

Programando a

WALL·E



© 2016 UH/DAI

Segundo Proyecto de Programación

Carrera de Ciencia de la Computación

Universidad de La Habana

2016-2017



INTRODUCCIÓN

Sobre un mundo rectangular dividido en casillas se ubica un conjunto de objetos. Algunos de estos objetos son robots que pueden ser programados para que efectúen tareas (incluso, competir entre ellos). Los robots pueden desplazarse hacia delante o hacia atrás, o girar 90 grados hacia la izquierda o hacia la derecha. La dirección del robot siempre será alguno de los ángulos (0°, 90°, 180° o 270° con respecto al norte). En el terreno puede haber objetos grandes (obstáculos), objetos

medianos (movibles) y objetos pequeños (cargables). Si un robot avanza sobre un obstáculo no se mueve, si avanza sobre un objeto movible (siempre que se pueda) lo desplaza y si avanza sobre un objeto pequeño lo recoge (lo almacena en su interior). Si la casilla de enfrente al robot está libre el robot puede soltar (descargar) el objeto en su interior.

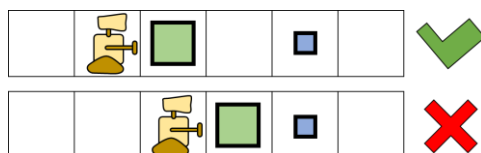
El robot tiene un núcleo central de procesamiento que es capaz de ejecutar un conjunto de instrucciones. Además tiene sensores que permiten “percibir” características del ambiente. Por ejemplo, un sensor ultrasónico para determinar distancia o una webcam para identificar el color del objeto enfrente más cercano. Los sensores permiten recibir información del medio y que el robot pueda tomar decisiones de acuerdo a ello.

OBJETOS

Los objetos pueden ser clasificados según distintos atributos (tamaño, forma y color). El tamaño y la forma del objeto determinan si éste puede ser recogido, empujado o si es un obstáculo que impide el avance del robot. El tamaño del objeto frente al robot se puede conocer a través del atributo **size** y su forma a partir del atributo **shape**. Además los objetos tienen un color (accesible a través del atributo **color**) y un código de barra (atributo **number**). Estos últimos no influyen en la interacción entre el robot y los objetos pero pueden ser utilizados para especificar objetivos del robot (i.e. encontrar la bola negra 8).

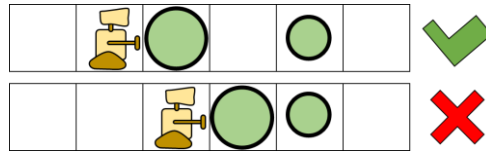
Si el objeto es pequeño (no importa la forma) el robot puede agarrarlo (moviéndose para la casilla donde se encuentra dicho objeto). Si el objeto es mediano (o pequeño y el robot ya contiene un objeto pequeño), este puede ser empujado. Las reglas para desplazar un objeto es la siguiente:

- Un robot nunca puede desplazar a otro robot.
- Una caja mediana o pequeña es desplazable siempre y cuando la siguiente casilla esté desocupada.

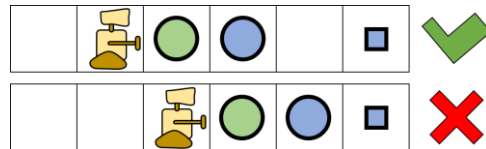


- Una bola grande es desplazable siempre y cuando la siguiente casilla esté desocupada.





- Una bola mediana o pequeña es desplazable siempre y cuando luego de una serie de bolas pequeñas o medianas contiguas haya una casilla desocupada.



- Una planta nunca es desplazable, con excepción de si es pequeña que el robot la puede recoger.

Si un robot suelta un objeto lo hace a la casilla del frente. Si ya existe un objeto en dicha casilla queda sin efecto la acción.

Para evaluar el comportamiento de los robots en un ambiente controlado se diseñó un lenguaje de programación denominado SINTIME. En este lenguaje cada línea representa una instrucción y cada instrucción comienza por un comando. En el lenguaje se pueden declarar tanto la dinámica del mundo como el programa de cada robot.



