



# 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

Tablero inicial

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

Tablero final

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

✓ Si hay menos de 10 bolitas, se mueve al Norte

Tablero inicial

	0	1	
1			1
0	3	2	0
	0	1	

Tablero final

	0	1	
1			1
0	3	2	0
	0	1	

Costó entender qué hace sin **ejecutarlo**, ¿no?

Eso es señal de que nos está faltando **dividir en subtareas...**

Esta guía fue desarrollada por Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# 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			1
0	3	2	0
	4	3	
	0	1	

✓ `nroBolitasTotal()` -> 5

Tablero inicial

	0	1	
1			1
0	3	2	0
	0	1	

### 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.

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# MoverSegunBolitas, versión 2

Ahora que ya logramos mover la cuenta de las bolitas a una subtarea, podemos mejorar el procedimiento que habíamos hecho antes.

3. MoverSegunBolitas, versión 2

Modificá la primera versión de `MoverSegunBolitas` para que use la función `nroBolitasTotal()` en vez de la expresión larga.

 Solución

 Biblioteca

```
1
2
3
4 procedure MoverSegunBolitas() {
5   if (nroBolitasTotal() > 10) {
6     Mover(Este)
7   } else {
8     Mover(Norte)
9   }
10 }
```

 Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

 Si hay más de 10 bolitas, se mueve al Este

Tablero inicial

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

Tablero final

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

✓ Si hay menos de 10 bolitas, se mueve al Norte

Tablero inicial

	0	1	
1			1
0	3	2	0
	0	1	

Tablero final

	0	1	
1			1
0	3	2	0
	0	1	

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 Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)









# 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).

Definé 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	3	2	0
	4	3	
	0	1	

✓ todasExcepto() -> 10

Tablero inicial

	0	1	
1			1
0	3	2	0
	4	3	
	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** parándonos sobre cosas que hicimos antes, logrando que lo que hacemos sea cada vez más poderoso.

Esta guía fue desarrollada por Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)







# 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	
0	4	3
	2	1
	0	

`rojoEsDominante()` **retorna** `False` (hay 2 bolitas rojas contra 8 de otros colores). Pero en este otro:

	0	
0	4	3
	9	1
	0	

`rojoEsDominante()` **retorna** `True` (hay 9 bolitas rojas contra 8 de otros colores)

Definé 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 }  
4
```



5



Enviar

## ✓ ¡Muy bien! Tu solución pasó todas las pruebas

### Resultados de las pruebas:

✓ rojoEsDominante() -> False

Tablero inicial

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

✓ rojoEsDominante() -> True

Tablero inicial

	0	1	
1			1
0	3 2		0
	10 3		
	0	1	

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](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# 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, ¿de qué **tipo** tiene que ser el valor que retorna 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 comunmente 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:

```
function esUnCasilleroCargadoConRojoDominante() {  
  return (nroBolitasTotal() > 5 && rojoEsDominante())  
}
```



Definé 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(0este))  
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
	0	1	2	

✅ 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](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Cualquier bolita nos deja bien

Definí la función `hayAlgunaBolita()` que responda a la pregunta ¿hay alguna bolita en la celda actual?

7. Cualquier bolita nos deja bien

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(Rojo) || hayBolitas(Verde) || hayBolitas(Negro) || hayBolitas(Azul))  
3 }
```

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

✅ hayAlgunaBolita() -> False

Tablero inicial

	0	1	
1			1
0			0
	0	1	

✅ hayAlgunaBolita() -> True

Tablero inicial

	0	1	
1			1
0	1	3	0
	0	1	

✔ hayAlgunaBolita() -&gt; True

Tablero inicial

	0	1	
1			1
0	8		0
	0	1	

✔ hayAlgunaBolita() -&gt; True

Tablero inicial

	0	1	
1			1
0	2	4	0
	0	1	

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 Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Siempre al borde...

Te recordamos los operadores lógicos que vimos hasta ahora:

8. Siempre al borde...

- **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 }  
4
```

▶ 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
	0	1	2	

✅ `estoyEnUnBorde()` -> True

Tablero inicial

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

✔ `estoyEnUnBorde()` -> False

Tablero inicial

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

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](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# Las compañeras ideales

Vamos a ver ahora funciones que **hacen cosas antes** de retornar un resultado. Para ejemplificar 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**.

9. Las compañeras

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 }  
5  
6  
7  
8
```



▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

✅ hayBolitasAl() -> True

Tablero inicial

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

✅ hayBolitasAl() -> False

Tablero inicial

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

✅ hayBolitasAl() -> True

Tablero inicial

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

¿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](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# 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

	0	1	2
2			
1	1		
0		1	
	0	1	2

Tablero final



BOOM

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





Tablero inicial

	0	1	2
2			
1	1		
0		1	
	0	1	2

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](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# ¿Hay bolitas lejos?

Ejercitemos un poco más esto de las **funciones con procesamiento**.

Te toca programar una nueva versión de `hayBolitasAl` que mire si hay bolitas a cierta distancia de la celda actual. A esta función la vamos a llamar `hayBolitasLejosAl` 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: `hayBolitasLejosAl(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			4
	1		
3			3
2			2
1			1
0			0
	0	1	

Y para este tablero devolvería *False*:

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

Definí la función `hayBolitasLejosAl(direccion, color, distancia)`.

💡 ¡Dame una pista!

☒ Solución [Biblioteca](#)

```
1 function hayBolitasLejosAl (direccion, color, distancia) {  
2   MoverN(distancia, direccion)  
3   return (hayBolitas(color))  
4 }  
5  
6
```

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

✅ hayBolitasLejosAl() -> True

Tablero inicial

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

✔ hayBolitasLejosAl() -> True

Tablero inicial

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

✔ hayBolitasLejosAl() -> False

Tablero inicial

	0	1	2	3	4	
1						1
0	1					0
	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 Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)





# 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!

☒ Solución  Biblioteca

```
1 function estoyRodeadoDe(color) {  
2   return(hayBolitasAl(Norte, color) && hayBolitasAl(Este, color) && hayBolitasAl(Sur, color) &&  
3   hayBolitasAl(Oeste, color))  
4 }  
5
```

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

✅ `estoyRodeadoDe()` -> True

Tablero inicial

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

✅ `estoyRodeadoDe()` -> True

Tablero inicial

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

✅ `estoyRodeadoDe()` -> False

Tablero inicial

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

✓ estoyRodeadoDe() -> False

Tablero inicial

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

✓ estoyRodeadoDe() -> False

Tablero inicial

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

✓ estoyRodeadoDe() -> False

Tablero inicial

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

Por si todavía no nos creías: a pesar de que el cabezal se *movi*ó cuatro veces por cada prueba, al finalizar la función vemos que siempre quedó en la posición inicial.

Reglas del Espacio de Consultas





# Sin límites

Para cerrar, vamos a definir la función `hayLimite()`, 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 `hayLimite`.

 Solución  Biblioteca

```
1 function hayLimite() {  
2   return (estoyRodeadoDe(Rojo) || estoyEnUnBorde())  
3 }  
4  
5  
6
```

 Enviar

 ¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

 `hayLimite() -> True`

Tablero inicial

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

✔ hayLimite() -> True

Tablero inicial

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

✔ hayLimite() -> False

Tablero inicial

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



Esta guía fue desarrollada por Federico Aloí bajo los términos de la [Licencia Creative Commons Compartir-Igual, 4.0](#).

© 2015-2022 Ikumi SRL

[Información importante](#)

[Términos y Condiciones](#)

[Reglas del Espacio de Consultas](#)

