



Gobstones - Respuesta

Fundamentos de Programación (Instituto Universitario Escuela Argentina de Negocios)



Capítulo 1: Fundamentos

¿Nunca programaste antes? Aprendé los fundamentos de la programación utilizando Gobstones, un innovador lenguaje gráfico en el que utilizás un tablero con bolitas para resolver problemas.

Lecciones

1. Primeros Programas

1. ¡Hola, computadora!
2. El Tablero
3. El Cabezal
4. Que comience el movimiento
5. Que siga el movimiento
6. Para todos lados
7. El orden de las cosas
8. Sí, esto también se puede romper
9. Nuestras primeras bolitas
10. Más y más bolitas
11. Poné tus primeras bolitas
12. Sacar Bolitas
13. Cuando no hay bolitas
14. Limpiar celda

2. Práctica Primeros Programas

1. Calentando motores
2. Combinando comandos
3. La fila roja
4. Una escalerita
5. Portugal
6. Y ahora una de más cerquita
7. Limpando el jardín
8. Reemplazar bolitas

3. Procedimientos

1. ¿De qué se trata?
2. Un programa un poco largo
3. Las cosas por su nombre
4. Enseñándole tareas a la computadora
5. Procedimientos en acción
6. Escribiendo procedimientos
7. Procedimiento, ¡te invoco!
8. Una definición, "infinitos" usos
9. Procedimientos dentro de otros
10. Dibujamos con imaginación
11. De punta a punta
12. Rojo al borde
13. Adornando el tablero
14. Colores, colores, colores
15. Cuadrado de colores

4. Repetición Simple

1. MoverOeste10
2. La computadora repite por nosotros
3. MoverOeste5 usando repeat
4. No todo es repetir
5. También vale después
6. Repitiendo varios comandos
7. ¿Dónde está el error?
8. Diagonal con una bolita
9. Diagonal "pesada"
10. El caso borde
11. De lado a lado, dibujamos un cuadrado

5. Parámetros

1. Pensando en subtareas
2. Dibujando un cuadrado con subtareas
3. Color esperanza
4. Los que faltan
5. Procedimientos con agujeritos
6. Llenando los espacios vacíos
7. DibujarLinea3
8. DibujarCuadradoDeLado3
9. Pasando varios parámetros
10. La ley, el orden y el BOOM
11. Un argumento para dos parámetros
12. La tercera es la vencida

6. Práctica Repetición simple

1. Entrando en calor... ¡Volviendo!
2. Una diagonal más ancha
3. Pongamos... ¡Todo lo que queramos!
4. Día de la Memoria
5. Escribir cualquier fecha
6. Movamos... ¡Todo lo que queramos!
7. Los números del reloj
8. Una línea heavy
9. Guarda con la guarda
10. Una guarda en L



7. Expresiones

1. Muchas formas de decir lo mismo
2. La suma de las partes
3. ¿Qué se hace antes?
4. La carrera del salmón
5. Dos pasos adelante, un paso atrás
6. Poner al lado
7. La línea que vuelve
8. Dibujando una L
9. Siga la flecha
10. Copiando bolitas
11. Sacando bolitas

8. Alternativa Condicional

1. Sacar con miedo
2. Sacar con miedo, segundo intento
3. Eliminando la bolita roja
4. Un ejemplo medio rebuscado
5. ¿Y sólo sirve para ver si hay bolitas?
6. Un poquito de matemática
7. Cómo decirle que no...
8. Dos caminos distintos
9. Un tablero de luces
10. Un desorden muy especial

9. Funciones

1. Y esto, ¿con qué se come?
2. La importancia de nombrar las cosas
3. MoverSegunBolitas, versión 2
4. todasExcepto
5. Una función de otro tipo
6. En libertad
7. Cualquier bolita nos deja bien
8. Siempre al borde...
9. Las compañeras ideales
10. Lo ideal también se puede romper
11. ¿Hay bolitas lejos?
12. Estoy rodeado de viejas bolitas
13. Sin límites

Apéndice



¡Hola, computadora!

En nuestra vida cotidiana sabemos cómo hacer para comunicarnos con otras personas. Si necesitamos que hagan algo, sabemos que tenemos que pedirles por favor ☺. También sabemos cómo darle órdenes a nuestras mascotas. Pero... ¿cómo hacemos para darle órdenes a una computadora? 😐

¡Enterate mirando este video!

Introducción a Fundamentos con Gobstiones





El Tablero

Para empezar a programar, el primer elemento que vamos a usar es un **tablero** cuadriculado, similar al del Ajedrez, Damas o [Go](http://es.wikipedia.org/wiki/Go) (<http://es.wikipedia.org/wiki/Go>).

Estos tableros pueden ser de cualquier tamaño, por ejemplo,

4x4

0	1	2	3
3			
2			
1			
0			

3x2

0	1	2
1		
0		

Siempre vamos a necesitar un tablero sobre el cual **ejecutar** nuestros **programas**, ¡pero despreocúpate! nos vamos a encargar de crearlos por vos en cada uno de los ejercicios. Lo interesante es que un mismo programa puede ejecutarse sobre distintos tableros, potencialmente produciendo resultados diferentes. 😊

Para que veas lo que te decimos, presioná el botón Continuar, y vamos a generar tu primer tablero: un tablero de 3x3. 😊

Continuar

¡Muy bien!

Tablero final

GBB/1.0 size 3 3 head 0 0

Perfecto, ¡este es tu primer **tablero** de 3x3! 🎉

Notá que ahora la **celda** (casillero) de la esquina inferior izquierda está pintada de otra forma. En el próximo ejercicio veremos por qué. 💡



El Cabezal

3. El Cabezal

El tablero generado en el ejercicio anterior tenía una **celda** marcada:

	0	1	2	
2				2
1				1
0				0
0	1	2		

¿Y eso por qué? 🤔 Porque nuestra máquina tiene un **cabezal**, que en todo momento está situado sobre una de las celdas del tablero y puede realizar distintas operaciones sobre ella (pacienza, ya las vamos a conocer 😊).

Por ejemplo, el siguiente es un tablero de 5x2, con el cabezal en la segunda fila y la cuarta columna.

	0	1	2	3	4	
1						1
0						0
0	1	2	3	4		

¡Algo importante! Contamos las filas hacia arriba, y las columnas hacia la derecha. La primera **fila** es la de **abajo** de todo, y la primera **columna** es la de **la izquierda**.

Vamos a ver otro ejemplo?

Presioná Enviar y generaremos un tablero 3x3 con el cabezal en la segunda columna y tercera fila.

Continuar

¡Muy bien!

Tablero final

	0	1	2	
2				2
1				1
0				0
0	1	2		

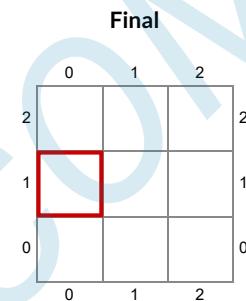
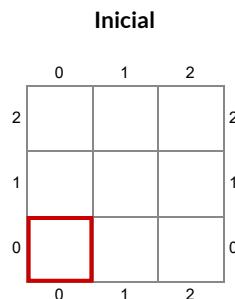
Como podrás ver podemos crear muchísimos tableros distintos, pero ¿el cabezal se va a quedar siempre en el mismo casillero? 😊



Que comience el movimiento

Hasta ahora lo que vimos no fue muy emocionante, porque no te enseñamos cómo darle instrucciones a la máquina y sólo te mostramos un tablero 🏆. En este ejercicio vamos a aprender una de las órdenes que podemos darle a la máquina: mover el cabezal.

Por ejemplo, partiendo de un tablero **inicial** vacío con el cabezal en el origen (abajo a la izquierda), podemos fácilmente crear un programa que mueva el cabezal una posición hacia el **norte**:



El **código** del programa (es decir, el **texto** de la descripción de la solución que le daremos a la computadora) que logra esto es el siguiente:

```
program {
    Mover(Norte)
}
```

¿No nos creés? Escribí el código anterior en el editor y dale Enviar.

¡Dame una pista!

```
1 program {
2     Mover(Norte)
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial			Tablero final		
0	1	2	0	1	2
2			2		
1			1		
0			0		

¡Felicitaciones, creaste tu primer programa! Éste, al ser ejecutado por la máquina, provoca que el cabezal se mueva una posición hacia el Norte.

Pero programar no se trata de copiar y pegar código. Acompañanos al próximo ejercicio para entender qué es lo que hicimos exactamente. ☺

Esta guía fue desarrollada por Franco Bulgarelli, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Que siga el movimiento

Entendamos qué es lo que acabamos de hacer: ¡crear un programa!

Todo programa tiene exactamente un `program`: una sección del código que declara los comandos (acciones) que queremos que la máquina realice sobre el tablero **inicial**. Al **ejecutar** un programa obtendremos un tablero **final**.

La sintaxis de un `program` es bastante simple:

1. escribimos una línea (renglón) que diga `program`, seguido de una llave de apertura: {
2. a continuación, los comandos: uno por línea
3. y finalmente, una última llave que cierra la que abrimos anteriormente }

Vamos a ver algunos ejemplos de `program`s:

- uno que no hace nada

```
program {
}
```

- uno que mueve el cabezal **una** posición hacia el norte

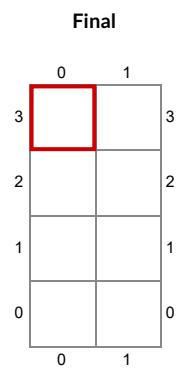
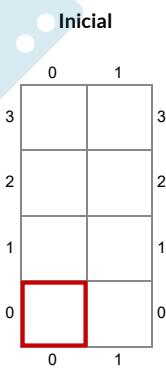
```
program {
    Mover(Norte)
}
```

- uno que mueve el cabezal **dos** posiciones hacia el norte

```
program {
    Mover(Norte)
    Mover(Norte)
}
```

¡Te toca a vos!

Creá un programa que en un tablero de 2x4 con el cabezal en el origen (la celda de abajo a la izquierda), mueva el cabezal tres veces hacia el norte:



¡Dame una pista!

```
1 program {
2     Mover(Norte)
```

```
3 Mover(Norte)  
4 Mover(Norte)  
5 }
```



► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial		Tablero final	
0	1	0	1
3		3	
2		2	
1		1	
0		0	
0	1	0	1

Los lenguajes de programación son creados con algunas palabras que solo se pueden utilizar con un fin determinado. Se las llama **palabras reservadas** . En Gobstones, el lenguaje que estamos utilizando, `program` es una palabra reservada.

Como ya sabemos que nuestros programas son ejecutados por la máquina, de ahora en más diremos "creá un programa que haga ..." en vez de "creá un programa que **provoque que la máquina haga ...**".

Pero ojo: la máquina sigue estando ahí y no hay que olvidarla, sólo hacemos esto para escribir un poco menos.

Esta guía fue desarrollada por Franco Bulgarelli, Federico Aloï, Mumuki Project bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)

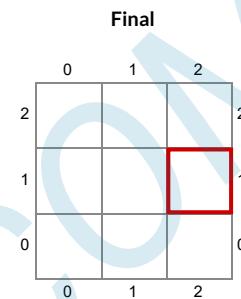
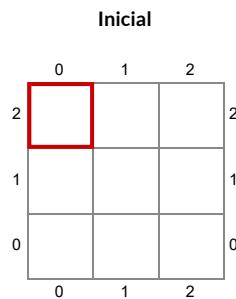




Para todos lados

Como te imaginarás, el cabezal no sólo se puede mover hacia el Norte, y un programa puede combinar cualquier tipo de movimientos.

Creá un programa que mueva el cabezal dos posiciones hacia el **Este** y una hacia el **Sur**, produciendo el siguiente efecto:



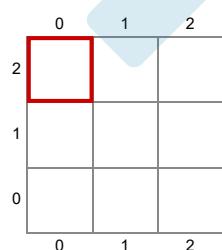
¡Dame una pista!

```
1 program {
2   Mover(Sur)
3   Mover(Este)
4   Mover(Este)
5 }
```

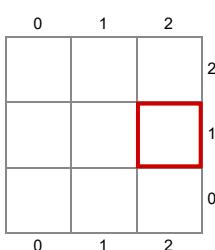
Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial



Tablero final



Notá que estos dos programas hacen lo mismo:

```
program {
    Mover(Este)
    Mover(Este)
    Mover(Sur)
}
```

```
program {
    Mover(Este)
    Mover(Sur)
    Mover(Este)
}
```

Moraleja: como te decíamos al principio ¡no hay una sola forma de resolver un problema!

Y además, el orden, **a veces**, no es tan importante. Acompañanos a entender mejor esto.

Esta guía fue desarrollada por Franco Bulgarelli, Federico Aloi, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





El orden de las cosas

Cuando trabajamos en Gobstones, hacemos las cosas en un cierto orden. Por ejemplo, si tenemos este programa:

```
program {
    Mover(Norte)
    Mover(Este)
}
```

una forma posible de leerlo (llamada **operacional**) es como lo haría una máquina, en orden, de arriba hacia abajo:

1. primero se mueve al norte: Mover(Norte)
2. luego se mueve al este: Mover(Este)

Y de hecho **se ejecuta de esa forma**. Esto es *cómo* lo hace.

Pero, los humanos, solemos pensar en función del resultado final, es decir, resaltamos el **objetivo** del programa. Nos importa más *qué* hace, y no *cómo*. Esta manera denotacional nos llevaría a decir que, simplemente, **mueve el cabezal al noreste**.

Por eso hay varias formas de resolver un mismo problema: podemos crear varios programas que hagan lo mismo (el *qué*), pero que lo hagan de forma diferente (el *cómo*).

Veamos si entendiste esto: creá otro programa que haga lo mismo que el de arriba (mover hacia el noreste), pero de manera distinta. **Ojo:** tiene que funcionar en un tablero de 2x2.

¡Dame una pista!

```
1 program {
2     Mover(Este)
3     Mover(Norte)
4 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1
1	
0	

Tablero final

0	1
1	
0	

¡Perfecto!

Recién creamos un programa en el que el orden no afecta lo que queremos hacer. Pero esto no será siempre así, en breve nos toparemos con problemas en los que hay que tener más cuidado con el orden de los comandos.

Esta guía fue desarrollada por Franco Bulgarelli, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Sí, esto también se puede romper

8. Sí, esto también se puede ro

Hasta ahora veníamos tratando de esquivar el asunto, pero seguro ya habrás pensado que tiene que haber alguna forma de romper esto (incluso es posible que ya te haya pasado).

Si bien **nunca** vamos a querer que nuestro programa se rompa, es algo que definitivamente **te va a pasar muchas veces**. Pero no es motivo para frustrarse ni mucho menos, te aseguramos que a todo el mundo le pasó alguna vez (bueno, también 2, 3, 100, 800 veces...).

Dicho esto, te vamos a mostrar una forma de hacerlo, simplemente para que no te asustes *tanto* cuando te pase de verdad 😊.

¿Y cómo es esa forma? Descubrilo vos: partiendo del tablero que te mostramos acá abajo, creá un programa que provoque que el cabezal se salga fuera de los límites.

	0	1	2	
2				2
1				1
0				0
	0	1	2	

```
1 program {
2   Mover(Sur)
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡BOOM!

Tablero inicial

Tablero final

	0	1	2	
2				2
1				1
0				0
	0	1	2	



BOOM

[2:3]: No se puede mover hacia la dirección Sur: cae afuera del tablero.

¡BOOOOOOOOOOOOOOM! 

Ey, ¿qué pasó?

Tu programa falló, se rompió, o como lo llamamos en el universo Gobstones: **hizo BOOM**.

Y, ¿qué significa esto?

Que los comandos que le diste a la computadora no se pueden ejecutar, y hay algo que vas a tener que cambiar para que funcione. En este ejercicio eso no tiene sentido porque lo hicimos a propósito, pero tenelo en cuenta para cuando falles en el futuro.

Esta guía fue desarrollada por Franco Bulgarelli, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Nuestras primeras bolitas

Genial, ya entendiste cómo mover el cabezal del tablero usando la operación `Mover` y las direcciones (`Sur`, `Oeste`, etc). Vayamos un paso más allá: las **bolitas**.

En cualquier celda de nuestro tablero podemos poner `bolitas`. Las hay de distintos colores:

- rojas (`Rojo`);
- azules (`Azul`);
- negras (`Negro`);
- y verdes (`Verde`).

Por ejemplo, este es un tablero con una bolita roja y una negra:

0	1	
1		1
0		1
0	1	0

Además de moverse, el cabezal también puede poner bolitas en la **celda actual**. Para eso contamos con la operación `Poner`, que le dice al cabezal que deposite una bolita del color dado:

```
program {
    Poner(Rojo)
}
```

¡Probá este programa! Escribí el código en el editor, envialo y verás lo que pasa al ejecutarlo sobre este tablero:

0	1	2	2
1			1
0			0
0	1	2	0

```
1 program {
2     Poner(Rojo)
3 }
```

Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2
2			
1			
0	1		

Tablero final

	0	1	2
2			
1			
0	1		

Felicitaciones! Acabás de escribir un programa que pone una bolita roja en la celda actual.

Esta guía fue desarrollada por Franco Bulgarelli, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Más y más bolitas

Algo interesante de nuestros tableros es que en sus celdas podemos poner cualquier cantidad de bolitas de cualquier color.

Por ejemplo, si tenemos este tablero:

0	1	2	3	4
1				
0				

y ejecutamos el siguiente programa:

```
program {
    Poner(Rojo)
    Poner(Rojo)
    Poner(Azul)
    Poner(Verde)
    Poner(Rojo)
}
```

el cabezal colocará en la celda actual tres bolitas rojas, una azul y una verde.

¡Escribí este programa en el editor y fijate cómo queda el tablero!

```
1 program {
2     Poner(Rojo)
3     Poner(Rojo)
4     Poner(Azul)
5     Poner(Verde)
6     Poner(Rojo)
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	3	4
1				
0				

Tablero final

0	1	2	3	4
1			1 3 1	
0				

Notá que en este problema, si cambiamos el orden en que *llamamos* (usamos a) `Poner`, el resultado no cambia: siempre nos terminará quedando un tablero con tres bolitas rojas, una azul y una verde.

Por ejemplo, los siguientes dos programas también resuelven este mismo problema:

```
program {
    Poner(Rojo)
    Poner(Rojo)
    Poner(Rojo)
    Poner(Verde)
    Poner(Azul)
}
```

```
program {
    Poner(Rojo)
    Poner(Azul)
    Poner(Rojo)
    Poner(Verde)
    Poner(Rojo)
}
```

Esta guía fue desarrollada por Franco Bulgarelli, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Poné tus primeras bolitas

Como te habrás dado cuenta, estos tableros son un poco mágicos, podemos poner en una celda tantas bolitas como queramos: 2, 4, 12, 50, 1000. ¡No hay ningún límite!

Esto es algo muy interesante que ocurre al programar: podemos trabajar con cantidades tan grandes como queramos.

Ah, y ahora te toca a vos: creá un programa que ponga cuatro bolitas rojas y tres bolitas negras en la celda actual.

¡Dame una pista!

```
1 program {
2   Poner(Rojo)
3   Poner(Rojo)
4   Poner(Rojo)
5   Poner(Rojo)
6   Poner(Negro)
7   Poner(Negro)
8   Poner(Negro)
9 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial			Tablero final		
0	1	2	0	1	2
2			2		
1			1		
0			0		
0			0		
0			0		
0			0		
0			0		



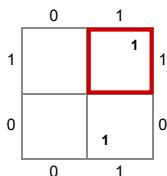
Sacar Bolitas

De la misma forma que hay un "poner bolita" (Poner), tenemos un "sacar bolita" (Sacar), que quita exactamente una bolita del color dado.

Por ejemplo, el siguiente programa saca dos bolitas de la posición inicial.