LENGUAJE SINTIME

El lenguaje SINTIME (**Single Instruction at a Time**) es un lenguaje imperativo con un único ámbito para las variables. En este se pueden declarar rutinas (método, procedimiento) o describir un conjunto de instrucciones a ser ejecutadas por el simulador.

DECLARACIONES

Las declaraciones de rutinas están delimitadas por las instrucciones `routine` y `return`. La rutina se define con un nombre con el que será identificada en otras partes del código. El siguiente listado muestra un código con dos rutinas. Las rutinas no se pueden declarar dentro de otras rutinas.

```
routine Accion1
// esto es un comentario
return

routine Accion2
/* esto es otra forma de comentario */
return
```

Las rutinas solo se ejecutan si se invocan mediante el comando `execute`. También pueden ser ejecutadas dependiendo de si cierta condición se cumpla, mediante el comando `execute ... if ...`.

Al terminar la ejecución de una rutina (comando `return`) se debe continuar con la instrucción que le sigue al comando `execute` que inició la ejecución en primer lugar. El lenguaje permite ejecutar cualquier rutina que esté declarada dentro del código, no importa si está antes o después de la declaración actual. **¡La recursión es permitida!**

Otra forma más general de factorización son las bibliotecas. Una biblioteca es un script SINTIME que sólo contiene declaraciones de rutina (o incluye otras bibliotecas). Para incluir en el script actual las definiciones de un script biblioteca se utiliza el comando `include <nombre_del_fichero>`. Estos comandos deben ubicarse al principio del código antes de cualquier declaración o instrucción. Por ejemplo:

main.stm	tools.stm
<pre>include "tools.stm" execute Saludar</pre>	<pre>routine Saludar message "Hola a todos" return</pre>

Si dos bibliotecas incluidas comparten un mismo nombre para una rutina se debe resolver el conflicto mediante la instrucción `include ... as <nombre>` y `execute ... from <nombre>`. El siguiente ejemplo muestra una posible resolución del conflicto.

main.stm	tools1.stm	tools2.stm
<pre>include "tools1.stm" as MiModulo1 include "tools2.stm" as MiModulo2 // imprime Hola desde 2 execute Saludar from MiModulo2</pre>	<pre>routine Saludar message "Hola desde 1" return</pre>	<pre>routine Saludar message "Hola desde 2" return</pre>

Si el propio script tiene una rutina con igual nombre que una importada entonces prevalece (sin error de ambigüedad) la rutina local, aunque la rutina importada puede ser accedida mediante la instrucción antes mencionada.

Cualquier punto del código puede ser “etiquetado” mediante un nombre para luego realizar un salto en la ejecución hasta dicha posición. Cada rutina determina un ámbito para las etiquetas, así como el script en sí mismo. Desde una rutina no se puede saltar hacia ninguna etiqueta que esté fuera de su ámbito, ni del código principal se puede saltar a una etiqueta dentro de una rutina. Para etiquetar se utiliza el comando `label <nombre>` y para saltar hacia un punto del código etiquetado se utilizará el comando `goto <nombre>`. En el mismo ámbito no pueden existir dos etiquetas con igual nombre. El siguiente listado muestra varias rutinas y con colores distintos cada uno de los ámbitos de etiquetas que quedan determinados.

```
routine Accion1
```

```
...  
label A  
...  
goto A  
...
```

```
return  
routine Accion2
```

```
...  
label A  
...  
goto A  
...
```

```
return
```

```
...  
label A  
...  
goto A  
...
```

MEMORIA

El simulador y los robots tienen memoria para almacenar el programa y las variables. Las variables tienen un nombre que las identifican y son todas de tipo entero (incluso las que se refieran a operaciones lógicas se tratarán con los valores 0 para falso y 1 para verdadero). Las variables no hay que declararlas y si se consulta su valor antes de ser asignadas tendrán valor por defecto 0 (ej. $x + 4 == 4$ si x no está inicializada).

