tortuga 0- procedería de la siguiente manera:

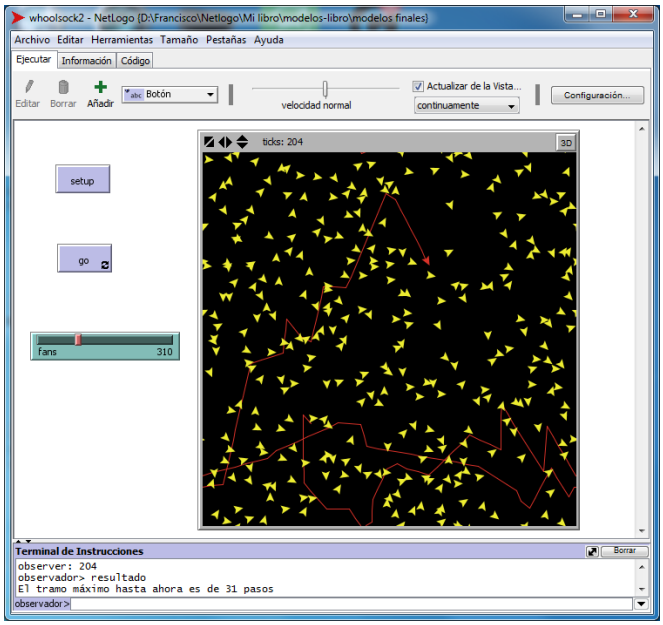
1. Si logra dar un nuevo paso hacia adelante entonces debe aumentar en una unidad el valor de la variable EsteTramo: fd 1 set EsteTramo EsteTramo + 1.
2. Si no puede seguir avanzando porque hay tortugas en la parcela frente a ella, antes de cambiar de dirección para iniciar un nuevo tramo recto debe comparar el valor de EsteTramo con TramoMax. Si EsteTramo es mayor que TramoMax, reemplazará TramoMax por el valor de EsteTramo. En código:
   if EsteTramo > TramoMax [set TramoMax EsteTramo]
3. Antes de iniciar el nuevo tramo recto debe asignar el valor 0 a la variable EsteTramo y probar una nueva dirección de avance:
   set EsteTramo 0 set heading random 180.

Las tres anteriores expresiones se encuentran inmersas dentro del código y los lectores las podrán identificar sin problema. La comparación de las variables indicada en el punto 2 se realiza en un procedimiento seprarado llamado “comparar”.

Preparativos: Plantar botones set-up y go (marcar casilla “Continuamente) y un deslizador llamado “fans” en el que se determina el valor de la variable global de ese mismo nombre. Mantener el mundo con la topología de la rosquilla. He aquí el código:

```
globals[TramoMax EsteTramo]
to setup
clear-all
crt fans [setxy random-xcor random-ycor set color yellow]
ask turtle 0 [set color red pd]
reset-ticks
end

to go
ask turtle 0 [ifelse any? turtles-on patch-ahead 1
[comparar set EsteTramo 0 set heading random 180]
[ fd 1 set EsteTramo EsteTramo + 1 ]
ask other turtles [ifelse any? turtles-on patch-ahead 1
[set heading random 180][fd 1]]
```

```
]
wait 0.1
tick
end


to comparar
if EsteTramo > TramoMax [set TramoMax EsteTramo
show TramoMax]
end

to resultado
type "El tramo máximo hasta ahora es de" type " " type TramoMax type " pasos"
end
```



La figura muestra en rojo la trayectoria que ha seguido la tortuga 0 luego de varias pasadas por "go".

Explicaciones y comentarios adicionales. La orden en la que las tortugas preguntan sobre la presencia de otras tortugas en la parcela de enfrente produciría un problema si el modelo se corriera con la topología del cuadrado: si una tortuga llega a la frontera del mundo apuntando en la dirección en que choca contra dicha frontera, la pregunta "¿hay tortugas en la parcela delante de mí?" carecería de sentido pues no habría ninguna parcela delante de la tortuga. Para evitar el error que esta situación generaría, el modelo debe correrse con la topología de la rosquilla.

Ejercicio 4.3. Modificar el código de manera que el modelo se pueda correr con la topología del cuadrado.

Ejercicio 4.5. Explicar el siguiente fenómeno: si se corre el programa varias veces, especialmente con pocas tortugas, se notará que algunas de ellas cambian de dirección "espontáneamente" sin que haya ninguna tortuga delante con la cual puedan colisionar.

Esta, en apariencia, extraña conducta es más fácil de notar si se corre el modelo con una sola tortuga, la cual, por ser la única habitante del mundo, no tendría motivo para cambiar de dirección, ya que no podría colisionar con ninguna otra tortuga. No obstante, si se corre el modelo varias veces, se observará que estos cambios de dirección que podríamos llamar "espontáneos", siguen ocurriendo. Desde el punto de vista de la coherencia con la historia, podemos pensar que las tortugas cambian de dirección porque decidieron esquivar un charco o una roca en el terreno. Invitamos a lectores y lectoras a encontrar la razón computacional de este comportamiento. Este es un ejemplo de las cosas que a veces no se anticipan mientras se escribe el código de un modelo, pero que se descubren cuando el modelo se corre e invitan al análisis.

Ejercicio 4.6. Si se empleara una lista para almacenar las longitudes de cada tramo recto en una variable llamada ListaDeTramos, no sería necesario definir la variable TramoMax pues se podría obtener este valor mediante la primitiva "max", la cual reporta el valor máximo de una lista. Modificar el código aplicando esta idea.

Promenade I.

La programación pone la mente en movimiento como pocas actividades lo hacen. En cuanto concluimos la construcción de un modelo o bien durante el proceso de construcción, con gran facilidad nos vemos naturalmente atraídos a considerar aspectos relacionados con el modelo mismo o incluso cuestiones en otros campos del conocimiento, sugeridas por el modelo. Cuando construimos el modelo Concierto en Whoolsock I, la idea original era muy simple: observar la conducta de una población de tortugas que se desplazan hacia una dirección cardinal evitando las colisiones entre sus miembros y modelar esta conducta con los agentes en NetLogo. Pero en cuanto se observa el modelo funcionar, de inmediato comienzan a surgir ideas y preguntas. ¿Cómo se vería la trayectoria típica de las tortugas? La respuesta no requiere otra acción que pedir a las tortugas que caminan con la pluma abajo. Cuando observamos la maraña de líneas que se produce, pensamos que sería mejor observar la trayectoria de una sola tortuga, pero ¿cuál de ellas? ¿Podemos escoger una cualquiera? ¿Son todas las trayectorias "estadísticamente equivalentes"? ¿Qué significa decir que las trayectorias son "estadísticamente equivalentes"? Y de pronto nos vemos haciendo preguntas que pertenecen al campo de la estadística. Corremos el modelo varias veces para observar la trayectoria de la tortuga seleccionada (la tortuga 0). Hacemos algunas pruebas con el color del trazo y al final nos decidimos por el blanco o el rojo, que resaltan bien sobre el fondo negro. Entonces es casi inevitable que surja la pregunta acerca de cuál sería la longitud del tramo recto de mayor longitud en la trayectoria que va dibujando la tortuga 0. Parece claro que entre más tiempo se corra el modelo mayor será la longitud del mayor tramo recto logrado. Si almacenamos las longitudes de un buen número de tramos rectos, ¿cuál sería entonces el promedio y la desviación típica de estas longitudes? ¿Cómo varía este promedio con respecto al tiempo o al número de tortugas? ¿Se estabiliza el promedio o la desviación típica? Parece claro que a menor número de tortugas se esperarían tramos rectos de mayor longitud, pero ¿podríamos hacer alguna estimación cuantitativa? Lo cierto es que cualquier persona que construya modelos en NetLogo o en cualquier otro ambiente experimentará este tipo de dinámica mental. Más adelante, después de haber expuesto varios modelos, volveremos sobre

este tema y el potencial que tiene en educación. No podemos dejar de señalar que la creación de modelos tiene también un enorme potencial en el campo de la expresión artística y el gozo creativo, similar al de actividades como pintar, tocar un instrumento, escribir o componer música. A diferencia de los lenguajes de programación de tipo general, en NetLogo se cuentan desde el inicio con un conjunto de elementos preinstalados que invitan a los usuarios a crear modelos, ya se trate de modelos que simulan aspectos de la realidad o modelos que expresan mundos virtuales o de fantasía en los cuales los agentes son los protagonistas principales. Pero esto no es mera casualidad: si nos fijamos con algún detenimiento, veremos que el mundo está hecho de agregados, de pluralidades que nos permiten encontrar ejemplos de agentes por todas partes: los empleados de una oficina, los votantes para alcalde de una ciudad, las moléculas de un cristal, las partículas elementales de la materia, las páginas Web de Internet, los depredadores y sus presas en un bosque, las hormigas en un hormiguero, los autos en un sistema vial, los genes en el DNA, las tiendas de un centro comercial, las mercancías de un almacén, las cuentas de los clientes de un banco, los libros en tu biblioteca personal, etcétera.

La programación y más específicamente la programación de modelos, actúa como un magneto que tiene la virtud de atraer ideas en los más diversos campos del conocimiento y la actividad humanas.

Capítulo 5: Modelos II

En este capítulo presentamos una segunda colección de modelos un poco más elaborados que los del capítulo precedente. Aún quedan primitivas importantes y aspectos del lenguaje que deseamos introducir para armarnos con un equipamiento que podríamos llamar "básico", que nos permita abordar la construcción de programas y modelos en un rango bastante amplio. Sin embargo, ya en los ejemplos del capítulo anterior hemos comenzado a notar que los obstáculos más importantes que un modelo plantea no serán los relativos al conocimiento y manejo del lenguaje de programación sino aquellos problemas que el modelo mismo plantea. Los ejemplos siguientes no sólo muestran el empleo de nuevas primitivas o recursos de la interfaz sino también algunas técnicas de programación basadas en los recursos previamente introducidos del lenguaje NetLogo, mediante las cuales es posible resolver los problemas que presentan los modelos. Estos problemas son intrínsecos al modelo y el diseño de estrategias de solución es en gran medida –aunque como veremos no en su totalidad- ajeno al lenguaje en que se programa el modelo. Una vez formulados los planes de ataque a los problemas que el modelo plantea, los mismos han de ser traducidos a código de NetLogo. Debido a que la variedad de problemas que pueden presentar los modelos es enorme, podríamos decir que inagotable, es prácticamente imposible cubrirlos mediante una lista de recetas *ad hoc*. El ingenio será una herramienta que no podremos guardar bajo llave. Igual que ocurre en el aprendizaje de cualquier materia, en la programación también se manifiestan los estilos individuales de aprendizaje y de resolución de problemas de las personas. Hay quienes comienzan a programar un modelo escribiendo los primeros trozos de código que llegan a sus mentes. Hay quienes prefieren hacer un planeamiento general previo, y si no de todo el modelo, al menos del núcleo central o de algunos de sus módulos. Es una buena práctica dedicar algún tiempo a concebir un plan general de ataque del modelo antes de lanzarse impulsivamente a escribir código. De cualquier manera, en programas o modelos de cierta complejidad, es muy difícil tener al inicio un plan general finalmente detallado y habrá ocasiones en que acudiremos al viejo recurso de la prueba y el error. El matemático húngaro George Pólya (1887-1985) en su famoso libro "Cómo resolver problemas" [12] enuncia una serie de recomendaciones útiles para resolver problemas de matemática. Una de esas recomendaciones, que consideramos muy aplicable a la programación, consiste en tratar versiones simplificadas de un problema complejo y difícil. Por ejemplo, antes de verificar si es correcto el código que ordena a muchos agentes comportarse e interactuar de cierta manera, probar una versión con pocos agentes y utilizando sólo las variables que son indispensables podría ser de gran ayuda para verificar si las interacciones se comportan del modo esperado.

Ejemplo 26: Diferencia entre las primitivas self y myself.

Presentamos dos primitivas de NetLogo parecida y muy útiles en la construcción de expresiones: "self" y "myself". En el modelo siguiente sólo utilizaremos "myself" pero aprovecharemos la oportunidad para resaltar la diferencia entre las dos. La primitiva "self" se podría traducir por "yo mismo", mientras que "myself" quizás es mejor traducirla como "a mí". Sin embargo, esto no puede tomarse como una regla rígida. Los

siguientes ejemplos ayudarán a esclarecer la diferencia entre ambas primitivas. Suponiendo que la tortuga 0 y la tortuga 1 se encuentran separadas por una distancia de 2.87 pasos, exploramos el resultado de las siguientes órdenes:

ask turtle 0 [ask turtle 1 [show distance myself]]
==> (turtle 1): 2.87, la tortuga 0 le pide a la tortuga 1 que reporte la distancia entre ella (tortuga 1) y quien le ha dado la orden (tortuga 0).
ask turtle 0 [ask turtle 1 [show distance self]]
==> (turtle 1): 0, la tortuga 0 le pide a la tortuga 1 que reporte la distancia entre ella (tortuga 1) y sí misma (tortuga 1), la cual obviamente es igual a 0.

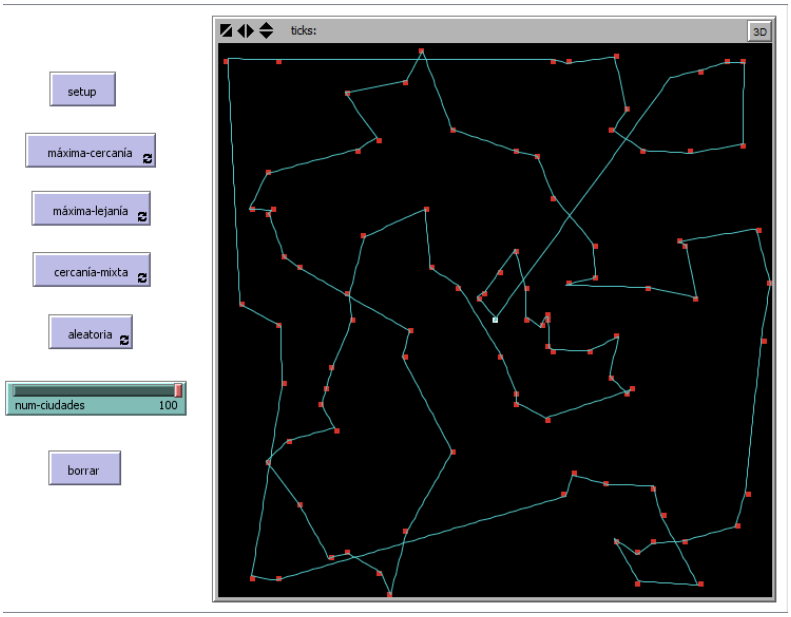
Modelo 9: El vendedor viajante.

Primitivas: face (encarar, mirar hacia), [ ] (constructor de listas), fput (poner de primero), patch-here (parcela-aquí), with-min (con-mínimo), with-max (con-máximo), myself (yo mismo), clear-drawing (limpiar-dibujo), sprout-<familia> (engendrar-<familia>), in-radius (dentro-del-radio), int (función parte entera), breed (familia).
Otros detalles: se crean dos familias y se da un ejemplo del uso de la primitiva myself.

El problema del vendedor viajante (the traveling salesman) fue formulado hace más de cien años y es uno de los problemas de optimización más estudiados. Consiste en lo siguiente: un vendedor debe visitar N ciudades siguiendo una ruta que pasa sólo una vez por cada ciudad y termina en la ciudad donde inició el recorrido. El problema consiste en encontrar la ruta con estas características que minimiza la distancia total recorrida por el agente. Una solución posible sería la de generar todas las posibles rutas, calculando la distancia recorrida para cada una y finalmente escoger aquella cuya distancia recorrida sea la mínima. Esto es viable para configuraciones en que el número de ciudades es muy pequeño, ya que el número de posibles rutas para N ciudades es igual a la cantidad conocida como "el factorial de N", a saber: 1 x 2 x 3 x 4 x 5 x 6 x .... x N, cantidad que crece muy rápidamente conforme aumenta el valor de N.[14] Existen métodos heurísticos para aproximar la solución siguiendo algunos tipos de estrategias. En este modelo presentamos cuatro tipos de estrategias para el agente vendedor y asignamos un botón a cada tipo, con lo que en realidad cada botón representa un modelo diferente. Las estrategias son las siguientes:
Máxima cercanía. El vendedor busca en todo momento del trayecto cuál es la ciudad más cercana al punto en que se encuentra y se dirige a ella. Si hay varias ciudades a igual distancia mínima, el vendedor selecciona una de éstas al azar.
Máxima lejanía. El vendedor busca en todo momento del trayecto cuál es la ciudad más lejana al punto en que se encuentra y se dirige a ella. Evidentemente ningún agente vendedor haría esto, a menos que anduviera buscando que su jefe lo despida. Incluimos este recorrido para tener una referencia al medir la distancia total recorrida.
Cercanía mixta. En cada momento del recorrido el vendedor pregunta si existen ciudades en un radio de 10 km (10 pasos). Si la respuesta es afirmativa escoge al azar

[14] Para dar una idea de cómo crece el factorial de un entero, el factorial de 30 ya supera la edad actual del Universo expresada en segundos.

una de estas ciudades y se dirige a ella, en caso contrario escoge al azar una de las restantes ciudades. El valor 10 podría ser cambiado por cualquier otro en el código.
Aleatoria. En cada momento del recorrido el vendedor escoge la siguiente ciudad al azar.



La figura muestra la trayectoria del vendedor visitando 100 ciudades bajo la estrategia de máxima cercanía. El cuadrito blanco señala el origen y final del recorrido.

Cómo correr el modelo. El número de ciudades se fija con el deslizador num-ciudades. El botón "setup" distribuye las ciudades aleatoriamente por el mundo, al cual se le da la topología del cuadrado. El modelo se puede correr en cualquiera de las cuatro modalidades descritas anteriormente haciendo clic sobre el correspondiente botón. A fin de poder comparar la distancia total recorrida por el agente según distintas estrategias, pero sin variar el número ni la disposición de las ciudades, se incluye el procedimiento "borrar", con su respectivo botón, el cual borra el trazo de la última trayectoria y mantiene la misma distribución espacial de las ciudades (atención: después de borrar se debe hacer clic sobre el botón de otra modalidad pero sin pulsar el botón setup pues esto generaría una nueva distribución de las ciudades).

Preparativos: Plantar botones "setup" y "borrar". Plantar botones para cada estrategia: máxima-cercanía, máxima-lejanía, cercanía-mixta y aleatoria, todos ellos con la casilla "Continuamente" marcada. Plantar el deslizador "num-ciudades". Configurar el mundo con la topología de la caja.

Plan general y problemas a resolver. Fuera del procedimiento setup que se aplica en cualquiera de las cuatro estrategias, el código se separa en cuatro bloques, uno para cada estrategia del vendedor. Los códigos de cada bloque difieren en una parte muy pequeña, a saber, la parte que define la estrategia. Se han utilizado tortugas para representar tanto a las ciudades como al vendedor. El número de ciudades se fija con el deslizador "num-ciudades" y la posición de cada una se fijan al azar con la orden

”setxy random-xcor random-ycor”, que ya se ha usado en anteriores ejemplos. Para diferenciar las ciudades del vendedor se han creado dos familias de tortugas: “ciudades” y “vendedores” con las declaraciones: “breed[ciudades ciudad]” y “breed[vendedores vendedor]”. La familia “vendedores” sólo tendrá un miembro, que seráel vendedor viajante. La ciudad donde se inicia el recorrido se ubica en la parcela llamada “origen”. Esta parcela se vuelve blanca al final del recorrido: ask origen [set pcolor white]. Durante la travesía, la siguiente ciudad que el vendedor visitará es la ciudad (tortuga) llamada “meta”, la cual se crea con la orden: “set meta one-of ciudades with...” El resto de la orden depende de la estrategia que está siguiendo el vendedor. Por ejemplo, en la modalidad de mínima cercanía sería:

“set meta one-of ciudades with-min [distance myself]”, es decir:
“asignar a la variable meta una de las ciudades cuya distancia a mí sea mínima”.

La distancia que ha recorrido el agente se almacena en la variable global “distancia-recorrida”. Cada vez que el vendedor visita una ciudad, vuelve roja la parcela donde la ciudad (la tortuga) se encuentra y la tortuga que representa la ciudad es eliminada (con la primitiva “die”), pero la tortuga vendedora sigue viva. Para saber cuándo el vendedor ha visitado todas las ciudades y debe regresar a la ciudad origen, se cuenta el número de tortugas-ciudades que quedan vivas. En el código del vendedor viajante, el uso de myself ocurre en la orden compuesta: ask vendedor 0 [set meta one-of ciudades with-min [distance myself]. Sabemos que al vendedor le corresponde el número who 0 por ser el primer agente en ser creado, motivo por el que podemos referirnos a él como “vendedor 0”. En la orden anterior “myself” (mí mismo) es el vendedor viajante, por ser él quien ha dado la orden. En los bloques de código que corresponden a cada estrategia, lo primero que el vendedor hace es verificar si aún quedan ciudades por visitar o si es hora de regresar a la ciudad “origen”, pintarla de blanco y mostrar la distancia total recorrida. La secuencia de estas órdenes es la siguiente:

```
if count ciudades = 0 [ask vendedores [face origen
set distancia-recorrida distancia-recorrida + distance origen
pd fd distance origen show int distancia-recorrida
ask origen [set pcolor white]] stop]
```

Analicemos las partes más importantes del anterior bloque de órdenes:
if count ciudades = 0 [ask vendedores [face origen
(si no quedan ciudades pedir al vendedor fijar su orientación hacia el origen).
set distancia-recorrida distancia-recorrida + distance origen
(sumar a distancia-recorrida la distancia al origen).
pd fd distance origen show int distancia-recorrida
(bajar pluma y avanzar la distancia que lo separa del origen y mostrar la distancia recorida como un número entero, eliminando los decimales).
ask origen [set pcolor white]] stop]
(pedir al origen [ponerse color blanco] y detener procedimiento).