```
program {
    Sacar(Rojo)
    Sacar(Rojo)
}
```

Sabiendo esto, creá un programa que elimine sólo la bolita roja de este tablero. ¡Tené cuidado! Prestá atención a la posición del cabezal ☺.



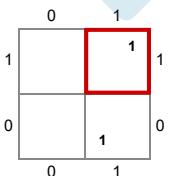
¡Dame una pista!

```
1 program {
2     Mover(Sur)
3     Sacar(Rojo)
4 }
```

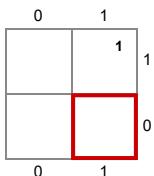
Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial



Tablero final



¿Y si no hubiera ninguna bolita para sacar? ☺



Cuando no hay bolitas

Cada vez que usamos `Sacar`, tenemos que tener más cuidado que con `Poner`, porque...

¿Querés saber por qué? Intentá sacar una bolita verde o azul de este tablero y descubrilo.

0	1
1	
	1
0	1

```
1 program {
2   Sacar(Verde)
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡BOOM!

Tablero inicial

0	1
1	
	1
0	1

Tablero final



BOOM

[1:9]: No se puede sacar una bolita de color Verde: no hay bolitas de ese color.

Claro, otra vez BOOM.

Esta vez lo que pasó fue que el cabezal intentó sacar una bolita de un color que no había, y como no sabía qué hacer se autodestruyó. Esto te va a pasar siempre que intentes sacar una bolita que no exista, así que ¡a prestar atención!



Limpiar celda

Un último esfuercito: usando `Sacar`, creá un programa que elimine todas las bolitas de este tablero:

	0	1	
1			1
0	1	1	0
0	1	1	0

```
1 program {
2   Sacar(Azul)
3   Sacar(Verde)
4   Sacar(Negro)
5   Sacar(Rojo)
6 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	
1			1
0	1	1	0
0	1	1	0

Tablero final

	0	1	
1			1
0			0
0			0



Calentando motores

¡Vemos un primer ejemplo!

El siguiente programa coloca una bolita roja en la posición inicial y una negra al este.

```
program {
    Poner(Rojo)
    Mover(Este)
    Poner(Negro)
}
```

Probá escribir y ejecutar este programa. Te mostraremos el resultado al ejecutarlo en un tablero de 2x2, y en otro de 3x2, ambos con el cabezal inicialmente en el origen.

¡Dame una pista!

```
1 program {
2     Poner(Rojo)
3     Mover(Este)
4     Poner(Negro)
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero de 2x2

		0	1
1	1		1
0	1		0
	0	1	

Tablero inicial

Tablero final

Tablero de 3x2

			0	1	2
1	1		1		0
0	1		0		2
	0	1	2		

Tablero inicial

Tablero final

Ahora que combinamos operaciones, la cosa se pone un poco mas complicada, porque hay que tener **más cuidado con el orden**.

Por ejemplo, mirá el programa que escribiste:

```
program {
    Poner(Rojo)
    Mover(Este)
    Poner(Negro)
}
```

Operacionalmente:

1. pone una roja
2. luego se mueve al este
3. luego pone una negra

Es decir: pone una roja en la posición inicial, y una negra al este

Y ahora mirá este otro:

```
program {
    Mover(Este)
    Poner(Rojo)
    Poner(Negro)
}
```

Operacionalmente:

1. se mueve al este
2. luego pone una roja
3. luego pone una negra

Es decir: pone una roja y una negra al este de la posición inicial.

Moraleja: ¡no hacen lo mismo! Cambiar el orden nos cambió el qué.

Esta guía fue desarrollada por Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Combinando comandos

Creá un programa que ponga dos bolitas en la posición inicial, y otras dos en la celda de al lado hacia el Este. Todas las bolitas deben ser rojas.

Acá te dejamos un ejemplo de cómo debería quedar el tablero:

	0	1	2	
2				2
1				1
0				0
	0	1	2	

```
1 program {
2   Poner(Rojo)
3   Poner(Rojo)
4   Mover(Este)
5   Poner(Rojo)
6   Poner(Rojo)
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	
1			1
0			0
	0	1	

Tablero final

	0	1	
1			1
0			0
	0	1	



La fila roja

Creá un programa que a partir de un tablero vacío con el cabezal en el origen, dibuje una linea de cuatro celdas **hacia el Este**. Las bolitas deben ser rojas y debe poner una bolita por celda.

Ademas, el cabezal debe quedar en el extremo final de la línea, como se ve en la imagen:

	0	1	2	3	
1					1
0	1	1	1	1	0

```
1 program {
2   Poner(Rojo)
3   Mover(Este)
4   Poner(Rojo)
5   Mover(Este)
6   Poner(Rojo)
7   Mover(Este)
8   Poner(Rojo)
9 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3	
1					1
0	1	1	1	1	0

Tablero final

	0	1	2	3	
1					1
0	1	1	1	1	0



Una escalera

Usando las herramientas que ya conocés, creá un programa que dibuje una escalera azul como la que se ve en la imagen. El cabezal empieza en el origen (o sea, en el borde Sur-Oeste) y debe quedar en el extremo inferior derecho de la escalera.

Mirá la imagen:

	0	1	2	3	
3					3
2	1				2
1	1				1
0	1	1	1		0
	0	1	2	3	

```
1 program {
2   Poner(Azul)
3   Mover(Norte)
4   Poner(Azul)
5   Mover(Norte)
6   Poner(Azul)
7   Mover(Este)
8   Mover(Sur)
9   Poner(Azul)
10  Mover(Sur)
11  Poner(Azul)
12  Mover(Este)
13  Poner(Azul)
14
15 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3	
3					3
2					2
1					1
0					0
	0	1	2	3	

Tablero final

	0	1	2	3	
3					3
2					2
1					1
0	1	1	1		0
	0	1	2	3	



Portugal

Creá un programa que dibuje una bandera portuguesa.

La bandera de Portugal se ve así:



Como no nos vamos a poner tan quisquillosos, te vamos a pedir una versión simplificada, que se tiene que ver así:

0	1	2	
1	1	1	1
0	1	1	0
0	1	2	0

Ah, el cabezal empieza en el origen.

```
1 program {
2   Poner(Verde)
3   Mover(Norte)
4   Poner(Verde)
5   Mover(Este)
6   Poner(Rojo)
7   Mover(Sur)
8   Poner(Rojo)
9   Mover(Este)
10  Poner(Rojo)
11  Mover(Norte)
12  Poner(Rojo)
13 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	
1			1
0			0
0	1	2	0

Tablero final

0	1	2	
1	1	1	1
0	1	1	0
0	1	2	0



Y ahora una de más cerquita

Ya lograste dibujar la bandera de Portugal con Gobstones. Ahora probemos hacer una bandera de Latinoamérica. ¿Te animás a dibujar la de Argentina?

Aunque como en Gobstones no hay amarillo, nos vamos a tomar el atrevimiento de cambiarlo por rojo (perdón Belgrano (https://es.wikipedia.org/wiki/Manuel_Belgrano), no nos queda otra 😊).

Con el cabezal en el origen, tu tarea es dibujar esta pseudo-bandera argentina:

0	1	2	3	4	2
1	1	1	1	1	1
			1		
1	1	1	1	1	1

```
1 program {
2   Poner(Azul)
3   Mover(Este)
4   Poner(Azul)
5   Mover(Este)
6   Poner(Azul)
7   Mover(Este)
8   Poner(Azul)
9   Mover(Este)
10  Poner(Azul)
11  Mover(Norte)
12  Mover(Norte)
13  Poner(Azul)
14  Mover(Oeste)
15  Poner(Azul)
16  Mover(Oeste)
17  Poner(Azul)
18  Mover(Sur)
19  Poner(Rojo)
20  Mover(Norte)
21  Mover(Oeste)
22  Poner(Azul)
23  Mover(Oeste)
24  Poner(Azul)
25  Mover(Sur)
26  Mover(Sur)
27 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3	4	
2						2
1						1
0	1					0

Tablero final

	0	1	2	3	4	
2	1	1	1	1	1	2
1			1			1
0	1	1	1	1	1	0

No quedó muy lindo el programa, ¿no?

Por ahora no podemos hacer nada mejor, pero para que lo vayas pensando: una particularidad de nuestra bandera es su simetría, la franja de arriba es exactamente igual a la de abajo. Si pudieramos crear un comando que dibuje la franja celeste nuestro programa quedaría mucho más simple...

Esta guía fue desarrollada por Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Limiando el jardín

¡Este jardín necesita una buena podada!

0	1	2	
2	1	1	1
1	1		1
0	1	1	1
0	1	2	



Aunque no sabemos bien por qué Willie piensa que nuestro tablero es un jardín, mejor hagámosle caso, no sea que tengamos que escucharlo por horas hablando de hazañas escocesas.

Con el cabezal en el origen, creá un programa que se encargue de "podar" el tablero de la imagen: sacar todas las bolitas verdes. Al finalizar, el cabezal debe terminar donde empezó.

```

1 program {
2   Sacar(Verde)
3   Mover(Este)
4   Sacar(Verde)
5   Mover(Este)
6   Sacar(Verde)
7   Mover(Norte)
8   Sacar(Verde)
9   Mover(Norte)
10  Sacar(Verde)
11  Mover(Oeste)
12  Sacar(Verde)
13  Mover(Oeste)
14  Sacar(Verde)
15  Mover(Sur)
16  Sacar(Verde)
17  Mover(Sur)
18 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	
2	1	1	1
1	1		1
0	1	1	1
0	1	2	

Tablero final

0	1	2	
2			
1			
0			
0	1	2	



Reemplazar bolitas

¿Ya te estás durmiendo? 😴

Pasemos a algo un poco más difícil entonces. Te vamos a dar un tablero de **2x2** (o sea, con 4 celdas) donde cada una de ellas tiene **una bolita roja**.

Tu tarea es crear un programa que **reemplace** todas las bolitas rojas por verdes.

Inicial	
0	1
1	1
0	1

Final	
0	1
1	1
0	1

¡Dame una pista!

```
1 program {
2   Sacar(Rojo)
3   Poner(Verde)
4   Mover(Este)
5   Sacar(Rojo)
6   Poner(Verde)
7   Mover(Norte)
8   Sacar(Rojo)
9   Poner(Verde)
10  Mover(Oeste)
11  Sacar(Rojo)
12  Poner(Verde)
13 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1
1	1
0	1

Tablero final

0	1
1	1
0	1



¿De qué se trata?

Tomate unos minutos (no más de 3 🕒) para tratar de descubrir qué es lo que hace el programa a continuación.

```
program {
    Poner(Negro)
    Mover(Este)
    Poner(Negro)
    Mover(Este)
    Poner(Negro)
    Mover(Norte)
    Poner(Negro)
    Mover(Oeste)
    Poner(Negro)
    Mover(Oeste)
    Poner(Negro)
    Mover(Norte)
    Poner(Negro)
    Mover(Este)
    Poner(Negro)
    Mover(Este)
    Poner(Negro)
}
```

¿Lo pensaste? Decinos qué es lo que hace 😊

- Llena el tablero de bolitas negras
- Dibuja una línea de bolitas negras
- Dibuja una cruz de bolitas negras
- Dibuja un cuadrado de bolitas negras
- Pone 9 bolitas negras en una celda

Enviar

¡La respuesta es correcta!

Costó un poco, ¿no?

Bueno, eso es normal porque **los humanos no somos máquinas** 🤖 y nos cuesta ejecutar un programa en nuestra cabeza. ¡Por suerte existen las computadoras, que se encargan de resolverlo!



Un programa un poco largo

2. Un programa un poco largo

Ahora tenés la posibilidad de ver en acción el programa.

```
program {
    Poner(Negro)
    Mover(Este)
    Poner(Negro)
    Mover(Este)
    Poner(Negro)
    Mover(Norte)
    Poner(Negro)
    Mover(Oeste)
    Poner(Negro)
    Mover(Oeste)
    Poner(Negro)
    Mover(Norte)
    Poner(Negro)
    Mover(Este)
    Poner(Negro)
    Mover(Este)
    Poner(Negro)
}
```

¡Presioná Continuar para comprobar tu respuesta anterior!

Continuar

¡Muy bien!

Tablero inicial			
0	1	2	3
3			
2			
1			
0	1		

Tablero final			
0	1	2	3
3			
2	1	1	1
1	1	1	1
0	1	1	1

Aunque ahora pudimos probarlo, sigue siendo un poco confuso saber de qué se trata el programa con solo leerlo ¿no sería mejor si también fuera fácil de entender para un humano? ☺



Las cosas por su nombre

Mirá esta nueva versión del mismo programa. Aunque todavía hay elementos de la **sintaxis** que no conocés, confiamos en que vas a tardar mucho menos en descubrir qué hace.

Envíá el código, así nos aseguramos de que **hace exactamente lo mismo** que el anterior.

```
1 procedure DibujarCuadradoNegroDeLado3() {  
2   Poner(Negro)  
3   Mover(Este)  
4   Poner(Negro)  
5   Mover(Este)  
6   Poner(Negro)  
7   Mover(Norte)  
8   Poner(Negro)  
9   Mover(Oeste)  
10  Poner(Negro)  
11  Mover(Oeste)  
12  Poner(Negro)  
13  Mover(Norte)  
14  Poner(Negro)  
15  Mover(Este)  
16  Poner(Negro)  
17  Mover(Este)  
18  Poner(Negro)  
19 }  
20  
21 program {  
22   DibujarCuadradoNegroDeLado3()  
23 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial			
0	1	2	3
3			.
2			
1			
0	1		

Tablero final			
0	1	2	3
3			
2	1	1	1
1	1	1	1
0	1	1	1

Mucho más fácil de entender, ¿no? 😊

Probablemente te estés preguntando *¿cómo supo la computadora lo que tenía que hacer DibujarCuadradoNegroDeLado3 ?* ;*Qué es eso de procedure ?* 🤔

¡Vamos a averiguarlo!



Enseñándole tareas a la computadora

Como viste en el ejemplo del cuadrado, se puede empezar a diferenciar dos tipos de comandos dentro de un programa:

- los que vienen definidos por el lenguaje y nos sirven para expresar operaciones básicas, como `Mover`, `Poner` y `Sacar`. A estos los llamaremos **comandos primitivos**, o simplemente **primitivas**;
- y los que **definimos nosotros**, que nos sirven para expresar tareas más complejas. Como el nombre de esta lección sugiere, estos son los **procedimientos**.

Cuando *definimos* un procedimiento estamos "enseñándole" a la computadora a realizar una tarea nueva, que originalmente no estaba incluida en el lenguaje.

Prestale atención a la sintaxis del ejemplo para ver bien cómo definimos un procedimiento y cómo lo *invocamos* en un `program`.

```
procedure Poner3Rojas() {
    Poner(Rojo)
    Poner(Rojo)
    Poner(Rojo)
}

program {
    Poner3Rojas()
}
```

¿Qué te parece que hace el nuevo procedimiento? Copiá y envíá el código para ver qué pasa.

```
1 procedure Poner3Rojas() {
2     Poner(Rojo)
3     Poner(Rojo)
4     Poner(Rojo)
5 }
6
7 program {
8     Poner3Rojas()
9 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	
2				2
1				1
0	3			0
	0	1	2	

Tablero final

	0	1	2	
2				2
1				1
3				0
	0	1	2	

Ahora que ya probamos cómo funcionan, podemos ver las diferencias entre las sintaxis de **programas** y **procedimientos**.

El procedimiento se define con la palabra `procedure` seguida por un nombre y paréntesis `()`. Luego escribimos entre llaves `{}` todas las acciones que incluya. Para ver un procedimiento en acción hay que invocarlo dentro de un programa, si no sólo será una descripción que nunca se va a ejecutar. ☹

El programa se crea con la palabra `program` seguida de llaves `{}`, y adentro de ellas lo que queremos que haga la computadora. ▲ ¡No lleva nombre ni paréntesis!

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Procedimientos en acción

Si bien las palabras que utilizamos para crear programas (`program`) y definir procedimientos (`procedure`) se escriben parecido son cosas muy distintas.

Cuando creamos un programa nuevo le estamos diciendo a la computadora lo que queremos que suceda luego de tocar Enviar.

Pero si queremos crear una tarea nueva podemos agrupar las acciones que requiere en un procedimiento. Los procedimientos se definen con un **nombre** que describa lo que hace.

Veamos cómo creamos un nuevo procedimiento llamado `PonerVerdeYAzul`.

```
procedure PonerVerdeYAzul() {  
    Poner(Verde)  
    Poner(Azul)  
}
```

La computadora solo va a seguir las instrucciones dentro de un procedimiento cuando sea **invocado** dentro de un `program`. ¿Cómo lo invocamos? Escribimos su nombre seguido por paréntesis () .

```
program {  
    PonerVerdeYAzul()  
}
```

Completa el código para que además de **definir** el procedimiento `PonerNegroYRojo` luego lo **invogue** en el `program`.

```
1 procedure PonerNegroYRojo() {  
2     Poner(Negro)  
3     Poner(Rojo)  
4 }  
5  
6 program {  
7     PonerNegroYRojo()  
8 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1
1	
0	1

Tablero final

0	1
1	
0	1

0	1
1	
1	0

A la hora de **definir** e **invocar** procedimientos tenemos que prestar mucha atención a la sintaxis para no perder de vista el objetivo del problema por un error de escritura. 

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloi bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Escribiendo procedimientos

6. Escribiendo procedimientos

Llegó el momento de programar desde cero. ¡No te preocupes! Como recién estamos empezando, repasemos lo que aprendimos.

A esta altura ya sabemos que para programar siempre tenemos que tener en cuenta la sintaxis y que para definir nuevos procedimientos también tenemos reglas:

- empezamos con la palabra reservada `procedure` ;
- elegimos un nombre que lo describa y lo escribimos con mayúscula seguido de paréntesis `()` ;
- encerramos las acciones que queremos que haga entre llaves `{}` .

Entonces, un procedimiento que se mueve cuatro celdas al Norte se va a definir así:

```
procedure Mover4AlNorte() {  
    Mover(Norte)  
    Mover(Norte)  
    Mover(Norte)  
    Mover(Norte)  
}
```

Y si lo queremos utilizar, tenemos que invocarlo dentro del `program` escribiendo su nombre tal cual y sin olvidar los paréntesis `()` ! Prestá atención a la sintaxis!

```
program {  
    Mover4AlNorte()  
}
```

¡Ahora te toca a vos! 🎉

Definí un procedimiento `Poner3Verdes` que ponga 3 bolitas verdes en la celda actual e **invocalo** en el `program`.

```
1 procedure Poner3Verdes() {  
2     Poner(Verde)  
3     Poner(Verde)  
4     Poner(Verde)  
5 }  
6  
7 program {  
8     Poner3Verdes()  
9 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1
1	
0	

Tablero final

0	1
1	
0	

Resumiendo, en lo que a un procedimiento respecta, se pueden distinguir dos momentos:

- la **definición**, que es cuando ponemos `procedure Poner3Verdes()` y el bloque de código que especifica qué hace.
- el **uso o invocación**, que es cuando escribimos `Poner3Verdes()` en alguna parte de `program` (o de otro procedimiento).

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloi bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Procedimiento, ¡te invoco!

Algo **MUY** importante es que un procedimiento se define **una sola vez** y luego se puede usar **todas las veces que necesitemos**, como cualquier otro comando.

Por eso, su nombre debe ser **único** dentro de todo el programa. Recordá que la computadora no hace más que seguir órdenes y si existiera más de un procedimiento con el mismo nombre, no sabría cuál elegir. 😞

Para facilitarte las cosas, a veces te ofreceremos partes del problema resuelto, para que sólo tengas que enfocarte en lo que falta resolver. ¿Y dónde están? Mirá el editor y fijate si encontrás la pestaña **Biblioteca** 📁. Lo que aparece en la Biblioteca no hace falta que lo escribas en el código, ¡si está ahí podés invocarlo directamente! 🎉

¡Vamos a probarlo! Queremos poner 3 bolitas verdes en dos celdas consecutivas como muestra el tablero:

	0	1	2	
1				1
0	3	3		0
	0	1	2	

Creá un **programa** que lo haga invocando el procedimiento `Poner3Verdes`. Recordá que ya te lo damos definido ¡no tenés que volver a escribirlo!

Solución

Biblioteca

```

1 program {
2   Poner3Verdes()
3   Mover(Este)
4   Poner3Verdes()
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	
1				1
0	3			0
	0	1	2	

Tablero final

	0	1	2	
1				1
0		3	3	0
	0	1	2	

Repasemos:

- Para definir un procedimiento nuevo usamos `procedure` y le ponemos un nombre que describa lo que hace seguido por paréntesis `()`. A continuación y entre llaves `{}` lo que querremos que haga.
- Los procedimientos se definen **una sola vez**. Si ya están definidos en la Biblioteca o en alguna parte del código no hay que volver a hacerlo (no se pueden repetir los nombres, Δ si se define un procedimiento más de una vez nuestro programa va a fallar).
- Para invocar un procedimiento escribimos su nombre (sin olvidar los paréntesis `()` al final). ¡Y podemos hacerlo todas las veces que sean necesarias! ☺
- Aunque también se escribe entre llaves `{}`, el `program` **nunca** lleva nombre ni paréntesis.