Mediante la instrucción asignación (`set <var> = <exp>`) se le podrá dar valores a las variables durante la ejecución. Las variables son globales a todas las rutinas (del propio código y las importadas). Eso significa que si una rutina deja el valor 3 en una variable ‘a’, las instrucciones fuera pueden acceder a dicho valor. El ejemplo siguiente es válido:

```
routine Sucesor  
set n = n + 1  
return
```

```
set n = 0  
execute Sucesor  
message n
```

Las variables pueden nombrarse con letras, guion bajo y números pero siempre deben empezar con letra (o guion bajo '_'). Si se utiliza el prefijo '@' es equivalente a iniciar la variable con el nombre de la rutina en la que se está trabajando (concatenando guion bajo antes del nombre de la variable). El ejemplo anterior es equivalente al siguiente.

```
routine Sucesor
  set @n = @n + 1
return

set Sucesor_n = 0
execute Sucesor
message Sucesor_n
```

Las rutinas no reciben parámetros ni devuelven valores (a diferencia de las funciones en otros lenguajes), en cambio, pueden utilizar variables globales para “leer” la entrada y “almacenar” la salida, de modo que estos puedan ser usados luego de la llamada. El símbolo @ sirve para “emular” un traspaso de parámetros a las rutinas. El siguiente código permite calcular el mínimo entre dos valores.

```
routine Minimo
  set @result = @a if @a < @b else set @result = @b
return

set Minimo_a = 5, Minimo_b = 7
execute Minimo
message Minimo_result
```

VARIABLES INDEXADAS

Un recurso importante que debe ser expresable en el lenguaje son las variables indexadas. Una variable puede representarse mediante un identificador (e.g. `cantidad`, `terminado`, `posicion`), o mediante un identificador indexado (e.g. `valores[0]`, `mapa[4;5]`).

El mismo identificador NO puede ser accedido con diferentes dimensiones. El siguiente código debe dar error de compilación:

```
set a=5
set a[4]=8
```

Se puede indexar entre corchetes con cualquier cantidad de expresiones enteras separadas por punto y coma. Son válidas por ejemplo: `a[3;2]`, `mapa[X;Y+1]`, `marcas[cantidad]`. Las variables pueden ser indexadas en valores negativos. Por ejemplo: `mapa[-4;3]`. Si se consulta en un índice que no ha sido asignado se devolverá el valor por defecto 0.

EXPRESIONES

Existen instrucciones que involucran valores. En el lenguaje se podrán especificar constantes numéricas (-3, 0, 100, 20,...) o cadenas de texto (“Hola”, “Terminé!”). Las variables son otro tipo de expresión (indexadas o no) y siempre representan valores enteros.

Existe un conjunto de operadores que pueden ser usados para formar expresiones más complejas: +, -, /, *, %, and, or, not, xor, >, >=, <, <=, ==, !=, & (and lógico), | (or lógico), ^ (xor lógico), ~ (not lógico), así como los paréntesis para agrupar y dar mayor prioridad a las operaciones internas. Las prioridades de estos operadores es la misma que en los demás lenguajes. La diferencia está en que los operadores condicionales y de comparación devuelven 1 y 0, en lugar de true y false. Ejemplo, la expresión (1 < 3)+(2 == 2) es válida e igual a 1+1 (2). Cualquier valor distinto que 0 evalúa verdadero en una expresión condicional, y el 0 evalúa falso. Por ejemplo: 2 or 0 es verdadero, es decir, 1. Los operadores lógicos &, |, ^ y ~ permiten trabajar con los enteros a nivel de bits.

Otro tipo de expresión que está presente en el lenguaje es la consulta a propiedades de los sensores. Esta consulta deberá hacerse refiriéndose al identificador del atributo específico. Ejemplo: distance devuelve la distancia al obstáculo enfrente más cercano. Estos nombres no pueden ser utilizados como variables por lo tanto el código:

```
set distance=3
```

Produce un error de compilación. Los nombres de comandos, rutinas y módulos tampoco pueden ser utilizados como nombres de variables.

Algunos ejemplos válidos de expresiones:

```
cantidad + 1
cantidad >= 0 or distance < 3
(a[3;1+cantidad] + 4) * 10
```

LOS VALORES ENTEROS

En el lenguaje todas las variables almacenan valores enteros, y todas las operaciones resultan enteras (incluidas las operaciones condicionales). La división es entera (es decir, 7/2==3).

