Lenguaje Gobstones

Primeros Programas

Aunque la programación parece una ciencia exacta, **programar es el arte de hacer que una computadora resuelva nuestros problemas**.

Momento... ¿arte? 😱 ¡Sí! Hay muchas formas de resolver un problema y encontrarlas es un proceso creativo 💡. El resultado de este proceso es un *programa*: una descripción de la solución al problema que puede ser *ejecutada* por una máquina.

Saber programar nos da un gran poder: en lugar de hacer tareas repetitivas y tediosas, **usaremos nuestros conocimientos e imaginación para automatizarlas** (por suerte, la computadora no se aburre 😉).

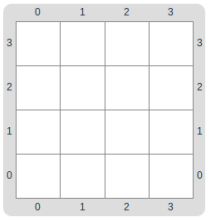
Pero, ¿cómo le explicamos a la máquina de qué forma resolver el problema? Necesitamos escribirla en un idioma que tanto ella como las personas podamos entender: el *lenguaje de programación*.

¡Aprendamos cómo hacerlo de la mano del lenguaje Gobstones!

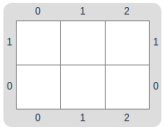
Para empezar a programar, el primer elemento que vamos a usar es un **tablero** cuadriculado, similar al del Ajedrez, Damas o Go.

Estos tableros pueden ser de cualquier tamaño, por ejemplo,

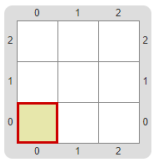
4x4



3x2

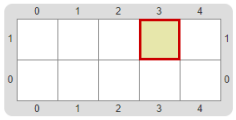


El tablero generado en el ejercicio anterior tenía una **celda** marcada:



¿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 (paciencia, ya las vamos a conocer 😁).

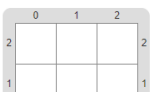
Por ejemplo, el siguiente es un tablero de 5x2, con el cabezal en la segunda fila y la cuarta columna.



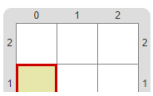
Hasta ahora lo que vimos no fue muy emocionante, porque no te enseñamos cómo darle instrucciones a la máquina y sólo te mostramos un tablero 😒. En este ejercicio vamos a aprender una de las órdenes que podemos darle a la máquina: mover el cabezal.

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**:

Inicial



Final




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

▶ Enviar

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

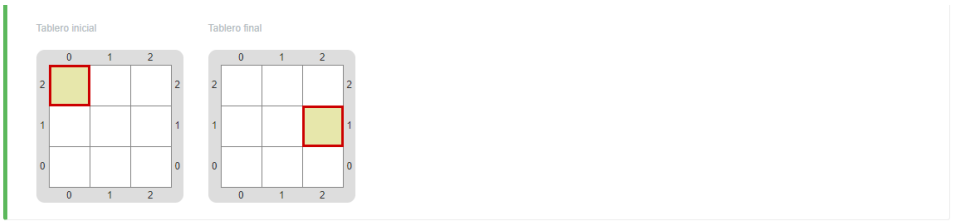
```
program {
  Mover(Norte)
}
```

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

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

¡Te toca a vos!

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



Notá que estos dos programas hacen lo mismo:

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

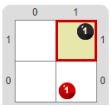
```
program {
  Mover(Este)
  Mover(Sur)
  Mover(Este)
}
```

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:



Se puede diferenciar destinos tipos de comandos:

- Comandos primitivos (mover, poner, sacar) (definidos por el lenguaje)
- Comandos definidos por nosotros (procedimientos)

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 enviá el código para ver qué pasa.

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!

Sintaxis

procedure NOMBRE () {

“COMANDOS”

}

program {

“COMANDOS “

}

IrAlBorde

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.

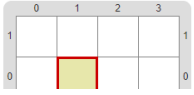
Imaginate que no sabés ni dónde está el cabezal ni qué tamaño tiene el tablero pero querés llegar a una esquina: La primitiva Mover no te va a ser de mucha ayuda. 😱

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**.

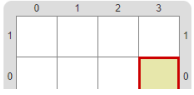
¿Cómo? Mirá el resultado del siguiente programa:

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

Inicial



Final



¡Vamos a aprovecharlo!

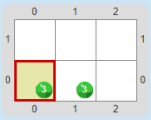
Definí el procedimiento `RojoAlBorde` que ponga una bolita roja en la esquina superior izquierda del tablero e invocalo en el `program`.

¡Dame una pista!

En este caso tenés que lograr que el cabezal quede en una esquina, por lo tanto tenés que pensar en **dos bordes**: `Norte` y `Oeste`.

```
procedure RojoAlBorde  () {
  IrAlBorde (Norte)

  IrAlBorde (Oeste)
 Poner(Rojo)
}

program {
  RojoAlBorde ()
}
```

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



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

```
program {
  Poner3Verdes ()



  Mover(Este)


  Poner3Verdes ()
}
```

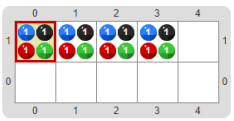
Bueno, ya sabemos cómo crear procedimientos, pero ¿por qué querríamos hacerlos?

