Muchas formas de decir lo mismo

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

- el número 5;
- la cantidad de dedos de una mano;
- la suma entre 3 y 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!

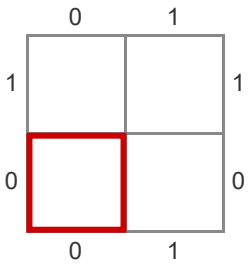
Solución | Biblioteca

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

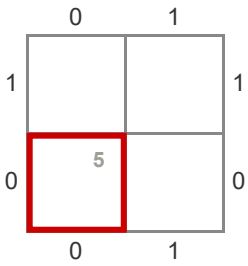
Enviar

¡Muy bien! Tu solución pasó todas las pruebas

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).

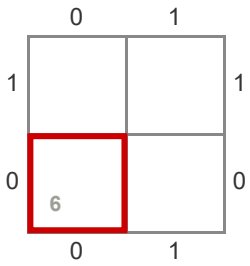
La suma de las partes

Juguemos un poco más con esto de hacer cuentas.

2. La suma de las partes

> 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).



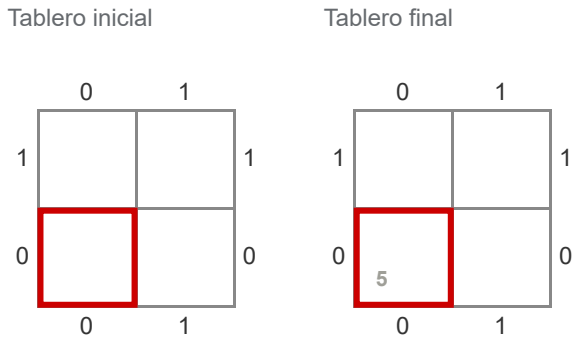
Solución | Biblioteca

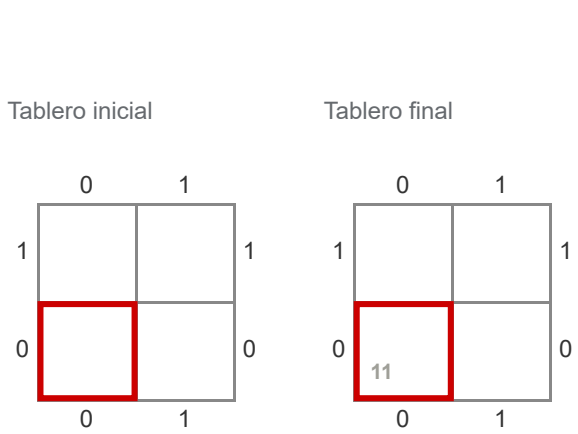
```
1 procedure PonerSuma(x, y) {
2   PonerN(x+y, Rojo)
3 }
4
5
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:





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

¿Qué se hace antes?

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

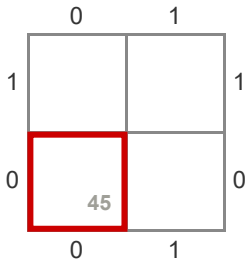
Contamos con la información de cuántos micros, autos y bicicletas asistieron y desde allí podemos hacer un cálculo siguiendo estas reglas:

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

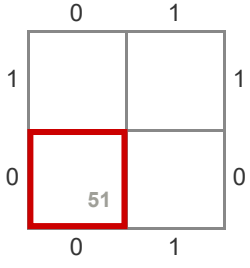
> Definí el procedimiento `ContarGente(micros, autos, bicicletas)` que a partir de la cantida de micros, autos y bicicletas que recibe como parámetro, haga las cuentas necesarias y refleje el resultado con bolitas de color verde.

Te dejamos un par de ejemplos que te pueden ayudar:

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



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



¡Dame una pista!

Solución | Biblioteca

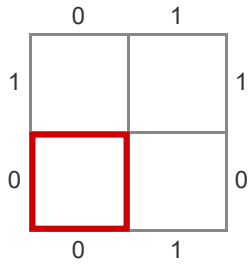
```
1 procedure ContarGente(micros, autos, bicicletas) {
2   PonerN(micros*40+autos*4+bicicletas,Verde)
3 }
4
5
```

Enviar

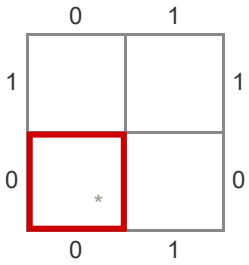