Existen varios valores enteros especiales: posinf, neginf e indet. Estos valores representan el infinito positivo, el infinito negativo y un valor indeterminado (por ejemplo: 4/0==posinf, 0/0==indet). Las operaciones de enteros están definidas de forma tal que contemplen la posibilidad de que un parámetro sea una de estas constantes especiales. Note que usted deberá proveer una resolución lógica para estos escenarios. Por ejemplo, las tablas de sumar y restar quedarían como se muestra a continuación:

+	n	posinf	neginf	indet
n	n	posinf	neginf	indet
posinf	posinf	posinf	indet	indet
neginf	neginf	indet	neginf	indet
indet	indet	indet	indet	indet

-	n	posinf	neginf	indet
n	n	neginf	posinf	indet
posinf	posinf	indet	posinf	indet
neginf	neginf	neginf	indet	indet
indet	indet	indet	indet	indet

INSTRUCCIONES CONDICIONADAS

Exceptuando las instrucciones de declaración (include, routine, return, label), las demás instrucciones pueden ser condicionadas, esto es, su ejecución puede depender de la evaluación de una condicional. Ejemplo:

```
goto Fin if a < b
```

En estos casos también se puede utilizar la cláusula else para ejecutar una segunda instrucción en caso de evaluar falsa la condicional. Ejemplo:

```
goto Fin if a < b else goto Inicio
```

Las instrucciones condicionadas agrupan a la derecha, es decir, el primer `else` se refiere a la instrucción conformada por todo el resto de la instrucción. Por ejemplo, la siguiente instrucción condicionada pudiera parecer ambigua.

```
goto Fin if a <= b else goto Medio if a >= b else goto Inicio
```

Normalmente esto podría tener dos interpretaciones (resaltado en cada caso la condición referida por el primer `else`):

```
(goto Fin if a <= b else goto Medio) if a >= b else goto Inicio
```

```
goto Fin if a <= b else (goto Medio if a >= b else goto Inicio)
```

En el primer caso el escenario `a==b` haría verdadera la condicional `a>=b` y por tanto se saltaría a la etiqueta `Inicio`. En el segundo caso, haría verdadera la condición `a<=b`, saltando a la posición etiquetada como `Fin`. Esta ambigüedad se resuelve suponiendo siempre que se agrupa la derecha (es decir, se asume siempre el segundo caso).

Esta regla puede reescribirse de la siguiente forma: una instrucción condicionada no se puede condicionar nuevamente.

COMANDOS

De forma general todas las instrucciones del lenguaje, ya sean de declaración o ejecución, pueden ser vistas como comandos. En cada línea se describe un comando (exceptuando los fragmentos de comentario). El comando más simple es el formado por una palabra (por ejemplo `return`). En otros casos se requiere de un argumento, valor o expresión (como es el caso de `include "Tools.stp", message 4`), o de un identificador (`routine Accion1, label Inicio`).

En el caso del `set` se recibe una lista de asignaciones de la forma `<variable(indexada o no)>=<expresión>`.

Dependiendo del entorno que se esté programando (el simulador o el robot), existirán otros comandos propios que pueden ser usados. Los comandos cumplen (de forma general) con la siguiente sintaxis:

```
<comando> <argumento> <subcomando> <argumento> <subcomando> <argumento> ...
```

Los argumentos pueden ser (dependiendo del comando) una expresión, un identificador o una lista separada por coma de asignaciones de la forma: `<atributo>=<expresión>`.

Los argumentos siempre se refieren al comando/subcomando inmediato de la izquierda. Ejemplo, en el caso siguiente:

```
move from row=3, column=5 to row=5, column=6
```


El comando es `move` (mover un objeto), el subcomando `from` tiene asociado las asignaciones a atributos `row=3` y `column=5` (indicando la casilla desde donde se mueve) y el subcomando `to` tiene asociado `row=5` y `column=6` (hacia donde se mueve).

Otros casos como `turn left` (girar a la izquierda) no requieren atributos, y casos como `drop`, no necesitan de especificar un subcomando.