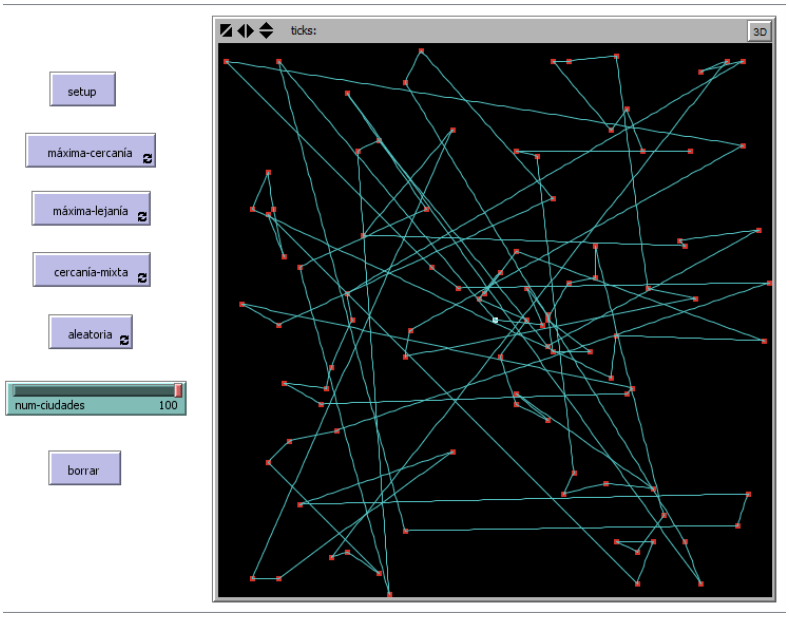
Hechas estas observaciones podemos presentar el código completo:

```
globals[origen meta distancia-recorrida]
```

```
breed[vendedores vendedor]
breed[ciudades ciudad]

to setup
clear-all
create-vendedores 1 [set origen patch-here]
create-ciudades num-ciudades [setxy random-xcor random-ycor set color yellow]
end

to máxima-cercanía
if count ciudades = 0 [ask vendedor 0 [face origen
set distancia-recorrida distancia-recorrida + distance origen pd fd distance origen
show int distancia-recorrida
ask origen [set pcolor white]] stop]
ask vendedor 0 [set pcolor red ]
ask vendedor 0 [set meta one-of ciudades with-min [distance myself]
face meta set distancia-recorrida distancia-recorrida + distance meta pd fd distance
meta]
ask meta [die]
wait 0.2
end

to máxima-lejanía
if count ciudades = 0 [ask vendedor 0 [face origen
set distancia-recorrida distancia-recorrida + distance origen pd fd distance origen
show int distancia-recorrida
ask origen [set pcolor white]] stop]
ask vendedor 0 [set pcolor red ]
ask vendedor 0 [set meta one-of ciudades with-max [distance myself]
face meta set distancia-recorrida distancia-recorrida + distance meta pd fd distance
meta]
ask meta [die]
wait 0.2
end

to cercanía-mixta
if count ciudades = 0 [ask vendedor 0 [face origen
set distancia-recorrida distancia-recorrida + distance origen pd fd distance origen
show int distancia-recorrida ask origen [set pcolor white]] stop]
ask vendedor 0 [set pcolor red ifelse (count ciudades in-radius 10) > 0
[set meta one-of ciudades in-radius 10 face meta
set distancia-recorrida distancia-recorrida + distance meta pd fd distance meta][set
meta one-of ciudades  face meta
set distancia-recorrida distancia-recorrida + distance meta pd fd distance meta] ]
ask meta [die]
wait 0.2
end



to aleatoria
if count ciudades = 0 [ask vendedor 0 [face origen
```

```
set distancia-recorrida distancia-recorrida + distance origen pd fd distance origen
show int distancia-recorrida
ask origen [set pcolor white]] stop]
ask vendedor 0 [set pcolor red ]
ask vendedor 0 [set meta one-of ciudades face meta
set distancia-recorrida distancia-recorrida + distance meta pd fd distance meta]
ask meta [die]
wait 0.2
end

to borrar
clear-drawing
ask patches with [pcolor = red] [sprout-ciudades 1] set distancia-recorrida 0
end
```

Explicaciones y comentarios adicionales. El procedimiento “máxima-lejanía” difiere de “máxima-cercanía” únicamente en una primitiva: “with-min” es reemplazada por “with-max”. En el procedimiento “borrar” figura la expresión “sprout-ciudades 1” en una orden dirigida a las parcelas cuyo color es rojo. Nótese que por ser “ciudades” una familia se puede usar su nombre en singular o en plural para construir órdenes en que se combina el nombre de la familia con otras primitivas. Aquí se ordena a las parcelas de color rojo engendrar un agente de la familia “ciudades” (sprout-ciudades 1). Si usted desea comprobar que el vendedor efectivamente está respetando la modalidad indicada del recorrido, por ejemplo dirigiéndose siempre a la ciudad más cercana bajo la estrategia de ”máxima-cercanía”, desmarque simplemente la casilla “Continuamente” en el respectivo botón de la modalidad y observe al vendedor avanzar a la siguiente ciudad cada vez que hace clic sobre dicho botón.



La figura muestra un recorrido con la estrategia de cercanía mixta. Se puede observar que el vendedor conectó ciudades muy cercanas en varios puntos del recorrido y también realizó grandes saltos entre dos ciudades.

Para la trayectoria de cercanía mixta de la figura, la distancia total recorrida fue de 2455 km. Con esta misma distribución de ciudades, la distancia en las otras modalidades fue: máxima cercanía 942 km, máxima lejánía 7732 km y recorrido aleatorio 5370 km. Una exploración interesante consiste en comparar los resultados cambiando la topología del mundo. Se observará una disminución de la distancia total con la topología de la rosquilla, lo mismo que con la del cilindro, aunque un poco menos en este útlimo caso (explicar por qué).

Ejercicio 5.1. Modificar el código para que las ciudades sean representadas por parcelas.

Asociación entre agentes.

La asociación de agentes en grupos que poseen ciertas afinidades o que desempeñan tareas específicas es común en muchos modelos. La realidad nos provee de incontables ejemplos de agentes que se agrupan de acuerdo a ciertas reglas. Los humanos construimos grupos basados en criterios como sexo, afinidad política, religiosa o deportiva, nacionalidad, profesión o pasatiempos, entre muchos otros criterios. Las partículas elementales se reúnen para formar átomos y estos se asocian en grupos llamados moléculas, las letras del alfabeto se agrupan para formar palabras y las palabras para formar frases. En todos estos casos la formación de los grupos ocurre siguiendo criterios o reglas que podríamos llamar *reglas de asociación*, las cuales permiten o excluyen la posibilidad de que dos o más agentes se asocien. En los fenómenos del mundo real, con frecuencia la mayor dificultad que afrontan las ciencias reside en descubrir cuáles son las reglas de asociación que rigen la conducta de los agentes involucrados en un fenómeno (¿por qué se asocian ciertos tipos de células para formar un órgano y cuándo dejan de hacerlo?, ¿por qué tantas personas votaron para presidente por el candidato Inutilio Cuestionblez?). En el siguiente ejemplo las tortugas se asocian en grupos llamados clubes. El mecanismo de asociación está basado en un intercambio de información entre las tortugas y las parcelas.

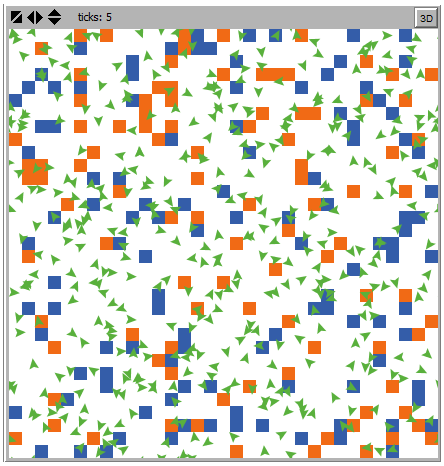
Modelo 10: La feria del libro.

Primitivas: length (longitud), count (contar), turtles-here (tortugas-aquí), die (morir), lput (poner de último), with (con), sum (suma), of (de).

La historia. En un parque de la ciudad tiene lugar una gran feria del libro en la que participan muchas casas editoriales de todo el mundo. Las editoriales se agrupan bajo las letras N, E, L y W, que corresponden a N = Norteamérica (Estados Unidos y Canadá), L = Latinoamérica, E = Europa y W = resto del mundo. En el parque se establecen muchos kioskos de lona o "stands", distribuidos por el área de la feria donde se venden los libros. Los kioskos son de dos tipos posibles: en los de tipo NL se venden libros de las editoriales N y L, en los de tipo EW se venden los libros de las editoriales E y W. Al ingresar a la feria una máquina dispensadora entrega un carnet a cada visitante, el cual tiene impresa alguna de las cuatro letras N, E, L o W, generada al azar por la máquina. El carnet permite a su portador o portadora comprar, con importantes rebajas y durante todo el año, los libros de las editoriales que corresponden a la letra impresa en su carátula. Pero para lograr estas rebajas el visitante debe antes asociarse a un club de lectura de alguno de los kioskos. Lo que se requiere para asociarse a un club de lectura de un kiosko es muy simple: básicamente sólo se requiere llegar temprano a la feria, para poder encontrar cupo en el club de alguno de los kioskos, ya que el número de cupos de cada club es limitado (un buen truco de mercadeo para lograr asistencia). Si, por ejemplo, usted posee un carnet de tipo E, debe visitar un kiosko de tipo EW y si aún quedan cupos libres en el kiosko, con sólo presentar su carnet usted queda automáticamente inscrito en el club de lectura de dicho kiosko, bajo la letra E, lo cual le permitirá beneficiarse de las rebajas de las editoriales del grupo E durante todo el año. Cada kiosko lleva un registro de los tipos de carnets de los socios que se registran en su club de lectura. Al final se puede consultar cómo ha quedado distribuida la conformación de los clubes.

Plan general y problemas a resolver. Hay que resolver los siguientes problemas:
1. Diseminar los kioskos por el parque (el mundo). Resolveremos esto asignando a cada parcela una de las tres siguientes cadenas: "NL", "EW", "O". La cadena "O" representará las parcelas donde no hay ningún kiosko. La asignación se hará aleatoriamente mediante la instrucción: ask patches [set letra-del-kiosko one-of ["NL" "EW" "O" "O" "O" "O" "O" "O" "O" "O"] donde se incluye "O" un mayor número de veces, a fin de que la mayoría de las parcelas no alojen ningún kiosko. Los kioskos de tipo "NL" se pintarán de azul y los de tipo "EW" se pintarán de color naranja.
2. Idear un mecanismo para que los visitantes se asocien a los clubes de lectura.
Modelaremos esto por medio de asignar variables a las parcelas y a las tortugas (los visitantes) y realizar comparaciones entre estas variables. Las parcelas tendrán dos variables de tipo patches-own: la variable "kiosko" almacenará alguna de las cadenas "NL", "EW"u "O" y la variable "club" almacenará una lista con los tipos de carnet de los socios que se han inscrito en su club de lectura. Por ejemplo ["E" "W" "W" "E"] indicaría que en el club del kiosko se asociaron dos personas con carnet de tipo E y otras dos con carnet de tipo W. Los visitantes tendrán sólo una variable de tipo turtles-own, en la cual guardan el tipo de carnet que la máquina dispensadora les ha asignado: N, L, E o W. Un deslizador "num-max-socios" determinará el número máximo de socios que los clubes pueden admitir y un deslizador "población-inicial" determinará el número inicial de visitantes a la feria.

El mecanismo de asociación a los clubes: Cuando una persona (tortuga) llega a una parcela, compara el valor de la variable “kiosko” de la parcela con el valor de su variable “carnet”. Si hay concordancia entre el valor de “carnet” y alguna de las dos letras de la variable “tipo-kiosko”, y si además hay cupo disponible, es decir, si “length club” no excede a “num-max-socios”, entonces la persona queda automáticamente inscrita en el club de lectura del kiosko. Esto se hace añadiendo la letra del carnet de la persona a la lista “club” de la parcela. Por ejemplo, supongamos que el carnet de Emilia es “L”, quien visita un kiosko (parcela) de tipo “NL” y el número máximo de socios permitidos es 6. Supongamos además que la variable “club” del kiosko tiene el valor [“L” “L” “N”]. Esto significa que “length club = 3” y aún hay tres lugares disponibles en el club del kiosko, por lo que Emilia quedaría incluida en el club de ese kiosko y la variable “club” de la parcela pasa a tomar el valor [“L” “L” “N” “L”]. Si una persona no logra inscribirse en el club de un kiosko, ya sea por que no queda cupo o porque el tipo de carnet de la persona no coincide con ninguna de las letras del kiosko, la persona entonces deberá seguir su marcha. Para evitar que las personas se inscriban en más de un club son eliminadas al momento de la inscripción (regresan felices a su casa después de haber comprado con rebajas). El procedimiento se detiene si no queda ninguna persona (tortuga) sin inscribirse en algún club o bien pulsando el botón “go”. Una vez que el modelo se detiene es posible hacer consultas sobre la distribución de personas en los clubes desde la Ventana del Observador.



Preparativos: Plantar los botones setup y go (con casilla “Continuamente marcada) y los deslizadores “población inicial” y “num-max-socios”. He aquí el código:

```
patches-own[club kiosko]
turtles-own[carnet]

to setup
clear-all
crt población-inicial [setxy random-xcor random-ycor
set color green set carnet one-of [″N″ “E″ “L″ “W″]]
ask patches [set kiosko one-of [″NL″ “EW″ “O″ “O″ “O″ “O″ “O″ “O″ “O″ “O″
“O″]]
ask patches [set club [] if kiosko = “O″ [set pcolor white] ]
reset-ticks
end

to go
ask turtles [ fd 2 if (length club < num-max-socios) and kiosko = “NL″
and (carnet = “N″ or carnet = “L″) [ set club lput carnet club die] ]
ask turtles [ fd 2 if (length club < num-max-socios) and kiosko = “EW″
and (carnet = “E″ or carnet = “W″) [ set club lput carnet club die] ]
ask patches  [if kiosko = “NL″   [set pcolor blue]
```

```
if kiosko = "EW" [set pcolor orange]]
if count turtles = 0 [stop]
wait 0.1
tick
end

to mostrar
ask patches with [length club > 0 ][show club]
end
```

Explicaciones y comentarios adicionales. El procedimiento se detendrá cuando no queden tortugas buscando asociarse a clubes. Sin embargo, puede ocurrir que queden unas pocas tortugas que no logran encontrar un club. Esto se debe a que por tener las tortugas una orientación fija, pueden quedar tortugas que caminen incesantemente sin encontrar un club al cual asociarse. En ese caso hay que detener el procedimiento manualmente pulsando el botón "go". El procedimiento "mostrar" es opcional y se debe llamar desde la Ventana del Observador. Este procedimiento muestra las parcelas donde hay clubes y cuál es su composición. Las parcelas se han coloreado de acuerdo al siguiente criterio. Inicialmente cuando se corre el procedimiento "setup" las parcelas en negro son aquellas en donde habrá kioskos de ventas y las blancas donde no habrá, mientras que todas las tortugas aparecen en color verde. Cuando se corre el procedimiento "go" las parcelas con kioskos de tipo NL se vuelven azules y las de tipo EW naranja, como se muestra en la figura anterior. Es posible obtener más información sobre la distribución de las tortugas en los clubes digitando órdenes en la Ventana del Observador después de correr el modelo. Para una corrida con la población inicial de 1000 tortugas y la variable num-max-socios con valor de 6 obtuvimos los siguientes resultados:

```
print count patches with [length club = 6]
```
==> 15, 15 kioskos lograron llenar sus clubes con 6 miembros.
```
print count patches with [num-socios > 3]
==> 92
print count patches with [estado = ["N" "N" "L"]]
```
==> 5, notar que por tratarse de listas, un club cuyo estado fuera ["N" "L" "N"] no se habría contado entre estos 5 clubes.
```
print count patches with [num-socios > 2 and num-socios < 6 ]
==>159
print count patches with [length estado = 4]
==>3
```

Cosas a explorar. Variando el valor de la población que asiste a la feria (deslizador población-inicial) y el número máximo de socios que se admiten en los clubes (deslizador "num-max-socios"), se obtienen distintos resultados, los cuales puede ser interesante explorar. Si el número de asistentes es suficientemente grande, todos los clubes se llenarán y quedará un sector de la población que no podrán comprar con descuentos por no haber encontrado clubes con lugares disponibles. Si la población es baja, habrá clubes llenos, otros a medio llenar e incluso puede haber kioskos cuyos clubes queden vacíos. El modelo se puede adaptar a varias otras situaciones en una población de agentes que busca asociarse para formar grupos. Ejemplos de esto podrían ser: personas

con determinados perfiles profesionales que buscan empleo en ciertos tipos de instituciones o empresas, o átomos que se unen a moléculas para formar moléculas de mayor tamaño (adelante se expondrá un modelo sobre este tema).

Ejercicio 5.2. Para mejorar las consultas sobre cómo están integrados los clubes, ordenar sus miembros por orden alfabético. (Sugerencia: utilizar la primitiva "sort-by").

Modelo 11: Buscando taxi en la ciudad.

Primitivas: ifelse, count, with-min (con-mínimo), myself (a mí mismo), one-of, create-link-with (crear-enlace-con), tie (atar), stop, distance.
Otros detalles: empleo de familias (breeds).

La historia. En un día laboral, en la gran ciudad hay muchas personas buscando un taxi para llegar o regresar de sus trabajos, para hacer turismo o para otras cumplir con actividades. Tanto personas como taxis se encuentran dispersos por toda la ciudad. En el modelo supondremos que cuando una persona solicita un taxi por su celular, un sistema basado en GPS envía a la persona el taxi disponible más cercano a ella. A los taxis no les está permitido recoger pasajeros mientras están realizando un servicio, aunque terminado el mismo pueden acudir al llamado de otro pasajero. El número de personas y de taxis se fija mediante deslizadores en la interfaz. El objetivo del modelo es observar cómo se distribuyen los servicios de taxi entre las personas.

Plan general y problemas a resolver. Las personas y los taxis se representarán por tortugas pertenecientes a dos familias: "taxis" y "personas" cuyos miembros se distribuyen al azar en el mundo. Los destinos adonde quieren llegar las personas están dados por parcelas, también escogidas al azar. Cada persona posee dos variables propias (turtles-own):

1) La variable "cercanos" tiene como valor el conjunto-agentes de los taxis cuya distancia a la persona es la más cercana (mínima).

2) La variable “destino” tiene como valor la parcela destino a que la persona desea llegar en taxi.

Los taxis, a su vez, poseen dos variables propias (taxis.own):

1) La variable “estado”, la cual puede tomar los valores 0 o 1 según el taxi esté disponible o esté realizando un servicio.
2) La variable “servicios”, la cual cuenta el número de servicios o carreras que el taxi ha realizado.

Todo servicio de taxi se representará por una línea quebrada con un primer tramo recto en amarillo y un segundo tramo recto en rojo. El tramo amarillo representa el trayecto que va del sitio donde se encuentra el taxi al lugar donde se encuentra la persona que ha solicitado el servicio, mientras que el tramo rojo marca el trayecto que va de donde el taxi recoge al pasajero al destino del mismo. A fin de que el taxi y el pasajero viajen juntos, en el momento en que el taxi llega al lugar donde se encuentra el pasajero, se crea un enlace entre ambos de tipo atadura (tie). Cuando ambos llegan al destino, la tortuga que representa al pasajero muere. Esto evita que el pasajero continúe demandando servicios de taxi y yendo de un lado a otro sin descanso. Una vez corrido el modelo resulta interesante realizar algunas consultas. La figura anterior muestra los servicios prestados por 2 taxis a 3 personas. Los servicios se inician con el tramo amarillo. La línea quebrada superior muestra que uno de los taxis realizó un sólo servicio (trayectoria amarillo-rojo). La línea quebrada inferior muestra una trayectoria que se compone de 4 trazos en secuencia amarillo-rojo-amarillo-rojo y nos dice que el taxi realizó dos servicios. La trayectoria se inicia en un tramo amarillo (el taxi recoge al primer cliente) seguido de un tramo rojo (el cliente es llevado a su destino), le sigue un segundo tramo amarillo (el taxi acude a recoger a la segunda cliente) y termina con un tramo rojo (la segunda cliente, la señora Robinson, ha sido llevada a su destino).

Una vez creados los conjunto-agentes de las personas y los taxis y definidas las variables en el procedimiento setup, las acciones más importantes se llevan a cabo en el procedimiento pedir-taxi, el cual es llamado por el procedimiento go:

1) La persona pide un taxi: *ask personas [pedir-taxi]*
   El procedimiento “pedir-taxi” desencadena las siguientes acciones
   a. Se selecciona un taxi del conjunto-agentes “cercanos”
      set cercanos taxis with-min [distance myself]
   b. Se verifica que hay taxis desocupados y que hay al menos un taxi cercano:
      ifelse (count taxis with [estado = 0]) > 0 [if count cercanos > 0
   c. Se le pide al taxi escogido que se vuelva color amarillo, se ponga en dirección al pasajero, avance hacia él, y defina su “estado” como “ocupado” y sume 1 al número de servicios que le han solicitado:

ask one-of cercanos [set color yellow face myself
pd fd distance myself set estado 1 set servicios servicios + 1

2) Luego se le pide al taxi que establezca un enlace con el pasajero o pasajera, que le pida se amarre el cinturón, ponga el taxi de color rojo y lleve al pasajero a su destino:
[….create-link-with myself [tie] set color red] ir-a-destino

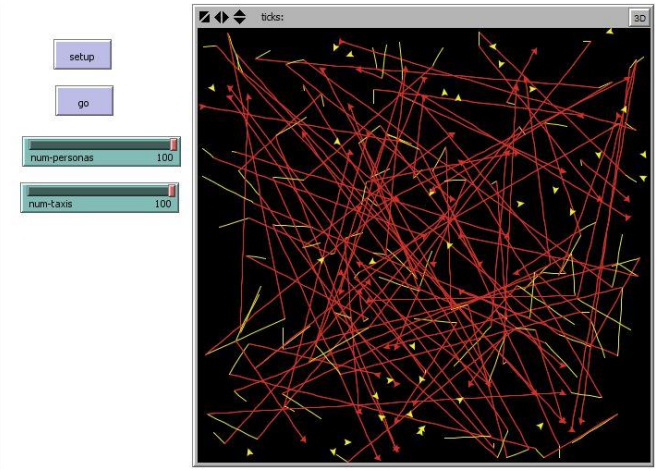
El procedimiento ir-a-destino no necesita explicaciones adicionales.

Preparativos: Plantar botones setup y go (casilla "Continuamente" marcada) y deslizadores "num-personas y num-taxis. Crear familias de personas y de taxis. Configurar el mundo con la topología del cuadrado. He aquí el código completo:

```
breed[taxis taxi]
breed[personas persona]
personas-own[destino cercanos]
taxis-own[ estado servicios]

to setup
clear-all
create-personas num-personas [setxy random-xcor random-ycor set color red
set destino one-of patches ]
create-taxis num-taxis [setxy random-xcor random-ycor set color yellow]
end

to go
ask personas [pedir-taxi]
end

to pedir-taxi
set cercanos taxis with-min [distance myself]
ifelse (count taxis with [estado = 0]) > 0 [if count cercanos > 0
[ask one-of cercanos [set color yellow face myself
pd fd distance myself ;; taxi avanza hacia mí
set estado 1 set servicios servicios + 1
create-link-with myself [tie] set color red ]]]
ir-a-destino
end

to ir-a-destino
wait 0.2 face destino pd fd distance destino die
end
```

Explicaciones y comentarios adicionales. Se desea que cuando la persona pida un taxi, se le asigne aquel que se encuentra a la mínima distancia de ella. Si hubiera más de uno el intérprete selecciona alguno de éstos al azar. Si se suprime la orden que elimina a las personas cuando llegan a su destino, ocurrirán cosas que se salen del libreto establecido, por ejemplo, veríamos trayectorias con trazos rojos seguidos de trazos rojos, que tendríamos que interpretar como personas viajando largas distancia a pie a otros destinos, cuando aún quedan taxis disponibles. El programa se detiene a pesar de que

en el código no aparece la primitiva “stop” ni hay ninguna condición explícita para detener el programa. Esto ocurre porque si en una instrucción del tipo:
“ask conjunto-agentes [..órdenes..]”, el intérprete encuentra que el conjunto-agentes se ha quedado vacío, esto da por terminado el programa.



La figura muestra los servicios brindados por 100 taxis a 100 personas.
Nótese que hay taxis que no realizaron ningún servicio.

Veamos algunas consultas correspondientes a la corrida de la figura anterior.
Número de taxis que no realizaron ningún servicio:
show count taxis with [estado = 0]
==>34
Número máximo de servicios realizado por algún taxi:
show max [servicios] of taxis
==>4
Número de taxis que realizaron 4 servicios:
show count taxis with [servicios = 4]
==>1
Otra información: 5 taxis realizaron 3 servicios, 21 realizaron 2 servicios y 39 realizaron 1 servicio.

Ejercicio 5.3: Adaptar el modelo para que cada cierto tiempo aparezcan nuevos peatones que necesitan un servicio de taxi.

Ejercicio 5.4: Modificar el código para que los taxis que realizan más de un servicio dejen un trazo en verde.

Modelo 12: Buscando un cargamento de droga en la selva.

Primitivas: remove (remover), run (ejecutar), first (primero), length (longitud), but-first (excepto el primero), write (escribir), word (palabra).
Otras características: cómo leer uno a uno los caracteres de una cadena o los miembros de una lista, cómo usar la primitiva “run”.