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Una definición, "infinitos" usos

Otro procedimiento que ya usamos antes es `Poner3Rojas`, que pone tres bolitas rojas en una celda. Te ahorraremos el trabajo de escribirlo, si lo buscás vas a ver que está definido en la Biblioteca. ¡Ahora podés utilizarlo tantas veces como quieras!

Creá un programa que ponga 9 bolitas rojas en la celda actual invocando el procedimiento `Poner3Rojas` todas las veces que sea necesario.

¡Dame una pista!

Solución

Biblioteca

```
1 program {
2   Poner3Rojas()
3   Poner3Rojas()
4   Poner3Rojas()
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial	Tablero final
 0 1 1 0	 0 1 1 0

No te olvides de revisar las herramientas que nos ofrece la Biblioteca para saber cuáles podemos aprovechar cuando resolvamos nuestro problema.

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Procedimientos dentro de otros

Cuando creamos procedimientos agrupamos varias acciones en una tarea que podemos reconocer y nombrar. Eso hace nuestros programas más claros, legibles y nos ahorra repeticiones innecesarias. ☺

Ya vimos que un procedimiento puede ser invocado tantas veces como queramos dentro de un programa 🎉, pero como su objetivo es agrupar los pasos de una tarea para usarla cuando haga falta, también lo podemos invocar dentro de otros procedimientos. ¡Vamos a probarlo!

Definí el procedimiento `Poner9Rojas` que, utilizando `Poner3Rojas`, ponga nueve bolitas rojas en una celda. Una vez definido, invocá el nuevo procedimiento en un program .

¡Dame una pista!

Solución

Biblioteca

```
1 procedure Poner9Rojas() {  
2   Poner3Rojas()  
3   Poner3Rojas()  
4   Poner3Rojas()  
5 }  
6  
7 program {  
8   Poner9Rojas()  
9 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial	Tablero final																
<p>Initial 2x2 board state:</p> <table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	0	1	1	0	0	1	1	0	<p>Final 2x2 board state:</p> <table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td></tr><tr><td>9</td><td>0</td></tr></table>	0	1	1	0	0	1	9	0
0	1																
1	0																
0	1																
1	0																
0	1																
1	0																
0	1																
9	0																

Tablero inicial	Tablero final																		
<p>Initial 3x3 board state:</p> <table border="1"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>	0	1	2	1	0	1	0	1	2	<p>Final 3x3 board state:</p> <table border="1"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>9</td></tr></table>	0	1	2	1	0	1	0	1	9
0	1	2																	
1	0	1																	
0	1	2																	
0	1	2																	
1	0	1																	
0	1	9																	

Bueno, ya sabemos cómo crear

This document is available free of charge on

StuDocu.com

Este archivo fue descargado de https://moodle.com/

Algunas posibles respuestas:

- para **simplificar código**, escribiendo una sola vez y en un solo lugar cosas que vamos a hacer muchas veces;
- para **escribir menos**, por qué queríramos hacer cosas de más; 
- para que el propósito de nuestro programa **sea más entendible para los humanos**, como vimos en el ejemplo de `DibujarCuadradoNegroDeLado3`. Para esto es fundamental **pensar buenos nombres**, que no sean muy largos (`DibujarCuadradoNegroDeLado3FormadoPor9BolistasDeArribaAAabajo`), ni demasiado cortos (`DibCuaNeg`), y sobre todo que **dejen en claro qué hace nuestro procedimiento**;
- para comunicar la **estrategia** que pensamos para resolver nuestro **problema**;
- y como consecuencia de todo esto: para **poder escribir programas más poderosos**. 

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloj bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Dibujamos con imaginación

10. Dibujan

Vamos a usar un poco la imaginación y vamos a hacer un procedimiento que dibuje una "punta" negra en la esquina inferior izquierda de esta forma:

0	1	2	
2			2
1	1		1
0	1	1	0
0	1	2	

Definí el procedimiento `DibujarPuntaNegra` e invocalo dentro de un `program`. El cabezal comienza en el origen.

```

1 procedure DibujarPuntaNegra() {
2   Poner(Negro)
3   Mover(Este)
4   Poner(Negro)
5   Mover(Norte)
6   Mover(Oeste)
7   Poner(Negro)
8 }
9
10 program {
11   DibujarPuntaNegra()
12 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial			
0	1	2	
2			2
1			1
0	1	1	0
0	1	2	

Tablero final			
0	1	2	
2			2
1	1		1
0	1	1	0
0	1	2	

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





De punta a punta

En el ejercicio anterior ya dibujamos una punta, ahora vamos a pensar cómo aprovechar el procedimiento que creamos para lograr un tablero como este:

0	1	2	3	
3			1	
2			1	1
1	1			
0	1	1		
0	1	2	3	

Definí el procedimiento `DibujarDosPuntas` e invocalo dentro un `program`. Acordate de utilizar `DibujarPuntaNegra`.

¡Dame una pista!

Solución

Biblioteca

```

1 procedure DibujarDosPuntas() {
2   DibujarPuntaNegra()
3   Mover(Este)
4   Mover(Norte)
5   Mover(Norte)
6   DibujarPuntaNegra()
7 }
8
9 program {
10   DibujarDosPuntas()
11 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	3	
3				
2				
1				
0				
0	1	2	3	

Tablero final

0	1	2	3	
3			1	
2			1	1
1				
0	1	1		
0	1	2	3	

Para resolver este problema lo que hicimos fue separarlo en partes, identificando las tareas más pequeñas que ya teníamos resueltas. ↗

Los procedimientos son muy útiles para esto, se ocupan de resolver una *subtarea* y nos permiten repetirla o combinarla para solucionar un problema mayor que la incluya.

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Rojo al borde

Ya vimos que los comandos que vienen definidos por el lenguaje se llaman **primitivas**. Hay una primitiva que no usaste hasta ahora que queremos presentarte.

Imagina que no sabés ni dónde está el cabezal ni qué tamaño tiene el tablero pero querés llegar a una esquina: La primitiva Mover no te va a ser de mucha ayuda. 🤔

Por suerte 🎉 existe una primitiva 📦 llamada **IrAlBorde**, que toma una dirección, y se mueve todo lo que pueda en esa dirección, **hasta llegar al borde**.

¿Cómo? Mirá el resultado del siguiente programa:

```
program {
    IrAlBorde(Este)
}
```



¡Vamos a aprovecharlo!

Definí el procedimiento **RojoAlBorde** que ponga una bolita roja en la esquina superior izquierda del tablero e invocalo en el **program**.

¡Dame una pista!

```
1 procedure RojoAlBorde() {
2     IrAlBorde(Oeste)
3     IrAlBorde(Norte)
4     Poner(Rojo)
5 }
6
7 program {
8     RojoAlBorde()
9 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero de 2x3 con el cabezal abajo a la izquierda

Tablero inicial

0	1
2	
1	
0	

Tablero final

0	1
2	
1	
0	

✓ Tablero de 3x3 con el cabezal en el medio

Tablero inicial

0	1	2
2		
1		
0		

Tablero final

0	1	2
2		
1		
0		

¡Excelente! 🙌

IrAlBorde es una primitiva muy útil para cuando no conocemos las condiciones de nuestro tablero. 😊

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Adornando el tablero

Para resolver un problema nos conviene comprender bien de qué se trata para elegir una estrategia. Es el momento de empezar a hacerlo aprovechando los procedimientos.

Uno de los objetivos al usar procedimientos es identificar y nombrar las subtareas ↗ que conforman un problema y combinar sus soluciones para poder resolverlo. Veamos un ejemplo:

Queremos decorar con guirnaldas las dos esquinas superiores de *cualquier* tablero como muestra la imagen.

0	1	2	3	2
2	3	3		
1				1
0				0
0	1	2	3	

Pensemos una estrategia distinguiendo subtareas:

Cada guirnalda se compone de 3 bolitas rojas y 3 bolitas verdes. Ya resolvimos cómo hacerlo en otros ejercicios 📚, hacer una guirnalda solo requerirá combinar esas soluciones. Y ponerla donde corresponda, claro.

¿Y que más? 😊 el procedimiento que decore el tablero debería poder aprovechar la creación de una guirnalda para usarla varias veces en las posiciones que querramos decorar. Nos vendría muy bien alguna primitiva que nos ayude a llegar a los bordes. 😊

¡Manos a la obra!

Definí dos procedimientos: el procedimiento `PonerGuirnalda` que coloque 3 bolitas rojas y 3 bolitas verdes en una celda y el procedimiento `DecorarTablero` que lo utilice y ponga una guirnalda en cada esquina superior. Invocá `DecorarTablero` en el program .

Dame una pista!

Solución

Biblioteca

```

1 procedure PonerGuirnalda() {
2   Poner3Rojas()
3   Poner3Verdes()
4 }
5
6 procedure DecorarTablero() {
7   IrAlBorde(Norte)
8   IrAlBorde(Oeste)
9   PonerGuirnalda()
10  IrAlBorde(Este)
11  PonerGuirnalda()
12 }
13
14 program {
15   DecorarTablero()
16
17 }
```

Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:



Tablero inicial

	0	1	2	3	
2					2
1					1
0					0

Tablero final

	0	1	2	3	
2	3	3			2
1					1
0					0



Tablero inicial

	0	1	2	3	4	
4						4
3						3
2						2
1						1
0						0

Tablero final

	0	1	2	3	4	
4	3	3			4	4
3					3	3
2						
1						
0						

Cuanto más complejo sea el problema, más útil nos va a ser pensar una estrategia y organizar la solución en subtareas ¡y los procedimientos están para ayudarnos!

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloi bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Colores, colores, colores

Vamos a darle un poco más de color a todo esto haciendo líneas multicolores como esta:

0	1	2	3	4	
1	1 1	1 1	1 1	1 1	1
0	1 1	1 1	1 1	1 1	0
0	1	2	3	4	

Como se ve en la imagen, cada celda de la línea debe tener una bolita de cada color (una roja, una negra, una verde y una azul).

¿Cómo podemos dibujarla? ¿Cuál es la tarea que se repite? ¿Se puede definir un nuevo procedimiento para resolverla y aprovecharlo para construir nuestra solución?

Definí un procedimiento `DibujarLineaColorida` que **dibuje una línea multicolor** de cuatro celdas hacia el `Este` y al finalizarla ubique el cabezal en la celda inicial. Invocá el nuevo procedimiento en un `program`.

¡Dame una pista!

```

1 procedure Poner4Bolitas() {
2   Poner(Azul)
3   Poner(Negro)
4   Poner(Rojo)
5   Poner(Verde)
6 }
7
8 procedure DibujarLineaColorida() {
9   Poner4Bolitas()
10  Mover(Este)
11  Poner4Bolitas()
12  Mover(Este)
13  Poner4Bolitas()
14  Mover(Este)
15  Poner4Bolitas()
16  IrAlBorde(Oeste)
17 }
18 program{
19   DibujarLineaColorida()
20 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial

	0	1	2	3
3				
2				
1	1			
0				

Tablero final

	0	1	2	3
3				
2				
1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1
0				



Tablero inicial

	0	1	2	3	4
1	1				
0					

Tablero final

	0	1	2	3	4
1	1 1 1 1				
0					

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Cuadrado de colores

Vamos a crear un procedimiento que nos permita dibujar un tablero como este:

	0	1	2	3	4
4					
3	1 1	1 1	1 1	1 1	
2	1 1	1 1	1 1	1 1	
1	1 1	1 1	1 1	1 1	
0	1 1	1 1	1 1	1 1	
	0	1	2	3	4

Definí un procedimiento `DibujarCuadradoColorido` que dibuje un cuadrado de 4x4 celdas en el que cada celda tenga una bolita de cada color e invocalo en el `program`. El cabezal debe quedar en la celda inicial.

[¡Dame una pista!](#)

[Solución](#)

[Biblioteca](#)

```

1 procedure DibujarCuadradoColorido() {
2   DibujarLineaColorida()
3   Mover(Norte)
4   DibujarLineaColorida()
5   Mover(Norte)
6   DibujarLineaColorida()
7   Mover(Norte)
8   DibujarLineaColorida()
9   IrAlBorde(Sur)
10 }
11 program{
12   DibujarCuadradoColorido()
13 }
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3
3				
2				
1				
0				
	0	1	2	3

Tablero final

	0	1	2	3
3	1 1	1 1	1 1	1 1
2	1 1	1 1	1 1	1 1
1	1 1	1 1	1 1	1 1
0	1 1	1 1	1 1	1 1
	0	1	2	3



MoverOeste10

Entremos en calor: definí un procedimiento `MoverOeste10` que mueva el cabezal 10 veces hacia el Oeste.

¡Dame una pista!

```
1 procedure MoverOeste10() {  
2   Mover(Oeste)  
3   Mover(Oeste)  
4   Mover(Oeste)  
5   Mover(Oeste)  
6   Mover(Oeste)  
7   Mover(Oeste)  
8   Mover(Oeste)  
9   Mover(Oeste)  
10  Mover(Oeste)  
11  Mover(Oeste)  
12 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	3	4	5	6	7	8	9	10
1										
0										

Tablero final

0	1	2	3	4	5	6	7	8	9	10
1										
0										

¿Te imaginás cómo hacer lo mismo pero 20, 100 o 5000 veces? Sería bastante molesto, ¿no?

Evidentemente tiene que haber una forma mejor de hacerlo, si no eso de "automatizar tareas repetitivas" sería una mentira. Y bueno, de hecho la hay: ¡vayamos al siguiente ejercicio!



La computadora repite por nosotros

Como te adelantamos en el ejercicio anterior, en Gobstones existe una forma de decir "quiero que **estos comandos se repitan esta cantidad de veces**".

Entonces, cuando es necesario repetir un comando (como `Mover` , `Poner` , `DibujarLineaNegra` , etc) un cierto número de veces, en lugar de copiar y pegar como veníamos haciendo hasta ahora, podemos utilizar la sentencia `repeat` .

Sabiendo esto, así es como quedaría `MoverOeste10` usando `repeat` :

```
procedure MoverOeste10() {
    repeat(10) {
        Mover(Oeste)
    }
}
```

Pero no tenés por qué creernos: ¡escribí este código en el editor y fijate si funciona!

```
1 procedure MoverOeste10() {
2     repeat(10) {
3         Mover(Oeste)
4     }
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3	4	5	6	7	8	9	10	
1												1
0												0

Tablero final

	0	1	2	3	4	5	6	7	8	9	10	
1												1
0												0

Ahora sí se empieza a poner interesante esto de la programación. 😊



MoverOeste5 usando repeat

Llegó tu turno de nuevo: definí un procedimiento `MoverOeste5` que se mueva 5 veces al Oeste.

Obvio, esta vez tenés que usar `repeat`.

¡Dame una pista!

```
1 procedure MoverOeste5() {  
2   repeat(5) {  
3     Mover(Oeste)  
4   }  
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial						Tablero final					
0	1	2	3	4	5	0	1	2	3	4	5
1						1					
0						0					
0	1	2	3	4	5	0	1	2	3	4	5

Como ya descubriste, el comando `repeat` consta básicamente de dos elementos:

Un **número entero** (o sea, sin decimales), que indica cuántas veces hay que repetir. Este número va entre paréntesis () luego de la palabra `repeat`.

Y un **bloque de código**, que va encerrado entre llaves ({}) y especifica qué comandos se quieren repetir. Es MUY importante que no te los olvides, porque sino la computadora no va a saber qué es lo que quisiste repetir (y fallará 😞).



No todo es repetir

Los ejemplos que hiciste en los ejercicios anteriores se solucionaban simplemente repitiendo cosas. Pero no todo es repetir, también podemos poner comandos tanto **antes** como **después** del `repeat`, al igual que veníamos haciendo hasta ahora.

Por ejemplo, este es un programa que se mueve al `Sur`, **luego** pone 4 bolitas de color `Rojo` y por último vuelve a moverse al `Norte`:

```
program {
  Mover(Sur)
  repeat(4) {
    Poner(Rojo)
  }
  Mover(Norte)
}
```

Fijate que `Mover(Sur)` lo pusimos **antes** del `repeat` y `Mover(Norte)` lo pusimos **después**. Por lo tanto cada movimiento se ejecuta solo una vez. Teniendo en cuenta esto:

Definí el procedimiento `Poner3AlNoreste()`, que ponga 3 bolitas negras en la primera celda al Noreste del cabezal.

Inicial

GBB/1.0 size 3 3 head 1 1

Final

GBB/1.0 size 3 3 cell 2 2 Negro 3 head 2 2

¡Dame una pista!

```
1 procedure Poner3AlNoreste() {
2   Mover(Norte)
3   Mover(Este)
4   repeat(3) {
5     Poner(Negro)
6   }
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

Tablero final

GBB/1.0 size 4 4 head 0 0 GBB/1.0 size 4 4 cell 1 1 Negro 3 head 1 1

¿Viste qué importante es definir bien qué comandos hay que repetir y cuáles no?

Es muy común, al principio, olvidarse de colocar las llaves o incluso pensar que no son importantes. Pero tené mucho cuidado: poner las llaves en el lugar erróneo puede cambiar por completo lo que hace tu programa. Mirá qué distinto sería el resultado si hubieras puesto el `Mover(Este)` adentro del `repeat`.

```
procedure Poner3AlNoreste() {
    Mover(Norte)

    repeat(3) {
        Mover(Este)
        Poner(Negro)
    }
}
```

GBB/1.0 size 4 4 cell 1 1 Negro 1 cell 2 1 Negro 1 cell 3 1 Negro 1 head 3 1

Esta guía fue desarrollada por Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





También vale después

Definí el procedimiento `PonerAzulLejos`, que coloque una bolita Azul 4 celdas hacia el Este:

	0	1	2	3	4	
1						1
0						0

	0	1	2	3	4	
1					1	1
0					4	0

¡Dame una pista!

```
1 procedure PonerAzulLejos() {  
2   repeat(4) {  
3     Mover(Este)  
4   }  
5   Poner(Azul)  
6 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3	4	
1						1
0						0

Tablero final

	0	1	2	3	4	
1					1	1
0					4	0

Como ya experimentaste, pueden ponerse comandos tanto antes como después del `repeat`. En definitiva... ¡es sólo un comando más!



Repetiendo varios comandos

Hasta el momento los ejemplos que vimos sólo repetían un comando, pero como mencionamos al comenzar es posible repetir cualquier **secuencia de comandos** - en definitiva lo que se repite es un **bloque** y, como ya sabíamos, en un bloque puede haber tantos comandos como se nos ocurra.