A continuación se describirán los escenarios en los que se utilizará el lenguaje SINTIME y se introducirán los comandos propios. Todos los comandos propios son de ejecución y por tanto pueden ser condicionados. Por ejemplo: `drop if column==6`



EL ENTORNO SIMULADO

Como se introdujo al principio del documento, los robots inicialmente se ubican en un terreno rectangular dividido en casillas cuadradas. En las casillas se pueden ubicar además objetos con diversas propiedades como tamaño, color, forma, código. 

El código que genera y controla el entorno se escribirá en lenguaje SINTIME. El propósito de este programa es agregar, mover y eliminar objetos del mapa (incluido los robots). Además en este programa se decide en qué momentos se ejecuta un nuevo paso para los robots del mapa (con un paso se refiere a un movimiento físico, por ejemplo: avanzar, girar, soltar, etc).

Al inicio se tiene un mapa por defecto vacío y de 10x20 celdas. Para la creación de un mapa nuevo se debe ejecutar el comando:

```
create map rows=<cantidad de filas>, columns=<cantidad de columnas>
```

Una vez creado el mapa se pueden ubicar objetos en él. El mapa anterior y los objetos (incluido robots) son destruidos (esto incluye además la memoria de los robots, con programas, etc.).

Para ubicar un nuevo objeto en el mapa se ejecutará el comando:

```
create object <atributo1>=<valor1>, ..., <atributoN>=<valorN>
```

Ejemplo:

```
create object row=2, column=3, shape=sphere, color=red, size=medium, number=4
```

Si al agregar un objeto en una posición, dicha posición no está vacía o no está en los límites del mapa, la acción no tiene efecto. El valor 0 en los atributos `shape`, `color` y `size` está reservado para la casilla vacía, por lo tanto si a un objeto se le asigna 0 a uno de estos atributos deberá tomar el valor por defecto para dicho atributo (ver anexo).

Para los robots es obligatorio que se le asigne el atributo `program`, especificando el fichero que contiene el programa del robot:

```
create robot color=red, number=4, program="code1.stm"
```

Los robots siempre tienen los atributos fijos `shape=bot` y `size=large` y no pueden ser cambiados. Si un atributo no es especificado se tomará el valor por defecto para el atributo. Cuando se compila un programa de simulación es necesario analizar TODOS los programas de los robots (incluso si se usan o no). Si el programa de un robot no se encuentra o no compila, se deberá mostrar un error de compilación. En SINTIME no hay lugar para las EXCEPCIONES!

Otros atributos como `stored` y `full` son de solo consulta, es decir, no se les puede cambiar su valor ni en la creación ni durante la ejecución. El atributo `stored` determina si un objeto está en el interior de un robot. El atributo `full` determina si un robot está almacenando un objeto.

Para mover un objeto de una posición (fila, columna) a otra (fila, columna) se utiliza el comando:

```
move from row=3, column=4 to row=2, column=5
```

Como regla general, si una acción no puede efectuarse por alguna razón, esta queda sin efecto.

Para eliminar los objetos (incluido los robots y objetos internos que estos puedan tener) que cumplan cierta condición se utiliza el comando: `destroy <condición>`

El siguiente comando elimina todas las bolas de una columna.

```
destroy shape==sphere and column==4
```

El comando `count` permite contar la cantidad de objetos en el mapa que cumplen cierta condición (incluido los objetos pequeños que pudieran estar contenidos dentro de los robots). En esta condicional se puede acceder a los atributos de un objeto (`column`, `row`, `shape`, `color`, `number`, `stored`, `full`, etc.). El resultado es accesible a través del atributo especial `result`.