La historia. La agente de policía Evans, mujer con gran experiencia en misiones de adentramiento en la selva tropical, es enviada con dos perros en busca de un cargamento de droga que se sabe ha sido enterrado en algún lugar de la jungla y el cual no está resguardado por los narcotraficantes. Se le encarga a Evans que haga una bitácora de todos los movimientos de su trayectoria por la selva y la envíe por radio a la Estación Central de Policía a fin de allanar el camino a otro agente que se reunirá con ella. La caminata de la agente Evans (tortuga 0) estará basada en cuatro movimientos: avanzar 1 paso hacia adelante, girar 90º a la derecha, girar 90º a la izquierda y girar 180º a la derecha (no se incluye el giro de 180 a la izquierda pues su efecto es el mismo que el giro a la derecha). Una vez que Evans encuentre el lugar donde está el cargamento, el agente García, del temible Comando Azul, será despachado al lugar para brindar asistencia a Evans y coordinar la confiscación del cargamento. García basará su trayectoria en la bitácora enviada por Evans, pero antes de partir a reunirse con ella, la bitácora debe ser depurada. Avanzar por una selva, donde además de la densa vegetación hay muchos otros obstáculos y peligros no es cosa fácil y en varias ocasiones Evans debe recurrir a la técnica de prueba y error. Esta es la razón por la que la bitácora de movimientos enviada por Evans contiene algunas secuencias de movimientos superfluos, los cuales es deseable eliminar para que el agente García pueda llegar más rápidamente al punto donde se encuentra Evans. Algunos ejemplos de las secuencias superfluas que podría contener la bitácora de la agente Evans son: un giro de 90º a la derecha seguido de uno de 90º a la izquierda o bien un avance de un paso en una dirección seguido inmediatamente de un avance de un paso en la dirección opuesta.

Tarzán y Roscoe, los invaluables compañeros que acompañaron a la agente Evans por la selva.

Cómo correr el modelo.

1. Pulsar el botón setup y luego el botón go.
2. Detener el modelo volviendo a pulsar el botón go cuando se supone que la agente Evans ha encontrado el cargamento. (Es el usuario quien decide en qué punto de la caminata se encuentra el cargamento).
3. Pulsar el botón “Depurar”, cuyo efecto es depurar la bitácora enviada por la agente Evans, eliminando las secuencias superfluas.
4. Pulsar el botón “comando-azul” para que el agente García se interne en la selva y se encuentre con la agente Evans siguiendo la trayectoria depurada.

5. En la Terminal de instrucciones aparece el número de movimientos de la trayectoria (la bitácora) de la agente Evans y de la trayectoria depurada del agente García.

Plan general y problemas a resolver. El modelo presenta dos problemas principales que se deben resolver:

1. Cómo guardar la información de la trayectoria de la agente Evans en un mensaje.
2. Cómo depurar el mensaje enviado por la agente Evans, eliminando las secuencias de movimientos superfluos.

El primer punto se resuelve guardando todos los movimientos de la agente en una variable, que llamaremos "bitácora" y la cual, en principio podría ser una lista, como por ejemplo: set bitácora ["fd 1" "rt 90" "rt 90" "fd 1" "fd 1"....] o bien una cadena, por ejemplo set bitácora "fd 1 rt 90 rt 90 fd 1 fd 1 lt 90". Para depurar el mensaje enviado por Evans de las secuencias superfluas, se debe comenzar por hacer una lista de este tipo de secuencias. Las siguientes secuencias de movimientos realizados por la agente Evans se consideran superfluas:

Giros en un mismo punto que no cambian la orientación de la tortuga:
1. rt 90 lt 90
2. lt 90 rt 90
3. rt 180 rt 180

Avance de 1 paso y regreso al mismo punto manteniendo la misma orientación inicial:
4. fd 1 rt 180 fd 1 rt 180.
5. fd 1 rt 180 fd 1 rt 90 rt 90.
6. fd 1 rt 180 fd 1 lt 90 lt 90.
7. fd 1 rt 90 rt 90 fd 1 rt 180.
8. fd 1 lt 90 lt 90 fd 1 rt 180.
9. fd 1 rt 90 rt 90 fd 1 rt 90 rt 90.
10. fd 1 rt 90 rt 90 fd 1 lt 90 lt 90.
11. fd 1 lt 90 lt 90 fd 1 rt 90 rt 90.
12. fd 1 lt 90 lt 90 fd 1 lt 90 lt 90.

Estas 12 secuencias de órdenes dejan a la tortuga en la misma posición y con la misma orientación que antes de ejecutarse la secuencia, por lo que pueden ser eliminadas. Hay otras secuencias superfluas que no se han incluido, debido a que su presencia es menos probable, como por ejemplo una secuencia de cuatro giros de 90 grados en una misma dirección, que equivaldría a un giro de 360 grados o secuencias en que la tortuga avanza más de un paso y regresa al mismo punto y con la misma orientación con que inició el recorrido. En principio el número de posibles secuencias superfluas es infinito, pero entre más largas son las secuencias más improbables son. En este modelo sólo consideraremos las 12 secuencias superfluas anteriormente mencionadas. Si decidiéramos representar el mensaje enviado por la agente Evans por medio de una lista, la depuración consistirá en eliminar de dicha lista las sublistas que corresponden a secuencias superfluas. Pero ocurre que NetLogo no tiene primitivas que puedan

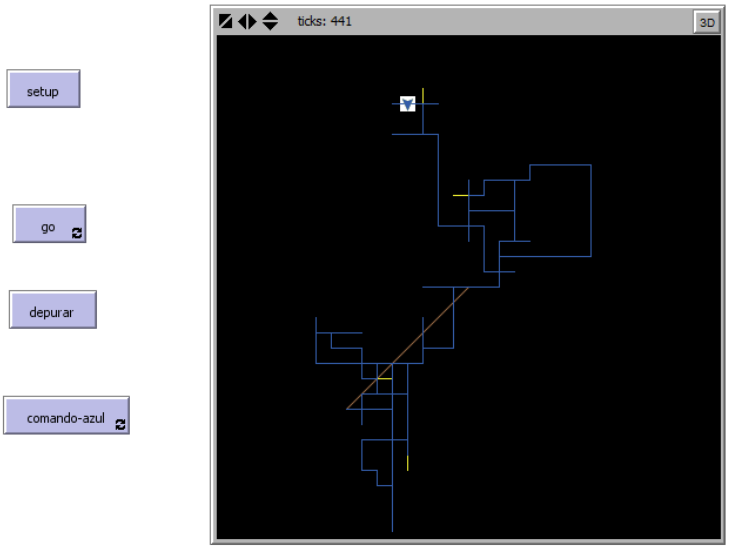
encontrar (tipo “search”) una sublista de una lista dada, sin tener que indicar también la posición donde comienza y termina la sublista. Pero hay un problema: no sabemos en qué posición de la lista se van a encontrar las sublistas que corresponden a secuencias superfluas. Para no tener que construir procedimientos que realicen esa función de búsqueda, resulta más económico representar la variable “bitácora” mediante una cadena, ya que para cadenas se dispone de la primitiva “remove”. Esta primitiva permite remover una subcadena, sin necesidad de indicar la posición que ocupa dentro de la cadena que la contiene. Para hacer el mensaje depurado aún más compacto, representaremos los cuatro movimientos permitidos a los agentes por medio de letras, según el siguiente convenio: los movimientos fd 1, rt 90, lt 90 y rt 180 se representarán mediante las letras f, r, l y h, donde a cada letra se le asigna un procedimiento con la acción correspondiente. De esta manera una cadena como, por ejemplo, “ffrfl” representa la secuencia de movimientos fd 1, fd 1, rt 90, fd 1 lt 90. Finalmente, el intérprete transformará la cadena en una sucesión de movimientos de tortuga asignando a cada letra un procedimiento. Por ejemplo. a la letra f se le asocia el procedimiento:

```
to f
fd 1
end
```

Preparativos: plantar botones para los procedimientos “setup”, “go” (casilla “Continuamente” marcada), “depurar” y “comando-azul”. Configurar el mundo con la topología del cuadrado. El código es el siguiente:

```
globals[movimientos bitácora]
to setup
clear-all
crt 1 [set bitácora “” ;; aquí se define “bitácora” igual a la cadena vacía “”
set heading 0 set color yellow setxy -8 -8 pd]
ask turtle 0 [set movimientos [”f” “f” “f” “r” “l” “h”]]
crt 1 [setxy -8 -8 set heading 0 pd set color blue]
reset-ticks
end

to go
ask turtle 0 [let paso one-of movimientos
run paso
wait 0.1  set bitácora word bitácora paso] ;; se añade “paso” a la cadena “bitácora”
tick
end

to f
fd 1
end

to r
rt 90
end
```

```
to l
lt 90
end

to h
rt 180
end

to depurar
type "Movimientos de la agente Evans: "   type " " write length bitácora
set bitácora remove "frrfrr" bitácora
set bitácora remove "fllfll" bitácora
set bitácora remove "fllfrr" bitácora
set bitácora remove "frrfll" bitácora
set bitácora remove "frrfh" bitácora
set bitácora remove "fllfh" bitácora
set bitácora remove "fhfh" bitácora
set bitácora remove "fhfll" bitácora
set bitácora remove "fhfrr" bitácora
set bitácora remove "rl" bitácora
set bitácora remove "lr" bitácora
set bitácora remove "hh" bitácora
type "  " type "Movimientos del agente García: " write length bitácora
print ""
end

to comando-azul
ask turtle 1 [ run first bitácora set bitácora but-first bitácora]
wait 0.1 if length bitácora = 0 [ask turtle 1 [set pcolor white]
beep stop]
end
```



La figura muestra trayectoria del agente García, la cual consta de 334 pasos. La trayectoria original de la agente Evans constaba de 441 pasos. El proceso de depuración redujo la trayectoria en un 24%, eliminando secuencias superfluas.

Explicaciones y comentarios adicionales:
La orden *set bitácora “”* asigna a la variable “bitácora” el valor de la cadena vacía “”. La variable movimientos se define como la lista de los posibles movimientos que pueden realizar los agentes: [”f” “f” “r” “l” “h”] donde cada letra es el nombre de un procedimiento que corresponde a un movimiento. La repetición de “f” es intencional para darle mayor probabilidad a los avances hacia adelante que a los giros. A la variable temporal “paso”, definida mediante la primitiva “let”, se le asigna como valor uno de los procedimientos de la lista “movimientos”: “let paso one-of movimientos”. Para correr el procedimiento alojado en la variable “paso” es necesario usar la primitiva “run”. La orden “run paso” se traduce como “correr el procedimiento contenido en la variable paso”. La primitiva “word” permite formar una cadena pegando dos cadenas. La expresión “word bitácora paso” agrega a la cadena bitácora la letra contenida en la variable “paso”, que corresponde al último movimiento realizado por la agente. Al inicio y al final del procedimiento “depurar” se han incluido órdenes que escriben en la Terminal de Instrucciones la longitud de la cadena “bitácora” (length bitácora) antes y después de efectuada la depuración, con el fin de conocer cuántos movimientos de diferencia hay entre la longitud de la bitácora enviada por Evans antes y después de ser depurada.

Como lo hemos señalado ya, la programación de modelos es un verdadero caldo de cultivo para la generación de ideas y para establecer conexiones con otros campos, que facilita:
1) La formulación de preguntas relacionadas con el funcionamiento del modelo.
2) El descubrimiento de conexiones del modelo con otras áreas de conocimiento.
3) La generación de ideas para extender el modelo.
4) El suplir explicaciones de conductas no previstas en el modelo.
5) La depuración de errores de programación y de planificación (errores de diseño y de sintaxis).
Utilicemos el modelo anterior como ejemplo. Algo que se puede observar al correr el presente modelo es que el agente García (tortuga 1) deja algunos pequeños trazos del trayecto -especie de puntas- en color amarillo, sin repintar. Después de analizar la situación podemos notar que se trata de puntas que tienen 1 paso de longitud y que no son sino pequeños trozos superfluos de los tipos 4 al 12 de la lista descrita anteriormente. Al buscar una explicación para la aparición de estas pequeñas puntas amarillas, surge otra cuestión, posiblemente más importante. En el proceso de depuración, supongamos que la cadena “bitácora” contiene una subcadena como la siguiente: “frrfrrlf”. Nótese que esta subcadena contiene dos secuencias superfluas que se solapan, las cuales son “frrfrr” (secuencia de tipo 9) y “rl” (secuencia de tipo 1). Si el intérprete decide eliminar primero la secuencia superflua “rl” se obtiene “frrfrf” pero si primero se elimina la secuencia superflua frrfrr, el resultado sería “lf”. La pregunta es ahora: ¿eliminar primero “frrfrf” produce el mismo resultado que eliminar primero “lf”. Un breve análisis de la situación revela que no lo son. Mientras eliminar “frrfrr” no altera el trayecto seguido por la agente García la eliminación de “lr” sí lo hace. Esto significa que, dependiendo del orden en que el intérprete suprima las secuencias superfluas, el agente García podría estar en peligro de desviarse de la trayectoria seguida por Evans y perderse en la jungla. Sin embargo, si forzamos al procedimiento “depurar” a eliminar

primero las secuencias superfluas largas de los tipos 4 al 12, este fenómeno no se podrá dar. Esta es la razón por la cual en el código del procedimiento “depurar”, las secuencias de los tipos 4 al 12 son las primeras que aparecen (un interesante tema de discusión para analizar en clase).

Ejercicio 5.5. Explicar la presencia de las “puntas amarillas” en la trayectoria del agente García.

Modelo 13: La sopa de la vida.

Primitivas: with (con), and (y), [ ] (constructor de listas), first, lput (poner de último), sort-by (ordenar según), != (diferente a).
Otros detalles: el modelo muestra cómo usar y manipular listas cuyos miembros son a su vez listas. También se muestra cómo simplificar expresiones complejas utilizando procedimientos reportadores.

La historia. Una de las teorías sobre la manera cómo surgió la vida en nuestro planeta plantea que, hace muchos millones de años, cuando La Tierra se encontraba en un estado muy primitivo, su superficie estaba cubierta por un caldo o sopa constituida en gran medida por los átomos que forman las moléculas de los compuestos orgánicos. Los elementos más comunes en esta sopa eran carbono, oxígeno, hidrógeno y nitrógeno. Se cree que de este caldo surgieron los primeros compuestos orgánicos que posteriormente dieron origen a los organismos unicelulares como las bacterias. La sopa de nuestro ejemplo estará formada por sólo tres de estos cuatro elementos, a saber, **C** (carbono), **O** (Oxígeno) y **H** (Hidrógeno). Las combinaciones o moléculas que en nuestro modelo se podrán formar son: **H, H_2, H_2O (agua), C, CH, CH_2, CH_3, CH_4 (metano), OH, O, O_2, O_3** (ozono), **CO** (monóxido de carbono), **CO_2** (dióxido de carbono)[15]. En total 14 moléculas diferentes, entre las cuales se cuenta a los átomos de los tres elementos originales H, C y O como moléculas mono-atómicas. El propósito del modelo es crear una estrategia para que los átomos de la sopa original formen moléculas, por medio de un proceso básicamente aleatorio e implementar dicha estrategia en lenguaje NetLogo. Después de correr el modelo podremos preguntar qué porcentajes de moléculas de cada tipo se han formado.

[15] En este modelo no se han incluido todas las moléculas que pueden formar los átomos H, C y O, ni siquiera las pequeñas de menos de 6 átomos.

Plan general y problemas a resolver. La idea central del plan consiste en lo siguiente:

1. Los átomos se repartirán entre las tortugas, asignando a cada tortuga una variable "átomo" que toma alguno de los tres valores "C", "H", "O" (carbono, hidrógen y oxígeno), mediante la orden:
   ask turtles [set átomo one-of ["H" "C" "O"]].
   Los átomos deambularán por el mundo en caminatas aleatorias buscando moléculas con las cuales combinarse.
2. En las parcelas se alojarán las moléculas, las cuales se representarán por listas. Inicialmente en las parcelas sólo habrá moléculas mono-atómicas de carbono, hidrógeno u oxígeno representadas por las listas ["C"], ["H"] y ["O"] respectivamente, pero conforme el modelo evoluciona estas moléculas aceptarán combinarse con los átomos deambulantes que "portan" las tortugas para formar moléculas de mayor tamaño, como por ejemplo la molécula de agua ["H" "H" "O"] o la de monóxido de carbono ["C" "O"]. Cada parcela guarda la molécula que tiene alojada en la variable llamada "molécula". El valor inicial de esta variable se fija con la orden: ask patches [set molécula one-of [["H"] ["C"] ["O"] ["V"] ["V"] ["V"] ["V"] ["V"] ["V"] ["V"]]] donde el valor ["V"] representa las "moléculas vacías", es decir, parcelas donde no habrá ningúna molécula mono-atómica de H, C u O. El mayor número de parcelas de tipo "V" obedece al deseo de que la mayoría de las parcelas del mundo se encuentren vacías.
3. A fin de reducir el número de listas que representan una misma molécula se conviene en representar las moléculas por listas ordenadas alfabéticamente, llamadas "moléculas ordenadas". Por ejemplo, la lista ["H" "O" "H"] no representa la molécula de agua porque no está ordenada alfabéticamente. La primitiva "sort-by" (ordenar-según) se encarga de realizar esta ordenación y la convertiría en ["H" "H" "O"].
4. Las tortugas (los átomos de "C", "O" y "H") deambularán por el mundo y cada vez que un átomo (una tortuga) llega a una parcela donde hay una molécula, el átomo pregunta si le es posible combinarse con la molécula de la parcela para formar una molécula de mayor tamaño. En caso de que esto sea posible el átomo está obligado a formar la molécula y luego debe desaparecer (morir), a fin de que no pueda volver a combinarse con otras moléculas (*principio de conservación de la materia)*. En caso de no poder combinarse con la molécula de la parcela, el átomo seguirá su camino en busca de una mejor oportunidad.
5. Cuando un átomo se combina con la molécula que hay en la parcela, su nombre se agrega de último a la lista de la molécula. Por ejemplo, si en la parcela hay una molécula de tipo ["O"] y es visitada por un átomo de tipo "H", en vista de que en nuestro modelo el hidrógeno se puede combinar con el oxígeno, la variable "molécula" de la parcela pasa del valor ["O"] al nuevo valor ["O" "H"], que luego del proceso de ordenación alfabética quedará como ["H" "O"].

Las anteriores reglas describen el mecanismo de formación de nuevas moléculas en la "sopa orgánica" del modelo. Las reglas establecen que las moléculas sólo se pueden formar por la interacción de una tortuga con una parcela, nunca por interacciones dos tortugas o dos parcelas. Las reglas 4 y 5 describen el mecanismo básico para la formación de moléculas y debemos detallarlo. Cuando una tortuga llega a una parcela donde hay una molécula alojada, debe formular la siguiente pregunta:

¿Es posible que el átomo que yo porto se pueda combinar con la molécula que hay en esta parcela a la que he llegado?

Hay que traducir esta pregunta a código de NetLogo en forma de una expresión condicional. Puesto que hay 3 tipos de átomos y 14 tipos de moléculas, existen 14 x 3 = 48 situaciones que se pueden presentar. Algunas de estas situaciones forman moléculas, otras no. Para que el átomo pueda responder la anterior pregunta, cuando llega a una parcela debe indentificar en cuál de estas 48 posibles situaciones se encuentra, lo que supondría un total de 48 expresiones condicionales que hay que traducir a código. Afortunadamente se puede reducir el número de expresiones condicionales a sólo 14 introduciendo lo que llamaremos "moléculas ampliadas". Para tomar la decisión de si debe combinarse o no, el átomo revisa una lista de 14 condicionales basadas en el concepto de molécula ampliada que explicaremos a continuación. Si el átomo se encuentra en una de las situaciones descritas por alguna de las 14 expresiones condicionales entonces se combinará, en caso contrario continuará su marcha. Dicho en palabras:

una molécula ampliada es una versión de la molécula que hay en la parcela, a la cual se le agrega la lista de átomos con que esa molécula se podría combinar.

Expresado en forma un poco más técnica: *una molécula ampliada consta de dos listas, la primera lista está formada por la molécula ordenada que se encuentra en la parcela y la segunda lista contiene los átomos con que la molécula ordenada se podría combinar.*
Un par de ejemplos terminarán de aclarar el punto:
Molécula ampliada del CO monóxido de carbono es: [["C" "O"] ["O"]]
La primera lista representa a la molécula de CO y la segunda lista contiene a "O" que es el único átomo con que a la molécula de CO se le permitiría combinarse (en este modelo).
Molécula ampliada del H_2O agua es: [["H" "H" "O"] [ ]]
La segunda lista está vacía porque a la molécula de agua no se le permite mezclarse con ningún otro átomo (en este modelo).
Enfatizamos el hecho que las moléculas ampliadas no se encuentran alojadas en las parcelas (no es una variable de tipo patches-own), sino un simple mecanismo de consulta en forma de expresiones condicionales que el intérprete verifica cada vez que un átomo llega a una parcela. He aquí el código completo:

```
turtles-own [átomo]
patches-own [molécula]

to setup
clear-all
crt tortugas [setxy random-xcor random-ycor] ;; tortugas se colocan al azar en el
mundo
ask turtles [set átomo one-of [″H″ "C″ "O″]] ;; cada tortuga escoge un átomo al azar
ask patches [set molécula one-of [[″H″] [″C″] [″O″] [″V″] [″V″] [″V″]
[″V″] [″V″] [″V″] ]] ;; parcelas escogen molécula monoatómica al azar o no alojar
;; ninguna
```

```
ask patches with [molécula != [″V″]] [set pcolor white] ;; parcelas que no están vacías
;; se pintan de color blanco
reset-ticks
end

to go
;; cada tortuga (átomo)  revisa en cuál de las 14 posibles situaciones se encuentra:
ask turtles [
  if molécula = first [[″C″]   [″H″ “O″]]  and
    member? átomo last [[″C″]   [″H″ “O″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″H″]  [″C″ “H″ “O″]]  and
member? átomo last [[″H″]  [″C″ “H″ “O″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″O″]   [″C″ “H″ “O″]]  and
  member? átomo last [[″O″]   [″C″ “H″ “O″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″H″ “H″]   [″O″]]  and
   member? átomo last [[″H″ “H″]   [″O″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″H″ “O″]   [″H″]]  and
  member? átomo last [[″H″ “O″]   [″H″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″C″ “H″]   [″H″]]  and
  member? átomo last [[″C″ “H″]   [″H″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″C″ “H″ “H″]  [″H″]]  and
member? átomo last [[″C″ “H″ “H″]  [″H″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″C″ “H″ “H″ “H″]   [″H″]]  and
  member? átomo last [[″C″ “H″ “H″ “H″]   [″H″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″C″ “H″ “H″ “H″ “H″]   []]  and
  member? átomo last [[″C″ “H″ “H″ “H″ “H″]   []]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″C″ “O″]   [″O″]]  and
  member? átomo last [[″C″ “O″]   [″O″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″C″ “O″ “O″]   []]  and
  member? átomo last [[″C″ “O″ “O″]   []]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″O″ “O″]   [″O″]]  and
  member? átomo last [[″O″ “O″]   [″O″]]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″O″ “O″ “O″]   []]  and
   member? átomo last [[″O″ “O″ “O″]   []]
[set molécula ordenar lput átomo molécula die]
if molécula = first [[″H″ “O″ “O″]   []]  and
    member? átomo last [[″H″ “O″ “O″]   []]
[set molécula ordenar lput átomo molécula die
set heading random 360 fd 1]
```

```
if count turtles = 0 [stop] ;; detener procedimientos si no quedan átomos libres
end

to-report ordenar [molec]
report sort-by [first ?1 < first ?2]  molec
end

to mostrar
ask patches with [first molécula != "V"] [show molécula]
end
```

Explicaciones y comentarios adicionales.
Analizaremos una de las 14 expresiones condicionales que constituyen el núcleo del código. Utilizaremos el ejemplo donde una tortuga que porta un átomo "O" de oxígeno llega a una parcela donde hay una molécula ["C"] de carbono, cuya moécula ampliada es [["C"] ["H" "O"]]. El átomo "O" inicia la revisión de las 14 expresiones condicionales basadas en las moléculas ampliadas, la primera de las cuales es:

if molécula = first [["C"] ["H" "O"]] and member? átomo last [["C"] ["H" "O"]]
[set molécula ordenar lput átomo molécula die]
Traducción: "Si la molécula en esta parcela es ["C"] y el átomo que portas es miembro de la lista ["H" "O"] agrega tu átomo de último a la lista ["C"], ordénala y muérete"
Esta situación calza a la perfección al átomo visitante "O", el cual se añadirá a la lista ["C"] y en la parcela quedará alojada la nueva molécula ["C" "O"] (la cual se somete al proceso de ordenación alfabética a pesar de que ya se encontraba ordenada).