Miremos el código de `DibujarLineaNegra6` que podríamos haber hecho sin usar `repeat`, con algunos espacios en blanco para ayudarnos a reconocer la secuencia que se repite:

```
procedure DibujarLineaNegra6() {  
    Poner(Negro)  
    Mover(Este)  
  
    Poner(Negro)  
    Mover(Este)  
  
    Poner(Negro)  
    Mover(Este)  
  
    Poner(Negro)  
    Mover(Este)  
  
    Poner(Negro)  
    Mover(Este)  
  
    Poner(Negro)  
    Mover(Este)  
}
```

¿Notás qué es lo que se repite y cuántas veces? Bueno, eso es lo que tenés que poner en el `repeat`.

Definí una versión superadora de `DibujarLineaNegra6`, esta vez usando `repeat`.

¡Dame una pista!

```
1 procedure DibujarLineaNegra6() {  
2     repeat(6) {  
3         Poner(Negro)  
4         Mover(Este)  
5     }  
6 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3	4	5	6	
1								1
0	1							0

Tablero final

	0	1	2	3	4	5	6	
1								1
0	1	1	1	1	1	1	1	0

Esta guía fue desarrollada por Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/).
(<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





¿Dónde está el error?

7. ¿Dónde está

Mientras corregíamos ejercicios de un compañero tuyo, nos encontramos con una solución para `LíneaRoja4` que no está del todo bien, mirá:

Tablero inicial

0	1
4	4
3	3
2	2
1	1
0	0

Lo que hace

0	1
1	
1	
1	
1	
0	

Lo que esperábamos

0	1
1	
1	
1	
1	
0	

¿Nos ayudás a corregirla? Te dejamos el código en el editor.

¡Dame una pista!

```
1 procedure LíneaRoja4() {  
2   repeat(4) {  
3     Poner(Rojo)  
4     Mover(Norte)  
5   }  
6 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas



Diagonal con una bolita

Definí un procedimiento Diagonal4Azul que dibuje una diagonal de longitud 4 hacia el Noreste, donde cada celda tenga una bolita azul. El cabezal debe quedar donde muestra la imagen.

0	1	2	3	4	4
3			1		3
2		1			2
1			1		1
0	1				0
0	1	2	3	4	0

```
1 procedure Diagonal4Azul(){
2   repeat(4){
3     Poner(Azul)
4     Mover(Norte)
5     Mover(Este)
6   }
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	3	4	4
3					3
2					2
1					1
0					0
0	1	2	3	4	0

Tablero final

0	1	2	3	4	4
3			1		3
2		1			2
1			1		1
0					0
0	1	2	3	4	0



Diagonal "pesada"

Ahora vamos a hacer lo mismo, pero en versión "pesada".

¿Qué quiere decir esto? Que en vez de poner 1 bolita en cada celda, ahora hay que poner 21. Mirá la imagen:

0	1	2	3	4
4				4
3			21	
2		21		
1	21			
0	21			
0	1	2	3	4

Definí un procedimiento `DiagonalPesada4Azul` que resuelva el problema.

¡Dame una pista!

```
1 procedure Poner21Azules() {
2   repeat(21) {
3     Poner(Azul)
4   }
5 }
6
7 procedure DiagonalPesada4Azul() {
8   repeat(4) {
9     Poner21Azules()
10    Mover(Norte)
11    Mover(Este)
12  }
13 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Con el cabezal en el origen

Tablero inicial

0	1	2	3	4
4				
3				
2				
1				
0				

Tablero final

0	1	2	3	4
4				
3			21	
2		21		
1	21			
0	21			

✓ Con el cabezal desplazado

Tablero inicial

0	1	2	3	4	5
4					
3					
2					
1					
0					

Tablero final

0	1	2	3	4	5
4					
3			21		
2		21			
1	21				
0	21				

Muy bien. Aunque el `repeat` es poderoso y nos ayuda a escribir menos código, sigue siendo igual de importante la división en subtareas.

¡No vale olvidarse de lo aprendido hasta ahora!

Esta guía fue desarrollada por Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





El caso borde

Muchas veces cuando usamos un `repeat` nos encontramos con que el último caso es levemente distinto a los anteriores, situación que solemos llamar **caso borde**. Pero mejor, veamos un ejemplo.

El procedimiento `LineaNegra4Este` que te presentamos dibuja una línea negra hacia el Este dejando el cabezal **fuera de la línea**, una celda hacia el Este.

```
procedure LineaNegra4Este() {
    repeat(4) {
        Poner(Negro)
        Mover(Este)
    }
}
```

Si ahora queremos hacer que deje el cabezal en la última celda de la línea, tenemos dos opciones:

- **Mover el cabezal al Oeste luego de dibujar la línea.** Un truco medio feo, porque para funcionar necesita que haya al menos 5 espacios al Este de la posición inicial, cuando nuestra línea sólo ocupará 4.
- **Tratar el último caso de manera especial.** Esta opción es más interesante y más fiel a lo que queremos hacer: la última vez no queremos que el cabezal se mueva, simplemente nos basta con poner la bolita negra.

0	1	2	3	
1				1
0	1	1	1	1

Teniendo en cuenta esto último, definí una nueva versión de `LineaNegra4Este` que deje el cabezal en la última celda de la línea.

¡Dame una pista!

```
1 procedure LineaNegra4Este() {
2     repeat(3) {
3         Poner(Negro)
4         Mover(Este)
5     }
6     Poner(Negro)
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Con lugares de sobra

Tablero inicial

	0	1	2	3	4	
1						1
0	1					0
	0	1	2	3	4	

Tablero final

	0	1	2	3	4	
1						1
0		1	1	1	1	0
	0	1	2	3	4	

✓ Con el espacio justo

Tablero inicial

	0	1	2	3	
1					1
0	1				0
	0	1	2	3	

Tablero final

	0	1	2	3	
1					1
0		1	1	1	0
	0	1	2	3	

Siempre que tengas problemas como este vas a poder solucionarlos de la misma manera: **procesando el último caso por separado**.

Otra variante menos común, y tal vez más difícil de construir también, es la de procesar el **primer** caso aparte:

```
procedure LineaNegra4Este() {
    Poner(Negro)
    repeat(3) {
        Mover(Este)
        Poner(Negro)
    }
}
```

Por convención, vamos a preferir la forma que procesa distinto al último caso, aunque a menudo ambas sean equivalentes (es decir, produzcan el mismo resultado).

Esta guía fue desarrollada por Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://mumuki.org/#social>)



De lado a lado, dibujamos un cuadrado

En el ejercicio anterior definimos el procedimiento `LineaNegra4Este`. Ahora vamos a utilizarlo para dibujar un cuadrado negro igualito a este:

0	1	2	3
3	1	1	1
2	1	1	1
1	1	1	1
0	1	1	1
0	1	2	3

Definí el procedimiento `CuadradoNegro4` para dibujar un cuadrado de 4x4 con bolitas negras dejando el cabezal en el extremo superior derecho del cuadrado. No te olvides de invocar `LineaNegra4Este`.

Tené en cuenta lo que hablamos en el ejercicio anterior sobre el **caso borde**.

[¡Dame una pista!](#)

[Solución](#)

[Biblioteca](#)

```
1 procedure CuadradoNegro4() {
2     repeat(3) {
3         LineaNegra4Este()
4         Mover(Norte)
5         repeat(3) {
6             Mover(Oeste)
7         }
8     }
9     LineaNegra4Este()
10 }
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Con lugares de sobra

Tablero inicial

	0	1	2	3
4				
3				
2				
1				
0	1			

Tablero final

	0	1	2	3
4				
3	1	1	1	1
2	1	1	1	1
1	1	1	1	1
0	1	1	1	1

Con el espacio justo

Tablero inicial

	0	1	2	3	4
3					
2					
1					
0	1	1			

Tablero final

	0	1	2	3	4
3		1	1	1	1
2		1	1	1	1
1		1	1	1	1
0		1	1	1	1

Esta guía fue desarrollada por Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Pensando en subtareas

¡Queremos dibujar un nuevo cuadrado! 🖌

0	1	2	
2	1	1	1
1	1	1	1
0	1	1	1

Dividiendo cada parte del problema en procedimientos más pequeños, podemos plantear la siguiente **estrategia**: construir el cuadrado como tres líneas de tres bolitas una encima de la otra. ¿A qué nos referimos con una línea de tres bolitas? A esto:

0	1	2	
2			2
1			1
0	1	1	1

¡Arranquemos por ahí! ⚡

Definí el procedimiento `DibujarLineaNegra3` que, como su nombre lo indica, *dibuje una línea* poniendo 3 bolitas negras consecutivas hacia el Este y dejando el cabezal donde comenzó. Invocalo en un `program`.

En la Biblioteca vas a encontrar el procedimiento `VolverAtras`. ¡Eso significa que podés invocarlo sin tener que definirlo! 📁

Solución

Biblioteca

```

1 procedure DibujarLineaNegra3() {
2   repeat(2) {
3     Poner(Negro)
4     Mover(Este)
5   }
6   Poner (Negro)
7   VolverAtras()
8 }
9
10 program {
11   DibujarLineaNegra3()
12 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero de 3 por 3

Tablero inicial

	0	1	2	
2				2
1				1
0	1			0
0	1	2	2	

Tablero final

	0	1	2	
2				2
1				1
0	1	1	1	0
0	1	2	2	

✓ Tablero de 4 por 4

Tablero inicial

	0	1	2	3	
3					3
2					2
1					1
0	1				0
0	1	2	2	3	

Tablero final

	0	1	2	3	
3					3
2					2
1					1
0	1	1	1	1	0
0	1	2	2	3	

¡Perfecto! 🎉 Ya tenemos nuestra línea, ya vamos a poder hacer un cuadrado negro. ■

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 ↗ [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://github.com/mumuki/mumuki>)



Dibujando un cuadrado con subtareas

2. Dibujando un cuadrado con subtareas

¡Ya podemos dibujar nuestro cuadrado! El cual debería verse así:

0	1	2
2	1	1
1	1	1
0	1	1

El cabezal comienza en el origen, es decir, en el casillero de abajo a la izquierda:

0	1	2
2		
1		
0		

Definí el procedimiento `DibujarCuadradoNegroDeLado3` que usando `DibujarLineaNegra3` dibuje un cuadrado negro sobre el tablero. Invocalo en un `program`.

¡Dame una pista!

```
1 procedure DibujarCuadradoNegroDeLado3() {  
2   repeat(2) {  
3     DibujarLineaNegra3()  
4     Mover(Norte)  
5   }  
6   DibujarLineaNegra3()  
7 }  
8  
9 program {  
10   DibujarCuadradoNegroDeLado3()  
11 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero de 3 por 3

Tablero inicial

	0	1	2	
2				2
1				1
0	1			0

Tablero final

	0	1	2	
2	1	1	1	2
1	1	1	1	1
0	1	1	1	0

✓ Tablero de 4 por 4

Tablero inicial

	0	1	2	3	
3					3
2					2
1					1
0	1				0

Tablero final

	0	1	2	3	
3					3
2	1	1	1		2
1	1	1	1		1
0	1	1	1		0

¡Muy bien! Ya tenemos un cuadrado negro, pero también queremos cuadrados de otros colores 😊. Solucionemos esto haciendo nuevos procedimientos. 😊

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Color esperanza

Ya sabemos como dibujar una línea negra con 3 bolitas 🎯. Usemos esa lógica para dibujar una línea verde. 😊

Definí el procedimiento `DibujarLineaVerde3` e invocalo en el `program`.

[¡Dame una pista!](#)

`DibujarLineaVerde3` es igual a `DibujarLineaNegra3` solo que dibuja una línea verde. 😊

[Solución](#)

[Biblioteca](#)

```

1 procedure DibujarLineaVerde3() {
2   repeat(2) {
3     Poner(Verde)
4     Mover(Este)
5   }
6   Poner(Verde)
7   VolverAtras()
8 }
9
10 program {
11   DibujarLineaVerde3()
12 }
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero de 3 por 3

			2
			1
2	1	2	
0	1	2	
1	2		1
0			2

Tablero final

			2
			1
2	1	2	
0	1	2	
1	2		1
0			1

Tablero de 4 por 4

				3
				2
				1
3	2	1	0	
0	1	2	3	
1	2	3		1
2	3			2
0				3

Tablero final

				3
				2
				1
3	2	1	0	
0	1	2	3	
1	2	3		1
2	3			2
0				3

¡Excelente! Pero todavía nos faltan dos colores más . Ya te imaginarás lo que hay que hacer. 😊

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Los que faltan

Solo nos faltan dos colores: Rojo y Azul .

Definí los procedimientos `DibujarLineaRoja3` y `DibujarLineaAzul3`.

Solución

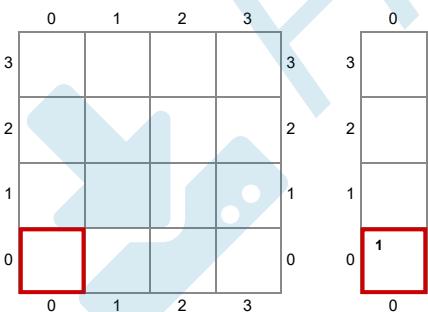
Biblioteca

```
1 procedure DibujarLineaRoja3() {
2   repeat(2) {
3     Poner(Rojo)
4     Mover(Este)
5   }
6   Poner(Rojo)
7   VolverAtras()
8 }
9
10 procedure DibujarLineaAzul3() {
11   repeat(2) {
12     Poner(Azul)
13     Mover(Este)
14   }
15   Poner(Azul)
16   VolverAtras()
17 }
```

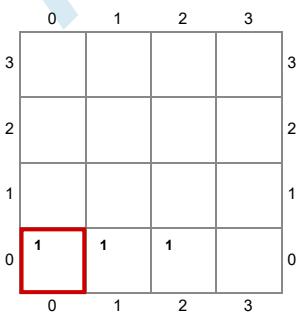
Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial



Tablero final



¡Muy bien! Peeero se volvió medio repetitiva la tarea de dibujar 4 procedimientos casi iguales ¿no? Sólo cambiaba el color, y si a esto le sumamos que además necesitamos un nuevo procedimiento por cada cuadrado, ¡la cosa se pone cada vez más aburrida! . ¿Se podrá solucionar de alguna manera?

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Procedimientos con agujeritos

5. Procedimientos con agujeritos

¡Empecemos con algo fácil! 🤖 Supongamos que tenemos un procedimiento llamado `Poner3Verdes`, que pone 3 bolitas verdes en un casillero, y lo queremos **generalizar** para que funcione con cualquier color que queramos (pero uno solo por vez). Lo que necesitamos es agregarle al procedimiento una especie de *agujero*...

```
procedure Poner3(color) {
    repeat(3) {
        Poner(color)
    }
}
```

...que luego pueda ser completado con el color que queramos:

```
program {
    Poner3(Negro)
    Poner3(Rojo)
}
```

Escribí los códigos anteriores en el editor y fijate qué pasa. ☺

```
1 procedure Poner3(color) {
2     repeat(3) {
3         Poner(color)
4     }
5 }
6 program {
7     Poner3(Negro)
8     Poner3(Rojo)
9 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	
2			2
1			1
0			0

Tablero final

	0	1	
2			2
1		3	1
0			0

¿Viste qué interesante lo que hicimos? ☺

Con un **mismo procedimiento** pudimos hacer 2 cosas casi iguales; por un lado pusimos 3 bolitas negras y por el otro 3 bolitas rojas. La diferencia estuvo en **cómo usamos Poner3**: la primera vez completamos el *agujero* con `Negro` y la segunda con `Rojo`.

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Llenando los espacios vacíos

Entendamos qué acabamos de hacer. 😊

Lo primero que hicimos fue definir un procedimiento, pero con una pequeña diferencia: toma un *parámetro*, llamado `color`.

```
procedure Poner3(color) {  
    Poner(color)  
    Poner(color)  
    Poner(color)  
}
```

¿Y qué es un *parámetro*? Son esos nombres que van entre paréntesis para ser reemplazados por valores concretos cuando invocamos al procedimiento. Por ejemplo, si lo invocamos así..

```
program {  
    Poner3(Negro)  
}
```

...lo que se ejecuta es:

```
Poner(Negro)  
Poner(Negro)  
Poner(Negro)
```

Y si lo invocamos así...

```
program {  
    Poner3(Rojo)  
}
```

lo que se ejecuta es:

```
Poner(Rojo)  
Poner(Rojo)  
Poner(Rojo)
```

Fijate como cada vez que aparece `color` se reemplaza por el valor que le *pasamos* a `Poner`. Veamos si se entiende:

Creá un programa que ponga tres bolitas verdes. No te olvides de invocar el procedimiento `Poner3`.

Solución

Biblioteca

```
1 program {  
2     Poner3(Verde)  
3 }
```

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1
2	
1	
0	

Tablero final

0	1
2	
1	3
0	

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/). (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





DibujarLinea3

7. DibujarLinea3

¡Ahora te toca a vos!

Ya hicimos cuatro procedimientos para dibujar líneas de cada color, pero si usamos parámetros podría ser sólo uno. 😊

Definí el procedimiento `DibujarLinea3` que reciba un color y dibuje una línea de ese color. Despreocúpate por los programas para invocarlo con cada uno de los colores, van por nuestra parte. 😊

¡Dame una pista!