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 querríamos 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 (`DibujarCuadradoNegroDeLado3FormadoPor9BolistasDeArribaAAbajo`), 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**.

Linea multicolor

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



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!

```
procedure Poner4Colores () {
  Poner(Rojo)
  Poner(Negro)
  Poner(Verde)
  Poner(Azul )
}

procedure MoverEste () {
  Mover(Este)

}

procedure IrBorde () {
  IrAlBorde (Oeste)
}

procedure DibujarLineaColorida () {
  Poner4Colores ()
  MoverEste ()
  Poner4Colores ()
  MoverEste ()
  Poner4Colores ()
  MoverEste ()
  Poner4Colores ()
```

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!

```
  Mover(Este)

}

procedure IrBorde () {
  IrAlBorde (Oeste)
}

procedure DibujarLineaColorida () {
  Poner4Colores ()
  MoverEste ()
  Poner4Colores ()
  MoverEste ()
  Poner4Colores ()
  MoverEste ()
  Poner4Colores ()
  IrBorde ()


}
program {
  DibujarLineaColorida ()
}
```

Repetición simple

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

¡Dame una pista!

```
procedure MoverOeste10 () {


  Mover(Oeste)
  Mover(Oeste)
  Mover(Oeste)
  Mover(Oeste)
  Mover(Oeste)
  Mover(Oeste)
  Mover(Oeste)
  Mover(Oeste)
  Mover(Oeste)
  Mover(Oeste)

  }
```

Como esto es molesto, hay un comando que repite por nosotros las veces que queramos

repeat

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`:

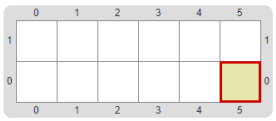
```
procedure MoverOeste10() {
  repeat(10) {
    Mover(Oeste)
  }
}
```

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

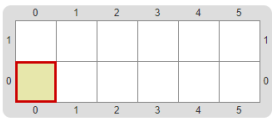
```
...escribí tu solución acá...
```

▶ Enviar

Tablero inicial



Tablero final



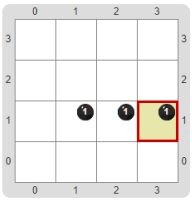
Como ya descubriste, el comando `repeat` consta básicamente de dos elementos:

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 😠).

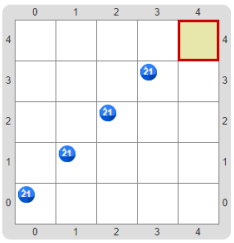
Es muy común, al principio, olvidarse de colocar las llaves o incluso pensar que no son importantes. Pero tené mucho cuidado: poner las llaves en el lugar erróneo puede cambiar por completo lo que hace tu programa. Mirá qué distinto sería el resultado si hubieras puesto el `Mover(Este)` adentro del `repeat`:

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

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



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:



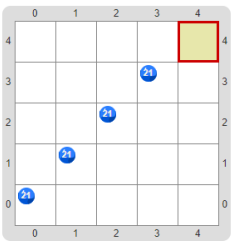
```
procedure DiagonalPesada4Azul () {

repeat (4) {

  repeat (21) {
    Poner(Azul)
  }
  Mover(Este)
  Mover(Norte)
  }
}
```

▶ Enviar

Otra forma de hacerlo

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:



Definí un procedimiento `DiagonalPesada4Azul` que resuelva el problema.

```
procedure Poner21 () {

  repeat(21) {
    Poner(Azul)
  }
   }

procedure DiagonalPesada4Azul () {
  repeat(4) {
      Poner21 ()
    Mover(Este)
    Mover(Norte)
  }

}
```

▶ Enviar

Caso borde

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).

Podemos pensar a un cuadrado de 4x4 como cuatro líneas de longitud 4, una arriba de la otra. Con el procedimiento que ya definiste tenés el problema de dibujar una línea resuelto, aunque el cabezal queda en una posición poco conveniente para dibujar la siguiente. ¿Qué deberíamos hacer después de dibujar cada línea?

Solución | </> Biblioteca

```
procedure CuadradoNegro4 () {
  repeat (3) {
    LineaNegra4Este ()
  Mover(Norte)
    Mover(Oeste)
    Mover(Oeste)
    Mover(Oeste)

  }
   LineaNegra4Este ()
}
```

Parámetros

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`.

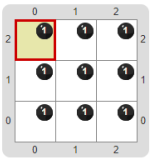
En la Biblioteca vas a encontrar el procedimiento `VolverAtras`. ¡Eso significa que podés invocarlo sin tener que definirlo!