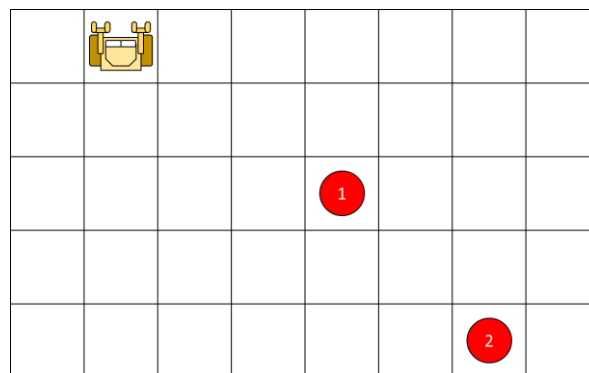
Para hacer que todos los robots avancen un paso (ejecuten su código hasta llegar a una acción física), en el programa del simulador deberá ejecutarse el comando `advance`. Para determinar que la simulación ha llegado a su fin se ejecutará el comando `exit`.

En todo momento se puede conocer el tiempo transcurrido desde el inicio de la simulación mediante el atributo `time` (este valor es equivalente a cuántos `advance` han ocurrido en la simulación). Esto permitiría “aparecer” o “desaparecer” objetos con el tiempo.

El siguiente código crea un mapa con varias esferas rojas, un robot, y termina la simulación si una esfera es “llevada” a la columna 7 o se han ejecutado 100 movimientos.

```
create map rows=5, columns=8
create object column=4, row=2, shape=sphere, color=red, size=medium, number=1
create object column=6, row=4, shape=sphere, color=red, size=medium, number=2
create robot column=1, program="bot1.stm"
label Inicio
advance
count shape==sphere and column==7
exit if result>0 or time==100 else goto Inicio
```

La figura muestra la configuración inicial que tendría el mapa al arribar a la etiqueta `Inicio`.



Si el programa de la simulación llega a su fin (final del código), la simulación continuará indefinidamente repitiendo el comando `advance`. Note que el hecho de poder describir condiciones de parada, creación y destrucción de objetos en el tiempo, etc. representa una herramienta para crear “misiones” más interesantes para los robots. Por ejemplo: determinar cuál encuentra primero una bola roja, o cuál mueve más bolas a su columna, etc.

El atributo `random` devuelve un valor entero positivo aleatorio, por lo tanto, la expresión `random%N` puede ser utilizada para generar un valor aleatorio entre 0 y N-1.

LOS ROBOTS



Cada robot tiene un programa que describe el comportamiento del mismo en todo momento. El lenguaje utilizado será SINTIME. No obstante, en este escenario se incluyen algunos comandos propios que permiten realizar acciones con el robot. Además, se tienen algunos atributos que expresan el estado de sus sensores.

Como existen varios robots en el mapa, el simulador debe simular la ejecución de las instrucciones de sus programas de forma concurrente. No obstante, en la realidad no ocurrirá así. Cada vez que en el simulador se alcance el comando `advance`, significará que todos los robots deben “moverse” un paso. Las instrucciones generales del programa se asumen ejecutan instantáneamente, por tanto lo único que puede significar “toma tiempo” es alguna acción física del robot, por ejemplo, avanzar, girar, soltar, etc. El tiempo de estas acciones será el mismo para todos los robots y aunque tentativamente pudiera ser un segundo, es un posible parámetro de la simulación y por tanto que se pueda ver más lento o más rápido el proceso.

El orden de los robots deberá ser asumido aleatorio en cada “ronda”. De esta forma ningún robot tendrá ventaja sobre otro por el orden en que fueron creados. Una vez establecido el orden en que se simulará el “siguiente paso” de cada robot, se ejecutará en cada uno de ellos el código (desde la última posición en que se había quedado de la ronda anterior) hasta que se ejecute una acción física o se alcance el final del código. Si se llega a una acción física se ejecuta y se pasa al siguiente robot. Si se llega al final del código (o se llega al comando `exit`) entonces el robot se queda quieto (similar al comando `wait`) y comenzará la ejecución desde el principio en la próxima ronda.

Si el robot realiza una acción que no tiene lugar por las características del mapa (por ejemplo, moverse contra un obstáculo) igual “pierde” su turno (como los carritos de control remoto cuando se traban contra una pared pero igual siguen moviendo las ruedas).

Los comandos propios del entorno del robot se muestran en la siguiente tabla.