Solución

Biblioteca

```
1 procedure DibujarLinea3(color) {
2     repeat(2) {
3         Poner(color)
4         Mover(Este)
5     }
6     Poner(color)
7     VolverAtras()
8 }
9
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial			
0	1	2	3
3			.
2			
1			
0	1		

Tablero final			
0	1	2	3
3			
2			
1			
0	1	1	1

Tablero inicial

	0	1	2	3
3				
2				
1				
0	1			

Tablero final

	0	1	2	3
3				
2				
1				
0	1	1	1	1



Tablero inicial

	0	1	2	3
3				
2				
1				
0	1			

Tablero final

	0	1	2	3
3				
2				
1				
0	1	1	1	1



Tablero inicial

	0	1	2	3
3				
2				
1				
0	1			

Tablero final

	0	1	2	3
3				
2				
1				
0	1	1	1	1

En vez de escribir `color` podríamos haber escrito `c`, `col` o incluso `pepe`, porque a la computadora no le importa el nombre que le demos a las cosas... pero a nosotros sí (y mucho 😊), así que por eso elegimos **usar nombres que expresen bien lo que queremos decir**.

Fijate también que elegimos un nuevo nombre para nuestro nuevo procedimiento porque ahora sirve para cualquier color. Esto tampoco es estrictamente necesario para la computadora, pero es super importante para los humanos que van a leer y escribir el código. Imaginate que si le pusieramos `DibujarLineaNegra3` (o `DibujarLineaAzul13`) al usarlo para dibujar una línea roja quedaría `DibujarLineaNegra3(Rojo)`. Si bien va a funcionar, no se entendería bien qué es lo que hace: ¿una línea negra? ¿una línea roja? ¿una línea negra y roja?. 😊

¡Es importantísimo poner buenos nombres para programar bien! 😊

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloï, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://github.com/mumuki/mumuki>)



DibujarCuadradoDeLado3

8. DibujarC

¡Hora del último pasito! 🎉 Ya definimos un procedimiento para poder dibujar cuadrados negros (`DibujarCuadradoNegroDeLado3`), pero todo este asunto de los parámetros surgió cuando quisimos hacer cuadrados de distintos colores. 🤔

Invocando `DibujarLinea3`, definí el procedimiento `DibujarCuadradoDeLado3` que recibe un `color` y dibuja un cuadrado de 3x3 de ese color.

¡Dame una pista!

Solución

Biblioteca

```
1 procedure DibujarCuadradoDeLado3(color) {
2   repeat(2) {
3     DibujarLinea3(color)
4     Mover(Norte)
5   }
6   DibujarLinea3(color)
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial				
3	0	1	2	3
2	0	1	2	3
1	0	1	2	3
0	0	1	2	3

Tablero final				
3	0	1	2	3
2	1	1	1	1
1	1	1	1	1
0	1	1	1	1

Tablero inicial

	0	1	2	3
3				
2				
1				
0	1			

Tablero final

	0	1	2	3
3				
2	1	1	1	
1	1	1	1	
0	1	1	1	



Tablero inicial

	0	1	2	3
3				
2				
1				
0	1			

Tablero final

	0	1	2	3
3				
2	1	1	1	
1	1	1	1	
0	1	1	1	



Tablero inicial

	0	1	2	3
3				
2				
1				
0	1			

Tablero final

	0	1	2	3
3				
2	1	1	1	1
1	1	1	1	1
0	1	1	1	1

Genial, ¡logramos crear un procedimiento que nos sirve para cualquier color! 🎉

Un procedimiento puede no tener parámetros (como pasa en `VolverAtras` o `DibujarLineaNegra3`) o un parámetro (como `Mover`, `Poner` o `DibujarLinea3`), pero ¿puede tener más de uno? 💬

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/). (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





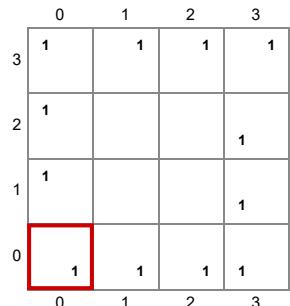
Pasando varios parámetros

¿Y si queremos que `DibujarLinea3` sirva también para dibujar líneas en cualquier dirección? Sin dudas tenemos que decirle al procedimiento, además del `color`, en qué `direccion` debe dibujar la línea; y para eso vamos a necesitar un nuevo **parámetro**. Por suerte, ¡los procedimientos también pueden tener más de un **parámetro**! 🤗

¿Y cómo se hace esto? Muy fácil, al igual que como hacemos al escribir, vamos a **separar cada parámetro usando comas** de esta manera:

```
procedure DibujarLinea3(color, direccion) {
    Poner(color)
    Mover(direccion)
    Poner(color)
    Mover(direccion)
    Poner(color)
}
```

Creá un `program` que invoque la nueva versión de `DibujarLinea3` (no tenés que definirla, sólo invocarla) y dibuje un cuadrado multicolor como este:



No te preocupes por la posición final del cabezal.

[¡Dame una pista!](#)

[Solución](#)

[Biblioteca](#)

```
1 program {
2     DibujarLinea3(Verde, Este)
3     Mover(Este)
4     DibujarLinea3(Rojo, Norte)
5     Mover(Norte)
6     DibujarLinea3(Negro, Oeste)
7     Mover(Oeste)
8     DibujarLinea3(Azul, Sur)
9     Mover(Sur)
10 }
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial				Tablero final			
0	1	2	3	0	1	2	3
3				3	1	1	1
2				2	1		1
1				1	1		1
0	1			0	1	1	1

Como habrás notado, al usar los procedimientos debemos darle un valor a cada uno de sus parámetros **respetando el orden** en que fueron definidos.

A estos valores concretos que usamos cuando invocamos a un procedimiento los llamamos *argumentos*. ☺

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloí, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





La ley, el orden y el BOOM

Recién te decíamos que el orden en que pasabamos los **argumentos** era importante pero nunca te dijimos por qué 😊. ¡Vamos a verlo!

Creá un programa cualquiera que invoque `DibujarLinea3`, pero esta vez intentá invocarlo con los argumentos invertidos. [-] [-]

¡Dame una pista!

Solución

Biblioteca

```
1 program {
2   DibujarLinea3(Norte,Azul)
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡BOOM!

Tablero inicial

	0	1	2
0			
1			
2			

Tablero final



BOOM

[5:3]: El parámetro de "Poner" debería ser un color pero es una dirección.

Ok, hizo BOOM 💣, pero ¿qué quiere decir eso de El parámetro de Poner debería ser un color? 🤔

Al pasar los argumentos al revés, donde se esperaba un color llegó una dirección; entonces cuando intentamos Poner una dirección provocamos la autodestrucción del cabezal 🎨. `Poner(Norte)` o `Poner(Este)` no se pueden ejecutar porque no tienen ningún sentido. 😅



Un argumento para dos parámetros

Ya vimos que pasa cuando pasamos los argumentos desordenados pero vamos a hacer un experimento más . ¿Qué crees que va a pasar si a un procedimiento le pasamos menos argumentos de los que necesita?

Creá un programa que invoque a `DibujarLinea3` pero pasándole sólo un argumento.

Solución

Biblioteca

```
1 program {
2   DibujarLinea3(Negro)
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

¡BOOM!

Tablero inicial		
0	1	2
2		
1		
0	■	

Tablero final



BOOM

[2:3]: El procedimiento "DibujarLinea3" espera recibir 2 parámetros pero se lo invoca con un argumento.

¡BOOM de nuevo ! Como te imaginarás, si le pasamos más argumentos de los que espera pasará lo mismo.

Entonces es importante recordar que al invocar un procedimiento **no debemos**:

- pasarle menos o más argumentos de los que necesita;
- pasarle los argumentos en un orden diferente al que espera.



La tercera es la vencida

Para terminar esta lección vamos a definir un procedimiento llamado `Triada` ¡que recibe tres parámetros! 😊

`Triada` recibe tres colores por parámetro y pone tres bolitas, una al lado de la otra hacia el Este, en el mismo orden en que se reciben. El cabezal empieza en el origen y debe terminar sobre la última bolita de la tríada.

Por ejemplo: `Triada(Rojo, Azul, Verde)` nos da como tablero resultante:

0	1	2
0	1	
1		
2		1

mientras que `Triada(Azul, Verde, Rojo)`:

0	1	2
0	1	
1		
2	1	1

Definí el procedimiento `Triada`.

¡Dame una pista!

```

1 procedure Triada(color1,color2,color3) {
2   Poner(color1)
3   Mover(Este)
4   Poner(color2)
5   Mover(Este)
6   Poner(color3)
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial

0	1	2
0		
1		
2		

Tablero final

0	1	2
0	1	
1		
2		1

Tablero inicial

0	1	2
0		
1		
2		

Tablero final

0	1	2
0	1	
1		
2	1	1

Esta guía fue desarrollada por Gustavo Trucco, Federico Aloi, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Entrando en calor... ¡Volviendo!

En la guía anterior, vimos que se podía usar `repeat` para hacer algo muchas veces. Hicimos un ejemplo que se llamaba `Diagonal4Azul`, que era más o menos así:

```
procedure Diagonal4Azul(){
    repeat(4){
        Poner(Azul)
        Mover(Este)
        Mover(Norte)
    }
}
```

¿Te animás a definir el procedimiento `Diagonal4AzulVolviendo`? Este procedimiento debería *hacer lo mismo* que `Diagonal4Azul`, pero tiene que dejar el cabezal **en la posición inicial**. Recordá que podés invocar todo lo que está en la Biblioteca sin necesidad de volver a definirlo. ☺

0	1	2	3	4
			1	
		1		
	1			
1				

Solución

Biblioteca

```
1 procedure Diagonal4AzulVolviendo() {
2     Diagonal4Azul()
3     repeat(4) {
4         Mover(Oeste)
5         Mover(Sur)
6     }
7 }
8 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3	4
4					
3					
2					
1					
0	1				

Tablero final

	0	1	2	3	4
4					
3				1	
2			1		
1		1			
0	1				

¡Bien!

Tenés que acostumbrarte a pensar... ¿No podría usar algún procedimiento que ya definí antes?

Una gran ventaja de los procedimientos es que, una vez que están escritos, podés guardártelos para volver a usarlos. Cuando ejercitás en Mumuki, nosotros ya los guardamos por vos y te dejamos la **Biblioteca** lista para usar, pero en la vida real ese trabajo vas a tener que hacerlo vos. ↗️💻

Esta guía fue desarrollada por Alfredo Sanzo, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 ↗️ [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://github.com/mumuki>)



Una diagonal más ancha

Sigamos probando tus habilidades para reutilizar...

Ahora, tenés que hacer este dibujo:

0	1	2	3	4	6
5			1		5
4		1	1		4
3	1	1	1		3
2	1	1	1		2
1	1				1
0	1				0
0	1	2	3	4	

El procedimiento debe llamarse `BandaDiagonal4` . ¡Ojo! prestá atención a la **posición final** del cabezal.

¡Dame una pista!

[Solución](#)

[Biblioteca](#)

```
1 procedure BandaDiagonal4() {
2     repeat(3) {
3         Diagonal4AzulVolviendo()
4         Mover(Norte)
5     }
6     repeat(3) {
7         Mover(Sur)
8     }
9 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

	0	1	2	3	4
6					
5					
4					
3					
2					
1					
0	1				

Tablero final

	0	1	2	3	4
6					
5				1	
4			1	1	
3		1	1	1	
2	1	1	1		
1	1	1			
0	1				

Esta guía fue desarrollada por Alfredo Sanzo, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Pongamos... ¡Todo lo que queramos!

Ahora que tenemos una idea de *reutilización*, y practicamos *repetición*, vamos a definir un procedimiento que nos va a servir de acá en adelante.

Necesitamos un procedimiento que nos ayude a poner muchas bolitas. Sí, podríamos simplemente usar un `repeat` para lograrlo, pero como es una tarea re-común que vamos a hacer un montón de veces, vamos a preferir definir un `procedure` llamado `PonerN`. Nuestro procedimiento debe poner la cantidad de bolitas indicada de un color dado.

Por ejemplo, `PonerN(3, Azul)` haría esto:

	0	1
1		
0	3	
0	1	0

Definí el procedimiento `PonerN(cantidad, color)`.

```
1 procedure PonerN(cantidad, color) {
2     repeat(cantidad) {
3         Poner(color)
4     }
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial

	0	1
1		
0		
0		1

Tablero final

	0	1
1		
0		
0	2	1

Tablero inicial

	0	1
1		
0		
0		1

Tablero final

	0	1
1		
0		
0	4	1

Aunque quizás no veas todavía la utilidad de este `procedure` que creamos, te contamos dos aspectos que es importante tener en cuenta al programar:

- **reutilización de código:** como poner muchas bolitas es una tarea común, está bueno tener un procedimiento que lo resuelva: lo escribimos una vez y lo usamos para siempre;
 - **declaratividad:** cuando tengamos que resolver un problema más complejo, tener este procedimiento nos va a ayudar a pensar a más alto nivel, ya que no vamos a tener que preocuparnos por **cómo** poner muchas bolitas sino en **qué** queremos construir con ellas.

Esta guía fue desarrollada por Alfredo Sanzo, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Día de la Memoria

4. Día de la Memoria

Muchas veces vamos a usar el tablero de Gobstones como memoria, o sea, para recordar algo importante que vamos a necesitar más adelante.

¿Qué podríamos representar con bolitas? Por ejemplo una fecha. Una fecha que debemos recordar es el *24 de Marzo de 1976*, hoy constituido Día de la Memoria por la Verdad y la Justicia (https://es.wikipedia.org/wiki/D%C3%ADa_Nacional_de_la_Memoria_por_la_Verdad_y_la_Justicia) en Argentina.

El objetivo, entonces, es definir un procedimiento `DiaDeLaMemoria()`:

- En la celda actual, poné 24 bolitas Azules, que **representan** el día.
- En la celda inmediatamente al Este, poné 3 bolitas Verdes, que **representan** el mes.
- En la celda a continuación, poné 1976 bolitas Negras, **representando** el año.

0	1	2	*	0
24				0
0	1	2		0

¡Dame una pista!

Solución

Biblioteca

```

1 procedure Fecha(dia, mes, anio) {
2   PonerN(dia,Azul)
3   Mover(Este)
4   PonerN(mes,Verde)
5   Mover(Este)
6   PonerN(anio,Negro)
7 }
8 procedure DiaDeLaMemoria() {
9   Fecha(24, 3, 1976)
10 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	0
0	1	2	0

Tablero final

0	1	2	*	0
24				0
0	1	2		0

¿Sabías que `Azul` es una expresión literal? ¡También `1976`! También son expresiones literales: `Verde`, `Negro`, `3` y `24`.

Cuando **usamos** un procedimiento que tiene parámetros como `PonerN`, `PonerN(56, Rojo)` tenemos que enviarle valores como argumento. ¡Y las expresiones sirven para eso!



Escribir cualquier fecha

Ahora que ya escribimos una fecha particular, hagamos un procedimiento que sirva para escribir cualquier fecha.

Definí el procedimiento `Fecha(dia, mes, anio)`, que recibe los **tres valores** correspondientes, y escribe la fecha que representan, de esta manera:

- En la celda actual, tantas bolitas azules para **representar** el día.
- En la celda inmediatamente al Este, tantas bolitas Verdes para **representar** el mes.
- En la celda a continuación, tantas bolitas Negras para **representar** el año.

Por ejemplo, `Fecha(12, 8, 1990)` produciría algo así:

0	1	2	*	0
12				0
0	1	2		

¡Dame una pista!

Solución

Biblioteca

```

1 procedure Fecha(dia, mes, anio) {
2   PonerN(dia,Azul)
3   Mover(Este)
4   PonerN(mes,Verde)
5   Mover(Este)
6   PonerN(anio,Negro)
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial

0	1	2	0
0	1	2	

Tablero final

0	1	2	0
24	3	*	0

Tablero inicial

0	1	2	0
0	1	2	

Tablero final

0	1	2	0
4	6	*	0

Dos cuestiones teóricas para pensar de este ejercicio:

1. Puede parecerse que estás **repetiendo código** con el ejercicio anterior. ¡Es cierto! Para arreglarlo, deberías volver al ejercicio anterior y allí usar el procedimiento `Fecha`. 😊
 2. Acá vemos que hay otro tipo de **expresiones**: ¡Los parámetros! Al **usar** un procedimiento, puedo enviarle tanto otros parámetros como literales: `PonerN(dia, Azul)`. Recordá en este caso que los nombres de los parámetros sólo nos sirven a los humanos, para la máquina sólo importa el orden.

Esta guía fue desarrollada por Alfredo Sanzo, Mumuki Project bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Movamos... ¡Todo lo que queramos!

Definí un procedimiento `MoverN(cantidad, dirección)` que haga que el cabezal se desplace la cantidad especificada de veces en la dirección indicada.

Por ejemplo, `MoverN(3, Oeste)` provocaría:

Inicial			
0	1	2	3
2			
1			
0			
0	1	2	3

Final			
0	1	2	3
2			
1			
0			
0	1	2	3

¡Dame una pista!

```
1 procedure MoverN(cantidad, dirección) {  
2   repeat(cantidad){  
3     Mover(dirección)  
4   }  
5 }  
6 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial			
0	1	2	0
0	1	2	0
0	1	2	0

Tablero final			
0	1	2	0
0	1	2	0
0	1	2	0

Tablero inicial			
0	1	2	0
1	2	0	0
0	1	2	0

Tablero final			
0	1	2	0
1	2	0	0
0	1	2	0

¡Perfecto!

Recordamos entonces que entre los paréntesis del `repeat` no sólo pueden ir números (como `7` o `42`), sino también otros tipos de expresiones que denoten valores numéricos (como `cantidad` en este caso).

Esta guía fue desarrollada por Alfredo Sanzo, Mumuki Project bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Los números del reloj

¡Ya sabés Kung Fu!

Ahora, tenés que mostrarnos que podés *dibujar un reloj*. Lo que haremos por ahora es solamente poner los números que aparecen en un típico reloj de agujas:

- El 12 arriba,
- El 3 a la derecha,
- El 9 a la izquierda, y
- el 6 abajo.

Definí un procedimiento `DibujarReloj(radio)` que ponga los números del reloj como se indica arriba: **alrededor del casillero actual**. El tamaño del reloj se indica con el `radio` que recibís como parámetro: mientras más grande es el radio, más alejados están los números del centro.

Dado el siguiente program:

```
program {
    DibujarReloj(2)
}
```

El reloj resultante es así:

0	1	2	3	4
4		12		
3				
2	9			3
1				
0		6		
0	1	2	3	4

¡Dame una pista!

Solución

Biblioteca

```
1 procedure DibujarReloj(radio) {
2     MoverN(radio,Este)
3     PonerN(3, Rojo)
4     repeat(2){
5         MoverN(radio,Oeste)
6     }
7     PonerN(9, Rojo)
8     MoverN(radio,Este)
9     MoverN(radio,Sur)
10    PonerN(6, Rojo)
11    repeat(2){
12        MoverN(radio,Norte)
13    }
14    PonerN(12, Rojo)
15    MoverN(radio,Sur)
16 }
```

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:



Tablero inicial

0	1	2	3	4
4				
3				
2				
1				
0				
0	1	2	3	4

Tablero final

0	1	2	3	4
4				
3				
2				
1				
0				
0	1	2	3	4



Tablero inicial

Tablero final

0	1	2
2		
1		
0		
0	1	2

0	1	2
2		
1		
0		
0	1	2

¿Te fijaste? Estamos usando bolitas para representar la hora de un reloj. Al programar, usamos las *abstracciones* que tenemos para modelar cosas del mundo real.

Y como siempre, es **muy importante** dividir el problema en subtareas. Y si puedo no repetir código, ¡Aún mejor!

Antes de pasar al siguiente ejercicio pregunta: ¿repetí código?

Esta guía fue desarrollada por Alfredo Sanzo, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://github.com/mumuki>)



Una línea heavy

El procedimiento `LíneaEstePesada(peso, color, longitud)` debe dibujar hacia el Este una línea del color dado, poniendo en cada celda tantas bolitas como indique el peso. La linea debe ser tan larga como la longitud.

A modo de ejemplo, `LíneaEstePesada(3, Verde, 5)` debería dibujar una línea verde, ocupando cinco celdas hacia el Este y poniendo tres bolitas en cada una de ellas:

	0	1	2	3	4	
1						1
0	3	3	3	3	3	0
	0	1	2	3	4	

Definí el procedimiento `LíneaEstePesada(peso, color, longitud)`. Tené en cuenta que el cabezal **debe regresar a la posición inicial**. Para eso vas a tener que invocar `MoverN`.

Solución

Biblioteca

```

1 procedure LíneaEstePesada(peso, color, longitud) {
2   repeat(longitud) {
3     PonerN(peso, color)
4     Mover(Este)
5   }
6   MoverN(longitud, Oeste)
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial

	0	1	2	3	4	
0						0
	0	1	2	3	4	

Tablero final

	0	1	2	3	4	
0	24	24	24	24	24	0
	0	1	2	3	4	

Tablero inicial

	0	1	2	3	4	5	6	7	
0									0
	0	1	2	3	4	5	6	7	

Tablero final

	0	1	2	3	4	5	6	7	
0	6	6	6	6	6	6	6	6	0
	0	1	2	3	4	5	6	7	

¡Son muy útiles!

Esta guía fue desarrollada por Alfredo Sanzo, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Guarda con la guarda

Bueno, estamos en tiempo para algún ejercicio integrador...

Definí un procedimiento `GuardaDe5()`, que haga una "guarda" de 5 azulejos (como las que decoran las paredes). Cada azulejo está conformado por 1 bolita verde, 5 negras y 9 rojas.

0	1	2	3	4	5	
1						1
0	5	5	5	5	5	0
9	1	9	1	9	1	5
0	1	2	3	4	5	

¡Dame una pista!

Solución

Biblioteca

```

1 procedure GuardaDe5() {
2   repeat(4){
3     PonerN(1,Verde)
4     PonerN(5,Negro)
5     PonerN(9,Rojo)
6     Mover(Este)
7   }
8   PonerN(1,Verde)
9   PonerN(5,Negro)
10  PonerN(9,Rojo)
11 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	3	4	
0					0
0	1	2	3	4	
0	1	2	3	4	

Tablero final

0	1	2	3	4	
0	5	5	5	5	5
0	9	1	9	1	9
0	1	2	3	4	0

¡Bien! Recordaste cómo considerar el caso borde.