Solución | </> Biblioteca

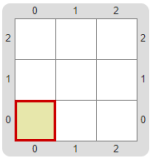
```
procedure DibujarLineaNegra3 () {

  Poner(Negro)

  repeat (2) {
    Mover(Este)
    Poner(Negro)

  }
   VolverAtras ()
}
program {
DibujarLineaNegra3 ()
}
```

```
procedure VolverAtras() {
    Mover(Oeste)
    Mover(Oeste)
}
```

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



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



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

```
procedure DibujarCuadradoNegroDeLado3 () {
  DibujarLineaNegra3 ()
  repeat (2) {
  Mover(Norte)
  DibujarLineaNegra3 ()
}
 }

program {
  DibujarCuadradoNegroDeLado3 ()
}
```

▶ Enviar

Capturas de pantalla guardadas
La captura de pantalla se agregó a tu OneDrive.
OneDrive

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!

Solución </> Biblioteca

```
procedure DibujarLineaVerde3 () {
  Poner(Verde)
  repeat (2) {


  Mover(Este)
  Poner(Verde)
}
  VolverAtras ()
}
program {
  DibujarLineaVerde3 ()
}
```

Solo cambia el color, se vuelve repetitivo y aburrido, para solucionar

¡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 tu solución acá...
```

▶ Enviar

¡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. 👀

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

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

▶ Enviar

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

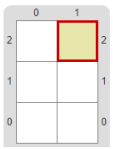
```
program {
  Poner3(Verde)
}
```

Solución | </> Biblioteca

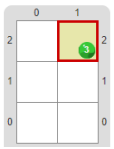
```
procedure Poner3(color) {
  Poner(color)
  Poner(color)
  Poner(color)
}
```

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial



Tablero final



¡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. Despreocupate por los `program`s para invocarlo con cada uno de los colores, van por nuestra parte. 😉

¡Dame una pista!

¡Ojo! 👀 `DibujarLinea3` tiene que servir para **cualquier** color como hicimos con `Poner3`.

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

Solución | </> Biblioteca

```
procedure DibujarLinea3(color) {
  repeat(2) {
    Poner(color)
    Mover(Este)
  }
  Poner(color)

  VolverAtras()
}
```

▶ Enviar

¡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!

Recordá que `DibujarLinea3` recibe un parámetro.

Solución | </> Biblioteca

```
procedure DibujarCuadradoDeLado3(color) {

  DibujarLinea3(color)

  repeat (2) {
    Mover(Norte)
    DibujarLinea3(color)
  }

}
```

Distintos 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)
}
```

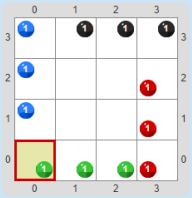
¡Terminaste Parámetros!

¡Felicitaciones! 🎉

En esta lección aprendimos como los **parámetros** facilitan nuestra tarea permitiéndonos crear procedimientos más genéricos. Cuidado al invocar estos procedimientos, ¡acordate que la cantidad de **argumentos** y su orden importa! Los nombres también pero para las personas, no para las computadoras. 😉

Siguiente Lección: Práctica Repetición simple ❯

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!

Prestá atención a la cantidad de parámetros que recibe el procedimiento y en qué orden. Y no te olvides de separarlos con comas. 😉

https://www.youtube.com/watch?v=jiLKeZISFx0

Prestá atención a la cantidad de parámetros que recibe el procedimiento y en qué orden. Y no te olvides de separarlos con comas. 😉

Solución | Biblioteca

```
program {
  DibujarLinea3(Verde,Este)
  Mover(Este)
 DibujarLinea3(Rojo,Norte)
  Mover(Norte)
  DibujarLinea3(Negro,Oeste)
  Mover(Oeste)
  DibujarLinea3(Azul,Sur)
  Mover(Sur)
}
```

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

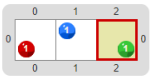
Entonces es importante recordar que al invocar un procedimiento **no debemos**:

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

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:



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

```
procedure Triada(color1,color2,color3) {

 Poner(color1)
  Mover(Este)
  Poner(color2)
  Mover(Este)
  Poner(color3)
  }
```

▶ Enviar

Practica de repetición simple

Ahora que ya sabés cómo repetir tareas, vamos a combinar eso con procedimientos y parámetros para solucionar problemas más complejos. 😉

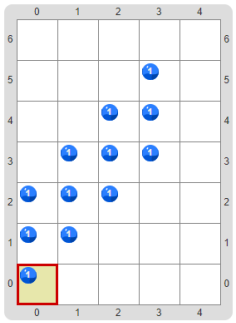
Además, vamos a ir introduciendo algunos temas nuevos en el camino... ¡No te los pierdas!