Comando	Descripción
move forward	Intenta mover al robot a la casilla próxima (empujando o cargando objetos en dependencia)
move backward	Intenta mover al robot marcha atrás (no puede empujar ni recoger nada en el proceso)
turn right	Gira a la derecha
turn left	Gira a la izquierda
drop	Intenta expulsar el objeto pequeño recogido a la casilla frente al robot (si ya existe un objeto en dicha casilla no tiene efecto)
wait	Decide no moverse en esta ronda
exit	Termina el programa en esta ronda, similar a haber alcanzado el final del código

La siguiente tabla muestra los sensores predeterminados del robot y los valores que devuelven en cada escenario.

Sensor	Atributos	Valores y descripción
Ultrasónico	distance	Entero indicando la cantidad de casillas desocupadas enfrente del robot (dentro de los límites del mundo).
Webcam	color	Entero indicando el color del objeto enfrente. Si no hay objeto se devuelve 0.
Kinect	shape	Entero indicando la forma del objeto enfrente. Si no hay objeto se devuelve 0.
Barcode scanner	number	Entero indicando el código de barra del objeto enfrente. Si no hay objeto se devuelve 0.
GPS	row column direction	Fila, columna y dirección del robot.
Pesa	full	true si hay un objeto en el interior del robot, false en caso contrario.
Cronómetro	time	Entero indicando el tiempo desde que comenzó la “vida” del robot.

En el lenguaje existen distintas constantes enteras para expresar los posibles valores para `color`, `shape`, `direction`.

Los robots tienen una memoria propia para su programa, variables, rutinas, etc. por lo tanto no existe forma de comunicación entre ellos que no sea a través del entorno.



LA APLICACIÓN

La aplicación visual debe permitir diseñar programáticamente un entorno en el que se ubiquen objetos, robots y se ejecute la simulación.

Todos los programas deberán poder guardarse en ficheros para un futuro uso. Los códigos que se hayan creado como bibliotecas también deberán salvarse.

Como parte de la especificación del proyecto se le proveerá de una biblioteca de clases y una aplicación de ejemplo que permite manejar un conjunto de códigos, y que tiene el proceso de análisis léxico resuelto. Usted puede utilizar este proyecto para transformarlo en su propio simulador.

En este documento se exponen un conjunto pequeño de instrucciones, sensores y objetos básicos... No se conforme con ello... Proporcionen nuevos tipos de sensores e instrucciones. Tenga en cuenta que un gran peso de la evaluación lo determina la extensibilidad propuesta. Es decir, cuán fácil es incorporar un nuevo operador, comando, objeto, etc.

Propóngase como meta hacer su solución lo suficientemente extensible como para que se pueda incorporar sin mucha dificultad un posible nuevo operador para la potencia ($2^3 == 8$), o un concepto nuevo como ciclos.

```
move forward until distance==0
```

¡Sea creativo!

No obstante, no puede obviar ninguno de los comandos descritos en este documento puesto que su proyecto será evaluado con códigos propuestos por los profesores.



SOBRE LEGIBILIDAD DEL CÓDIGO

IDENTIFICADORES

Todos los identificadores (nombres de variables, métodos, clases, etc) deben ser establecidos cuidadosamente, con el objetivo de que una persona distinta del programador original pueda comprender fácilmente para qué se emplea cada uno.

Los nombres de las variables deben indicar con la mayor exactitud posible la información que se almacena en ellas. Por ejemplo, si en una variable se almacena “la cantidad de obstáculos que se ha encontrado hasta el momento”, su nombre debería ser `cantidadObstaculosEncontrados` o `cantObstaculos` si el primero le parece demasiado largo, pero nunca `Ob`, `aux`, `temp`, `miVariable`, `juan`, `contador`, `contando` o `paraQueNoCheque`. Note que el último identificador incorrecto es perfectamente legible, pero no indica “qué información se guarda en la variable”, sino quizás “para qué utilizo la información que almaceno ahí”, lo cual tampoco es lo deseado.

- Entre un nombre de variable un poco largo y descriptivo y uno que no pueda ser fácilmente comprensible por cualquiera, es preferible el largo.
- Como regla general, los nombres de variables **no** deben ser palabras o frases que indiquen acciones, como ~~eliminando~~, ~~saltar~~ o ~~parar~~.