Además, en este ejercicio hay que dividir en subtareas para **evitar la repetición de código**. Esto es muy importante a la hora de programar.

¡Asegurate que tu solución no repita código!



Una guarda en L

Definí un procedimiento `GuardaEnL()` que haga una guarda en L como muestra la figura, pero **dejando el cabezal en la posición inicial**.

0	1	2	
2	5 9 1		2
1	5 9 1		1
0	5 9 1	5 9 1	5 9 1
0	1	2	0

La ventaja ahora, es que te regalamos el procedimiento `PonerAzulejo`. ¡Pero ojo que necesitás dividir en más subtareas!

¡Dame una pista!

Solución

Biblioteca

```

1 procedure Poner2Azulejos(direccion) {
2   repeat(2) {
3     Mover(direccion)
4     PonerAzulejo()
5   }
6 }
7
8 procedure GuardaEnL() {
9   PonerAzulejo()
10  Poner2Azulejos(Este)
11  MoverN(2,Oeste)
12  Poner2Azulejos(Norte)
13  MoverN(2,Sur)
14 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1	2	
2			2
1			1
0			0
0	1	2	0

Tablero final

0	1	2	
2	5 9 1		2
1	5 9 1		1
0	5 9 1	5 9 1	5 9 1
0	1	2	0

Esta guía fue desarrollada por Alfredo Sanzo, Mumuki Project bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Muchas formas de decir lo mismo

Cuando nos comunicamos con alguien más, usamos palabras o frases para describir una idea. Por ejemplo, todas las siguientes **expresiones** hablan de lo mismo, aunque lo hacen de distintas formas:

- el número 5; **[5]**
- la cantidad de dedos de una mano; **⌚**
- la suma entre 3 y 2; **[3] + [2]**
- el número de continentes que existen en el planeta, [según la ONU](https://es.wikipedia.org/wiki/Continente#Modelos_continuales) (https://es.wikipedia.org/wiki/Continente#Modelos_continuales). **⌚**

Todas las frases anteriores hablan del **valor** cinco, aunque no lo digan de forma explícita.

Con esta idea e invocando `PonerN`, creá un programa que ponga cinco bolitas negras, PERO sin escribir el número 5.

[¡Dame una pista!](#)

[Solución](#)

[Biblioteca](#)

```
1 program {
2   PonerN(4+1,Negro)
3 }
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial

0	1
1	
0	

Tablero final

0	1
1	
0	5

Algunas variantes válidas:

```
program {
  PonerN(4 + 1, Negro)
}
```

```
program {
    PonerN(12 - 7, Negro)
}
```

Y así se nos pueden ocurrir infinitas formas de "decir 5" y sólo una de ellas lo hace de manera **literal** (o sea, escribiendo 5).

Esta guía fue desarrollada por Federico Aloí, Alfredo Sanzo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





La suma de las partes

Juguemos un poco más con esto de hacer cuentas.

Definí un procedimiento `PonerSuma(x, y)` que reciba dos parámetros y ponga la cantidad de bolitas rojas que surge de sumar x e y .

Ejemplo: `PonerSuma(4, 2)` debería poner 6 bolitas rojas en la celda actual (porque 6 es el resultado de sumar 4 y 2).

0	1
1	
6	
0	1

Solución

Biblioteca

```
1 procedure PonerSuma(x, y) {
2   PonerN(x+y,Rojo)
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial

0	1
1	
6	
0	1

Tablero final

0	1
1	
5	
0	1

Tablero inicial

0	1
1	
11	
0	1

Tablero final

0	1
1	
11	
0	1

Como ya descubriste, podemos escribir **expresiones aritméticas** (o sea, cuentas matemáticas) en los mismos lugares donde hasta ahora veníamos escribiendo números.

Esta guía fue desarrollada por Federico Aloï, Alfredo Sanzo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





¿Qué se hace antes?

De un conocido diario (no podemos revelar su nombre por temas de confidencialidad) nos pidieron definir un procedimiento para contar, aproximadamente, cuánta gente asistió a una determinada manifestación.

Contamos con la información de cuántos micros, autos y bicicletas asistieron y desde allí podemos hacer un cálculo siguiendo estas reglas:

- en cada **micro** viajan **40 personas**;
- en cada **auto** viajan **4 personas**;
- en cada **bicicleta** viaja **1 persona**.

Definí el procedimiento `ContarGente(micros, autos, bicicletas)` que a partir de la cantidad de micros, autos y bicicletas que recibe como parámetro, haga las cuentas necesarias y refleje el resultado con bolitas de color verde.

Te dejamos un par de ejemplos que te pueden ayudar:

- `ContarGente(1, 1, 1)` generaría este tablero:

0	1	1
1		
0	45	0

- `ContarGente(1, 2, 3)` generaría este tablero:

0	1	1
1		
0	51	0

¡Dame una pista!

Solución

Biblioteca

```
1 procedure ContarGente(micros, autos, bicicletas) {  
2   PonerN(40*micros, Verde)  
3   PonerN(4*autos, Verde)  
4   PonerN(1*bicicletas, Verde)  
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:



Tablero inicial

	0	1
1		
0	*	

Tablero final

	0	1
1		
0	*	



Tablero inicial

	0	1
1		
0	*	

Tablero final

	0	1
1		
0	*	

En Gobstones, como en la matemática, existe la idea de **precedencia de operadores**. En criollo, esto quiere decir que hay ciertas operaciones que se hacen antes que otras, sin la necesidad de usar paréntesis para ello. En particular, el orden es: primero las multiplicaciones y divisiones, luego las sumas y las restas (de nuevo, como en matemática).

Por lo tanto, la expresión $(10 * 4) + (8 * 7)$ es equivalente a $10 * 4 + 8 * 7$.

Esta guía fue desarrollada por Federico Aloí, Alfredo Sanzo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





La carrera del salmón

Bueno, basta de números (por un ratito). Ahora vamos a aprender a hacer "cuentas" con las direcciones.

Para hacer esto, simularemos el movimiento de un salmón: en contra de la corriente. Nuestro objetivo será definir un procedimiento `MoverComoSalmon(direccion)` que reciba una dirección y se mueva exactamente una vez en la dirección **opuesta**. Veamos en una tabla cómo debería comportarse este procedimiento:

- `MoverComoSalmon(Norte)` se mueve hacia el **Sur**.
- `MoverComoSalmon(Este)` se mueve hacia el **Oeste**.
- `MoverComoSalmon(Sur)` se mueve hacia el **Norte**.
- `MoverComoSalmon(Oeste)` se mueve hacia el **Este**.

Como la dirección va a ser un parámetro de nuestro procedimiento, necesitamos una forma de decir "*la dirección opuesta a X*" para poder luego usar esto como argumento de `Mover`. Gobstones nos provee un mecanismo para hacer esto, la primitiva `opuesto(dir)`. En criollo: **opuesto** (¡sí, en minúsculas!) nos dice la dirección contraria a la `dir` que nosotros le pasemos.

Sabiendo esto, podríamos definir fácilmente el procedimiento que queríamos:

```
procedure MoverComoSalmon(direccion) {
    Mover(opuesto(direccion))
}
```

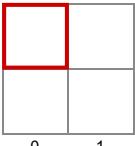
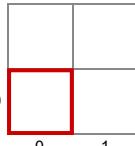
Escribí la solución en el editor y dale Enviar. Vas a ver cómo se mueve el cabezal...

```
1 procedure MoverComoSalmon(direccion) {
2     Mover(opuesto(direccion))
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial	Tablero final
	

Tablero inicial

	0	1
1		
0	0	1

Tablero final

	0	1
1		
0	1	0

Las expresiones no sólo pueden **denotar** números, también nos sirven para realizar transformaciones sobre **direcciones**. Más sobre esto en los próximos ejercicios. ☺

Esta guía fue desarrollada por Federico Aloí, Alfredo Sanzo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Dos pasos adelante, un paso atrás

5. Dos pasos adelante, un paso atrás

Tenemos un amigo llamado Carlos, que es bastante desconfiado. En su vida, eso se manifiesta en muchos aspectos, pero el más notorio es su forma de caminar: sólo camina hacia el Este y siempre que da dos pasos hacia adelante automáticamente da un paso hacia atrás.

Por ejemplo, si le pidieramos que diera 2 pasos, terminaría dando 1; si le pidieramos 4, daría 2; y así sucesivamente. En definitiva, lo que termina pasando es que nuestro amigo da la **mitad** de los pasos que le pedimos.

Importante: en Gobstones usamos el operador `div` para dividir; por ejemplo "4 dividido 2" se escribe `4 div 2`.

Definí el procedimiento `CaminarDesconfiado(pasos)` que simule el caminar de Carlos: debe recibir la cantidad de pasos que debería dar y dar la mitad. Siempre se mueve al Este .

¡Dame una pista!

Solución

Biblioteca

```
1 procedure CaminarDesconfiado(pasos) {
2   MoverN(pasos div 2, Este)
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial

0	1	2	3	4	5	0
0						0
0	1	2	3	4	5	0

Tablero final

0	1	2	3	4	5	0
0					0	
0	1	2	3	4	5	0

Tablero inicial

0	1	2	3	4	5	0
0					0	
0	1	2	3	4	5	0

Tablero final

0	1	2	3	4	5	0
0					0	
0	1	2	3	4	5	0

Sobre el ejemplo de 4 pasos, no hay dudas: Carlos dio 2 pasos. Ahora, cuando le pedimos que diera 7, ¿por qué dio 3?

En Gobstones, la **división es entera**: se ignoran los decimales. De esta forma, `7 div 2` termina dando 3 en vez de 3.5 .



Poner al lado

Para ver si entendiste lo anterior, te toca ahora resolver por tu cuenta.

Queremos definir un procedimiento que nos sirva para poner una bolita **al lado** de donde se encuentre el cabezal, dejándolo en la posición original. Por ejemplo, al invocar `PonerAl(Norte, Verde)` debería poner una bolita verde una posición hacia el Norte, **sin mover** el cabezal (bueno, ya sabemos que en realidad sí se mueve, pero el punto es que en el **resultado final** esto no se tiene que ver).



Definí el procedimiento `PonerAl(direccion, color)`.

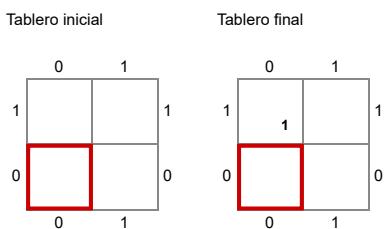
¡Dame una pista!

```
1 procedure PonerAl(direccion, color) {
2   Mover(direccion)
3   Poner(color)
4   Mover(opuesto(direccion))
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:



Tablero inicial	Tablero final												
<table border="1"> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td></td></tr> <tr> <td>0</td><td>1</td></tr> </table>	0	1	1		0	1	<table border="1"> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td></td></tr> <tr> <td>0</td><td>1</td></tr> </table>	0	1	1		0	1
0	1												
1													
0	1												
0	1												
1													
0	1												

Tanto `opuesto(dir)` como otras herramientas que veremos más adelante forman parte de lo que conocemos como **funciones** y, como las expresiones aritméticas que ya vimos, se pueden usar en cualquier lugar donde hasta ahora poníamos valores o argumentos.

O sea, donde hasta ahora podrías usar `dir` ahora también podrías poner `opuesto(dir)`, ya que ambas expresiones denotan direcciones. Obviamente te queda a vos decidir en cada caso si tiene sentido usar `opuesto` o no.

Esta guía fue desarrollada por Federico Aloí, Alfredo Sanzo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://github.com/filadd>)



La línea que vuelve

Ahora que sabés usar la función `opuesto`, podemos finalmente resolver el problema de definir un procedimiento que dibuje una línea en cualquier dirección y deje el cabezal en la posición inicial.

La versión que sabíamos hacer hasta ahora era esta:

```
procedure Linea(direccion, color, longitud) {
    repeat(longitud) {
        Poner(color)
        Mover(direccion)
    }
}
```

Valiéndote de tus nuevos conocimientos sobre expresiones, modifícá el procedimiento `Línea` para que el cabezal quede en el lugar donde empezó.

¡Dame una pista!

Solución

Biblioteca

```
1 procedure Linea(direccion, color, longitud) {
2     repeat(longitud) {
3         Poner(color)
4         Mover(direccion)
5     }
6     MoverN(longitud, opuesto(direccion))
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial			
0	1	2	3
3			
2			
1			
0	1		
0	0	1	
1	0	0	1
2	1	0	0
3	0	1	0

Tablero final			
0	1	2	3
3			
2			
1			
0	1		
0	0	1	
1	0	0	1
2	1	0	0
3	0	1	0

Tablero inicial				Tablero final			
	0	1	2	3		0	1
1					1		
0					0		
	0	1	2	3		0	1

Esta guía fue desarrollada por Federico Aloí, Alfredo Sanzo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

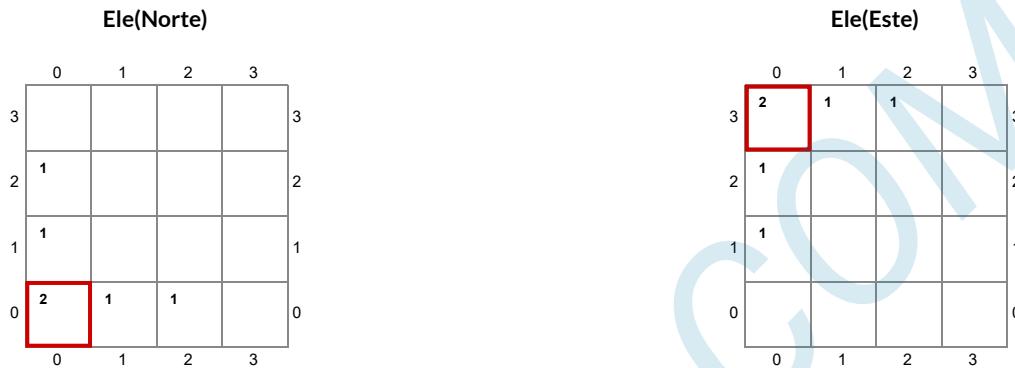
© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





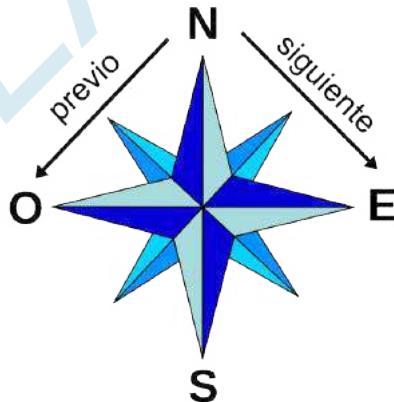
Dibujando una L

Nuestro objetivo en este ejercicio será definir un procedimiento capaz de dibujar una letra L de color Azul, pero con la posibilidad de elegir hacia dónde está orientada. A continuación, algunos ejemplos de cómo debería comportarse:



Indudablemente, una L consta de dos líneas y dibujar una línea es la tarea que ya resolviste en el ejercicio anterior. Así que por ese lado, tenemos la mitad del problema resuelto.

La primera línea es fácil, porque coincide con la dirección que recibimos por parámetro... ¿pero la segunda? Bueno, ahí viene lo interesante: además de opuesto, Gobstones nos provee dos funciones más para operar sobre las direcciones, `siguiente` y `previo`. `siguiente(direccion)` retorna la dirección siguiente a la especificada, mientras que `previo(direccion)` retorna la anterior, siempre pensándolo en el sentido de las agujas del reloj:



Descubrí cuál de las funciones nuevas tenés que invocar y definí el procedimiento `Ele(direccion)`. No te preocupes por la posición inicial del cabezal, nosotros nos encargaremos de ubicarlo en el lugar correspondiente para que la L se pueda dibujar.

¡Dame una pista!

[Solución](#)

[Biblioteca](#)

```

1 procedure Ele(direccion) {
2   Linea(direccion, Azul, 3)
3   Linea(siguiente(direccion), Azul, 3)
4
5 }
```

► Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:



Tablero inicial

0	1	2	3
3			
2			
1			
0			

Tablero final

0	1	2	3
3			
2	1		
1	1		
0	2	1	1



Tablero inicial

0	1	2	3
3			
2			
1			
0			

Tablero final

0	1	2	3
3	2	1	1
2	1		
1	1		
0	0		

Esta guía fue desarrollada por Federico Aloï, Alfredo Sanzo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Siga la flecha

Ya vimos distintas funciones que a partir de una dirección nos permiten obtener otra distintas.

Como siempre en programación, lo interesante es combinar nuestras herramientas para lograr nuevos objetivos 😊. Por ejemplo podemos dibujar flechas en una dirección determinada de la siguiente forma:

Flecha(Norte)

	0	1	2	
2		1		2
1	1		1	1
0				0
	0	1	2	

Flecha(Oeste)

	0	1	2	
2		1		2
1	1		1	1
0				0
	0	1	2	

Definí el procedimiento `Flecha(direccion)` que dibuje una flecha roja en la dirección correspondiente. El cabezal empieza y debe quedar siempre en el centro, como se ve en los tableros de ejemplo.

¡Dame una pista!

```

1 procedure Punta(direccion) {
2   Mover(direccion)
3   Poner(Rojo)
4   Mover(opuesto(direccion))
5 }
6
7 procedure Flecha(direccion) {
8   Punta(previo(direccion))
9   Punta(direccion)
10  Punta(siguiente(direccion))
11 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial

	0	1	2	
2		1		2
1	1		1	1
0				0
	0	1	2	

Tablero final

	0	1	2	
2		1		2
1	1		1	1
0				0
	0	1	2	



Tablero inicial

	0	1	2	
2				2
1				1
0				0

Tablero final

	0	1	2	
2				2
1		1		1
0	1		1	0

Esta guía fue desarrollada por Federico Aloí, Alfredo Sanzo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://github.com/filadd>)



Copiando bolitas

Supongamos ahora que queremos "copiar" las bolitas verdes, haciendo que haya la misma cantidad de rojas y pensemos cómo podría ser ese procedimiento.

Una tarea que seguro tenemos que hacer es poner muchas bolitas, y para eso ya sabemos que existe el procedimiento `PonerN` que construimos varios ejercicios atrás. El color de las bolitas que tenemos que poner también lo sabemos: Rojo, pero... ¿cómo sabemos cuántas poner?

Miremos algunos ejemplos:

- Si hay 4 bolitas verdes, hay que poner 4 bolitas rojas.
- Si hay 2 bolitas verdes, hay que poner 2 bolitas rojas.
- Si no hay bolitas verdes, no hay que poner ninguna roja.

Lo que nos está faltando es una forma de **contar cuántas bolitas verdes hay**, y para eso necesitamos otra función que nos da Gobstones: `nroBolitas(color)`. Lo que hace es simple: nos retorna la cantidad de bolitas de un color determinado **en la posición actual**.

Invocando `nroBolitas`, definí el procedimiento `CopiarVerdesEnRojas`.

¡Dame una pista!

Solución

Biblioteca

```
1 procedure CopiarVerdesEnRojas() {  
2   PonerN(nroBolitas(Verde), Rojo)  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Si no hay verdes, no hace nada

Tablero inicial

Tablero final

	0	1	
1			1
0	1		0
0			0

Si hay 2 verdes, pone 2 rojas

Tablero inicial

	0	1
1		
0	2	

Tablero final

	0	1
1		
0	2	2

✓ Si hay 7 verdes, pone 7 rojas

Tablero inicial

	0	1
1		
0	7	

Tablero final

	0	1
1		
0	7	7

En este ejercicio, además de aprender una expresión nueva, hiciste algo que nunca habías hecho hasta ahora: un programa cuyo efecto depende del estado del tablero inicial.