Objetivos

- Vamos a aprender qué son las **expresiones** más sencillas (¿sabías que `Azul` es una expresión?)
- Vamos a empezar a **reutilizar** el código que ya escribimos. Recordá lo que dijo un sabio: *"Cada vez que alguien repite código, se muere un gatito"* 🐱
- Profundizaremos el **uso de parámetros**.
- Vamos a encontrar **patrones que se repiten** y aprenderemos a utilizarlos.
- Comenzaremos a usar bolitas para **representar información**. Es decir, vamos a darle un pequeño **dominio** a nuestros problemas, usando las bolitas para representar otra cosa: personas, autos, flores, ¡lo que se te ocurra!

Sigamos probando tus habilidades para reutilizar...

Ahora, tenés que hacer este dibujo:



Solución | Biblioteca

```
procedure BandaDiagonal4 () {
  repeat (3) {
  Diagonal4AzulVolviendo ()
  Mover(Norte)
}
    repeat (3) {
    Mover(Sur)

}
    }
```

Enviar

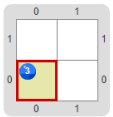
> El procedimiento debe llamarse `BandaDiagonal4`. ¡Ojo! prestá atención a la **posición final** del cabezal.

Poner N

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

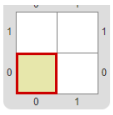
Necesitamos un procedimiento que nos ayude a poner muchas bolitas. Sí, podríamos simplemente usar un `repeat` para lograrlo, pero como es una tarea re-común que vamos a hacer un montón de veces, vamos a preferir definir un `procedure` llamado `PonerN`. Nuestro procedimiento debe poner la cantidad de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

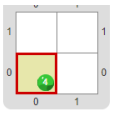
Por ejemplo, `PonerN(3, Azul)` haría esto:



```
procedure PonerN(cantidad, color) {
  repeat (cantidad) {
  Poner(color)
    }
  }

```

Enviar





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

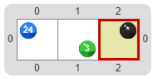
- **reutilización de código:** como poner muchas bolitas es una tarea común, está bueno tener un procedimiento que lo resuelva: lo escribimos una vez y lo usamos para siempre;
- **declaratividad:** cuando tengamos que resolver un problema más complejo, tener este procedimiento nos va a ayudar a pensar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

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 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.



Solución | Biblioteca

```
procedure DiaDeLaMemoria () {
  PonerN (24, Azul)
  Mover(Este)
  PonerN(3,Verde)
  Mover(Este)
  PonerN (1976,Negro)
}
```

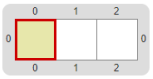
Enviar

Capturas de pantalla guardadas
La captura de pantalla se agregó a tu OneDrive.

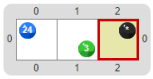
Ya que definimos `PonerN(cantidad, color)`, ahora podemos **invocarlo**, ¿no?.

¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial



Tablero final



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

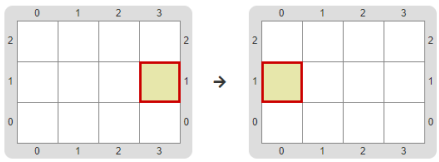
Cuando **usamos** un procedimiento que tiene parámetros como PonerN, `PonerN(56, Rojo)` tenemos que enviarle valores como argumento. ¡Y las expresiones sirven para eso!

Mover N

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

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

Inicial → Final



```
procedure MoverN (cantidad, direccion ) {
  repeat(cantidad) {
  Mover(direccion)
}
  }
```

Enviar

¡Dame una pista!

¡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.

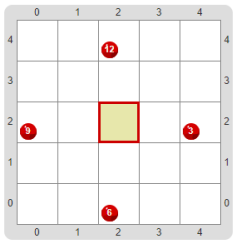
Dado el siguiente program:

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

Solución | Biblioteca

```
procedure Mover3 (radio) {
  MoverN(radio,Este)
  PonerN(3,Rojo)
   MoverN(radio,Oeste)

}

  procedure Mover9 (radio) {
    MoverN(radio,Oeste)
  PonerN(9,Rojo)
    MoverN(radio,Este)
}
 procedure Mover12 (radio) {
    MoverN(radio,Norte)
  PonerN(12,Rojo)
    MoverN(radio,Sur)
}
 procedure Mover6 (radio) {
    MoverN(radio,Sur)
  PonerN(6,Rojo)
    MoverN(radio,Norte)
}

```

El reloj resultante es así:



¡Dame una pista!

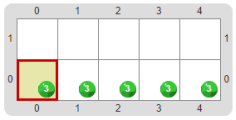
```
23
24 procedure DibujarReloj(radio) {
25   Mover3 (radio)
26   Mover9 (radio)
27   Mover12 (radio)
28   Mover6 (radio)
29 }
30
31
```

Enviar

Ejemplo con 3 parámetros: cantidad, color y longitud

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

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

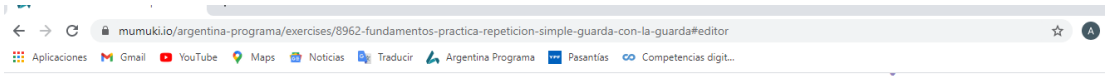


Definí el procedimiento `LineaEstePesada(peso, color, longitud)`. Tené en cuenta que el cabezal **debe regresar a la posición inicial**. Para eso vas a tener que invocar `MoverN`.