Existen algunos (muy pocos casos) en que se pueden emplear identificadores no tan descriptivos para las variables. Se trata generalmente de pequeños fragmentos de código muy comunes que “todo el mundo sabe para qué son”. Por ejemplo:

```
int temp = a;  
a = b;  
b = temp;
```

“Todo el mundo” sabe que el código anterior constituye un intercambio o *swap* entre los valores de las variables `a` y `b`, así como que la variable `temp` se emplea para almacenar por un instante uno de los dos valores. En casi cualquier otro contexto, utilizar `temp` como nombre de variable resulta incorrecto, ya que solo indica que se empleará para almacenar “temporalmente” un valor y en definitiva todas las variables se utilizan para eso.

Como segundo ejemplo, si se quiere ejecutar algo diez veces, se puede hacer

```
for (int i = 0; i < 10; i++)  
...
```

en lugar de

```
for (int iteracionActual = 0; iteracionActual < 10; iteracionActual++)  
...
```

Para los nombres de las propiedades (*properties* en inglés) se aplica el mismo principio que para las variables, o sea, expresar “qué devuelven” o “qué representan”, solo que los identificadores deben comenzar por mayúsculas. No deben ser frases que denoten acciones, abreviaturas incomprensibles, etc.

Los nombres de los métodos deben reflejar “qué hace el método” y generalmente es una buena idea utilizar para ello un verbo en infinitivo o imperativo: `Agregar`, `Eliminar`, `ConcatenarArrays`, `ContarPalabras`, `Arranca`, `Para`, etc.

En el caso de las clases, obviamente, también se espera que sus identificadores dejen claro qué representa la clase: Robot, Obstaculo, Ambiente, Programa.

COMENTARIOS

Los comentarios también son un elemento esencial en la comprensión del código por una persona que lo necesite adaptar o arreglar y que no necesariamente fue quien lo programó o no lo hizo recientemente. Al incluir comentarios en su código, tome en cuenta que no van dirigidos solo a Ud., sino a cualquier programador. Por ejemplo, a lo mejor a Ud. le basta con el siguiente comentario para entender qué hace determinado fragmento de código o para qué se emplea una variable:

```
// lo que se me ocurrió aquel día
```

Evidentemente, a otra persona no le resultarán muy útiles esos comentarios.

Algunas recomendaciones sobre dónde incluir comentarios

- Al declarar una variable, si incluso empleando un buen nombre para ella pueden quedar dudas sobre la información que almacena o la forma en que se utiliza
- Prácticamente en la definición de todos los métodos para indicar qué hacen, las características de los parámetros que reciben y el resultado que devuelven
- En el interior de los métodos que no sean demasiado breves, para indicar qué hace cada parte del método

Es cierto que siempre resulta difícil determinar dentro del código qué es lo obvio y qué es lo que requiere ser comentado, especialmente para Ud. que probablemente no tiene mucha experiencia programando y trabajando con código hecho por otras personas. Es preferible entonces que “por si acaso” comente su código lo más posible.

Otro aspecto a tener en cuenta es que los comentarios son fragmentos de texto en lenguaje natural, en los cuales deberá expresarse lo más claramente posible, cuidando la ortografía, gramática, coherencia, y demás elementos indispensables para escribir correctamente.

Todos estos elementos son importantes para la calidad de todo el código que produzca a lo largo de su carrera, pero además **tendrán un peso considerable en la evaluación de su proyecto de programación.**

ANEXOS - CONSTANTES

Motivo	Nombre	Valor
Valores boolean	true	1
	false	0
Atributo size	empty	0
	small	1
	medium	2
	large	3
Atributo shape	nothing	0
	sphere	1
	box	2
	plant	3
	bot	4
Atributo color	transparent	0
	red	1
	yellow	2
	green	3
	blue	4
	brown	5
	black	6
	white	7
Atributo direction	north	0
	east	1
	south	2
	west	3

Los valores por defecto de creación para los objetos son:

`row=0, column=0, size=2, shape=1, color=blue, number=0`

y para los robots:

`row=0, column=0, color=brown, direction=0, number=0`