¿Qué quiere decir esto? Que el código de `CopiarVerdesEnRojas()` se comporta diferente dependiendo de cómo estaba el tablero **antes** de ejecutarlo.

Esta guía fue desarrollada por Federico Aloia, Alfredo Sanzo bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Sacando bolitas

Siguiendo con esto de contar las bolitas, te toca ahora definir un procedimiento que sirva para sacar **todas** las bolitas de **un color**.

Pensemos las subtareas necesarias:

1. Poder sacar muchas bolitas: ya está resuelto con `SacarN`.
2. Contar cuántas bolitas hay que sacar: se puede hacer con `nroBolitas`.
3. Sacar todas las bolitas de un color: hay que combinar las 2 anteriores.

Definí `SacarTodas(color)`, que recibe un color y saca todas las bolitas que hay de ese color (no debe hacer nada con el resto de los colores).

¡Dame una pista!

[Solución](#)

[Biblioteca](#)

```
1 procedure SacarTodas(color) {  
2   SacarN(nroBolitas(color), color)  
3 }
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial	Tablero final
 A 2x2 grid with columns labeled 0 and 1, and rows labeled 0 and 1. The bottom-left square (0,0) contains the value 3. All other squares are empty (containing 0). A red box highlights the (0,0) square.	 The same 2x2 grid. The bottom-left square (0,0) now contains the value 0, indicating it has been removed. All other squares remain empty (containing 0).

Tablero inicial	Tablero final
 A 2x2 grid with columns labeled 0 and 1, and rows labeled 0 and 1. The bottom-left square (0,0) contains the value 5. All other squares are empty (containing 0). A red box highlights the (0,0) square.	 The same 2x2 grid. The bottom-left square (0,0) now contains the value 0, indicating it has been removed. All other squares remain empty (containing 0).

Tablero inicial

Tablero final

Esta guía fue desarrollada por Federico Aloí, Alfredo Sanzo bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





Sacar con miedo

Vamos a definir un procedimiento que saque una bolita azul "con miedo": no tiene que producirse un BOOM, aún cuando no haya ninguna bolita en la celda actual.

Con lo que sabés hasta ahora, probablemente tu primera idea sea hacer algo como esto:

```
procedure SacarAzulConMiedo() {
    Sacar(Azul)
}
```

¡Probalo! Copiá el código anterior en el editor y apretá *Enviar*.

```
1 procedure SacarAzulConMiedo() {
2     Sacar(Azul)
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Cuando hay una bolita azul, la saca

Tablero inicial	Tablero final																
<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td></td></tr><tr><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td></tr></table>	0	1	1		0	1	0	0	<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td></td></tr><tr><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td></tr></table>	0	1	1		0	0	0	1
0	1																
1																	
0	1																
0	0																
0	1																
1																	
0	0																
0	1																

Cuando no hay ninguna bolita azul, hace BOOM

Tablero inicial	Tablero final								
<table border="1"><tr><td>0</td><td>1</td></tr><tr><td>1</td><td></td></tr><tr><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td></tr></table>	0	1	1		0	0	0	1	
0	1								
1									
0	0								
0	1								

BOOM

[2:3]: No se puede sacar una bolita de color Azul: no hay bolitas de ese color.

¿Te diste cuenta qué pasó?

Funcionó para el primer tablero porque tenía una bolita azul, pero hizo BOOM para el segundo porque estaba vacío, claro.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Sacar con miedo, segundo intento

Ahora probá esta segunda versión que agrega una **alternativa condicional**. No te preocupes por la sintaxis, ya te lo vamos a explicar. 😊

```
procedure SacarAzulConMiedo() {
    if (hayBolitas(Azul)) {
        Sacar(Azul)
    }
}
```

Copíá el código anterior en el editor y apretá **Enviar**.

```
1 procedure SacarAzulConMiedo() {
2     if (hayBolitas(Azul)) {
3         Sacar(Azul)
4     }
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Cuando hay una bolita azul, la saca

Tablero inicial	Tablero final

Cuando no hay ninguna bolita azul, no hace nada

Tablero inicial	Tablero final

¡Bien!

Hiciste tu primer procedimiento que **decide** antes de ejecutar. Seguí con nosotros para entender de qué se trata esto...



Eliminando la bolita roja

Analicemos el procedimiento del ejercicio anterior:

```
procedure SacarAzulConMiedo() {
    if (hayBolitas(Azul)) {
        Sacar(Azul)
    }
}
```

Como notarás, introdujimos una nueva estructura de control: el **if**, que en castellano significa *si*; entendiendo al *si* como **condicional** ("*si* tuviera hambre me comería una empanada") y no como afirmación ("*sí*, yo rompí el teléfono").

Entonces, lo que le estamos diciendo a la computadora es "*si hay bolitas azules, sacá una bolita azul*", que dicho así suena un poco tonto ¡y lo es!. Ya te dijimos que la computadora sólo sabe cumplir órdenes.

¡Ahora te toca a vos! Modificá el procedimiento que te dimos para que saque una bolita roja, sólo si hay alguna.

¡Dame una pista!

```
1 procedure SacarRojoConMiedo() {
2     if (hayBolitas(Rojo)) {
3         Sacar(Rojo)
4     }
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Cuando hay una bolita roja, la saca

Tablero inicial

Tablero final

0	1	
1		
0		

Cuando hay más de una bolita roja, saca una

Tablero inicial

0	1
1	
0	4

Tablero final

0	1
1	
0	3

Cuando no hay ninguna bolita roja, no hace nada

Tablero inicial

0	1
1	
0	4

Tablero final

0	1
1	
0	3

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Un ejemplo medio rebuscado

Vamos a ponerle nombre a las partes del `if`.

En primer lugar, tenemos la **condición**. Por ahora siempre fue `hayBolitas(color)` pero podría ser cualquier otra cosa, ya veremos más ejemplos. Lo importante acá es que eso es lo que **decide** si la **acción** se va a ejecutar o no.

¿Y qué es la **acción**? Básicamente, cualquier cosa que queramos hacer sobre el tablero. Al igual que en el `repeat`, podemos hacer cuantas cosas se nos ocurran, no necesariamente tiene que ser una sola.

Para ejercitarte con esto, te vamos a pedir que definas un procedimiento `CompletarCelda()` que, si ya hay alguna bolita negra, complete la celda poniendo una roja, una azul y una verde.

¡Dame una pista!

```

1 procedure CompletarCelda() {
2   if (hayBolitas(Negro)) {
3     Poner(Azul)
4     Poner(Rojo)
5     Poner(Verde)
6   }
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Cuando hay alguna bolita negra, completa la celda

Tablero inicial	Tablero final
0 1	1
1	1
0 2	1 2
0 1	1 1
0 1	0 1

Cuando no hay ninguna bolita negra, no hace nada

Tablero inicial	Tablero final
0 1	1
1	1
0 1	1
0 1	0 1
0 1	0 1



¿Y sólo sirve para ver si hay bolitas?

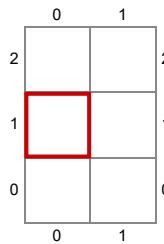
Claro que no, ¡por suerte! 🙌

La condición puede ser cualquier expresión booleana. En criollo: cualquier cosa que represente una "pregunta" que se pueda responder con **sí** o **no**. En Gobstones el **sí** se representa con el valor **True** (**Verdadero** en castellano) y el **no** con el valor **False** (**Falso** en castellano).

En los ejercicios anteriores te mostramos una de las expresiones que trae Gobstones, `hayBolitas(color)`, que recibe un `color` y retorna `True` o `False`.

Otra que trae `True` o `False` (y que vas a tener que usar ahora) es `puedeMover(direccion)` que nos sirve para saber si el cabezal puede moverse en una cierta dirección.

Por ejemplo, si tenemos este tablero:



- `puedeMover(Norte)` será `True`.
- `puedeMover(Sur)` será `True`.
- `puedeMover(Este)` será `True`.
- Pero `puedeMover(Oeste)` será `False`

Creá un programa que se mueva al **Este** sólo si es posible. Recordá utilizar `puedeMover(direccion)`.

¡Dame una pista!

```
1 program {
2   if (puedeMover(Este)) {
3     Mover(Este)
4   }
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Si hay celdas al Este, se mueve

Tablero inicial

0	1
1	
0	

Tablero final

0	1
1	
0	

✓ Si no hay celdas al Este, no hace nada

Tablero inicial

0	1
1	
0	

Tablero final

0	1
1	
0	

¿Y si hubieramos querido movernos hacia el Norte en caso de que **no** hubiera celdas al Este?

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Un poquito de matemática

Otra cosa que se puede hacer adentro de un `if` es comparar números, como seguramente alguna vez hiciste en matemática.

Por suerte, esto se escribe en Gobstones igual que en la matemática tradicional, con un `<` para el menor y un `>` para el mayor. Ejemplo: `nroBolitas(Verde) > 5` nos indica si hay más de 5 bolitas verdes.

Sabiendo esto, intentá crear un programa que ponga 1 bolita negra sólo si hay menos de 5 bolitas negras.

```
1 program {
2   if (nroBolitas(Negro) < 5) {
3     Poner(Negro)
4   }
5 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Si hay menos de 5 bolitas negras, agrega una

Tablero inicial	Tablero final
 0 1 2 1 1 0 0 0	 0 1 2 1 1 0 0 0

Si hay más de 5 bolitas negras, no hace nada

Tablero inicial	Tablero final
 0 1 2 1 1 0 0 0	 0 1 2 1 1 0 0 0



Cómo decirle que no...

En todos los problemas que hicimos hasta ahora, siempre preguntamos si una cierta condición se cumplía: ¿hay alguna bolita roja? ¿me puedo mover al Este? ¿hay más de 3 bolitas azules?

Algo que también se puede hacer es **negar** una condición, algo que en castellano puede sonar medio raro pero que en programación se hace un montón. Los ejemplos anteriores quedarían: **no** hay alguna bolita roja? **no** me puedo mover al Este? **no** hay más de 3 bolitas azules?

¿Y cómo se hace en Gobstones? Fácil, se agrega la palabra clave **not** antes de la expresión que ya teníamos.

Original	Negada
hayBolitas(Rojo)	not hayBolitas(Rojo)
puedeMover(Este)	not puedeMover(Este)
nroBolitas(Azul) > 3	not nroBolitas(Azul) > 3

Definí un procedimiento **AsegurarUnaBolitaVerde()** que se asegure que en la celda actual hay al menos una bolita verde. Esto es: si ya hay bolitas verdes no hay que hacer nada, pero si **no** hay tendría que poner una.

```

1 procedure AsegurarUnaBolitaVerde() {
2   if (not hayBolitas(Verde)) {
3     Poner(Verde)
4   }
5 }
```

Enviar

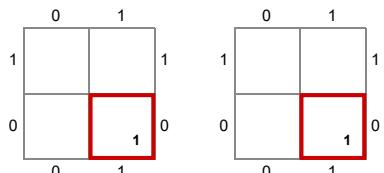
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Si hay bolitas verdes, no hace nada

Tablero inicial

Tablero final



Si no hay bolitas verdes, agrega una

Tablero inicial

	0	1
1		
0		

Tablero final

	0	1
1		
0		1

A lo que acabás de hacer, en lógica se lo llama **negación** y al anteponer el `not` decimos que se está **negando** una expresión. Cualquier expresión booleana (o sea, que devuelve `True` o `False`) se puede negar.

Esta guía fue desarrollada por Federico Aloi bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Dos caminos distintos

En lo cotidiano, se presentan muchas situaciones donde debemos elegir entre dos acciones diferentes, dependiendo de si se cumple una cierta condición o no.

- Si la remera está limpia me la pongo, *si no* la lavo.
- Si tengo aceite para freir las milanesas lo uso, *si no* le pongo un poco de manteca.
- Si me puedo mover al Este lo hago, *si no* me muevo al Norte.

Para estos casos, en Gobstones tenemos una nueva palabra clave que nos ayuda a cumplir nuestra tarea: el **else**. En castellano significa *si no* y hace justamente lo que necesitamos: ejecuta una serie de acciones *si no se cumple* la condición que pusimos en el **if**.

Supongamos que queremos definir un procedimiento que se mueva al Oeste y, en caso de que no pueda, lo haga hacia el Norte. Haciendo uso del **else**, podemos definirlo de la siguiente manera:

```
procedure MoverComoSea() {
    if (puedeMover(Oeste)) {
        Mover(Oeste)
    } else {
        Mover(Norte)
    }
}
```

Escribí ese código en el editor y fijate cómo resuelve el problema.

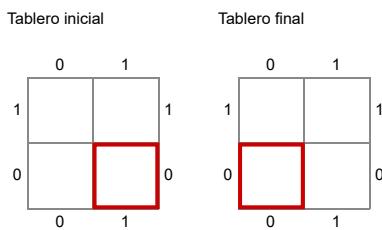
```
1 procedure MoverComoSea() {
2     if (puedeMover(Oeste)) {
3         Mover(Oeste)
4     } else {
5         Mover(Norte)
6     }
7 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Si hay celdas al Oeste, se mueve



Si no hay celdas al Oeste, se mueve al Norte

Tablero inicial	Tablero final												
<table border="1"> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td></td></tr> <tr> <td>0</td><td></td></tr> </table>	0	1	1		0		<table border="1"> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td></td></tr> <tr> <td>0</td><td></td></tr> </table>	0	1	1		0	
0	1												
1													
0													
0	1												
1													
0													

¡Espectacular!

Ya conocés la herramienta que usan todas las aplicaciones que conociste en tu vida para decidir qué hacer, el viejo y querido **if / if...else**.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Un tablero de luces

Como ejemplo final, imaginemos que nuestro tablero está lleno de luces que están prendidas o apagadas. Vamos a decir que las celdas con una bolita verde están prendidas y las celdas con una bolita negra están apagadas.

Definí un procedimiento `PrenderOApagarLuz()` que se encargue de prender las luces que estén apagadas o apagar las luces encendidas, según corresponda.

¡Dame una pista!

```

1 procedure PrenderOApagarLuz() {
2   if (hayBolitas(Verde)) {
3     Sacar(Verde)
4     Poner(Negro)
5   } else {
6     Sacar(Negro)
7     Poner(Verde)
8   }
9 }
10

```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Si la celda está apagada, la prende

Tablero inicial			Tablero final		
0	1	2	0	1	2
2	1	1	1	2	2
1	1	1	1	1	1
0	1	1	1	1	0
0	1	1	1	1	0
1	1	1	1	1	1
2	1	1	1	2	2

Si la celda está prendida, la apaga

Tablero inicial			Tablero final		
0	1	2	0	1	2
2	1	1	1	2	2
1	1	1	1	1	1
0	1	1	1	1	0
0	1	1	1	1	0
1	1	1	1	1	1
2	1	1	1	2	2



Un desorden muy especial

Como no queremos que se termine esta maravillosa lección 😊 vamos a hacer un poco de desorden. Necesitamos un procedimiento DesordenarCelda que:

- reemplace las bolitas azules por verdes;
- duplique las bolitas rojas;
- saque las bolitas negras.

¿Y qué tiene de especial este desorden? 😊 Aunque como siempre podés enviar tu solución las veces que quieras, no la vamos a evaluar automáticamente por lo que el ejercicio quedará en color celeste 😊. Si querés verla en funcionamiento, ¡te invitamos a que la copies en la página de Gobstones (<https://gobstones.github.io/gobstones-sr/>)!

Definí un procedimiento DesordenarCelda que se comporte como te explicamos arriba.

```
1 procedure SacarN(cantidad, color) {
2   repeat(cantidad) {
3     Sacar(color)
4   }
5 }
6
7 procedure PonerN(cantidad, color) {
8   repeat(cantidad) {
9     Poner(color)
10  }
11 }
12
13 procedure DesordenarCelda() {
14   if (hayBolitas(Azul)) {
15     PonerN(nroBolitas(Azul), Verde)
16     SacarN(nroBolitas(Azul), Azul)
17   }
18   if (hayBolitas(Rojo)) {
19     PonerN(nroBolitas(Rojo), Rojo)
20   }
21   if (hayBolitas(Negro)) {
22     SacarN(nroBolitas(Negro), Negro)
23   }
24 }
25
26 program {
27   DesordenarCelda()
28 }
29
```

Enviar

¡Gracias por enviar tu solución!



Y esto, ¿con qué se come?

Tomate unos pocos minutos y tratá de entender qué hace este procedimiento:

```
procedure MoverSegunBolitas() {
    if (nroBolitas(Azul) + nroBolitas(Negro) + nroBolitas(Rojo) + nroBolitas(Verde) > 10) {
        Mover(Este)
    } else {
        Mover(Norte)
    }
}
```

Cuando lo logres interpretar (o te canses 🤦), presioná **Enviar** y mirá el resultado.

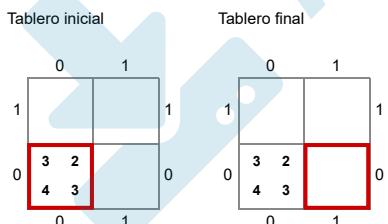
```
1 procedure MoverSegunBolitas() {
2     if (nroBolitas(Azul) + nroBolitas(Negro) + nroBolitas(Rojo) + nroBolitas(Verde) > 10) {
3         Mover(Este)
4     } else {
5         Mover(Norte)
6     }
7 }
```

Enviar

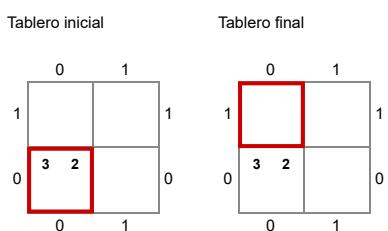
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Si hay más de 10 bolitas, se mueve al Este



Si hay menos de 10 bolitas, se mueve al Norte



Costó entender qué hace sin **ejecutarlo**, ¿no?

Eso es señal de que nos está faltando **dividir en subtareas**...



La importancia de nombrar las cosas

Como vimos, el problema de lo anterior era la falta de **división en subtareas**: la expresión que cuenta la cantidad de bolitas que hay en la celda es demasiado **compleja**, y cuesta entender a simple vista que hace eso.

Entonces, lo que nos está faltando es algún mecanismo para poder **darle un nombre** a esa **expresión compleja**; algo análogo a los **procedimientos** pero que sirva para encapsular expresiones.