Algunas consultas. Finalicemos mirando los resultados de algunas consultas en la Ventana del Observador, después de correr el modelo con una población inicial de 1000 tortugas y el mundo con su estructura (por omisión) de 1089 parcelas. Para saber cuántos átomos de C, H u O hay inicialmente en la sopa, hay que sumar los 1000 que portan las tortugas más los que están alojados inicialmente en las parcelas:

show count patches with [molécula != ["V"]]
==> 762, esto significa que en total el número de átomos es 1000 + 762 = 1762
Veamos cuántas moléculas de agua se han formado después de correr el modelo:
show count patches with [molécula = ["H" "H" "O"]]
==> 96, que corresponde a 5.4% de agua.
Examinemos la cantidad de dióxido de carbono:
show count patches with [molécula = ["C" "O" "O"]]
==> 102, no alarmarse, en esta etapa es posible que el dióxido de carbono sea incluso beneficioso para el proceso evolutivo de la vida. También encontramos 26 moléculas de metano CH4, pero ninguna de monóxido de carbono CO. Si se cambia la frecuencia de cada letra H, C, O, V, en la lista del procedimiento setup, se producirán cambios en la cantidad y tipos de moléculas que se forman.

Ejercicio 5.6: Agregar el nitrógeno “N” al modelo y algunas nuevas moléculas, por ejemplo: NO_2 dióxido de nitrógeno, NH_3 amoníaco, NH_4 amonio, HNO_3, ácido nítrico, HNO_2 ácido nitroso.

Promenade II.

Se podría decir que en el mundo actual de la programación hay dos clases de programadores. Por un lado, están los programadores profesionales, constituidos por personas con una fuerte formación en el campo de la computación. La mayoría de los programadores profesionales se han formado en universidades o escuelas de computación como ingenieros, científicos o técnicos en alguno de los diversos campos de la computación e informática y durante su formación adquirieron dominio de uno o varios de los grandes lenguajes de programación de propósito general como Pascal, C, C++, Lisp o Java, por mencionar sólo algunos posibles. Además del conocimiento de los lenguajes, los programadores profesionales han debido estudiar con profundidad varios otros aspectos relacionados con la tecnología de las computadoras (bases de datos, compiladores, sistemas operativos, entre otros temas). La otra clase de programadores está constituida por lo que podríamos llamar “programadores usuarios”. Este grupo de programadores, entre los cuales se encuentra el autor de este libro, lo integramos personas que somos usuarios de un lenguaje de programación de propósito específico. Aprendemos a programar en esos lenguajes para desarrollar proyectos en áreas específicas de nuestro interés, como podría ser MBA, matemática, estadística, finanzas, animación o diseño de páginas Web, entre otras opciones. Ejemplos de estos lenguajes son Logo, NegLogo, Mathematica, R, Excel, Blender, Scratch o HTML, entre muchos otros. Entre los programadores usuarios nos parece pertinente incluir también a personas que aprenden a programar en uno de los lenguajes llamados “scripting languajes”. Estos lenguajes permiten insertar código en varios tipos de aplicaciones para que realicen tareas específicas. Un ejemplo de scripting language lo constituye Javascript, cuyo código se puede insertar en las páginas Web para ser interpretado por los navegadores de Internet. Este libro está dirigido específicamente a programadores usuarios interesados en el campo de la programación basada en agentes utilizando el lenguaje NetLogo. No está dirigido a programadiores profesionales y ese es el motivo por el que en el libro se han incluido explicaciones sobre los aspectos más básicos de la programación. Los lectores y lectoras que quisieran profundizar un poco más en el apasionante mundo de la programación o en otros temas de las ciencias de la computación encontrán muchos recursos en la red. Una muy valiosa fuente de información, que no podemos dejar de recomendar, en la cual se exponen los principios fundamentales de la ciencia-ingeniería de la programación es el excelente libro de Abelson y Sussman [1], del cual hay versión pdf gratuita en la Web. Lamentablemente no existe aún –que nosotros sepamos- una traducción al español de este importante libro.

Interacción entre agentes II.

En el anterior encuentro con el tema de las interacciones entre agentes tratamos las interacciones basadas en proximidad espacial. En esta sección vamos a tratar las

interacciones basadas en la posibilidad que tienen los agentes de consultar o modificar los valores de las variables de otros agentes. A este respecto establezcamos algunos principios que puedan servir de guía.

1. Un agente puede consultar el valor de las variables de otro agente, aún cuando no es de su mismo tipo o familia, si especifica la identidad del otro agente.
2. Un agente puede modificar el valor de la variable propia de otro agente solicitando al otro que haga él mismo la modificación. La excepción a esta regla la constituyen las tortugas, quienes pueden cambiar los valores de las variables de la parcela sobre la que se encuentran como si fuera una de sus propias variables. Se vieron ejemplos de esta interacción en los modelos "Contando visitas, "La Feria del libro" y "La sopa de la vida".

El siguiente conjunto de órdenes sobre interacciones entre agentes ayudará a comprender el tema. Para mostrar los ejemplos se construye un pequeño ambiente donde hay agentes de varios tipos con sus variables. Las consultas se hacen desde la Ventana del Observador.

Ejemplo 27: Un ambiente para la consulta de variables entre agentes.

En este ejemplo se construye un pequeño ambiente para que algunos agentes realicen consultas sobre sus variables. En el ambiente hay: dos tortugas, una familia de tortugas llamada "lobos" integrada por un lobo, la totalidad de parcelas del mundo, y un enlace entre dos tortugas. Cada tipo de agente posee una variable propia: las tortugas poseen "edad", los lobos "velocidad", las parcelas "altura", los enlaces "elasticidad".

Definición del ambiente:

```
turtles-own[edad]
patches-own[altura]
links-own[elasticidad]
breed[lobos lobo]
lobos-own[velocidad]

to setup
clear-all
crt 2 [set color red set edad random 50 fd 8]
create-lobos 1 [set color white set velocidad 50 fd 5]
ask patches [set altura random 1000] ;; cada parcela tiene su propia altura
ask turtle 0 [create-link-with turtle 1] ;; se crea solamente un enlace
ask link 0 1 [set elasticidad 5]
end
```

Después de haber corrido los anteriores procedimientos que conforman el ambiente, escribimos las siguientes órdenes en la Ventana del Observador:

```
ask turtle 0 [show [edad] of turtle 1]
```

==> 32, la tortuga 0 consultó la edad de la tortuga 1. La primitiva "of" requiere que la variable "edad" se incluya entre corchetes para que reporte su valor.

ask turtle 0 [show edad]
==> 25, cuando un agente consulta el valor de una de sus variables propias los corchetes no son necesarios.
ask turtle 0 [show [edad] of turtle 0]
==> 25, "of" exige corchetes aún cuando un agente consulta su propia variable.
ask turtle 0 [show [edad] of self], otro ejemplo en que "of" exige corchetes.
==> 25
ask turtle 0 [show [edad] of myself], se genera mensaje de error, no se puede usar myself en esta situación.

Examinemos unos cuantos casos más entre agentes distintos:
ask turtle 0 [show [altura] of patch 8 8]
==> 325, tortuga muestra valor de variable de una parcela
ask turtle 1 [show altura]
==> 598, ¡sorpresa! la tortuga 1 muestra la altura de la parcela sobre la que se encuentra sin necesidad de explicitar la identidad de la parcela ni de usar "of" ni de encerrar "altura" entre corchetes. Esto se debe a que las tortugas pueden tratar las variables de la parcela sobre la que se encuentran como si fueran sus propias variables. Este privilegio de las tortugas no funciona a la inversa, de las parcelas hacia las tortugas y la razón es simple, la orden:
ask patch 0 0 [show edad], produce un error porque sobre la parcela podría haber varias tortugas y el intérprete no sabría a cuál de ellas está dirigida la pregunta. La forma correcta debe especificar al agente:
ask patch 0 0 [show [edad] of turtle 1], "of" exige corchetes a "edad"
==> (patch 0 0): 22
ask link 0 1 [show [edad] of turtle 1]
==> (link 0 1): 22
ask lobo 2 [show [elasticidad] of link 0 1]
==> 5, lobo consulta variable propia de enlace.
ask link 0 1 [show [velocidad] of lobo 2]
==> 50, enlace consulta variable propia de lobo.
ask link 0 1 [show elasticidad]
==> 5, no requiere corchetes pues se trata de un enlace preguntando por el valor de una de sus propias variables.
ask patch 8 9 [set altura [edad] of turtle 0], la parcela 8 9 fija su altura en 25, que es la edad de la tortuga 0.

Veamos ahora ejemplos donde un agente modifica el valor de una variable de otro agente. Comencemos por preguntar a la tortuga 1 por su edad:
ask turtle 1 [show edad]
==> (turtle 1): 34
Ahora la tortuga 0 quiere cambiar la edad de la tortuga 1:
ask turtle 0 [ask turtle 1 [set [edad] 99], comprobemos que realmente funcionó:
ask turtle 1 [show edad]
==> (turtle 1): 99

Además de hacer consultas sin pedir permiso, las tortugas también pueden modificar los valores de las variables de la parcela sobre la que se encuentran.
ask turtle 0 [show altura]

==> (patch -8 2): 932, la tortuga 0 se encuentra sobre la parcela (-8, 2).
ask turtle 0 [set altura 19]
ask patch -8 2 [show altura]
==> (patch -8 2): 19

Promenade III.

Las computadoras hicieron su primer ingreso en las aulas a finales de los años 80. En los años siguientes fuimos testigos de un espectacular desarrollo en la capacidad de las computadoras personales y en la aparición de numerosas aplicaciones de software, desarrollo que ha contínuado hasta nuestros días, extendiéndose al campo de las comunicaciones: hoy día un teléfono celular es una pequeña computadora portátil que además ofrece conectividad con el resto del mundo y acceso a la Web, un recurso que no existió sino hasta finales de los 90. Este impresionante desarrollo inevitablemente ha influido de muchas maneras en la forma como la tecnología ha modificado nuestras vidas y en la discusión sobre el rol de la tecnología de las computadoras y las telecomunicaciones en las aulas. Un viejo capítulo de esta discusión lo ha constituido el debate sobre la conveniencia o no de introducir el arte-ciencia de la programación en las aulas. Este debate también se remonta a finales de los años 80. Fue en esta época cuando surgieron los primeros "ambientes de programación" que también se llamaron "ambientes de apendizaje". Estos ambientes permiten a los estudiantes fijar su atención en proyectos ligados al curriculum escolar haciendo uso de un lenguaje de programación subayacente, el cual se constituye en herramienta y no en fin en sí mismo. Pero pese a no ser un fin en sí mismo de manera explícita, lo es de manera implícita pues el desarrollo de los proyectos exige que los estudiantes giren instrucciones a la máquina, ya sea en forma de texto o por medio íconos y el ratón. En cualquiera de los casos esto pone a los estudiantes en contacto con el razonamiento lógico y el acatamiento de un cierto número de reglas formales que el lenguaje subyacente impone. El ejemplo más conocido de este tipo de ambientes y posiblemente el primero en aparecer fue el lenguaje Logo, creado en 1967 por Wally Fuerzeig, Seymour Papert y Cynthia Solomon (a Papert se debió la posterior introducción del concepto de tortuga). La célebre frase de Papert "aprender con computadoras y no sobre computadoras" resume muy bien el enfoque de este ambiente de aprendizaje. Posteriormente han surgido otras alternativas, algunas de las cuales, como el ambiente Scratch, debido a Mitchel Resnick [10], facilitan la programación a estudiantes incluso desde la edad pre-escolar jugando a construir historias por medio de la manipulación de objetos gráficos. Aún hoy en día, la discusión sobre la introducción de la programación en las aulas nos sigue confrontando con preguntas como las siguientes:

1. ¿Cuáles ambientes de programación son apropiados?
2. ¿Cómo se deben introducir en el aula?
3. ¿Para qué edades es apropiado hacerlo?
4. ¿Cuántas horas semanales se recomienda dedicar a esta actividad?
5. ¿Se debe abrir espacio en el apretado calendario escolar, aún sacrificando tiempo de otras asignaturas?
6. ¿Es tiempo de hacer cambios en el curriculum para abrir espacio a la programación?

7. ¿Debe la programación ser para todos los estudiantes o sólo para grupos de interés y fuera del horario escolar?

Aún existe diversidad de respuestas a las preguntas anteriores y una variedad de valiosas opciones para los distintos niveles han surgido, desde el nivel escolar hasta el nivel universitario. Sin lugar a dudas NetLogo es una de estas opciones. No intentaremos dar respuesta a ninguna de las anteriores preguntas en este libro. Nos limitaremos a hacer algunas observaciones sobre puntos menos controversiales y enumeraremos algunos de los beneficios que creemos proporciona la actividad de programar.

La programación refuerza las siguientes áreas cognitivas:

1. **Fomenta la correcta expresión de las ideas y el uso del lenguaje con exactitud.** El código de un programa exige un alto grado de exactitud en la formulación de las unidades semánticas que lo conforman, similar al que exige las argumentaciones en las ciencias, particularmente en matemática.
2. **Constituye un entrenamiento en resolución de problemas.** Cada modelo presenta problemas que hay que resolver, para lo cual hay que diseñar y coordinar estrategias de resolución. La variedad de problemas que se presentan es prácticamente inagotable, como inagotables son los modelos que es posible concebir. La resolución de problemas atraviesa por las siguientes cuatro etapas:
    a. Identificación del problema.
    b. Diseño y formulación de una estrategia de solución en lenguaje ordinario (español, por ejemplo).
    c. Traslado de la estrategia de solución a código del lenguaje de programación.
    d. Verificación de que la solución propuesta funciona.

3. **Constituye un entrenamiento en el análisis de fallas.** La detección de las causas por las que secciones del código no producen los efectos previstos es un continuo ejercicio de análisis que requiere paciencia y perseverancia. De algún modo la detección y corrección de fallas también constituye un entrenamiento en resolución de problemas, aunque se trata de problemas en un dominio más restringido.
4. **Fomenta la concentración y focalización.** Los problemas que presenta la creación de un modelo o programa exigen momentos de considerable concentración y focalización, en los cuales la persona que programa debe aislar el foco de su atención de detalles ajenos al tema. La programación puede ser un ejercicio benéfico para las personas que padecen dificultades de concentración como por ejemplo el llamado “déficit atencional”.
5. **Provee de un entrenamiento para trabajar en equipo.** La programación ofrece una gran oportunidad para trabajar en equipo, pues un programa ofrece muchas formas de asignar distintas tareas a las personas que conforman un grupo que trabaja en un mismo proyecto.
6. **Ofrece un terreno fértil para el ejercicio de la creatividad.** Tratándos de una palabra de tan amplio y elusivo significado como lo es la palabra “creatividad”, la mejor respuesta que se nos ocurre dar a la pregunta *¿desarrolla la*

programación la creatividad? es la de echar un vistazo a los modelos creados en NetLogo, por ejemplo, en la Biblioteca de Modelos del menú Archivos o los muchos modelos disponibles en la Web. Si bien es muy difícil establecer cuando una actividad "desarrolla" la creatividad, sin duda podemos afirmar que la programación, lo mismo que la matemática, la pintura o la música ofrece un campo muy amplio para "ejercitar" la creatividad.

7. **Ventajas adicionales del modelado basado en agentes.** Si bien es cierto que los beneficios mencionados anteriormente son aplicables a cualquier lenguaje de programación, sea éste de propósito general o propósito específico, hay dos aspectos de la programación basada en agentes que aporta beneficios adicionales sobre otros lenguajes de programacón.
    a. En muchos de los modelos en que una tarea debe ser llevada a cabo por un conjunto de agentes, el problema no se resuelve simplemente diseñando una estragegia para un agente, replicándola para cada agente que el modelo utiliza. Con frecuencia se requiere el diseño de mecanismos de coordinación entre los agentes. Algunos de estos modelos ofrecen una excelente oportunidad para discusiones en clase, en donde los agentes se pueden representar por personas y las estrategias de organización se pueden formular e incluso "actuar teatralmente" en términos de experiencias cercanas a las vivencias cotidianas de los estudiantes.
    b. Uno de los aspectos que más se ha destacado del Modelado Basado en Agentes es que brinda la oportunidad de explorar el interesante tema del llamado "comportamiento emergente [14 y 15], el cual estudia la relación entre las interacciones a nivel individual o "micro" entre los agentes y la conducta global o macro que resulta de estas interacciones.
    c. Fuente inagotable de disfrute y esparcimiento creativo**.** Este punto no es muy frecuentemente tocado cuando se habla de los posibles beneficios de la programación. En este sentido, el autor considera la programación igualmente cerca del campo de las artes que el de las ciencias: una fuente inagotable de gozo donde la creación emerge como la fusión de la libre imaginación y el rigor que requiere la escritura del código de un programa.

Tres ejemplos de proyectos para desarrollar en el aula.

Una de las ventajas de trabajar en el ámbito educativo con modelado basado en agentes es la facilidad con que es dable encontrar fenómenos de la Naturaleza o de la conducta humana, fenómenos incluso de la vida cotidiana, los cuales se prestan idóneamente para ser modelados en los ambientes de MBA como NetLogo. Ya anteriormente hemos esbozado una de las razones para ello: la presencia por doquier de fenómenos constituidos por agregados de entidades que poseen algún grado de autonomía. Presentamos a continuación tres ejemplos de lo que podrían ser proyectos para desarrollo colectivo en clase. Nos limitamos a presentar el tema de los proyectos e indicar unas pocas de las ventajas que podrían ofrecer al ser desarrollados como proyectos grupales. No se incluye el código de estos proyectos, el cual se deja como proyecto opcional a ser desarrollado por el tutor o tutora.

Proyecto #1: Un tornado destruye una bodega.
Una bodega que almacenaba materiales para reciclar es destruida por un tornado. La bodega contenía trozos laminados en porciones de tres materiales: aluminio, plástico y cartón, los cuales han quedado dispersos por un área considerable alrededor del punto donde se encontraba la bodega. Por fortuna el área alrededor de este punto es un descampado grande dedicado a la agricultura y totalmente despoblado, lo que facilitará la recuperación de los materiales. La empresa Reciclados Modernos, dueña de la bodega, tanto en atención a su responsabilidad ambiental como conocedora de que los materiales no han perdido su valor comercial por el hecho de estar diseminados, decide contratar a una compañía local para que se encargue de la recolección de los retazos y limpieza del terreno. La compañía recolectora VLT (jocosamente llamada por los lugareños Vivan Los Tornados) envía un grupo de sus empleados para que hagan la recolección a mano. Modelar el mecanismo de recolección pensando que los trozos han quedado diseminados en un área grande, la cual una persona no puede cubrir a simple vista (para lo cual se requiere coordinación entre varias personas).

Proyecto #2: Distribución de la mercadería en la bodega de la gran tienda on-line.
En uno de los centros de acopio de una compañía que vende artículos por Internet, continuamente están entrando los artículos que envían los proveedores y por otro lado están saliendo los pedidos de los artículos que los clientes han comprado. El dueño de la compañía, el multimillonario de nombre Beg Jezus, desea comparar dos tipos de estrategias de almacenamiento de los artículos, para lo cual en sus bodegas sigue dos modelos de almacenamiento distintos. En algunas bodegas se clasifican los artículos en categorías, asignándoles áreas y estantes donde se agrupan artículos del mismo tipo. Por ejemplo, cuando ingresa una cámara fotográfica, la misma se ubica en el sector de artículos electrónicos, subsector de audio y video y en los estantes para las cámaras y en gabinetes de las distintas marcas. En las bodegas del segundo tipo, un artículo que ingresa a la bodega es colocado en el primer estante desocupado que encuentra el empleado en su camino. Con esta estrategia uno podría encontrar en el mismo gabinete artículos tan dispares como son un colchón inflable, seguido de un DVD con la séptima sinfonía de Mahler y luego un frasco de pastillas contra el insomnio Dormilak (claro, no faltará quien piense que los tres artículos persiguen el mismo fin). Se trata de modelar ambas estrategias y diseñar un procedimiento que permita comparar la eficiencia de ambos métodos.

Proyecto #3: Rotación del inventario en un almacén.
Uno de los problemas que los medianos y grandes almacenes deben resolver es el de mantener al día el inventario de los productos que ofrecen para su venta. Desde una farmacia, una ferretería, un almacén que vende repuestos de automóvil hasta una gran tienda de departamentos, evitar que los clentes tengan que escuchar el estribillo “lo siento señora, en este momento no tenemos ese artículo, pero en unos días nos estaría entrando”, es una meta perseguida por los gerentes de la tienda. Discutir con los y las estudiantes cómo se debe organizar el control del inventario, cuáles variables asignar a cada artículo o cada categoría de artículos brinda un terreno muy rico para la planificación y discusión. Por ejemplo, en una farmacia o almacén en donde se venden alimentos, el control de las fechas de vencimiento de los artículos es un factor que debe

tomarse en cuenta en el modelo. Este es un ejemplo en donde puede resultar de gran provecho la práctica de implementar primero un modelo de pequeño tamaño (pocos artículos con pocas variables cada uno) antes de pasar a un modelo de tamaño mayor. El modelo se puede desarrollar en forma incremental, haciéndolo más realista en cada nueva implementación. He aquí algunas de las variables que debería tener cada artículo: precio de compra, margen de utilidad, fecha de ingreso, y fecha de vencimiento.

Capítulo 6: Temas suplementarios.

Promenade IV.

El acomodador a la Condesa:
Estimadísima Condesa, si no es molestia y tuviera usted a bien, yo le solicitaría repetuosamente trasladarse un par de asientos hacia adelante y ocupar la silla número 25 que a su ilustrísima persona le ha sido reservada.

Lo mismo dicho en lenguaje NetLogo:
ask condesa 1 [forward 2 sentarse 25]

En los lenguajes de programación la exactitud y la brevedad constituyen la preocupación primordial y posiblemente sus mejores virtudes. En esto, los lenguajes de programación guardan similitud con el lenguaje de la matemática. Ambas disciplinas aman la precisión y detestan la ambigüedad. Tanto en matemática como en programación se busca que los conceptos y las ideas tengan un significado preciso y único, no sujeto a más de una interpretación[16]. Es en este sentido en que a la programación se la puede considerar una ciencia exacta. Esta similitud en el lenguaje de ambas disciplinas a veces puede inducir la errónea creencia de que para programar es necesario poseer muchos conocimientos de matemática. Hay ocasiones en que un modelo o programa en particular requiere de conocimientos matemáticos específicos, como se podría requerir de conocimientos específicos sobre química o sobre leyes. Sabemos que hay ciencias altamente matematizadas como la física, la estadística, la meteorología o la economía. Cuando hay que desarrollar programas en estas áreas, lo normal es que los conocimientos específicos en estos campos sean suministrados a los programadores por los especialistas en el campo y que ambos grupos, programadores y especialistas trabajen en equipo. Otras veces ocurre que el especialista en un campo determinado aprende a programar en un lenguaje específico para hacer sus investigaciones. Hay incluso quienes han ido más allá: el físico Stephen Wolfram inventó el lenguaje Mathematica para ayudarse en sus investigaciones sobre física y matemática. Por otra parte, también hay numerosos ejemplos de programadores que han construido sofisticados programas o grandes sistemas de información en muy diversos campos, sin que su nivel de conocimientos matemáticos rebasara el nivel de lo aprendido en la secundaria. Se trataba de proyectos que no demandaban de los programadores conocimientos específicos sobre matemática sino saber razonar con exactitud y una cierta dosis de ingenio para resolver problemas. La programación es un campo en donde la creatividad y la exactitud deben ir de la mano. Si se examinan los modelos de la Biblioteca de Modelos sorprenderá descubrir los pocos ejemplos en que se hace uso de conocimientos sobre matemática, incluso la más elemental. Para citar un ejemplo concreto, en el modelo llamado “Heroes y Cobardes” (carpeta IABM Textbook, Capítulo 2) hay que desempolvar un poco la geometría analítica de rectas vista en el colegio, para

[16] No obstante, algunos filósofos de la matemática, como Imre Lakatos, han cuestionado el hecho de que todas las afirmaciones de la matemática no formal tengan un significado único y preciso.

determinar las coordenadas del punto medio entre dos puntos dados del plano, algo que, de no recordarse, se puede consultar en un texto de matemática básica o incluso preguntar a alguna persona del entorno familiar o de trabajo. Por otra parte, no sería correcto pensar que tener conocimientos avanzados sobre matemática resulte completamente inútil en programación o, más aún, en ciencias de la computación. Si hablamos sobre personas que han contribuido al desarrollo de esta ciencia, no es casualidad encontrar que un número importante de estas personas venían del campo de la matemática. El gran matemático John von Neumann es uno de los padres de las computadoras modernas y el matemático John McCarthy fue el inventor del lenguaje Lisp. Como ejemplos más recientes y cercanos podemos citar a Seymour Papert (Logo) y Uri Wilensky (NetLogo), quienes se graduaron primero como matemáticos y a Stephen Wolfram (Mathematica) y Mitch Resnick (Scratch) quienes provenían del campo de la física.