Solución | Biblioteca

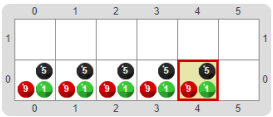
```
1  procedure LineaEstePesada(peso,color,longitud) {
2
3   repeat (longitud) {
4
5    PonerN(peso, color)
6    Mover(Este)
7         }
8
9    MoverN(longitud,Oeste)
10
11 }
```

Enviar



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.



¡Dame una pista!

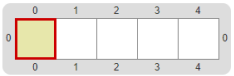
No te olvides de **dividir en subtareas** y además de considerar **el caso borde**.

¡A **no repetir código**! Cada azulejo puede resolverse con un procedimiento... 😉

Solución | Biblioteca

```
1  procedure MoverNegro () {
2    PonerN(5,Negro)
3  }
4  procedure MoverRojo () {
5    PonerN(9,Rojo)
6  }
7  procedure MoverVerde () {
8    PonerN(1,Verde)
9  }
10 procedure MoverTodos () {
11     MoverNegro ()
12     MoverRojo ()
13     MoverVerde ()
14   }
15  procedure GuardaDe5() {
16      MoverTodos ()
17      repeat(4) {
18      Mover(Este)
19      MoverTodos ()
20    }
21   IrAlBorde (Este)
22 }
23
```

Tablero inicial

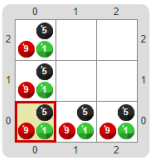


Tablero final

¡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!

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



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

¡Dame una pista!

Para dividir en *subtareas* vas a tener que pensar si podés partir el problema principal de dibujar una L en problemas más chicos, y cada una de ellas será un procedimiento nuevo que deberás usar desde `GuardaEnL`.

Solución | Biblioteca

```
procedure Horizontal () {
  PonerAzulejo()
  repeat (2) {
    Mover(Este)
      PonerAzulejo()
  }
repeat (2) {
Mover(Oeste)
  }
}
procedure Vertical () {
  repeat (2) {
    Mover(Norte)
    PonerAzulejo()
  }
repeat (2) {
Mover(Sur)
}
  }
  procedure GuardaEnL () {
    Horizontal ()
    Vertical ()
  }
```

Enviar

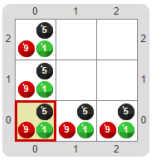
Tu solución funcionó, pero hay cosas que mejorar

Objetivos que no se cumplieron:

- **GuardaEnL** debe utilizar **MoverN**
- La solución parece tener un error de tipeo: debe usar **MoverN**, pero usa **Mover**. ¿Quizás quisiste decir **MoverN**?

Tablero inicial Tablero final

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



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

¡Dame una pista!

Para dividir en *subtareas* vas a tener que pensar si podés partir el problema principal de dibujar una L en problemas más chicos, y cada una de ellas será un procedimiento nuevo que deberás usar desde `GuardaEnL`.

```
procedure Horizontal () {
  PonerAzulejo()
  repeat (2) {
    Mover(Este)
      PonerAzulejo()
  }
MoverN (2,Oeste)
  }

procedure Vertical () {
  repeat (2) {
    Mover(Norte)
    PonerAzulejo()
  }
MoverN (2,Sur)
  }

  procedure GuardaEnL () {
    Horizontal ()
    Vertical ()
  }
```

Enviar

¡Terminaste Práctica Repetición simple!

¡Muy bien!

En esta guía:

- **Reusaste** procedimientos definidos anteriormente.
- Usaste **repetición**.
- Consideraste los **casos borde**.
- Hiciste un repaso grande de **parámetros**.
- ¡Evitaste **repetir código**! 😻
- Empezaste a usar las bolitas para **representar varios dominios**: Fechas, Relojes y Guardas.
- Aprendiste que los *literales* y los *parámetros* son **expresiones** que se pueden pasar por argumento a los procedimientos.

Expresiones

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.

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!

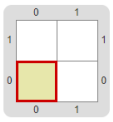
¿Y cómo se escribe una suma o una resta en Gobstones?
Igual que en la vida real.

Solución | Biblioteca

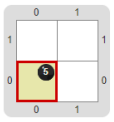
```
program {
  PonerN(3+2,Negro)

}
```

Enviar

Tablero inicial



Tablero final



Algunas variantes válidas:

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

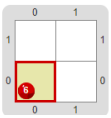
```
program {
  PonerN(12 - 7, Negro)
}
```

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).

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).



Solución | Biblioteca

```
procedure PonerSuma(x,y) {
  PonerN(x,Rojo)
  PonerN(y,Rojo)
}
```

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**.