La buena noticia es que Gobstones nos permite hacer esto, y la herramienta para ello es definir una **función**, que se escribe así:

```
function nroBolitasTotal() {  
    return (nroBolitas(Azul) + nroBolitas(Negro) + nroBolitas(Rojo) + nroBolitas(Verde))  
}
```

Pegá el código anterior en el editor y observá el resultado.

```
1 function nroBolitasTotal() {  
2     return (nroBolitas(Azul) + nroBolitas(Negro) + nroBolitas(Rojo) + nroBolitas(Verde))  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

nroBolitasTotal() -> 12

Tablero inicial

	0	1
1		
0	3 2 4 3	

nroBolitasTotal() -> 5

Tablero inicial

	0	1
1		
0	3 2	

Algunas aclaraciones sobre las funciones:

- son un caso particular de las **expresiones**, y por lo tanto siguen las mismas reglas que ellas: se escriben con la **primera letra minúscula** y siempre **denotan** algún valor (en este caso, un número);
- en la última línea de su definición siempre va un `return`, seguido de una expresión **entre paréntesis**: el **valor** que la función va a retornar.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





MoverSegunBolitas, versión 2

Ahora que ya logramos mover la cuenta de las bolitas a una subtarea, podemos mejorar el procedimiento que habíamos hecho antes.

Modificá la primera versión de `MoverSegunBolitas` para que use la función `nroBolitasTotal()` en vez de la expresión larga.

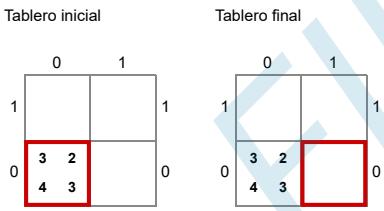
```
1 procedure MoverSegunBolitas() {  
2   if (nroBolitasTotal() > 10) {  
3     Mover(Este)  
4   } else {  
5     Mover(Norte)  
6   }  
7 }
```

Enviar

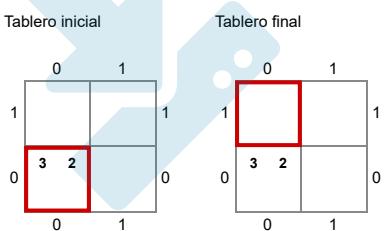
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Si hay más de 10 bolitas, se mueve al Este



Si hay menos de 10 bolitas, se mueve al Norte



Las **funciones** son una herramienta importantísima, que nos ayuda a escribir programas de mayor calidad.

Sólo mirando el código de esta nueva versión del procedimiento podemos entender de qué va nuestro problema, lo que **reduce la distancia** entre el problema real y la **estrategia** que elegimos para resolverlo.

Esta guía fue desarrollada por Federico Aloi bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](https://creativecommons.org/licenses/by-sa/4.0/) (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020 Mumuki (<http://mumuki.org/>)





todasExcepto

4. todasExcepto

Te toca ahora definir tu primera función: `todasExcepto(color)`. Lo que tiene que hacer es sencillo, contar cuántas bolitas hay en la celda actual **sin tener en cuenta** las del color recibido por parámetro.

Por ejemplo, `todasExcepto(Verde)` debería contar todas las bolitas azules, negras y rojas que hay en la celda actual (o dicho de otra forma: todas las bolitas que hay **menos** las verdes).

Definí la función `todasExcepto` para que retorne la cantidad de bolitas que **no** sean del color que se le pasa por parámetro.

¡Dame una pista!

```
1 function todasExcepto(color) {  
2   return (nroBolitasTotal() - nroBolitas(color))  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

`todasExcepto() -> 8`

Tablero inicial

0	1
1	1
0	0
0	1

`todasExcepto() -> 10`

Tablero inicial

0	1
1	1
0	0
0	1

Las funciones, como cualquier otra **expresión**, se pueden usar para definir nuevas funciones.

Y volvemos así a lo más lindo de la programación: la posibilidad de **construir nuestras propias herramientas** parandonos sobre cosas que hicimos antes, logrando que lo



Una función de otro tipo

Como ya sabés, las expresiones no sólo sirven para operar con números. Vamos a definir ahora una función que retorne un valor **booleano** (True / False).

Lo que queremos averiguar es si el color Rojo es dominante dentro de una celda. Veamos algunos ejemplos.

En este casillero:

0			
4	3	0	
2	1		
0			

rojoEsDominante() **retorna** False (hay 2 bolitas rojas contra 8 de otros colores). Pero en este otro:

0			
4	3	0	
9	1		
0			

rojoEsDominante() **retorna** True (hay 9 bolitas rojas contra 8 de otros colores)

Definí la función `rojoEsDominante()` que nos diga si la cantidad de bolitas rojas **es mayor** que la suma de las bolitas de los otros colores. En la Biblioteca está `todasExcepto(color)` lista para ser invocada. ☺

¡Dame una pista!

Solución

Biblioteca

```
1 function rojoEsDominante() {  
2   return (nroBolitas(Rojo) > todasExcepto(Rojo))  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

rojoEsDominante() -> False

Tablero inicial

	0	1
1		
0	3 2 4 3	0

rojoEsDominante() -> True

Tablero inicial

	0	1
1		
0	3 2 10 3	0

Las funciones pueden retornar distintos **tipos**: un color, una dirección, un número o un booleano.

Básicamente, lo que diferencia a un tipo de otro son las **operaciones que se pueden hacer con sus elementos**: tiene sentido sumar números, pero no colores ni direcciones; tiene sentido usar `Poner` con un color, pero no con un booleano. Muchas veces, pensar en el tipo de una función es un primer indicador útil de si lo que estamos haciendo está bien.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





En libertad

Queremos definir la función `esLibreCostados()`, que determine si el cabezal tiene libertad para moverse hacia los costados (es decir, Este y Oeste).

Antes que nada, pensemos, ¿qué **tipo** tiene que denotar nuestra función? Será...

- ... ¿un **color**? No.
- ... ¿un **número**? Tampoco.
- ... ¿una **dirección**? Podría, pero no. Fijate que lo que pide es "saber si puede moverse" y no hacia dónde.
- ... ¿un **booleano**? ¡Sí! 🎉 Cómo nos dimos cuenta: lo que está pidiendo tiene pinta de **pregunta** que se responde con sí o no, y eso es exactamente lo que podemos representar con un valor booleano: **Verdadero** o **Falso**.

Pero, ups, hay un problema más; hay que hacer DOS preguntas: ¿se **puede mover** al Este? Y ¿se **puede mover** al Oeste? 🤔

Bueno, existe el operador `&&` que sirve justamente para eso: toma dos expresiones booleanas y devuelve `True` solo si **ambas** son verdaderas. Si sabés algo de lógica, esto es lo que comúnmente se denomina **conjunción** y se lo suele representar con el símbolo \wedge .

Por ejemplo, si quisieramos saber si un casillero tiene más de 5 bolitas y el **Rojo** es el color dominante podríamos escribir:

```
nroBolitasTotal() > 5 && rojoEsDominante()
```

Definí la función `esLibreCostados()` que indique si el cabezal puede moverse tanto al Este como al Oeste.

¡Dame una pista!

```
1 function esLibreCostados() {
2   return (puedeMover(Este) && puedeMover(Oeste))
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

`esLibreCostados() -> False`

Tablero inicial

	0	1	2	
1				1
0	■			0

esLibreCostados() -> True

Tablero inicial

	0	1	2	
1				1
0				0
	0	1	2	

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Cualquier bolita nos deja bien

Definí la función `hayAlgunaBolita()` que responda a la pregunta ¿hay alguna bolita en la celda actual?

Otra vez una pregunta, por lo tanto hay que retornar un **booleano**. Además, podemos ver que acá también hay que hacer **más de una pregunta**, en particular cuatro: una por cada una de los colores.

A diferencia del ejercicio anterior, lo que queremos saber es si **alguna** de ellas es verdadera, por lo tanto hay que usar otro operador: la **disyunción**, que se escribe `||` y retorna verdadero si al menos **alguna de las dos** preguntas es verdadera.

De nuevo, si sabés algo de lógica, esta operación suele representarse con el símbolo \vee .

¡Dame una pista!

```
1 function hayAlgunaBolita() {  
2   return (hayBolitas(Azul) || hayBolitas(Verde) || hayBolitas(Negro) || hayBolitas(Rojo) )  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

`hayAlgunaBolita() -> False`

Tablero inicial

0	1
1	1
0	0

`hayAlgunaBolita() -> True`

Tablero inicial

0	1
1	1
0	0

`hayAlgunaBolita() -> True`

Tablero inicial

	0	1
1		
0	8	0

✓ hayAlgunaBolita() -> True

Tablero inicial

	0	1
1		
0	2	4

Tanto `&&` como `||` pueden usarse varias veces sin la necesidad de usar paréntesis, siempre y cuando tengan expresiones booleanas a ambos lados.

Esta guía fue desarrollada por Federico Aloi bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Siempre al borde...

Te recordamos los operadores lógicos que vimos hasta ahora:

- **Negación:** "da vuelta" una expresión booleana - ejemplo: `not hayBolitas(Rojo)` .
- **Conjunción:** determina si se cumplen **ambas** condiciones - ejemplo: `puedeMover(Norte) && puedeMover(Sur)` .
- **Disyunción:** determina si se cumple **alguna** de las condiciones - ejemplo: `esInteligente() || tieneBuenaOnda()` .

Con la ayuda de esa tablita, definí la función `estoyEnUnBorde()` que determine si el cabezal está parado en algún borde.

¡Dame una pista!

```
1 function estoyEnUnBorde() {  
2     return (not puedeMover(Norte) || not puedeMover(Este) || not puedeMover(Sur) || not puedeMover(Oeste))  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

`estoyEnUnBorde() -> True`

Tablero inicial

0	1	2	2
2			
1			1
0			0

`estoyEnUnBorde() -> True`

Tablero inicial

0	1	2	2
2			
1			1
0			0

`estoyEnUnBorde() -> False`

Tablero inicial

0	1	2	2
2			
1			1
0			0

Como en la aritmética, en la lógica también existe el concepto de **precedencia** y ciertas operaciones se resuelven antes que otras: primero la negación (`not`), después la conjunción (`&&`) y por último la disyunción (`||`).

Por esta razón, la expresión `not puedeMover(Norte) || not puedeMover(Este) || not puedeMover(Sur) || not puedeMover(Oeste)` se puede escribir sin tener que poner paréntesis en el medio.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Las compañeras ideales

Vamos a ver ahora funciones que **hacen cosas antes** de retornar un resultado. Para exemplificar esto, vamos a querer que definas una función que nos diga si hay una bolita de un color específico, pero **en la celda de al lado**.

Definí la función `hayBolitasAl(direccion, color)` que informe si hay alguna bolita del color especificado en la celda vecina hacia la dirección dada.

Ojo: como ya dijimos, **la última línea** siempre tiene que tener un `return`.

¡Dame una pista!

```
1 function hayBolitasAl(direccion, color) {  
2   Mover(direccion)  
3   return (hayBolitas(color))  
4 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

`hayBolitasAl() -> True`

Tablero inicial

	0	1	2
2			
1	1		
0	1	1	0

`hayBolitasAl() -> False`

Tablero inicial

	0	1	2
2			
1	1		
0	1	1	0

`hayBolitasAl() -> True`

Tablero inicial

	0	1	2	
2				2
1	1			1
0	1	1	2	0

¿Viste qué pasó? El cabezal "no se movió" y sin embargo la función devolvió el resultado correcto.

Esto pasa porque en Gobstones las funciones son **puras**, no tienen **efecto real** sobre el tablero. En ese sentido decimos que son las compañeras ideales: después de cumplir su tarea **dejan todo como lo encontraron**.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Lo ideal también se puede romper

Como en la definición de `hayBolitasA1` se usa `Mover`, es obvio que hay casos en los cuales podría romperse: basta con posicionar el cabezal en el origen y preguntar si `hayBolitas` de algún color al Oeste.

Pero, ¿no era que las funciones eran **puras** y no tenían efecto real? ¿Qué pasa si una función hace BOOM?

Hagamos la prueba: vamos a probar la función `hayBolitasA1` del ejercicio anterior con casos donde no pueda moverse el cabezal. Presioná Enviar y mirá el resultado.

```
1 function hayBolitasA1(direccion, color) {  
2   Mover(direccion)  
3   return (hayBolitas(color))  
4 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

Tablero inicial			Tablero final		
2	1	2	2	1	0
1					
0	1	1	1	1	1
0	1	2	2	1	0

Tablero final



BOOM

[2:3]: No se puede mover hacia la dirección Sur: cae afuera del tablero.

Tablero inicial			Tablero final		
2	1	2	2	1	0
1					
0	1	1	1	1	1
0	1	2	2	1	0

Tablero final



BOOM

[2:3]: No se puede mover hacia la dirección Oeste: cae afuera del tablero.

¡BOOM! 

Las funciones también pueden **producir BOOM** y por lo tanto tenés que tener el mismo cuidado que al programar un procedimiento: que el cabezal no salga del tablero, no intentar sacar bolitas de un color que no hay, etc.

Pensándolo así, podemos decir que las funciones **deshacen sus efectos** una vez que terminan, pero para poder devolver un resultado necesitan que sus acciones puedan ejecutarse.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





¿Hay bolitas lejos?

Ejercitemos un poco más esto de las funciones con procesamiento.

Te toca programar una nueva versión de `hayBolitasA1` que mire si hay bolitas a cierta distancia de la celda actual. A esta función la vamos a llamar `hayBolitasLejosA1` y recibirá tres parámetros: una **dirección** hacia donde deberá moverse, un **color** por el cual preguntar y una **distancia** que será la cantidad de veces que habrá que moverse.

Por ejemplo: `hayBolitasLejosA1(Norte, Verde, 4)` indica si hay alguna bolita Verde cuatro celdas al Norte de la posición actual.

Para este tablero devolvería **True**:

	0	1	
4	1		4
3			3
2			2
1			1
0			0
0	1		0
1			1
2			2
3			3
4			4

Y para este tablero devolvería **False**:

	0	1	
4			4
3			3
2			2
1			1
0			0
0	1		0
1			1
2			2
3			3
4			4

Definí la función `hayBolitasLejosA1(direccion, color, distancia)`.

¡Dame una pista!

Solución

Biblioteca

```
1 function hayBolitasLejosA1(direccion, color, distancia) {  
2   repeat(distancia) {  
3     Mover(direccion)  
4   }  
5   return (hayBolitas(color))  
6 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

`hayBolitasLejosA1()` -> True

Tablero inicial

	0	1	2	
2	1			2
1				1
0				0
	0	1	2	

✓ hayBolitasLejosAI() -> True

Tablero inicial

	0	1	2	3	4	
1						1
0	1					0
	0	1	2	3	4	
	0	1	2	3	4	

✓ hayBolitasLejosAI() -> False

Tablero inicial

	0	1	2	3	4	
1						1
0	1					0
	0	1	2	3	4	
	0	1	2	3	4	

Se puede realizar cualquier tipo de acción antes de **retornar un valor**, y nada de lo que hagamos tendrá **efecto real** sobre el tablero.

Interesante esto de las funciones, ¿no?

Esta guía fue desarrollada por Federico Aloi bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Estoy rodeado de viejas bolitas

Valiéndote de `hayBolitasAl`, definí la función `estoyRodeadoDe(color)` que indica si el cabezal está rodeado de bolitas de ese color.

Decimos que el cabezal "está rodeado" si hay bolitas de ese color en las cuatro direcciones: Norte, Este, Sur y Oeste.

[¡Dame una pista!](#)

Ya tenés una forma de determinar si hay bolitas en **una dirección**. Combiná eso con el **conectivo lógico** que corresponda y tendrás tu nueva función andando.

[Solución](#)

[Biblioteca](#)

```
1 function estoyRodeadoDe(color) {  
2   return (hayBolitasAl(Norte, color) && hayBolitasAl(Este, color) && hayBolitasAl(Sur, color) &&  
3     hayBolitasAl(Oeste, color))  
4 }
```

[Enviar](#)

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

`estoyRodeadoDe() -> True`

Tablero inicial

	0	1	2
2		1	
1	1		1
0	1		0

`estoyRodeadoDe() -> False`

Tablero inicial

	0	1	2
2		1	
1	1		1
0		1	0

`estoyRodeadoDe() -> False`

Tablero inicial

	0	1	2
2			
1	1		
0		1	
	0	1	2

estoyRodeadoDe() -> False

Tablero inicial

	0	1	2
2		1	
1		1	1
0	1		
	0	1	2

Por si todavía no nos creías: a pesar de que el cabezal se movió cuatro veces por cada prueba, al finalizar la función vemos que siempre quedó en la posición inicial.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)





Sin límites

Para cerrar, vamos a definir la función `hayLímite()`, que determina si hay algún tipo de límite a la hora de mover el cabezal.

El límite puede ser por alguno de dos factores: porque **estoy en un borde** y entonces no me puedo mover en alguna dirección, o porque **estoy rodeado de bolitas rojas** que me cortan el paso. Si ocurre **alguna** de esas dos condiciones, quiere decir que hay un límite.

Usando `estoyEnUnBorde` y `estoyRodeadoDe`, definí `hayLímite`.

¡Dame una pista!

Solución

Biblioteca

```
1 function hayLímite() {  
2     return (estoyEnUnBorde() || estoyRodeadoDe(Rojo))  
3 }
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

`hayLímite() -> True`

Tablero inicial

	0	1	2
2		1	
1	1		1
0		1	

`hayLímite() -> True`

Tablero inicial

	0	1	2
2			
1			
0	1		

`hayLímite() -> False`

Tablero inicial

0	1	2
2		
1	1	
0	1	2

Esta guía fue desarrollada por Federico Aloí bajo los términos de la Licencia Creative Commons Compartir-Igual, 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>).

© Copyright 2015-2020  Mumuki (<http://mumuki.org/>)





Apéndice

Procedimientos Primitivos

Poner(color)

Pone una bolita del color indicado en la casilla actual. Ejemplo:

```
program {
    Poner(Rojo) // pone una bolita roja en la casilla actual.
}
```

Sacar(color)

Saca una bolita del color indicado de la casilla actual. Ejemplo:

```
program {
    Sacar(Negro) // saca una bolita negra de las que hay en la casilla actual.
}
```

Mover(direccion)

Mueve el cabezal indicador de la casilla actual un paso hacia la dirección indicada. Ejemplo:

```
program {
    Mover(Este) // mueve el cabezal una vez hacia el Este.
}
```

IrAlBorde(direccion)

Lleva el cabezal todo lo que se puede hacia la dirección indicada. Ejemplo:

```
program {
    IrAlBorde(Norte) // mueve el cabezal de la celda actual a la última celda en la dirección Norte.
}
```

VaciarTablero()

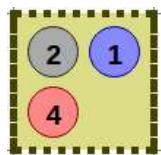
Saca todas las bolitas del tablero, dejando el cabezal en la posición en la que estaba. Ejemplo:

```
program {
    VaciarTablero() // En un tablero con alguna o muchas bolitas, las saca todas.
}
```

Funciones primitivas

nroBolitas(color)

Denota un número: la cantidad de bolitas del color indicado que hay en la casilla actual. Ejemplo, asumiendo la siguiente celda actual:



```
nroBolitas(Rojo) // denota 4
```

opuesto(direccion)

Denota una dirección: la dirección opuesta a la provista. Ejemplo:

```
opuesto(Norte) // denota Sur
```

opuesto(numero)

Denota un número: el original, negado. Ejemplo:

```
opuesto(59) // denota -59
```

siguiente(direccion)

Denota una dirección: la siguiente a la provista, es decir, la próxima en sentido horario. Ejemplo:

```
siguiente(Oeste) // denota Norte  
siguiente(Norte) // denota Este  
siguiente(Este) // denota Sur  
siguiente(Sur) // denota Oeste
```

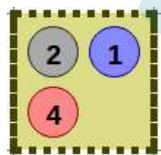
previo(direccion)

Denota una dirección: la anterior a la provista, es decir, la próxima en sentido anti horario. Ejemplo:

```
previo(Sur) // denota Este
```

hayBolitas(color)

Denota un booleano: es cierto cuando en la casilla actual hay al menos una bolita del valor indicado. Ejemplo, asumiendo la siguiente celda actual:



```
hayBolitas(Rojo) // denota cierto  
hayBolitas(Verde) // denota falso
```

puedeMover(direccion)

Denota un booleano: si el cabezal puede moverse en esa dirección (o sea, no está en el borde). Por ejemplo, estando el cabezal en la esquina de abajo a la izquierda:

```
puedeMover(Norte) // denota cierto  
puedeMover(Oeste) // denota falso
```



Expresiones lógicas y matemáticas

- $p \wedge q$ (*and*): `p && q`
- $p \vee q$ (*or*): `p || q`
- $\neg p$ (*not*): `not p`
- $x = y$ (*igual*): `x == y`
- $x \neq y$ (*no igual*): `x /= y`
- $x \geq y$ (*mayor o igual*): `x >= y`

© Copyright 2015-2020  [Mumuki](http://mumuki.org/) (<http://mumuki.org/>)



(<https://github.com/filadd>)