Interacción del usuario con el modelo por medio del ratón.

En NetLogo es posible usar el ratón para interactuar con un modelo o programa. Para ello se dispone de cuatro primitivas que permiten esta interacción:

mouse-down? Esta primitiva reporta verdadero si el puntero del ratón se encuentra dentro del cuadrado del mundo y el botón izquierdo del ratón se encuentra oprimido. En caso contrario reporta falso.

mouse-inside? Esta primitiva reporta verdadero si el puntero del ratón se encuentra dentro del cuadrado del mundo y falso en caso contrario.

mouse-xcor. Esta primitiva reporta la coordenada "x" del puntero del ratón, cuando éste se encuentra dentro del cuadrado del mundo.

mouse-ycor. Esta primitiva reporta la coordenada "y" del puntero del ratón, cuando éste se encuentra dentro del cuadrado del mundo.

Ejemplo 28: Uso del ratón para detener acciones.

Primitivas: create-link-with, one-of, other (otro), mouse-down? (ratón-abajo?), mouse-inside? (ratón-dentro?), thickness (grosor), let, sprout.
Otros detalles: El usuario puede interactuar con el modelo por medio del ratón.

En este ejemplo se crean tortugas en aproximadamente una quinta parte de las parcelas y hay dos procedimientos independientes llamados "go" y "go2". En el procedimiento "go" se crean enlaces entre las tortugas cada segundo y se detiene el procedimiento introduciendo el puntero del ratón en el mundo y oprimiendo el botón izquierdo. En el procedimiento "go2" se crean enlaces cada medio segundo y se dejan de crear los enlaces con solo introducir el puntero del ratón en el mundo. Sin embargo, esta última acción no detiene el procedimiento, tan solo detiene el proceso de crear más enlaces. Para detener el procedimiento hay que pulsar de nuevo el botón "go2". La primitiva "thickness" (grosor) permite fijar el grosor de los enlaces.

Preparativos: Plantar botones “setup”, “go” y “go2”. Marcar la casilla “Continuamente” de los dos últimos procedimientos.

```
to setup
clear-all
ask patches [let estado one-of [0 1 2 3 4]
if estado = 1 [sprout 1] ]
;; en aprox. la quinta parte de las parcelas la variable estado es igual a 1
end

to go
ask one-of turtles [create-link-with one-of other turtles]
ask links [set thickness 0.2]
wait 1
if mouse-down? [stop]
end

to go2
if mouse-inside? [ask one-of turtles [create-link-with one-of other turtles
ask links [set thickness 0.2] ] ]
wait 0.5
end
```

Explicaciones y comentarios adicionales. La presencia de la primitiva “other” en la expresión: “create-link-with other turtles” tiene por objeto evitar que la tortuga intente crear un enlace consigo misma, lo cual generaría un error. Para poblar el mundo de modo que aproximadamente una quinta parte de las parcelas engendren (sprout) una tortuga, se crea una variable local (estado) y se asigna un valor a cada parcela, escogido de la lista de 5 valores [0 1 2 3 4]. Sólo las parcelas a las que se asignó el valor 1 engendrarán tortugas y estas constituyen aproximadamente la quinta parte del total.
En muchas situaciones los efectos que se logran por la intervención del usuario con el ratón sólo constituyen una opción cuyo idéntico efecto bien podría lograrse por medio de botones en la interfaz. Este es el caso de los dos procedimientos anteriores “go” y “go2”. En el siguiente ejemplo, la acción del ratón no se podría lograr por medio de botones.

Modelo 14: El temido regreso de la tortuga aniquiladora.

Primitivas: any?, one-of, other, turtles-on, neighbors (vecinas), mouse-down?
Otros detalles: El usuario puede interactuar con el modelo por medio del ratón.

Un conjunto de 50 tortugas ocupa posiciones estáticas en el mundo y una de ellas, la tortuga asesina, realiza una trayectoria de tal modo que aquellas tortugas que se encuentran en parcelas vecinas a su paso son eliminadas. El usuario puede crear nuevas tortugas en cualquier momento y en el punto que desee, oprimiendo el botón del ratón en el punto escogido dentro del cuadrado del mundo. El procedimiento se detiene cuando la tortuga aniquiladora ha acabado con todas las demás tortugas.

Preparativos: Plantar botones “setup” y “go”. Marcar la casilla “Continuamente” del botón “go”.

```
to setup
clear-all
crt 50 [setxy random-xcor random-ycor]
end

to go
ask turtle 0 [fd 1 set heading random 30 ]
ask turtle 0 [if any? other turtles-on neighbors [ask one-of turtles-on neighbors
[die]]]
wait 0.1
if count turtles = 1 [stop]
nacimiento
end

to nacimiento
if mouse-down? [ask patch mouse-xcor mouse-ycor [sprout 1]]
end
```

Explicaciones y comentarios adicionales. Este es el primer ejemplo en que hemos usado la primitiva “neighbors”, la cual reporta todas las parcelas vecinas al agente que da la orden. En la topología de la rosquilla cada parcela tiene 8 vecinas. La orden “ask one-of turtles-on neighbors” se traduce como “pedir a una tortuga en alguna de las parcelas vecinas”. La primitiva “neighbors4” reporta solamente las parcelas vecinas por los 4 lados, sin incluir las de las esquinas. La orden “set heading random 30” tiene como efecto que la tortuga avance zigzagueando, en una trayectoria aproximadamente recta, con oscilaciones de un máximo de 30 grados. Este zigzagueo hace muy difícil que alguna tortuga pueda escapar a la tortuga aniquiladora.

En las páginas anteriores hemos podido conocer algunas de las maneras como NetLogo permite a los usuarios ingresar información y las áreas hacia donde dirige la salida. En las dos secciones siguientes completaremos este tema, mostrando otras opciones tanto para entrada como para salida de datos.

Opciones para el ingreso de datos.

En NetLogo el ingreso de datos se puede hacer de cuatro maneras posibles: 1. Mediante un procedimiento que acepta parámetros de entrada, 2. A través de una ventana de entrada de datos, 3. A través de la primitiva “user-input” y 4. Leyendo datos desde un archivo.

1. Procedimientos con parámetros. Este modo fue descrito en el capítulo 3, sección “Procedimientos con datos de entrada”.
2. Objeto “Ventana Entrada”. Esta ventana es una de las opciones que ofrece el ícono “Seleccionador” de la Barra de Herramientas de la interfaz bajo el nombre “Entrada”. En el momento en que se planta un objeto tipo Entrada se despliegan

dos ventanas, una ventana de edición y una ventana de menor tamaño para ingreso de datos. En la ventana de edición hay que escoger un nombre para la Ventana de Entrada de datos, así como el tipo de dato que se va a almacenar en ella. Las opciones son:

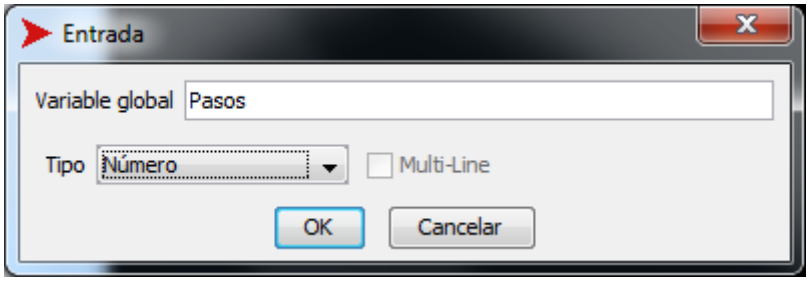
a. Número
b. texto
c. color
d. texto de código
e. texto de función

El valor por omisión es "texto". Se cierra la ventana de edición haciendo clic sobre el botón OK. Si se desea volver a abrir la ventana de edición, se debe hacer clic derecho sobre la banda superior verde de la ventanita de entrada de datos.

Ejemplo 29: Ingreso de datos con la opción Ventana de Entrada.

Detalle: Se planta un objeto tipo "Ventana de Entrada" y se ingresan datos por la misma.

Preparativos: Mediante el Seleccionador de Objetos, seleccionar una Ventana de Entrada escogiendo la opción "Entrada" y plantarla en la interaz. En la ventana de edición, a la derecha de "Variable global" escribir el nombre "Pasos" y en la casilla "Tipo" escoger la opción "Número". Pulsar el botón "OK".



Una pequeña ventanita verde, como la que se muestra en la figura, queda plantada en la interfaz:

Ingresar el valor 10 en la ventanita. Seguidamente escribiremos un pequeño procedimiento en el área de edición de código, en el que se usa la variable numérica "pasos" definida en la Ventana de Entrada:

```
to andar
clear-all
crt 3
ask turtles [fd pasos]
end
```

Si el procedimiento “andar” se llama desde la Ventana del Observador, se crearán 3 tortugas, las cuales avanzarán 10 pasos.

3. Ventana “user-input”. Cuando la primitiva se invoca, al momento en que el intérprete lee la primitiva “user-input”, entonces emerge una ventana en la interfaz donde el usuario puede ingresar datos. La ventana debe tener un nombre y un contenido. El dato ingresado en la ventana (el contenido) se comporta como una variable local y es del tipo cadena (string). Sin embargo, es posible ingresar variables de otro tipo si se les aplica antes la primitiva “read-from-string”, la cual cambia el tipo de la variable al adecuado. El formato de la primitiva es: user-input “nombre” donde “nombre” es el nombre de la ventana. Cuando la ventana emerge, se ingresa el dato y se valida la acción pulsando el botón OK.

4. Lectura de datos de un archivo de texto. Para ello se deben usar las primitivas “file-read”, “file-read-line”, “file-read-characters n”, “file-at-end?”. La cadena es leída del archivo como si la hubiéramos escrito en el centro de mandos. Es preciso abrir el archivo previamente mediante la orden “file-open”.

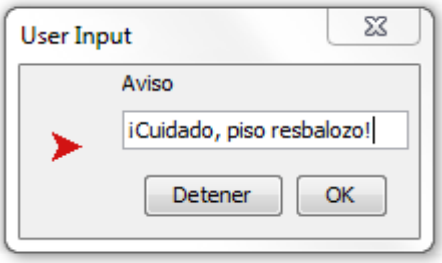
Ejemplo 30: ingreso de datos con la primitiva “user-input”.

Primitivas: user-input, read-from-string
Otros detalles: Se ingresan datos por la ventana emergente que genera “user-input”.

Los dos procedimientos “tener-cuidado” y “andar” muestran el uso de la primitiva “user-input” con variables de dos tipos diferentes: texto y número.

Ejemplo 30 a.

```
to tener-cuidado
clear-all
show user-input “Aviso”
end
```



Efecto: el texto ingresado en la ventana emergente llamada “Aviso” es enviado a la Terminal de Instrucciones. En ese ejemplo el texto ingresado es “¡Cuidado, piso resbalozo!”.

Ejemplo 30 b.

to andar
clear-all
crt 5
ask turtles [fd read-from string user-input "Pasos"]
end

Efecto: La primera tortuga avanza el número ingresado en la ventana emergente "Pasos", la cual permanece abierta para recibir el número de pasos de la segunda tortuga y así sucesivamente hasta terminar con la última tortuga. Si se desea interrumplir este proceso se debe oprimir el botón "Detener". La primitiva "read-from-string" convierte la cadena ingresada a tipo numérico.

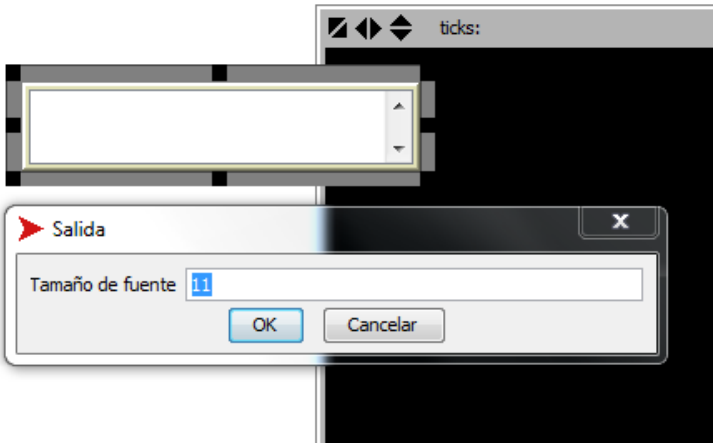
Ejemplo 30 c.
Si qusiéramos guardar el contenido de la ventana user-input para alguna acción ulterior, podemos hacerlo usando una variable local o global. Ahora lo guardaremos en una variable local llamada "institución":

to bienvenida
clear-all
let institución user-input "Nombre de organismo"
type "Estimados amigos del organismo " type institución type " os damos una calurosa bienvenida"
end

Aquí el nombre de la ventana es "Nombre de organismo" y su contenido es almacenado en la variable local llamada "institución". Si, por ejemplo, en la ventana user-input ingresamos el texto "Ministerio de Educación", en la Terminal de Instrucciones veríamos la salida: "Estimados amigos del Ministerio de Educación, os damos una calurosa bienvenida." Notar que el espacio en blanco después de la palabra "organismo" es necesario para separar las palabras.

Dirigiendo la salida a otras áreas.

En lo que atañe a la salida de datos, hasta ahora hemos visto dos opciones, la cuales son: 1) la Terminal de Instrucciones y 2) dirigir la salida a un archivo de texto sin formato. Vamos a añadir una tercera opción que consiste en dirigir la salida a un objeto tipo "Ventana de Salida". Para poner en marcha esta opción, lo primero que se debe hacer es plantar la ventana en la interfaz, siguiendo el mismo método usado para plantar botones: se selecciona primero la opción "Salida" y luego se planta el objeto. Cuando se planta una Ventana de Salida, debajo de la misma emerge una segunda ventana en que se pide al usuario ingresar el tamaño de la fuente que desea para el texto de salida (el valor por omisión es 11 puntos).



La Ventana de Salida se puede ampliar o mover de lugar mediante las marcas cuadradas en sus lados. Para dirigir la salida a la Ventana de Salida se emplean cualquiera de las primitivas output-show, output-type, output-write o output-print.

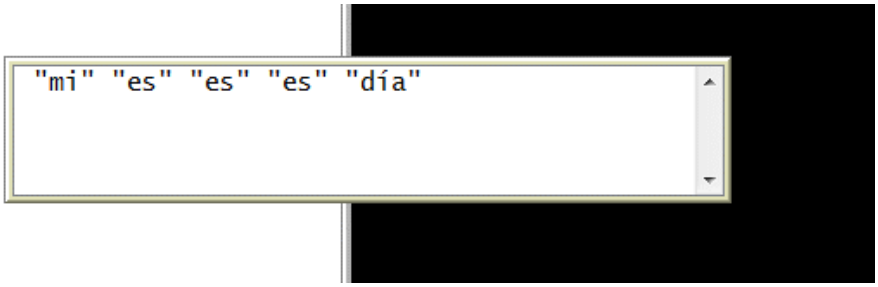
Ejemplo 31: Enviando la salida a un objeto tipo “Ventana de Salida”.

Primitivas: output-write, one-of, let, while (mientras).
Otros detalles: Se utiliza la primitiva “while” para producir iteraciones y se planta una ventana de salida.

Se generan palabras al azar de la frase [“Hoy” “es” “mi” “día” “de” “suerte”] y se escriben en la Ventana de Salida. El ejemplo se corre llamando el procedimiento “setup” desde la Ventana del Observador. He aquí el código:

Preparativos: Plantar una Ventana de Salida.

```
to setup
  clear-all
escribir
end

to escribir
while [random 5 != 3]
[let mensaje ["Hoy" "es" "mi" "día" "de" "suerte"]
  output-write one-of mensaje]
end
```



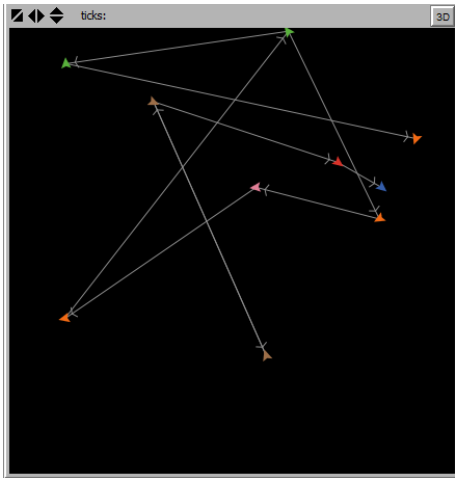
Explicaciones y comentarios adicionales. En este ejemplo introducimos la primitiva "while" (mientras). Esta primitiva permite crear mecanismos de repetición de bloques de órdenes. Su formato es *while [condición] [bloque de órdenes]*. Mientras la condición se cumple el bloque de órdenes se ejecuta. Cuando la condición deja de cumplirse el bloque de órdenes se ignora y el intérprete prosigue con las órdenes siguientes al bloque. En el ejemplo la condición de "while" consiste en sacar un entero al azar del conjunto (0, 1, 2, 3, 4) y verificar que dicho entero es diferente de 3. Cuando éste es el caso se escoge al azar (one-of) una de las palabras de la lista "mensaje" y se envía a la Ventana de Salida.

Expresiones "ask" encapsuladas.

En la programación basada en agentes es frecuente encontrar situaciones en que un agente pide a otro agente que haga algo o incluso situaciones en que un agente le pide a otro agente que le pida a otro agente que haga algo. Este tipo de situaciones se suelen resolver mediante órdenes que consiste en varios niveles de primitivas "ask" encapsuladas. La mayoría de las veces no se conoce la identidad (el número who) de los agentes que participan en las órdenes, pues las mismas están dirigidas a conjunto-agentes constituidos por más de un agente. El ejemplo siguiente muestra dos órdenes "ask" encapsuladas en un procedimiento que construye un grafo aleatorio con 20 nodos. Un grafo o red consiste en un conjunto de puntos llamados "nodos", algunos de los cuales se conectan entre sí por medio de líneas llamadas "aristas". Podemos representar fácilmente los nodos en NetLogo usando las tortugas como nodos y los enlaces como aristas. En los últimos años y gracias a la tecnología de las computadoras, las redes se han convertido en un tema central de estudio e investigación en las ciencias económicas y sociales [9] así como en ingeniería, matemática pura [4] y matemática aplicada. NetLogo resulta particularmente adecuado para construir modelos basados en redes, tanto así que existe una extensión dedicada a este tema.

Ejemplo 32: Un grafo aleatorio I.

```
to grafo
clear-all
crt 10 [setxy random-xcor random-ycor]
ask turtles [ask one-of other turtles [create-link-to myself]]
end
```



El encapsulamiento también es común en el lenguaje ordinario. Si se conoce la identidad de las personas que intervienen en una oración con encapsulamiento, hay menos riesgo de que la oración resulte confusa. Por ejemplo: *"Dile a Pedro que le diga a Carmen que le pregunte a Roberto que si va a invitar a Marcela a la fiesta".* Una orden NetLogo con tres niveles "ask" encajados, en que se conoce la identidad de los agentes a los que se ordenan actuar, podría ser la siguiente:

ask turtle 0 [ask turtle 1 [ask turtle 2 [show distance turtle 0] create-link-with turtle 1]]

El siguiente ejemplo muestra un encapsulamiento de cuatro niveles, en donde se establecen enlaces entre dos parejas de tortugas cuyas identidades (sus números who) se conocen. Se asignan colores distintivos a las tortugas para verificar mejor las identidades de las mismas.

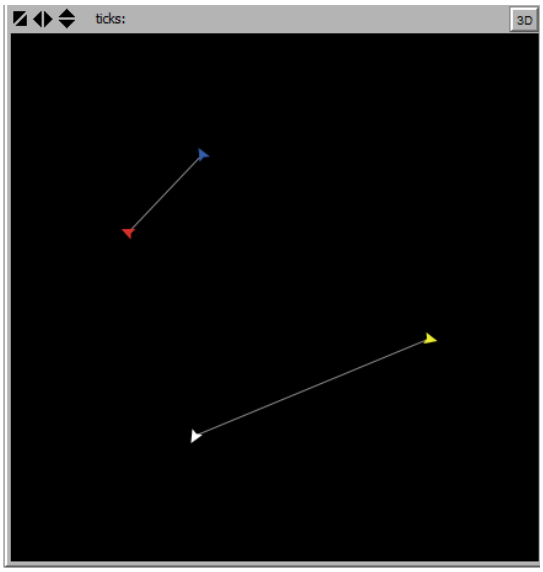
Ejemplo 33: Encapsulamientos con agentes cuya identidad es conocida.

```
to ligaduras1
clear-all
crt 4 [fd 10]
ask turtle 0 [set color white ask  turtle 1
[set color yellow ask turtle 2
[set color red ask turtle 3 [set color blue
create-link-with turtle 2] ]
create-link-with turtle 0]]
end
```

La primitiva "myself" se puede usar de modo que haga referencia a un agente cuya identidad es conocida, como en la siguiente versión, cuyo código es equivalente al del ejemplo anterior.

```
to ligaduras2
clear-all
```

crt 4 [fd 10]
ask turtle 0 [set color white ask turtle 1 [set color yellow ask turtle 2
[set color red ask turtle 3 [set color blue create-link-with myself]]
create-link-with myself]]
end

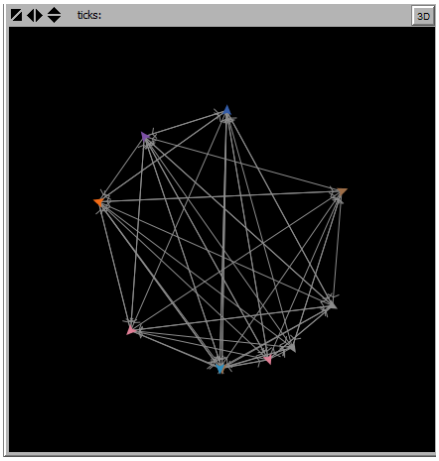


En las dos versiones anteriores los enlaces se producen entre las tortugas 0 y 1, blanca y amarilla y entre las tortugas 2 y 3, roja y azul. El orden en que aparecen los corchetes regulan las relaciones entre los agentes, del mismo modo que lo hacen los paréntesis en las fórmulas matemáticas.