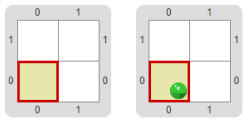
> Definí el procedimiento `ContarGente(micros, autos, bicicletas)` que a partir de la cantida de micros, autos y bicicletas que recibe como parámetro, haga las cuentas necesarias y refleje el resultado con bolitas de color verde.

Te dejamos un par de ejemplos que te pueden ayudar:

Solución | Biblioteca

```
procedure ContarGente(micros,autos,bicicletas) {
  PonerN (micros*40+autos*4+bicicletas,Verde)

}
```

Enviar



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`.

Opuesto

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...

Div

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 pidiéramos que diera 2 pasos, terminaría dando 1; si le pidiéramos 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`.

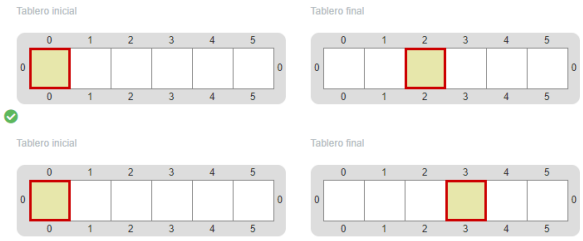
Definí el procedimiento `CaminarDesconfiado(pasos)` que simule el caminar de Carlos: debe recibir la cantidad de pasos que debería dar y dar la mitad. Siempre se mueve al `Este`.

Solución | Biblioteca

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

Enviar

Resultados de las pruebas:



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 la bolita al lado sin que se mueva el cabezal

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).

Inicial **Final**

```
procedure PonerAl(direccion,color) {
  Mover(direccion)
  Poner(color)
  Mover(opuesto(direccion))

}
```

Enviar

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, modificá el procedimiento `Linea` para que el cabezal quede en el lugar donde empezó.

¡Dame una pista!

`MoverN` y `opuesto` parecieran ser buenos aliados para el problema que tenés que resolver.

Solución Biblioteca

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

Enviar

Siguiente y previo

> 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.

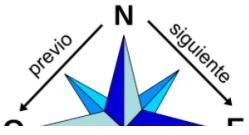
¡Dame una pista!

Solución Biblioteca

```
procedure Ele(direccion) {
  Linea(direccion, Azul, 3)

  Linea(siguiente(direccion), Azul, 3)


}
```

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.

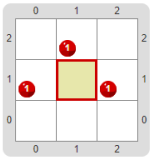
La primera línea es fácil, porque coincide con la dirección que recibimos por parámetro... ¿pero la segunda? Bueno, ahí viene lo interesante: además de `opuesto`, Gobstones nos provee dos funciones más para operar sobre las direcciones, `siguiente` y `previo`. `siguiente(direccion)` retorna la dirección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



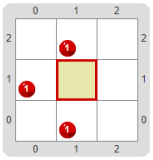
Ya vimos distintas funciones que a partir de una dirección nos permiten obtener otra distintas.

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)



Flecha(Oeste)



```
procedure Flecha(direccion) {
  Mover(direccion)
  Poner(Rojo)
  Mover(opuesto(direccion))
  Mover(previo(direccion))
  Poner(Rojo)
  Mover(siguiente(direccion))
  Mover(siguiente(direccion))
  Poner(Rojo)
  Mover(previo(direccion))
}
```

Enviar

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

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

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!

- `SacarTodas(Rojo)` produciría esto:

Solución | Biblioteca

```
procedure SacarTodas(color) {
  SacarN(nroBolitas(color), color)
}
```

Enviar

¡Terminaste Expresiones!

Repasemos todo lo que aprendiste en esta guía:

Además de los **literales** que ya venías usando, existen también otro tipo de **expresiones** que realizan algún tipo de operación y pueden depender de dos cosas: **de sus parámetros** y del **estado del tablero**.

En el camino, conociste unas cuantas expresiones nuevas:

- **Aritméticas:** `+`, `-`, `*`, `div`.
- **De direcciones:** `opuesto`, `siguiente`, `previo`.
- **Numéricas:** `nroBolitas`.

Y de yapa, también te adentraste un poquito en el arte de los programas que **hacen cosas distintas** según el tablero que haya.

Alternativa Condicional

Alternativa Condicional

Hasta ahora, todos los programas y procedimientos que hicimos fueron sobre tableros **conocidos**, sabíamos exactamente de qué tamaño era el tablero, dónde estaba el cabezal y cuántas bolitas había en cada celda.

Nos introduciremos ahora en el mundo de lo **desconocido**, donde la programación cobra aún mucho más sentido. Con la herramienta que veremos podremos resolver problemas nuevos y evitar errores que hasta ahora no podíamos: sacar una bolita Roja *sólo si hay alguna*, movernos al Norte *si eso no provoca que nos caigamos del tablero* o agregar una bolita Azul *sólo si ya hay una Verde*.