Hay ocasiones en que hay que emitir órdenes sin conocer la identidad de los agentes, como cuando la dueña del negocio le dice a su empleado: *"Dile a cada cliente que llega a la tienda que llame a uno de sus familiares y que éste le pida a una de sus vecinas que forme un equipo con él, para que puedan participar en el concurso que estamos organizando en la tienda".* En esta orden con encapsulamiento no se conoce quién es el cliente ni quién será su familiar y menos aún quien es la vecina de su familiar. La efectividad de la orden reside precisamente en el anonimato de esas personas, pues eso es lo que permite aplicarla a muchas personas, las cuales aún no sabemos quiénes son. Sus identidades quedan resueltas conforme la orden se ejecuta, es decir, conforme se van presentando los clientes en la tienda. Al final del día las identidades ya serán conocidas y se podría incluso elaborar una bitácora: la cliente Marta llamó a su primo Ernesto quien le pidió a su vecina Ramona que formara equipo con él. Luego llegó el cliente Fernando quien llamó su hermana Ruth, etcétera. En la construcción de modelos es importante poder construir órdenes con encapsulamiento donde los agentes son anónimos. NetLogo puede soportar varios niveles de encapsulamiento (en principio una cantidad ilimitada). El siguente ejemplo contiene una orden con tres niveles de encapsulamiento, donde los agentes son anónimos. El resultado es nuevamente un grafo aleatorio.

Ejemplo 34: Grafo aleatorio con encapsulamiento y agentes anónimos.

```
to grafo
clear-all
crt 10 [fd 10]
ask turtles [ask other turtles [ask one-of other turtles [create-link-to myself ]]]
end
```



La figura anterior muestra el resultado de: “pedir a las tortugas que le pidan a cada una de las otras tortugas que le pidan a una de las otras tortugas crear un enlace con ellas”. Si bien es cierto que muchos paréntesis en los lenguajes de programación no confunden al intérprete, para nosotros los humanos es más difícil el manejo e interpretación de expresiones con muchos paréntesis encapsulados y agentes anónimos. Un recurso que podemos usar en Netlogo para controlar las relaciones entre los agentes en expresiones de este tipo es utilizar variables locales como especie de comodines[17] (jokers) o puntos de apoyo para dar nombres temporales a algunos agentes anónimos, como en el ejemplo siguiente.

Ejemplo 35: Un grafo aleatorio usando variables comodines.

```
to setup
clear-all
crt 20 [setxy random-xcor random-ycor]
go
end

to go
ask turtles [let Johnny one-of other turtles
ask Johnny [create-link-to myself]]
```

[17] Nombre que se da en algunas partes a las cartas del naipe que pueden tomar el valor que el jugador quiera darles.

end

El nombre Johnny es usado por cada tortuga y luego desechado para que lo use la tortuga siguiente en la orden "ask turtles". Cada una de las 20 tortugas del conjunto-agentes "turtles" se llama Johnny en algún momento, pero nunca hay dos tortugas usando ese nombre en un mismo momento. Una variable comodín también puede usarse como alias de un conjunto-agentes. Si llamáramos "amigas" al conjunto-agentes de "las otras tortugas" sería perfectamente lícito escribir:
ask turtles [let other turtles amigas ask amigas [create-link-to myself]].

Primitivas relacionadas con operaciones sobre listas.

NetLogo tiene varias primitivas que realizan operaciones sobre las listas. Presentaremos aquí cuatro de ellas: map, foreach, filter y reduce. La combinación de estas primitivas con el símbolo de interrogación "?" permite construir expresiones que aumentan el poder expresivo del lenguaje NetLogo.

map.
map tarea-reportadora lista1 lista2...
Lo que esencialmente hace map es aplicar una operación o tarea a los elementos de una o varias listas. El resultado de aplicar la tarea debe reportar un valor. Por ejemplo, la aplicación de la función abs (valor absoluto) a una lista de números mediante "map":
show map abs [1 -2 -3]
=> [1 2 3], la primitiva map aplica o "mapea" la primitiva abs a cada número de la lista.
show map is-number? [1 "banana" 3]
==>true false true, pues "banana" no es un número.

En el siguiente ejemplo utilizamos la primitiva "?". Este símbolo juega un papel parecido al de una variable (por ejemplo x) en una fórmula matemática. La expresión ? * ? puede verse como x * x, con la salvedad de que ? tomará uno a uno los valores de la lista de entrada.
show map [? * ? - 1] [1 2 3]
=> [0 3 8], equivale a la fórmula x * x – 1. En efecto: 1 * 1 - 1 = 0, 2 * 2 - 1 = 3, 3 * 3 - 1 = 8.
Con múltiples listas éstas deben tener el mismo tamaño.
show (map * [1 2 3][4 5 6])
=> [4 10 18], como la operación * requiere de dos entradas, por eso se se requieren dos listas, a fin de ir tomando por turno un elemento de cada una: 1 * 4 = 4, 2 * 5 = 10, 3 * 6 = 18.
Esta orden también se podría escribir como:
show (map [?1 * ?2] [1 2 3][4 5 6])
=> [4 10 18], en donde "?1" indica los elementos de la primera lista y "?2" los de la segunda.
Un ejemplo con tres listas requiere una tarea con tres argumentos, por ejemplo:
show (map [?1 + ?2 > ?3][1 2 3][3 3 1][5 3 8])
=> [false true false], 1 + 3 > 5 reporta falso, 2 + 3 > 3 reporta verdadero, etc.
Un ejemplo con cuatro listas:

show (map [?1 * ?2 = ?3 * ?4] [1 2 4 3][4 5 3 6] [3 3 2 2][1 3 6 9])
=> [false false true true], el primero es falso porque 1 * 4 no es igual a 3 * 1, el tercero es verdadero porque 4 * 3 = 2 * 6.

Como se ha podido apreciar, el número de listas que sigue a la tarea reportadora debe coincidir con el número de entradas (argumentos) de la tarea reportadora y todas estas listas deben ser del mismo tamaño.

foreach
Formato: foreach lista comando.
Con una sola lista ejecuta el comando aplicado a cada ítem de la lista:
foreach [1 2 3] write
==>1 2 3, la tarea está dada por la primitiva write, la cual se aplica a cada uno de los miembros de la lista [1 2 3].
foreach [1 2 3] [write 2 * ?]
==>2 4 6, el símbolo ? se sustituye por cada uno de los miembros de la lista. Esta primitiva también se puede usar con dos o más listas. Las listas deben tener el mismo tamaño.
(foreach [1 2 3] [2 4 6] [show word "the sum is: " (?1 + ?2)])
==>observer: "the sum is:3"
==>observer: "the sum is:6"
==>observer: "the sum is:9"
La primitiva word reporta la fusión de dos cadenas como una sola cadena. En este caso se trata de la expresión entre comillas y la expresión: ?1 + ?2 reporta la suma de los respectivos números de ambas listas: 1 + 2, 2 + 4 y 3 + 6.

filter.
Filtra una lista reportando sólo los miembros que cumplen una condición que reporta verdadero o falso. Los tres ejemplos que brinda el diccionario explican bien la función de esta primitiva:
show filter is-number? [1 "2" 3]
=> [1 3], la orden muestra la lista cuyos miembros cumplen la condición de ser números.
show filter [i -> i < 3] [1 3 2]
=> [1 2], aquí el símbolo "->" actúa con el significado de "tales que", es decir, la orden se traduce como "mostrar los ítemes i de la lista [1 2 3] *tales que* el valor de i es menor que 3".
show filter [s -> first s != "t"] ["hola" "todo" "mundo"]
=> ["hola" "mundo"], se traduce como "mostrar los ítemes "s" de la lista ["hola" "todo" "mundo"] tales que el primer elemento de cada miembro "s" es diferente de la letra "t".

reduce.
Esta primitiva reduce los ítemes de una lista a uno solo, según la operación indicada como entrada.
show reduce + [1 2 3 4]
==>10, se reduce la lista en la manera: 1 + (2 + (3 + 4)) = 1 + (2 + 7) = 1 + 9 = 10.

Invitamos a los lectores a examinar los restantes ejemplos en el Diccionario de Primitivas de NetLogo.

La exportación e importación de datos de NetLogo.

Como hemos visto ya, el ambiente NetLogo está conformado por varios elementos o componentes. Algunas veces es deseable exportar un componente del modelo para integrarlo a otra aplicación o bien importar alguno de los componentes previamente exportados de un modelo. También es posible importar ciertos formatos de archivos gráficos para integrarlos a un modelo de NetLogo. Las acciones de exportación o importación se activan desde el menú Archivo, mediante las opciones "Exportar" o "Importar". Para cada opción se abre una ventana mostrando los posibles elementos elegibles. Varios de los componentes se pueden exportar haciendo clic derecho sobre sus respectivas áreas o ventanas o desde el código mediante órdenes.

Componentes exportables:

Exportar Mundo. Esta opción guarda en un archivo de datos los valores de las variables, el estado de las tortugas y las parcelas, la capa de dibujo y los gráficos, incluyendo algunos parámetros que definen la configuración del ambiente. La exportación también se puede realizar desde código indicando el nombre del archivo al que se desea exportar, con la orden *export-world "nombre-de-archivo"*.

Exportar Gráfico. Guarda el gráfico seleccionado como un archivo en formato PNG. Una ventana emergente permite seleccionar el gráfico que se desea exportar. Se puede efectuar la exportación haciendo clic con el botón derecho del ratón sobre el gráfico. La exportación también se puede realizar desde código indicando el nombre del archivo al que se desea exportar, con la orden *export-plot "nombre-de-archivo"*.

Exportar Todos los Gráficos. Guarda en un archivo todos los gráficos. La exportación también se puede realizar desde código indicando el nombre del archivo al que se desea exportar, con la orden *export-all-plots "nombre-de-archivo"*.

Exportar Vista. Guarda en un archivo en formato gráfico PNG el estado en el que se encuentra la vista, es decir, exporta "una foto" del cuadrado donde se encuentran las parcelas y habitan las tortugas. Se puede efectuar la exportación haciendo clic con el botón derecho del ratón sobre alguna parte de la vista. La exportación también se puede realizar desde código indicando el nombre del archivo al que se desea exportar, con la orden *export-view "nombre-de-archivo"*.

Exportar Interfaz. Guarda en un archivo en formato gráfico PNG el estado en que se encuentra la interfaz. La figura incluye no sólo la vista del mundo sino los elementos adicionales como por ejemplo los botones, deslizadores y gráficos, pero no incluye las barras de menús, la Ventanas del Observador ni la Terminal de Instrucciones. Se puede efectuar la exportación haciendo clic con el ratón sobre una zona de la interfaz libre de elementos. La exportación también se puede realizar desde código indicando el nombre del archivo al que se desea exportar, con la orden *export-interface "nombre-de-archivo"*.

Exportar Salida. Guarda el contenido del área adonde se haya dirigido la salida, ya sea la Terminal de Instrucciones o una Ventana de Salida. La exportación también se puede realizar desde código indicando el nombre del archivo al que se desea exportar, con la orden *export-output "nombre-de-archivo"*.

Diferencia entre enviar la salida a un archivo y exportar una componente de un modelo. Cuando se envía la salida a un archivo mediante las primitivas file-show, file-type, etc., el archivo debe haberse creado y abierto previamente, de lo contrario la orden no surtirá efecto alguno. Cuando se exportan elementos de un modelo mediante cualquiera de las tres opciones disponibles (opción "Exportar" del menú Archivo, botón derecho del elemento que se desea exportar o desde código con órdenes del tipo *export-elemento "nombre-de-archivo"*), el efecto equivale a una acción del tipo "guardar como" (save as), la cual provoca la apertura de una ventana para que se asigne nombre al archivo que va a ser creado y el cual contendrá la información que se desea exportar.

Componentes importables

En el menú Archivo, la opción Importar ofrece las siguientes cinco opciones:

Importar Mundo. Carga un archivo que contiene todos los datos que se han guardado en una exportación previa del mundo. La acción se puede realizar desde código mediante la orden *import-world "nombre-de-archivo"*.

Importar Imagen. Carga una imagen sobre la capa de dibujo del mundo ajustándola a su tamaño. Se explicó anteriormente que, aunque las tortugas pueden realizar e incluso borrar los trazos que realizan sobre esta capa, sin embargo, no pueden detectar dichos trazos, lo que les impide interactuar con los mismos. La acción se puede realizar desde código mediante la orden *import-drawing "nombre-de-archivo"*.

Para las tres restantes opciones: Importar Colores de Parcelas, Importar Colores de Parcelas en formato RGB e Importar Interfaz del Cliente Hubnet, dirigimos a los lectores al Manual [16].

Capítulo 7: Recursividad.

El esquema de recursividad.

En computación, un procedimiento recursivo es un procedimiento que se invoca a sí mismo en alguna parte de su propio código. Los procedimientos recursivos brindan un poderoso recurso para resolver ciertos tipos de problemas. La recursividad es uno de los mecanismos por medio de los cuales se puede lograr la repetición o iteración de un procedimiento o un conjunto de órdenes. Ya hemos empleado la primitiva "repeat" para construir iteraciones. Las primitivas "while" y "loop" también sirven a este propósito. Ahora examinaremos el recurso de programación conocido como "recursividad" o "recursión". En este libro, cuyo enfoque es eminentemente práctico, usaremos el concepto de recursividad en el sentido sintáctico: si un procedimiento se invoca a sí mismo directa o indirectamente diremos que el procedimiento es recursivo. Como primer ejemplo consideremos el siguiente pequeño procedimiento:

```
to iterar
crt 1
ask turtles [fd 1]
iterar
end
```

Este procedimiento comienza creando una tortuga, seguidamente se pide a todas las tortugas -al principio no hay más que una- que avancen un paso, después de lo cual el procedimiento se invoca a sí mismo, por lo que el intérprete lo ejecuta por segunda vez: vuelve a crear una tortuga -ahora hay dos- les ordena que avancen un paso y el procedimiento se invoca a sí mismo por tercera vez creando una tercera tortuga y así sucesivamente. El procedimiento continuará invocándose a sí mismo, creando más tortuga que avanzan un paso y en principio, si no existieran limitaciones físicas, el procedimiento continuaría repitiéndose por siempre. El procedimiento de este ejemplo no realiza ninguna tarea interesante, pero ilustra que NetLogo es un lenguaje que soporta la existencia de procedimientos que se invocan a sí mismos. En la práctica es común que un procedimiento recursivo no corra todas las veces de manera idéntica, ya que podría contener variables que cambian con cada llamada recursiva. También es común que en un procedimiento recursivo se incluya un mecanismo para que el procedimiento se detenga cuando se cumple una cierta condición. El siguiente ejemplo muestra un procedimiento que muestra en la Terminal de Instrucciones los enteros a partir de un entero que se suministra como entrada. El procedimiento se detiene al llegar al entero 101.

```
to mostrar-enteros [número]
if número > 100 [stop] show número
wait 0.2
mostrar-enteros número + 1
end
```

El procedimiento se llama asignando un valor numérico al parámetro “número”, por ejemplo “mostrar-enteros 10”. En la Terminal de instrucciones se muestran los números:
observer: 10
observer: 11
observer: 12
……..
observer: 99
observer: 100

La condición de parada detiene el procedimiento al llegar al valor 101.
En el Capítulo 2 vimos un ejemplo en que una tortuga traza un círculo y lo recorre incansablemente sin detenerse debido que su procedimiento “go” tenía la casilla “Contínamente” marcada. El mismo efecto se podría lograr si desmarcamos esta casilla y colocamos como última orden una llamada al procedimiento “go”:

```
to go
ask turtle 0 [fd 0.3 right 3]
wait 0.1
go
end
```

Procesamiento recursivo de una lista o una cadena.

Un mecanismo frecuentemente empleado en programación, es el procesamiento recursivo de una lista o cadena: el procedimiento inicia con la lista o cadena y en cada llamada recursiva, es decir, cada vez que el procedimiento se repite, se realiza alguna acción sobre la lista o cadena. Las acciones más comunes son eliminar o agregar un nuevo miembro a la lista o cadena. En el caso de listas vimos que la primitiva “map” también permite realizar ciertos tipos de operaciones sobre las listas. Seguidamente presentamos tres formas de exhibir un anuncio mediante un rótulo. En los tres casos el mecanismo de procesamiento del rotulo es recursivo.

Ejemplo 36: Impresión de un rótulo I.

En el presente ejemplo, se introduce el texto de un rótulo como parámetro de un procedimiento. Cuando el procedimiento se corre el texto es impreso en la Terminal de Instrucciones. El rótulo se representa como una lista de palabras (cada palabra del rótulo se debe encerrar entre comillas). El texto también se puede representar como una cadena, como se explicará seguidamente, lo cual nos dispensa de tener que usar las comillas para cada palabra. El procesamiento de la lista se realiza recursivamente de la siguiente manera:

1. Se imprimie el primer miembro de la lista “rótulo”: *type first rótulo.*
2. Se elimina el primer miembro que ya se imprimió: *set rótulo but-first rótulo*.
3. Se vuelve a imprimir el primer miembro de la lista, que ahora ocupa el segundo lugar de la lista (segunda palabra del rótulo) y así sucesivamente.
4. Cuando la lista se queda vacía se usa esto para detener el procedimiento.

```
to procesar [rótulo]
if rótulo = [ ] [stop] ;; condición de parada
type first rótulo
type " "  ;; se imprime un espacio en blanco para separar las palabras
wait 0.3
set rótulo butfirst rótulo
procesar rótulo
end
```

Explicaciones y comentarios adicionales.
Para correr el procedimiento, en la Ventana del Observador digitar, por ejemplo: *procesar ["Estimado" "cliente:" "Nos" "hemos" "trasladado" "150" "metros" "al" "sur" "de" "este" "local."]*. Como resultado, en la Terminal de Instrucciones se imprime la anterior frase con una separación de 0.3 segundos entre cada palabra. Las palabras se imprimen sin las comillas (la primitiva type no las incluye). El procedimieno se detiene cuando la lista queda vacía. Se puede evitar tener que encerrar cada palabra del rótulo entre comillas ingresando el rótulo como cadena en vez de lista: *"Estimado cliente: Nos hemos trasladado 150 metros al sur de este local",* pero sería necesario adaptar la condición de parada para cuando la cadena se quede vacía:

if rótulo = "" [stop], sería la nueva condición de parada.

Otra diferencia que notaríamos al correr el procedimiento con una cadena es que el rótulo se imprimiría caracter por caracter en vez de palabra por palabra, pues los ítemes de una cadena son sus caracteres. La inclusión de un parámetro, cuya función es contar el número de repeticiones del procedimiento recursivo se usa con frecuencia para construir la condición de parada.

Ejemplo 37: Un mensaje aleatorio.

Se imprimen caracteres de una cadena de manera aleatoria. El parámetro llamado "contador" lleva la cuenta de las llamadas recursivas y el procedimiento se detiene cuando este parámetro alcanza el valor de 50. El ejemplo se pone en marcha llamando al procedimiento "setup" desde la Ventana del Observador.

```
globals[mensaje]

to setup
clear-all
set mensaje "Lo mejor que le puede pasar a una persona es trabajar haciendo lo que
le gusta hacer"
procesar mensaje 1
end

to procesar [texto contador]
if contador = 50 [stop] ;; condición de parada
type item random 84 mensaje
wait 0.2
procesar texto contador + 1 ;;  llamada  recursiva
```

end

Explicaciones y comentarios adicionales: El procedimiento “procesar” es llamado desde el procedimiento “setup” dando a “texto” el valor de la variabla “mensaje” y a “contador” el valor 1. El mensaje del cual se extraen los caracteres tiene 84 caracteres. El texto obtenido es una cadena aleatoria de 50 caracteres sin significado alguno. Por ejemplo, en una corrida obtuvimos: “aqdrjnsdtae r ensu a e as eua aeuee h nra oehn t”.

En el siguiente ejemplo procesaremos un rótulo recursivamente de manera que aparezca escrito sobre las parcelas del mundo. Para ello usaremos la primitiva “plabel” (etiqueta de parcela).

Modelo 15: Un rótulo se mueve sobre el terreno.

Primitivas: user-input, length, item, pxcor, pycor, max-pxcor, reset-ticks, ticks, plabel (etiqueta-de-parcela), plabel-color (color-de-etiqueta-de-parcela) or, stop.
Otros detalles: Se utiliza una ventana para ingresar el texto que el usuario desea ver.

En este ejemplo modelaremos un rótulo que se mueve por el mundo de derecha a izquierda. El texto del rótulo es ingresado por los usuarios mediante la primitiva “user-input”, la cual, como se explicó anteriormente, abre una ventana para ingresar el texto.

Preparativos: Plantar los botones “setup” y “go” (marcando la casilla “Continuamente” de go), seleccionar un mayor tamaño para los caracteres del rótulo (por ejemplo 16) en el botón “Configuración” de la interfaz, para que el rótulo se vea más claro.

Plan general y problemas a resolver. Hay muchas maneras de modelar un rótulo móvil. El texto del rótulo será ingresado por los usuarios en la ventana emergente que abre la primitiva user-input y almacenado en la variable global “rótulo”. Se ha diseñado un plan en que a las parcelas del centro del mundo (patches with [pycor = 0]) se les ordenará mostrar un carácter del rótulo y conforme el procedimiento go se repite, cada parcela leerá los siguientes caracteres. Esto producirá el efecto de movimiento. Para determinar el carácter que una parcela debe leer se deben tomar en cuenta dos cosas: 1) el tiempo transcurrido desde que el rótulo ha comenzado a desplazarse por la pantalla, el cual lo determina la variable ticks y 2) la posición horizontal de la parcela, que viene dada por pxcor. En cada momento, el carácter que la parcela debe leer (plabel) viene dado por la expresión:
set plabel item (pxcor + ticks) rótulo
que se traduce como “fijar la etiqueda de la parcela como el ítem que ocupa la posición “pxcor + ticks” de la cadena rótulo”. Se debe controlar que no se invoquen parcelas que se salen del mundo por la izquierda “(pxcor + ticks) < 0” o por la derecha “(ticks + pxcor > length rótulo - 1)”. He aquí el código:

globals[rótulo]

to setup

```
clear-all
set rótulo user-input "Escribir el rótulo"
reset-ticks
end

to go
if ticks > max-pxcor + length rótulo [stop]
ask patches with [pycor = 0]
[ifelse (pxcor + ticks) < 0  or (ticks + pxcor > length rótulo - 1)
[set plabel "" stop]
[set plabel-color red set plabel item (pxcor + ticks ) rótulo]]
wait 0.3
tick
go
end
```

Explicaciones y comentarios adicionales: La primitiva max-pxcor reporta la coordenada "x" de las parcelas que ocupan las posiciones más a la derecha, es decir, cuyo valor de esta coordenada es máximo.

Recursividad de cola.

La historia de Ana. Supongamos que Ana vive en una comunidad en donde hay muchos niños y niñas y su tarea (su procedimiento) consiste en leerles cuentos, un servicio que Ana presta a la comunidad mediante apoyo municipal. Cada vez que un niño o niña la llama al teléfono, ella debe acudir a su casa y leer el cuento que le es solicitado. Un día Ana recibe una llamada de Carmen, quien dice a Ana que desea escuchar el cuento de Blanca Nieves. Ana comienza a leer el cuento a Carmen, pero cuando va por la página 6 recibe una llamada del niño José, quien quiere que le lean el cuento de Simbad el Marino. Ana interrumpe la lectura de Blanca Nieves, después de prometer a Carmen que volverá en algún momento a finalizar la lectura de Blanca Nieves, pero antes de anotar la dirección de José y el cuento que él desea, debe también anotar en su libreta la dirección de Carmen, avenida del Olmo 7, el nombre del cuento que le estaba leyendo y el número de página por donde iba cuando tuvo que suspender la lectura. Una vez donde José, comienza a leer Simbad el Marino, pero cuando va por la página 11 Ana recibe una llamada de Teresita, quien desea que le lean el cuento de Pulgarcito. Ahora Ana agrega a su libreta de apuntes la dirección de Teresita y el cuento que desea, pero además debe agregar la dirección de José, calle de La Paciencia 18, el cuento que le estaba contando y el número de página por donde iba cuando Teresita llamó. Si esto continuara ocurriendo y el número de niños fuera muy grande, llegaría un momento en que a Ana no le quedaría espacio en su libreta de apuntes. Algo similar ocurre cuando en un procedimiento recursivo, con cada nueva llamada del procedimiento hay que guardar el estado en que quedaron las llamadas recursivas anteriores. Ahora bien, Ana podría adoptar una estrategia diferente, la cual le permitiría trabajar usando sólo una pequeña hoja de papel, un lápiz y un borrador. Lo que Ana tendría que hacer es rehusarse a atender nuevas solicitudes antes de haber finalizado un cuento. De este modo, cuando atiende la llamada de Ramón, el siguiente niño, llamada que atiende

porque ha finalizado la lectura de un cuento, todo lo que necesita apuntar en su libreta es la dirección de Ramón y el cuento que solicita. Puede hacer esto en una sola hoja de papel. Cada vez que le llega una nueva solicitud borra los datos de la solicitud anterior y apunta los nuevos datos. De esta manera Ana puede desempeñar su trabajo y cumplir con todas las solicitudes empleando solamente una sola hoja de papel, un lápiz y un borrador. A esta forma de organizar un procedimiento recursivo se le conoce comúnmente como "recursividad de cola" pues la llamada recursiva se coloca en la cola, como última instrucción de procedimiento recursivo.

Las siguientes tres secciones están dedicadas a temas de matemática. Para su comprensión no se requiere mayores conocimientos de matemática que los vistos en los primeros años de la secundaria. Si el lector o lectora desea omitir la lectura de estos temas puede pasar, sin consecuencia alguna, directamente a la sección "Interacción del usuario con el modelo por medio del ratón" y continuar la lectura a partir de ese punto.

La recursividad aplicada a ejemplos numéricos.

La noción de recursividad es particularmente útil en algunos procedimientos que realizan cálculos matemáticos. De hecho, esta noción tuvo sus orígenes en la noción matemática de función recursiva, y uno de los primeros en usarla fue el gran lógico y matemático del siglo veinte Kurt Gödel, en la demostración de su famoso teorema de incompletitud [7]. Como ejemplo usaremos la recursividad para calcular la función factorial de un entero positivo. El factorial de un entero positivo se define como el producto de todos los enteros positivos menores o iguales al número dado y mayores que cero. Por ejemplo: factorial 3 = 1 x 2 x 3 = 6, factorial 4 = 1 x 2 x 3 x 4 = 24. Se suele convenir que el factorial de 1 es igual a 1. Mostraremos dos esquemas diferentes para calcular recursivamente el factorial de un entero positivo.

Ejemplo 38: Primer esquema.

En el primer esquema, el procedimiento "factorial [num]", cuya entrada "num" es el entero cuyo factorial se desea calcular, se encarga de lanzar el procedimiento recursivo "fact", el cual tiene tres parámetros de entrada:
"fact [num prod contador]"
El parámetro "num" de este procedimiento almacena el número cuyo factorial se pide calcular, "prod" almacena los productos que van formando el factorial y "contador" es el siguiente factor por el que hay que multiplicar a "prod" en cada pasada recursiva. He aquí el código:

```
to factorial [num]
fact num 1 1
end

to fact [número prod contador]
if contador = número [type prod stop] ;; condición de parada
let producto prod * (contador + 1)
fact número producto  contador + 1
```

end

Explicaciones y comentarios adicionales. A fin de entender lo que hace el intérprete, vamos a seguir la evolución del procedimiento factorial en un ejemplo. Calcularemos el factorial de 3.
Cuando escribimos "factorial 3" este procedimiento llama al procedimiento fact con los valores número = 3, prod = 1 y contador = 1. Veamos cómo evoluciona el procedimiento "fact 3 1 1".
Primera llamada recursiva: fact 3 1 1.
la condición 1 = 3 (contador = número) reporta falso.
let producto prod * (contador + 1) se convierte en let producto 1 * (1 + 1), o sea producto toma el valor 2
fact número producto contador + 1 es una llamada a fact con los nuevos valores:
fact 3 2 2:
Segunda llamada recursiva: fact 3 2 2
la condición 2 = 3 (contador = número) reporta nuevamente falso.
let producto prod * (contador + 1) se convierte en let producto 2 * (2 + 1), o sea producto = 6
fact número producto contador + 1 es una llamada a fact con los nuevos valores:
fact 3 6 3
Tercera llamada recursiva: fact 3 6 3
la condición 3 = 3 (contador = número) reporta verdadero,
se aplican las órdenes "type prod stop" que equivalen a "type 6 stop".
Se imprime 6 en la Terminal de Instrucciones y el procedimiento fact termina, pero el intérprete debe regresar al procedimiento factorial 3, desde donde fue llamado.
Puesto que la siguiente orden después de la llamada factorial 3 1 1 es "end" todo el proceso termina.

Ejemplo 39: Segundo esquema.

Consideremos el mismo ejemplo del factorial de un entero positivo pero empleando un esquema recursivo diferente. En este esquema nos ahorramos varias cosas: no se necesita un procedimiento lanzador desde donde se invoca al procedimiento recursivo "fact" y también se podrá prescindir de los parámetros adicionales "prod" para almacenar el producto y del parámetro "contador", que almacena el siguiente factor. Pero, como veremos, no todo son ventajas pues este esquema replica la conducta de Ana cuando tomaba llamadas sin haber terminado un cuento. He aquí el código:

```
to-report factorial [num]
ifelse num < 1 [report 1]
[report num *  factorial (num - 1)]
end
```

Veamos cómo evoluciona la evaluación de este procedimiento cuando se invoca "factorial 4".
Primera llamada recursiva: factorial 4
La condición 4 < 1 (num < 1) es falsa, por lo que se ejecuta el segundo corchete:

report num * factorial (num - 1), es decir
report 4 * factorial 3, que se convierte en
report 4 * (3 * factorial 2) , que se convierte en
report 4 * (3 * (2 * factorial 1)) que se convierte en
report 4 * (3 * (2 * factorial 0))
pero en el cálculo de factorial 0 ahora se tiene que 0 < 1 factorial 0 reporta 1
report 4 * (3 * (2 * 1))
report 4 * (3 * 2)
report 4 * 6
report 24

Sin duda sorprende lo pequeño y eficiente del código con este segundo esquema. Pero lo más importante de notar es que cada llamada recursiva, excepto la última, es interrumpida para tener que realizar la llamada siguiente y esta es interrumpida para realizar la siguiente y así hasta llegar a la llamada factorial 0. Este proceso, en que quedan operaciones pendientes en cada llamada recursiva supone un gasto creciente de memoria pues en cada llamada hay que acarrear el estado en que quedaron las llamadas anteriores. Si el número sobre el que se desea calcular el factoria fuera muy grande, la memoria podría colmarse y el procedimiento tendría que finalizar por falta de espacio. Con el primer esquema esto no llega a ocurrir porque la información necesaria para calcular cada siguiente llamada recursiva del procedimiento es almacenada en los parámetros “prod” y “contador” y es renovada cada vez que el procedimiento es invocado, de modo que el espacio de memoria utilizado es constante[18]. Este segundo esquema procede según el primer método que usaba Ana, la chica que contaba cuentos a los niños, cuando se dejaba interrumpir en medio de un cuento, mientras que el primer esquema corresponde al método que adoptó finalmente, llevando sólo una pequeña hoja de papel un lápiz y un borrador.

Los números primos.

Los dos siguientes ejemplos están dedicados a un tema de la matemática, a saber, generar listas de números primos [6] y en ambos se emplea el primer esquema de recursividad. A fin de mantener uniformidad con los nombres decidimos llamar a estos ejemplos modelos, aunque tal vez sería más apropiado llamarlos “programas”. Afortunadamente los conocimientos matemáticos necesarios para entender estos modelos son mínimos y no exceden el conocimiento de la división entre enteros que aprendimos en la escuela primaria. El principal reto no está pues, tanto en la matemática como en diseñar la estrategia de cálculo y luego plasmarla en código de NetLogo. Se examina si un número es primo tratando de encontrar divisores menores que el número. No obstante, si el lector o lectora decide saltar estos ejemplos, esto no afectará en nada la comprensión de los temas siguientes. En el primer modelo se construye una lista de números primos y en el segundo se extraen de esta lista los llamados “primos gemelos”. En la red se puede encontrar cuantioso material en pdf sobre teoría elemental de números y sobre los números primos (ver por ejemplo [6]).

[18] Salvo por el tamaño del número.

Comenzamos por recordar que un número entero N se considera primo si posee exactamente dos divisores, a saber, la unidad y el número mismo. Los primeros doce enteros positivos primos son: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 y 37. A los enteros que no son primos se les llama números compuestos. Todos los números compuestos poseen tres o más divisores. Al número 1 no se le considera ni primo ni compuesto pues posee un solo divisor. Por otra parte, a una pareja de primos que se encuentran muy cercanos, separados sólo por un entero, se le conoce como "primos gemelos". En la anterior lista de 12 números primos podemos encontrar 5 parejas de primos gemelos, a saber: 3 y 5, 5 y 7, 11 y 13, 17 y 19, 29 y 31. Se sabe desde hace más de dos mil años que existe un número infinito de primos. La primera demostración conocida se debe a Euclides, cerca de 300 años antes de nuestra era. En cambio, hasta el día de hoy no se sabe si existe un número infinito de parejas de primos gemelos. Este problema de tan sencilla formulación ha probado ser de una dificultad enorme y sin duda es uno de esos problemas que llenaría de gloria a quien logre resolverlo.

Una lista de números primos.

La manera más directa de averiguar si un número N es primo es buscando divisores entre los números menores que N y mayores que 1. Mostraremos tres métodos de implementar esta búsqueda, cada uno más eficiente que el anterior. El código de los tres métodos hace uso del esquema de recursividad de cola, pero sólo se mostrará el código de los dos últimos métodos.

Primer método: Para establecer si un entero N es primo, se prueba dividiendo N por cada uno de los enteros mayores que 1 y menores que el entero N. Si no aparece ningún divisor entonces se concluye que el entero N es primo. El mayor número de divisiones que habría que realizar ocurre precisamente cuando el entero investigado es primo, en cuyo caso hay que efectuar N - 2 divisiones (excluyendo las divisiones por 1 y por el entero N mismo). Por ejemplo, si no supiéramos de antemano que 101 es primo y quisierámos averiguarlo usando este método, sólo llegaríamos al resultado después de haber realizado 99 divisiones. Este método funciona de modo infalible, pero es ineficiente y –como veremos- se puede mejorar sustancialmente reduciendo el número de divisiones de prueba que hay que hacer.

Segundo método: Este método se basa en la observación de que cada vez que encontramos un divisor de un entero, en realidad hemos encontrado dos: tanto el divisor como el cociente o resultado de la división son divisores del número. En la escuela nos enseñan que si dividimos 24 entre 6, el resultado es 4 y el residuo es 0. Nos enseñana también a probar que la división es correcta multiplicando 6 x 4 y comprobando que da 24. Esta multiplicación nos muestra que 6 es divisor de 24 pero también muestra que 4 es divisor de 24 (aunque esta parte, en la escuela primaria, al menos a mí, nunca me la contaron). Vemos entonces que los divisores de un entero N vienen en parejas que podemos llamar *parejas de divisores complementarios*. Esta observación nos permite buscar los divisores de un entero centrando la búsqueda sólo en uno de los miembros de cada pareja. Para ello separaremos los divisores de un entero en dos tramos, de manera que si un divisor está en uno de los tramos su pareja o divisor

complementario está en el otro tramo. Por ejemplo, si quitamos el 1 y el 24, los dos tramos de la lista de divisores de 24 son:

Primer tramo: 2, 3, 4 segundo tramo: 6, 8, 12.

Efectivamente, cada divisor del primer tramo tiene una pareja complementaria en el segundo tramo: 2 tiene a 12 como divisor complementario, 3 tiene a 8 y 4 tiene a 6. Podemos aprovechar esto de la siguiente manera: 1) Para saber si un número tiene divisores basta con realizar la búsqueda en uno de los tramos ya que, si no encontramos ningún divisor en el tramo inspeccionado, podemos asegurar que tampoco hay divisores en el otro tramo, donde deberían estar los divisores complementarios (no existen divisores "solteros", sin pareja, salvo la excepción que ya veremos). 2) Es más rápido hacer la búsqueda en el primer tramo porque este tramo tiene menos números. Por ejemplo, en el caso de 24 en el primero tramo buscaríamos divisores en el intervalo que va del 2 al 4 (tres números), mientras que en el segundo tramo tendríamos que buscarlos entre los números que van del 5 al 23 (19 números). Un caso especial se da cuando el entero es un cuadrado perfecto. En este caso su raíz cuadrada es un divisor que se encuentran exactamente en la frontera entre los dos tramos. Por ejemplo, ocurre con el número 36, cuyos tramos de divisores son:

Primer tramo 2, 3, 4, 6 segundo tramo: 6, 9, 12, 18

Este es el único caso de un divisore cuya pareja es él mismo y nos dice que la raíz cuadrada del número marca la frontera entre los dos tramos. Con estas observaciones en mano podemos reducir drásticamente el número de pruebas que hay que hacer para establecer si un entero es primo: *Buscamos divisores en el primer tramo, el cual va de 2 al entero más cercano a la raíz cuadrada del número y que no exceda a dicha raíz. Si no aparece ningún divisor en este tramo, sabremos que el número es primo pues tampoco podría haber divisores en el segundo tramo.* Ilustremos esto aplicando esta idea al número 101, suponiendo que no sabemos que es primo. Como la raíz cuadrada de 101 es aproximadamente 10.049, el entero más cercano a 10.49 que no excede a este número es 10 (el primer número del segundo tramo es 11). Buscaremos divisores de 101 sólo entre los números de su primer tramo: 2, 3, 4, 5, 6, 7, 8, 9 y 10. Con esta nueva estrategia, en vez de tener que realizar 99 divisiones sólo tendríamos que realizar 9, una diferencia nada despreciable. Conforme el tamaño de N crece, la diferencia en el número de divisiones que hay que examinar entre ambos métodos aumenta dramáticamente. Para un número en el orden del millón, como por ejemplo el número primo 1000003 (un millón tres), con el primer método habría que hacer 1000001 (un míllón uno) divisiones mientras que con el segundo método sólo mil pues la raíz cuadrada de 1000003 es aproximadamente 1000. Esto supone una diferencia de 999001 (novecientos noventa y nueva mil uno) divisiones.

Tercer método: Con este método vamos a reducir aún más el número de divisiones. Buscaremos posibles divisores en el primer tramo, es decir, entre los números que no exceden a la raíz cuadrada de N, pero que además sean primos. La justificación de este último requisito es muy simple: si ya sabemos que N no es divisible por un primo, ¿por qué habríamos de buscar divisores de N entre los múltiplos de dicho primo?, aunque se encuentren en el primer tramo. Por ejemplo, si sabemos que N no es divisible por 2, evidentemente tampoco podría ser divisible por 4, por 6 o por ningún otro múltipo de 2. Por eso no probaremos a aquellos números del primer tramo que no son primos. Para establecer, por ejemplo, que 101 es primo bastaría comprobar que este número no es

divisible por ninguno de los primos del primer tramo que son 2, 3, 5 y 7, lo que reduce aún más el número de divisiones de 9 a sólo 4. Para el número 1000003 el número de divisiones se reduce de las 1000 del segundo método a 168 con este nuevo método, ya que sólo hay 168 primos que no exceden a la raíz cuadrada de 1000003. A contínuación mostramos el código de los métodos segundo y tercero. Se podrá comprobar la poca diferencia que existe en el código de ambos. Las pruebas de divisibilidad entre dos enteros se efectúan con la primitiva "remainder", la cual ya usamos anteriormente y la cual reporta el residuo de la división entre dos enteros. Un número es divisible por otro si el residuo de la división es igual a cero, es decir si "remainder N d" reporta el valor 0.

Modelo 16: generación de primos utilizando el segundo método.

Este algoritmo divide por los enteros menores que la raíz cuadrada del número n cuya primalidad está siendo examinanda. El programa se pone en marcha llamando al procedimiento "listar-primos" desde la Ventana del Observador.

```
globals[primos]

to listar-primos
clear-all
set primos [2]
generar 2 primos ;; se comienza probando como posible divisor a 2
end

to generar [num listado]
divide num 2
generar num + 1 primos
end

to divide [dividendo divisor]
if (remainder dividendo divisor) = 0 [stop]
if divisor > sqrt dividendo
[set primos se primos dividendo  stop]
divide dividendo divisor + 1
end
```

Explicaciones y comentarios adicionales. El procedimiento se debe detener manualmente con la opción "Detener" del menú Herramientas pues de lo contrario seguirá generando primos, los cuales no se envían a la Terminal de Instrucciones, sino que son almacenados en la variable "primos". Una vez detenido el procedimiento, se puede consultar la lista de primos generada o el tamaño de la lista en la Terminal de Instrucciones con las órdenes "show primos" o "show length primos" respectivamente.

Modelo 17: generación de primos utilizando el tercer método.

Este algoritmo prueba divisiones por los primos que son menores o a lo sumo iguales que la raíz cuadrado del número n cuya primalidad está siendo examinada. Los primos que hay que probar como posibles divisores se toman de la propia lista de primos que

el programa ha ido generando, la cual se va almacenando en la variable “primos”. El programa se pone en marcha llamando al procedimiento “listar-primos” desde la Ventana del Observador.

```
globals[primos]

to listar-primos
clear-all
set primos [2]
generar 2 primos
end

to generar [dividendo listado]
let i 0
divide dividendo i
generar dividendo + 1 primos
end


to divide [dividendo i]
let i-ésimo-primo item i primos
if (remainder dividendo i-ésimo-primo) = 0 [stop]
if i-ésimo-primo > sqrt dividendo [set primos se primos dividendo  stop]
divide dividendo i + 1
end
```

Se puede consultar la lista de primos generados en la Terminal de Instrucciones como se indicó en el modelo anterior. Sin embargo, resulta más cómodo leer grandes listas enviando la salida a un archivo externo. Ambos métodos pueden usar el siguiente procedimiento llamado “resultados”, para ver la lista de primos generada en el archivo “primos.txt”, pero sería necesario crear primero este archivo. En este ejemplo suponemos que el archivo será ubicado en el mismo directorio que el modelo. De no ser así habría que poner la ruta completa en la instrucción “file-open”.

```
to resultados
show last primos
file-open “primos.txt”
file-type “Este es un listado de algunos números primos.”
file-type “\r\n”
file-type “La lista contiene “ file-type length primos
file-type “ elementos”
file-type “\r\n”
file-write primos
file-close
end
```

Explicaciones y comentarios adicionales. La diferencia fundamental del tercer método con respecto al segundo consiste en que los divisores que el programa va probando se toman de la lista “primos” que el programa mismo va generando. El siguiente primo a probarse como posible divisor del número llamado “dividendo” se llama “i-ésimo-

primo" y su valor toma el del i-ésimo ítem de la lista "primos" en la instrucción: *let i-ésimo-primo item i primos*. El procedimiento "resultados" se puede invocar después de haber corrido el procedimiento "listar-primos" siguiendo cualquiera de los dos métodos[19]. En la colección de modelos del libro los dos métodos se encuentran bajo los nombres de "primos2" y "primos3". Nótese que hay dos procedimientos recursivos, cuyos nombres son "generar" y "divide". El procedimiento "generar" invoca a "divide", el cual al terminar de realizar su rutina recursiva regresa a "generar" para que este procedimiento se invoque a sí mismo con nuevos valores de sus variables. He aquí un trozo de la última parte del listado de 182239 primos generados por el método 3 en 10 segundos:

2482889 2482903 2482913 2482933 2482937 2482943 2482967 2482973 2482981
2482993 2482999 2483017 2483027 2483059 2483077 2483093 2483099 2483113
2483119 2483137 2483141 2483147 2483161 2483171 2483179 2483219 2483233
2483291 2483381 2483417 2483431 2483447 2483461 2483483 2483519 2483521
2483543 2483549 2483561 2483567 2483599 2483603 2483617 2483641 2483653
2483659 2483669 2483671 2483687 2483693 2483707 2483711 2483713 2483729
2483743 2483749 2483753 2483777 2483797 2483827 2483837 2483861 2483867
2483869 2483881 2483911 2483917 2483939 2483953 2484011 2484017 2484019
2484037 2484049 2484059 2484089 2484109 2484113 2484127 2484133 2484151
2484179 2484191 2484197 2484199 2484203 2484233 2484241 2484259 2484271
2484289 2484311 2484319 2484323 2484331 2484353 2484359 2484379 2484473
2484491 2484509 2484523 2484527 2484539 2484563 2484569 2484571 2484589
2484593 2484617 2484623 2484631 2484653 2484673 2484679 2484683 2484689
2484697 2484707 2484721 2484731 2484733 2484739 2484751 2484803 2484827
2484857 2484863 2484871 2484893 2484899 2484901 2484917 2484919 2484931
2484959 2484961 2484971 2484973 2485001 2485003 2485027 2485033 2485037
2485061 2485069 2485073 2485121 2485123 2485129 2485159 2485169 2485183
2485187 2485193 2485207 2485211 2485243 2485277 2485279 2485283 2485303
2485319 2485339 2485367 2485381 2485391 2485393 2485397 2485421 2485429
2485453 2485477 2485481 2485489 2485507 2485513 2485537 2485547 2485559
2485573 2485579 2485607 2485627 2485631 2485643 2485649 2485657 2485663
2485667 2485669 2485687 2485727 2485733 2485739 2485759 2485801 2485807
2485831 2485849 2485867 2485897 2485907 2485937 2485939 2485949 2485991
2485997 2485999 2486009 2486027 2486039 2486041 2486059 2486069 2486089
2486101 2486123 2486137 2486147 2486149 2486153 2486167 2486189 2486191
2486203 2486219 2486221 2486243 2486251 2486269 2486273 2486287 2486291
2486333 2486371 2486381 2486383 2486387 2486423 2486443 2486459 2486467
2486483 2486501 2486509 2486513 2486521 2486531 2486551 2486557 2486563
2486567 2486579 2486581 2486591 2486593 2486611 2486623 2486639 2486651
2486669 2486677 2486681 2486689 2486699 2486713 2486717 2486747 2486753
2486761 2486767 2486801 2486831 2486833 2486843 2486857 2486863 2486867
2486873 2486951 2486963 2486969 2486971 2486987 2486993 2487047 2487061

[19] En enero de 2018 hemos leído en el período que un ingeniero eléctrico acaba de encontrar el número primo más grande que se conoce hasta el momento. El nuevo campeón es el número: $2^{77.232.917} - 1$, el cual supera los veintitrés millones de cifras decimales (exactamente 23.249.425).

2487071 2487073 2487091 2487097 2487113 2487137 2487139 2487143 2487167
2487203 2487211 2487227 2487229 2487259 2487269 2487281 2487293 2487299
2487307 2487313 2487319 2487341 2487349 2487367 2487383 2487391 2487413
2487431 2487439 2487467 2487481 2487493 2487497 2487517 2487521 2487523
2487557 2487571 2487581 2487587 2487599 2487601 2487619 2487623 2487629
2487637

Ejercicio: Modificar el procedimiento anterior listar-primos, en cualquiera de sus dos versiones, para que inicie la búsqueda de primos a partir de un número inicial dado como entrada. Por ejemplo, si se llama al procedimiento en la forma listar-primos 1000, se generaría la lista de primos a partir del número 1000.

Modelo 18: Una lista de primos gemelos.