¿Y cómo haremos esto? Utilizando la capacidad de la computadora para **decidir**: la **alternativa condicional** o **sentencia if**.

If

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)
  }
}
```

> Copiá el código anterior en el editor y apretá *Enviar*.

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

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 ejercitar esto ultimo, 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
2
3
4
5 procedure CompletarCelda () {
6   if(hayBolitas(Negro)){
7   Poner(Rojo)
8   Poner(Azul)
9   Poner(Verde)
10
11 }
12   }
```

▶ Enviar

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:

```
1 program {
2
3
4 if (puedeMover(Este)) {
5
6   Mover(Este)
7  }
8  }
```

▶ Enviar

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**.

```
program {
  if (nroBolitas(Negro)<=5) {
  Poner(Negro)
}
  }
```

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)`

```
procedure AsegurarUnaBolitaVerde() {
 if ( not hayBolitas(Verde)) {
  Poner(Verde)
}
}
```

▶ Enviar

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`

```
...escribí tu solución acá...
```

▶ Enviar

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.

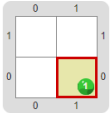
> Definí un procedimiento `AsegurarUnaBolitaVerde()` que se asegure que en la celda actual hay al menos una bolita verde. Esto es: si ya hay bolitas verdes no hay que hacer nada, pero si **no** hay tendría que poner una.

✅ **¡Muy bien! Tu solución pasó todas las pruebas**

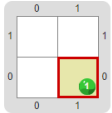
Resultados de las pruebas:

✅ Si hay bolitas verdes, no hace nada

Tablero inicial

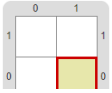


Tablero final

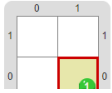


✅ Si no hay bolitas verdes, agrega una

Tablero inicial



Tablero final



Else

- 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.

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.

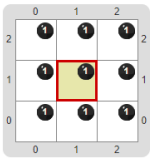
¡Dame una pista!

```
procedure PrenderOApagarLuz() {

  if (hayBolitas(Verde)) {
    Poner(Negro)
    Sacar(Verde)


    } else  {
     Poner(Verde)
    Sacar(Negro)


    }
}
```

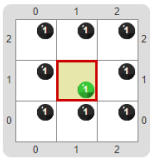
Resultados de las pruebas:

Si la celda está apagada, la prende

Tablero inicial

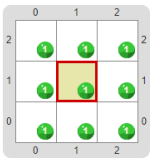


Tablero final

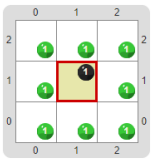


Si la celda está prendida, la apaga

Tablero inicial



Tablero final



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!

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

```
procedure DesordenarCelda () {
  if (hayBolitas(Azul)) {
  Poner(Verde)
  Sacar(Azul)
}
  if(hayBolitas(Rojo)) {
    Poner(nroBolitas*2(Rojo)) {
  }

    if(hayBolitas(Negro)) {
      Sacar(Negro)
    }
    }
```

¡Terminaste Alternativa Condicional!

Ahora conocés una de las herramientas más poderosas que tiene la computación: la **alternativa condicional**. Aunque parezca algo sencillo, esto es lo que permite que los programas puedan reaccionar diferente ante distintos estímulos, dando lugar así a que las computadoras puedan **tomar decisiones**.

Algunos ejemplos de esto:

- en las redes sociales, probablemente exista un `if` que determine si podés ver el perfil de alguien o no;
- cuando te tomás un colectivo y pagás con la SUBE (o tarjetas similares), la máquina decide si podés viajar o no dependiendo de si te alcanza el saldo.

Te dejamos como ejercicio pensar (y por qué no intentar escribirlas) qué partes de los sistemas con los que interactuás todos los días parecerían estar resueltas con un `if`.

Funciones

Funciones

Cuando introdujimos la noción de **procedimientos**, dijimos que:

- son una forma de **darle nombre** a un grupo de comandos, logrando así que nuestros programas fueran más entendibles;
- nos posiblitan la **división en subtareas**: para resolver un problema grande, basta con dividirlo en problemas más chicos y luego combinarlos;
- nos ayudan a **no repetir código**, no volver a escribir lo mismo muchas veces.

Al trabajar con expresiones complejas, rápidamente surge la necesidad de contar con un mecanismo similar, por los motivos que acabamos de esbozar.

¿Querés saber cuál es ese mecanismo? ¡Empecemos! 👊

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))
}
```

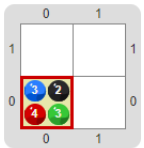
> Pegá el código anterior en el editor y observá el resultado.