En este modelo se genera una lista que contenga parejas de primos gemelos. Como se dijo anteriormente, dos primos p y q se dice que son gemelos si la diferencia entre ambos es igual a 2, o sea, si entre ambos sólo media un entero. Tal es el caso de las parejas de primos 5 y 7, 11 y 13 o 29 y 31. Fabricaremos un procedimiento que tome como entrada una lista de primos y de esta lista extraiga solamente las parejas de primos gemelos. La lista de primos de entrada será generada por el procedimiento del modelo anterior generador de números primos. Todo lo que el nuevo procedimiento listar-gemelos debe hacer es comparar cada entero de la lista de entrada (primo o no) con el entero que le antecede y verificar si la diferencia entre ambos es igual a dos. En caso afirmativo ambos enteros se incluyen en la nueva lista de primos gemelos. Presentamos el código de este procedimiento precedido del código del procedimiento anterior generador de primos, agregando la variable “gemelos” a la lista de las variables globales. Esta variable se usa solamente en el procedimiento “listargemelos”. He aquí el código:

```
globals[primos gemelos]
to listar-primos
clear-all
set primos [2]
generar 2 primos
end

to generar [dividendo listado]
let i 0
divide dividendo i
generar dividendo + 1 primos
end


to divide [dividendo i]
let i-ésimo-primo item i primos
if (remainder dividendo i-ésimo-primo) = 0 [stop]
if i-ésimo-primo > sqrt dividendo [set primos se primos dividendo stop]
divide dividendo i + 1
end
```

```
;;  Aquí comienza el nuevo procedimiento que selecciona los pares de primo
;;  gemelos generados por el procedimiento anterior.

to listar-gemelos
set gemelos []
procesargemelos primos 1
end

to procesargemelos [lista i]
if lista = [] [type "La lista de primos está vacía" stop]
if (item (i + 1) primos - item i primos = 2)
[set gemelos lput (item i primos) gemelos
set gemelos lput (item (i + 1) primos) gemelos]
if (item (i + 1) primos) = last primos [stop]
procesargemelos primos i + 1
end
```

Los siguientes procedimientos deben llamarse desde la Ventana del Observador y envían la lista de primos o de gemelos a un archivo externo llamado "primos.txt", el cual debe ser previamente creado:

```
to ver-primos
file-open "primos.txt"
file-type  "Este es un listado de los primeros números primos."
file-type "\r\n"
file-type "La  lista contiene "
file-type length primos
file-type " primos"
file-type "\r\n"
file-write primos
file-type "\r\n"   ;; es buena idea cerrar con un retorno de carro
;;  por si más tarde añadiéramos algo al archivo
file-close
end

to ver-gemelos
file-open "primos.txt"
file-type "Esta es una lista de primos gemelos"
file-type "\r\n"
file-type "La lista contiene "
file-type length gemelos
file-type " primos gemelos"
file-type "\r\n"
file-write gemelos
file-type "\r\n"
file-close
end
```

Explicaciones y comentarios adicionales. El procedimiento "listar-gemelos" se debe correr después de haber corrido primero el procedimiento "listar-primos". De no

hacerse de este modo el programa mismo se encargará de mostrar un mensaje indicando que la lista de primos se encuentra vacía. En la lista de 182239 primos generada anteriormente en 10 segundos se encontraron 35750 primos gemelos (17875 parejas). Estas son las últimas parejas de primos gemelos de la lista:

2468099 2468101 2468129 2468131 2468447 2468449 2468951 2468953 2468969
2468971 2469281 2469283 2469317 2469319 2469407 2469409 2469431 2469433
2469557 2469559 2469581 2469583 2469869 2469871 2470001 2470003 2470121
2470123 2470199 2470201 2470241 2470243 2470331 2470333 2470337 2470339
2470691 2470693 2470889 2470891 2471057 2471059 2471087 2471089 2471321
2471323 2471471 2471473 2471531 2471533 2472179 2472181 2472539 2472541
2472557 2472559 2472851 2472853 2472929 2472931 2472959 2472961 2473127
2473129 2473181 2473183 2473421 2473423 2473451 2473453 2473607 2473609
2473631 2473633 2474051 2474053 2474117 2474119 2474207 2474209 2474711
2474713 2474861 2474863 2475089 2475091 2475287 2475289 2475437 2475439
2475797 2475799 2475857 2475859 2475959 2475961 2476037 2476039 2476079
2476081 2476391 2476393 2476421 2476423 2476751 2476753 2477129 2477131
2477159 2477161 2477171 2477173 2477309 2477311 2477327 2477329 2477411
2477413 2477609 2477611 2477639 2477641 2478239 2478241 2478269 2478271
2478347 2478349 2478521 2478523 2478527 2478529 2478587 2478589 2479487
2479489 2479661 2479663 2479667 2479669 2479691 2479693 2479847 2479849
2479901 2479903 2480081 2480083 2480207 2480209 2480501 2480503 2480717
2480719 2480909 2480911 2481137 2481139 2481179 2481181 2481317 2481319
2481497 2481499 2481839 2481841 2481887 2481889 2481977 2481979 2482349
2482351 2482619 2482621 2482769 2482771 2483519 2483521 2483669 2483671
2483711 2483713 2483867 2483869 2484017 2484019 2484197 2484199 2484569
2484571 2484731 2484733 2484899 2484901 2484917 2484919 2484959 2484961
2484971 2484973 2485001 2485003 2485121 2485123 2485277 2485279 2485391
2485393 2485667 2485669 2485937 2485939 2485997 2485999 2486039 2486041
2486147 2486149 2486189 2486191 2486219 2486221 2486381 2486383 2486579
2486581 2486591 2486593 2486831 2486833 2486969 2486971 2487071 2487073
2487137 2487139 2487227 2487229 2487521 2487523 2487599 2487601

Apéndice A: Aspectos de NetLogo no tratados en este libro.

Como se estableció en la Introducción, el propósito de este libro no ha sido otro que el de brindar una introducción al ambiente de programación NetLogo con especial énfasis en su lenguaje. Los lectores deben saber que hay varios aspectos de NetLogo que no se han tratado en este libro, los cuales –si se considera necesario- lectores y lectoras no tendrían problema en completar por su cuenta mediante el Manual [16] y otros recursos disponibles en la red. Entre los aspectos no cubiertos se incluye principalmente un buen número de primitivas, algunos temas relacionados con la interfaz y las varias extensiones del lenguaje. Toda esta información se encuentra bien documentada en el Manual de NetLogo [16]. En lo que al modelado basado en agentes se refiere, el tema se ha tocado sólo en la relación que guarda con los ejemplos y modelos presentados en el libro y no se ha desarrollado como un tema en sí mismo. Para mayor información sobre MBA recomendamos el libro de Wilensky y Rand [15], el cual trata el tema de forma muy amplia, tanto desde el punto de vista general de los ambientes para el modelado basado en agentes, como desde la perspectiva particular de NetLogo. Los lectores también pueden consultar el artículo on-line [13]. Información sobre todos los demás aspectos mencionados a continuación se puede encontrar en el Manual del Usuario de NetLogo. Algunas de las capacidades adicionales de NetLogo no mencionadas en el libro se encuentran disponibles desde el menú de Herramientas mientras otras hay que cargarlas como extensiones. Desde el menú de Herramientas se puede acceder a:

HubNet.

El Manual del Usuario dice: *HubNet es una tecnología que usa NetLogo para correr simulaciones participativas en el aula. En una simulación participativa la conducta de un sistema es determinada por toda la clase, donde cada estudiante controla una parte del sistema usando algún dispositivo individual, como por ejemplo una computadora conectada a la red.*

Behavior Space (Analizador de comportamiento).

El Manual del Usuario dice: *BehaviorSpace es una herramienta de software integrada dentro de NetLogo, la cual permite realizar experimentos con los modelos. BehaviorSpace corre un modelo muchas veces, variando sistemáticamente la configuración del modelo y grabando los resultados de cada corrida. Este proceso se conoce a veces como "barrido de parámetros" y permite explorar el "espacio" de posibles comportamientos del modelo y determinar cuáles combinaciones de su configuración causan comportamientos interesantes. Si el procesador de su computador posee varios núcleos de procesamiento, entonces por omisión, las corridas del modelo ocurrirán en paralelo, una por núcleo".*

En el menú Herramientas también se puede encontrar:

- Editor de formas para para tortugas: permite cambiar la apariencia de las tortugas.

- Editor de Formas para Enlaces: permite cambiar apariencia de los enlaces.
- Mostrario de colores disponibles.
- Monitores que dan información sobre el estado de:
    - Variables globales
    - Tortugas
    - Parcelas
    - Enlaces

Enlace hacia el programa Mathematica: permite cargar y correr modelos de NetLogo desde el poderoso software Mathematica.

NetLogo 3D.

Permite construir y ver modelos en tres dimensiones. Se carga desde el menú de Herramientas. También se puede cargar directamente desde su propio ícono independiante.

Dinámica de Sistemas (System Dynamics).

Permite construir modelos basados en la disciplina conocida como Dinámica de Sistemas [2]. Se carga desde el menú de Herramientas.

Extensiones.

Las extensiones son colecciones de herramientas que se pueden cargar desde NetLogo con la primitiva “extensions”. Cada extensión provee de capacidades adicionales al programa NetLogo. Seguidamente mencionamos las que, a gusto del autor, le parecen las más relevantes.

Array (arreglos).

Permite constuir y manipular arreglos de datos, como por ejemplo matrices numéricas.

Arduino.

Permite conectarse con un tablero electrónico Arduino, para el aprendizaje y creación de proyectos en electrónica.

Bitmap.

Permite manipular e importar imágenes en las parcelas y dibujos. Ofrece primitivas no incluidas dentro del conjunto básico de primitivas de NetLogo.

CSV.

Permite manipular archivos de tipo CSV (valores separados por comas) de gran tamaño.

GIS.

El Manual del Usuario dice: “Esta extensión añade soporte de GIS (Sistema de Información Geográfica) a NetLogo. Agrega la capacidad de cargar datos GIS vectoriales (puntos, líneas y polígonos), y datos GIS de tipo raster (celdillas) en sus modelos”.

Matrix.

Agrega las matrices como una estructura de datos adicional.

Network.

Agrega herramientas para construir y manipular redes (grafos).

Palette.

Aumenta el poder de NetLogo para trabajar con colores.

R.
Provee de primitivas para trabajar con la aplicación de estadística R desde NetLogo [3].
Sound.
Permite trabajar con sonidos MIDI y sonidos pregrabados.
Table.
Ofrece la posibilidad de trabajar con tablas.
Vid.
Permite conectar una fuente de video para integrarla a los proyectos o modelos.

Apéndice B.

Lista de las primitivas presentadas en el libro.

A continuación se ofrece una lista de las primitivas usadas en los ejemplos, modelos y explicaciones del libro. Cada primitiva se acompaña de una breve traducción (salvo cuando la traducción es obvia) y de una breve descripción de lo que la primitiva hace. En muchos casos las descripciones no ofrecen una explicación completa de la función que realiza la primitiva, para lo cual deberá consultarse el Diccionario de Primitivas incluido en el Manual del Usuario. Las primitivas se pueden usar con la primera letra en mayúscula o en minúscula pues NetLogo es insensitivo a este aspecto.

And. Conjunción "y" (conectiva lógica).
Any? Algún?/alguna? (reporta verdadero si hay algún agente).
Ask. Pedir, solicitar (a un agente o conjunto de agentes que ejecute algunas órdenes).
Beep. Bip (produce un sonido corto).
Blue. Azul (reporta el color azul, #105).
Breed. Familias (se usa para crear familias (breeds) de agentes).
Butfirst. Menos-el-primero. Reporta una lista sin su primer miembro.
Butlast. Menos-el-último. Reporta una lista sin su último miembro.
Ca. Abreviatura de "clear-all".
Clear-all. Limpiar todo (Elimina las tortugas y borra los trazos y las variables).
Clear-drawing. Limpiar-dibujo (borra los trazos hechos por las tortugas).
Color. Color (reporta el color de la tortuga).
Count. Reporta el número de agentes de un conjunto-agentes.
Create. Crear (se usa para crear agentes).
Create-link-from, **create-links-from**. Crear enlace desde, crear enlaces desde.
Create-link-to, **create-links-to**. Crear enlace hacia, crear enlaces hacia.
Create-link-with, **create-links-with**. Crear enlace con, crear enlaces con.
Die. Morir, muere (se usa para eliminar tortugas).
Distance. Distancia (reporta la distancia entre dos agentes).
End. Fin (indica el final de un procedimiento).
Export-interface, exportar-interfaz (guarda la interfaz en un archivo en formato PNG).
Export-plot, exportar-gráfico (guarda el gráfico en un archivo).
Export-plots, exportar-gráfico (guarda todos los gráficos en un archivo).
Export-view, exportar-vista (guarda la vista en un archivo en formato PNG).
Export-world, exportar-mundo (guarda el estado del mundo en un archivo).
Extensions (se usa para cargar una extensión del lenguaje)
Face. Encarar, orientarse en dirección a un agente.
File-close. Cerrar-archivo (cierra un archivo previamente abierto).
File-open. Abrir-archivo (abre un archivo previamente creado).
File-print. Imprimir-archivo (envía la salida a un archivo previamente abierto). Las primitivas **file-type**, **file-show** y **file-write** operan de modo similar, con pequeñas diferencias.
First. Primero (reporta el primer miebro de una lista o cadena).
Filter. Filtra los miembros de una lista según que cumplan una condición.
Foreach. (para-cada-uno). Aplica una operación a cada miembro de una lista.
Forward. Adelante (se usa para avanzar hacia adelante).
Fput. Abreviatura de "first-put" (se usa para colocar de primero en una lista).
Globals. Globales (se usa para declarar las variables globales).
Gray. Gris (reporta el color gris, #5).
Green. Verde (reporta el color verde, #55).
Heading. Orientación (reporta la dirección en que mira la tortuga).

If. Si (se usa para crear expresiones condicionales con una alternativa).
Ifelse, si-otro (se usa para crear expresiones condicionales con dos alternativas).
Import-drawing, importar-dibujo (carga un dibujo en la capa de dibujo del mundo).
Import-pcolors, importar-pcolores (ver Manual [16]).
Import-pcolors-rgb, importar-pcolores-rgb (ver Manual [16]).
Import-world, importar-mundo (carga un archivo con los datos del mundo).
Int. Abreviatura de "integer" (entero). Reporta la parte entera de un número.
In-radius. Dentro-del-radio (reporta los agentes incluidos en un círculo de radio dado.
Item. Item (reporta un ítem de una lista o cadena).
Last. Último (repota el último miembro de una lista o cadena).
Length. Longitud (reporta el número de miembros de una lista).
Left. Izquierda (se usa para que la tortuga realice giros hacia la izquierda).
Let. Asigna un valor a las variables locales.
List. Lista (se usa para crear listas).
Lput. Abreviatura de "last-put" (se usa para colocar de último en una lista).
Lt. Abreviatura de "left".
Map. Aplica una operación a los miembros de una o varias listas.
Max. Reporta el máximo de una lista de valores.
Max-one-of. Reporta el agente con el máximo valor de una cantidad.
Max-pxcor. Reporta la coordenada x de la parcela cuyo valor es el máximo.
Max-pycor. Reporta la coordenada y de la parcela cuyo valor es el máximo.
Mean. Promedio (reporta el promedio de un conjunto de valores).
Member? Miembro? (reporta verdadero si un objeto es miembro de una lista o cadena o conjunto-agentes).
Min. Reporta el mínimo de una lista de valores.
Mouse-down? Ratón-abajo? (reporta verdadero cuando el botón del ratón está oprimido).
Mouse-inside? Ratón-dentro? (reporta verdadero cuando el ratón se encuentra dentro de la vista o view).
Mouse-xcor. Ratón-xcor (reporta la coordenada x del puntero del ratón).
Mouse-ycor. Ratón-ycor (reporta la coordenada y del puntero del ratón).
Myself. Yo mismo (se usa en la construcción de órdenes compuestas).
Neighbors. Vecinas (reporta las parcelas vecinas a una parcela dada).
Neighbors4. Vecinas4 (reporta las 4 parcelas vecinas de los lados de una parcela dada).
Not. Negación lógica "no".
Of. Preposición "de" (se usa para construir órdenes compuestas).
One-of. Uno-de (selecciona un miembro de una lista, cadena o conjunto-agentes).
Or. Disyunción "o" inclusiva (conectiva lógica).
Other. Otro (se usa para referirse a otro agente de aquel que da la orden).
Patch, **patches**. Parcela, parcelas.
Patch-ahead. Parcela-adelante (reporta los agentes en dicha parcela).
Patch-here. Parcela-aquí (reporta los agentes en esta parcela).
Patch-left-and-ahead. Parcela-izquierda-adelante (reporta los agentes en dicha parcela).
Patch-right-and-ahead. Parcela-derecha-adelante (reporta los agentes en dicha parcela).
Patches-own. Parcelas-poseen (se usa para declarar las variables propias de las parcelas).
Pcolor. Reporta el color de la parcela indicada.
Pd. Abreviatura de "pendown".
Pendown. Pluma abajo (se usa para activar la pluma de la tortuga).
Penup. Pluma arriba (se usa para desactivar la pluma de la tortuga).
Plabel. Petiqueta (reporta la etiqueta de la parcela).
Play-note. Tocar-nota (toca la nota indicada en la extensión "sound").
Print. Imprime (escribe el resultado en la Terminal de Instrucciones u otra salida).
Pu. Abreviatura de "penup".

Pxcor. Reporta la coordenada x del centro de la parcela sobre la que se encuentra la tortuga.
Pycor. Reporta la coordenada y del centro de la parcela sobre la que se encuentra la tortuga.
Random. Azar (genera y reporta números aleatorios).
Random-seed. Semilla-aleatoria (genera una nueva semilla aleatoria).
Random-**xcor**. Reporta la coordenada x al azar.
Random-**ycor**. Reporta la coordenada y al azar.
Red. Rojo (reporta el color rojo, #15).
Reduce. Aplica un proceso de reducción a los miembros de una lista.
Remainder. Residuo (reporta el residuo de una división entre enteros).
Remove. Remover (remueve un miembro de una lista o cadena).
Repeat. Repetir (se usa para repetir un conjunto de órdenes).
Report. Reportar (se usa en conjunción con "to-report" para reporta un resultado).
Reset-**ticks**. Reinicia la variable ticks en el valor 0.
Right. Derecha (se usa para girar a la derecha).
Run. Correr (corre el bloque de instrucciones dado).
Self. Yo, a mí (se usa en la construcción de órdenes compuestas).
Sentence. Frase (fusiona dos listas o cadenas en una sola lista).
Set. Asignar (se usa para asignar valores a las variables).
Setxy. Asignarxy (se usa para asignar las coordenadas de posición a una tortuga).
Show. Mostrar (escribe el resultado en la Terminal de Instrucciones u otra salida).
Sort-**by**. Ordenar o sortear según (reporta una lista ordenada según un criterio dado).
Sound. Sonido (Habilita la extensión de los sonidos).
Sprout. Brotar, germinar (se usa para hacer brotar tortugas en na parcela).
Sqrt. Reporta la raíz cuadrada de un número.
Stamp. Estampar (la tortuga estampa su figura).
Standard-**deviation**. Reporta la desviación estándar de un conjunto de valores.
Stop. Detener (detiene un bloque de instrucciones o un procedimiento).
Sum. Reporta la suma de una lista de números.
Thickness. Grosor (reporta el grosor de un enlace).
Tick. Incrementa el valor de la variable ticks.
Ticks. Reporta el valor de la variable "ticks" preinstalada en el sistema.
Tie. Atar (crea una atadura entre dos tortugas).
To. Para (preposición empleada para dar nombre a un procedimiento).
To-**report**. Para-reportar (se usa para construir procedimientos reportadores).
Turtle, **turtles**. Tortuga, tortugas.
Turtles-at. Tortugas-ahí (reporta las tortugas en la parcela situada a una distancia dx, dy de la parcela actual).
Turtles-**here**. Tortugas.aquí (reporta las tortugas sobre esta parcela).
Turtles-on. Tortugas-en (reporta las tortugas en una parcela o conjunto de parcelas).
Turtles-**own**. Tortugas-poseen (se usa para declarar las variables propias de las tortugas).
Type. Escribir (escribe el resultado en la Terminal de Instrucciones u otra salida).
User-**input**. Ingreso-del-usuario (reporta la entrada que el usuario ingresa).
Xcor. Reporta la coordenada x de la tortuga.
Ycor. Reporta la coordenada y de la tortuga.
Yellow. Amarillo (reporta el color amarillo, #45).
Wait. Esperar (detiene las acciones durante los segundos indicados).
While. Mientras (un bloque de órdenes se repite mientras se cumple una conidición).
White. Blanco (reporta el color blanco, #9.9).
With. Preposición "con" (se usa para construir órdenes compuestas).
With-**max**. Con-el-máximo (se usa para reportar agentes cuya cualidad tiene un valor máximo).
With-**min**. Con-el-mínimo (se usa para reportar agentes cuya cualidad tiene un valor mínimo).
Who. Quién (reporta el número de identidad de la tortuga).

Word. Palabra (reporta la lista o cadena formada por la unión de dos listas o cadenas).
Write. Escribir (escribe el resultado en la Terminal de Instrucciones u otra salida).

+. Signo de adición.
- . Signo de sustracción o resta.
*****. Signo de multiplicación.
/. Signo de división.
". Signo de comillas (se usa para definir cadenas).
=. Signo "igual a ".
¡=. Signo "diferente a".
<. Signo "menor que".
<=. Signo "menor o igual que".
>. Signo "mayor que".
>=. Signo "mayor o igual que".
[ ]. Corchetes (se usan para crear listas).
( ). Paréntesis para agrupar expresiones.
\r\n. Fuerza un retorno de carro (cambio de línea forzado).

Referencias.

1. Abelson H., Sussman G. J., Sussman J., 1985. Structure and Interpretation of Computer Programs, Cambridge, Mass. The MIT Press. Una version gratuita pdf se puede bajar de la Web: https://mitpress.mit.edu/sites/default/files/6515.pdf
2. Forrester Jay, Lagarda Ernesto (s.f.), Introducción a la dinámica de sistemas. Artículo en: http://jmonzo.net/blogeps/ids1.pdf
3. De Vries A., Meys J., 2012. R for dummies. John Wiley & Sons, Ltd., England.
4. Distel R. Graph theory, 2000, electronic edition (pdf): http://www.esi2.us.es/~mbilbao/pdffiles/DiestelGT.pdf
5. García Vázquez J. C., Sancho Caparrini F. (s.f.), NetLogo: una herramienta de modelado.
6. Gracián E. Los números primos (s.f.). http://www.librosmaravillosos.com/losnumerosprimos/pdf/Los%20Numeros%20Primos%20-%20Enrique%20Gracian.pdf.
7. Hamilton, A. (1981). Lógica para matemáticos. Madrid: Paraninfo.
8. Hillis, D. W. (1985). The connection machine. The MIT PRESS.
9. Jackson M. O., Social and economic networks, 2008. Princeton University Press, 504 pp.
10. Manual del software Scratch (Resnick M.) en español: https://nuestratecnologia.files.wordpress.com/2011/09/manual-scratch.pdf
11. Papert S., 1987. Desafío a la mente (Titulo orignal en inglés: Mindstorms) . Ediciones Galápagos, quinta edición. Versión on line en: http://neoparaiso.com/logo/desafio-mente.html#s104
12. Polya G., 1975. Cómo plantear y resolver problemas. Editoria Trillas, México. Original en inglés “How to solve it”, 1946, Princeton University Press, USA. (se puede conseguir en la red en formato pdf en algunos sitios que piden membresía o mediante torrent). Un resumen se puede ver en: http://ficus.pntic.mec.es/fheb0005/Hojas_varias/Material_de_apoyo/Estrategias%20de%20Polya.pdf
13. Quesada A., Canessa E., Modelado Basado en Agentes: una herramienta para completar el análisis de fenómenos sociales. http://www.scielo.org.co/pdf/apl/v28n2/v28n2a7.pdf
14. Resnick, M. (1994). Learning about life. Artificial Life, vol. 1, no. 1-2.
15. Wilensky U., Rand W. (2015). An introduction to Agent Based Modeling: modeling natural, social and engineered complex systems with NetLogo. MIT Press, 482 pp.
16. Wilensky y colaboradores. NetLogo User Manual, 2017, versión 6.02: https://ccl.northwestern.edu/netlogo/docs/

ÍNDICE ALFABÉTICO