```
function nroBolitasTotal() {
  return (nroBolitas(Azul) + nroBolitas(Negro) + nroBolitas(Rojo) + nroBolitas(Verde))
}
```

Resultados de las pruebas:

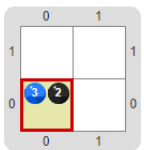
✅ nroBolitasTotal() -> 12

Tablero inicial



✅ nroBolitasTotal() -> 5

Tablero inicial



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.

```
procedure MoverSegunBolitas() {
if (nroBolitasTotal()>10)
  {
    Mover(Este)
  } else {
    Mover(Norte)
  }
}
```

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!

Ya definimos una función para contar todas las bolitas (`nroBolitasTotal()`) y Gobstones ya trae una para contar las de un color en particular (`nroBolitas(color)`).

Sólo te queda pensar cómo combinarlas. 😉

```
function todasExcepto(color) {
  return ( nroBolitasTotal () - nroBolitas(color))
}
```

Enviar

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:

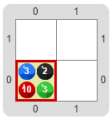
`rojoEsDominante()` **retorna** `False` (hay 2 bolitas rojas contra 8 de otros colores). Pero en este otro:

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

Solución Biblioteca

```
function rojoEsDominante() {
  return (nroBolitas(Rojo) > todasExcepto(Rojo))
}
```

Enviar



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.

Booleanos y &&

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 comunmente se denomina **conjunción** y se lo suele representar con el símbolo ∧.

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()
```

esLibreCostados

> Definí la función `esLibreCostados()` que indique si el cabezal puede moverse tanto al Este como al Oeste.

¡Dame una pista!

¡No te olvides de que existe una función `puedeMover(direccion)`! 😉

```
function esLibreCostados () {
  return (puedeMover(Este) && puedeMover(Oeste))
}
```

||

> 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 ∨.

¡Dame una pista!

Recordá que existe la función `hayBolitas(color)`, que indica si hay alguna bolita del color especificado. Además, el operador `||` se puede usar varias veces, como si fuera una suma: `unaCosa || otraCosa || otraCosaMas`.

```
function hayAlgunaBolita() {
 return ( hayBolitas(Rojo) || hayBolitas(Azul) || hayBolitas(Negro) || hayBolitas(Verde))
}
```

hayAlgunaBolita

Tanto `&&` como `||` pueden usarse varias veces sin la necesidad de usar paréntesis, siempre y cuando tengan expresiones booleanas a ambos lados.

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.

estoyEnUnBorde

Si el cabezal está en un borde, **no** se **puede mover** en **alguna** dirección. 😉

```
function estoyEnUnBorde () {
  return (not puedeMover(Este) || not puedeMover(Oeste))
  }
```

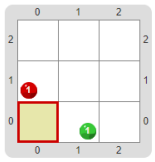
Vamos a ver ahora funciones que **hacen cosas antes** de retornar un resultado. Para ejemplificar esto, vamos a querer que definas una función que nos diga si hay una bolita de un color específico, pero **en la celda de al lado**.

> 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!

```
function hayBolitasAl (direccion, color) {
 Mover(direccion)

  return
   (hayBolitas(color))

}
```



¿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**.

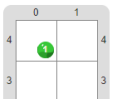
hayBolitasLejosAl

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

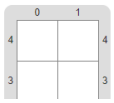
Te toca programar una nueva versión de `hayBolitasAl` que mire si hay bolitas a cierta distancia de la celda actual. A esta función la vamos a llamar `hayBolitasLejosAl` y recibirá tres parámetros: una **dirección** hacia donde deberá moverse, un **color** por el cual preguntar y una **distancia** que 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*:



Y para este tablero devolvería *False*:



Solución | Biblioteca

```
function hayBolitasLejosAl(direccion,color,distancia) {
  MoverN(distancia,direccion)

  return (hayBolitas(color))


}
```

Enviar

estoyRodeadoDe

> 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

```
function estoyRodeadoDe(color) {

  return (hayBolitasAl(Norte, color) &&
hayBolitasAl(Sur, color) && hayBolitasAl(Este, color)
&& hayBolitasAl(Oeste, color))
}
```

hayLimite

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`.

¡Dame una pista!

Cuidado con el **orden** de las expresiones: para poder preguntar si **está rodeado**, primero deberías chequear si **está en un borde**.

Solución Biblioteca

```
function hayLimite() {

  return (estoyEnUnBorde() || estoyRodeadoDe(Rojo))
}
```

Envi

Capturas de pantalla guardac
La captura de pantalla se agre
tu OneDrive

¡Terminaste Funciones!

Esta lección fue bastante intensa, aprendiste unas cuantas cosas:

- conociste las **funciones**, que son una buena forma de **nombrar expresiones compuestas**;
- también vimos que se pueden usar funciones para calcular cosas que necesitan **salir de la celda actual** y que el **efecto desaparece** una vez que la función se ejecuta;
- ejercitamos los **conectivos lógicos** `||` y `&&`, y vimos que ambos funcionan con **cortocircuito**.

¡Sigamos programando! 💪