¡Muy bien! Tu solución pasó todas las pruebas

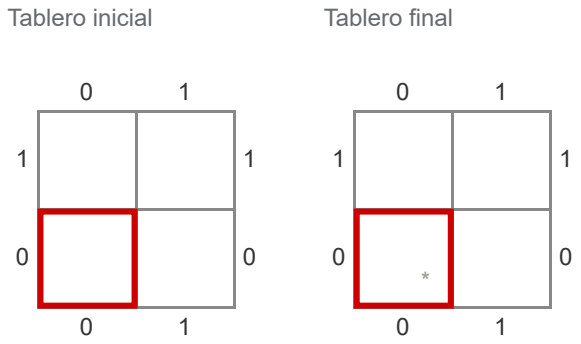
Resultados de las pruebas:

Tablero inicial



Tablero final





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`.

La carrera del salmón

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

4. La carrera del salmón

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

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

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

Sabiendo esto, podríamos definir fácilmente el procedimiento que queríamos:

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

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

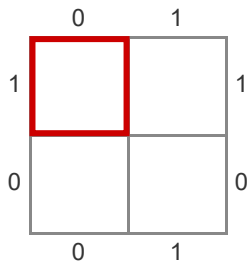
```
1 procedure MoverComoSalmon(direccion) {
2   Mover(opuesto(direccion))
3 }
4
5
```

Enviar

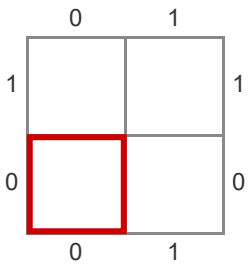
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

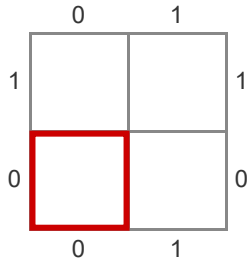
Tablero inicial



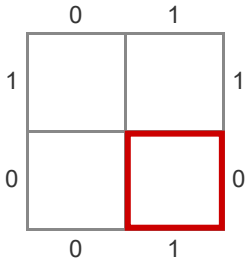
Tablero final



Tablero inicial



Tablero final



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

Dos pasos adelante, un paso atrás

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

5. Dos pasos adelante, un paso 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 recorrer la mitad. Siempre se mueve al `Este`.

¡Dame una pista!

Solución | Biblioteca

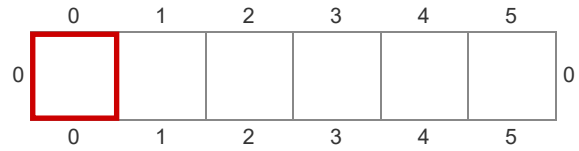
```
1 procedure CaminarDesconfiado(pasos) {
2   MoverN(pasos div 2, Este)
3 }
4
5
```

Enviar

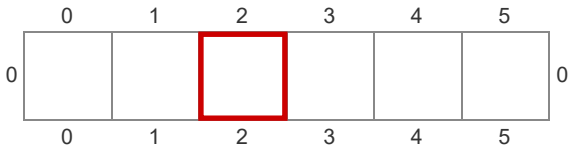
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

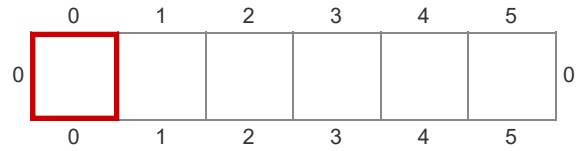
Tablero inicial



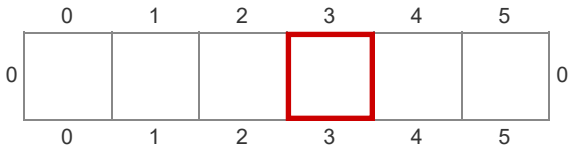
Tablero final



Tablero inicial



Tablero final



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

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

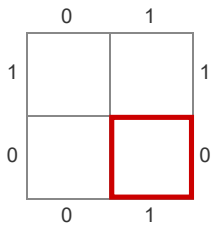
Poner al lado

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

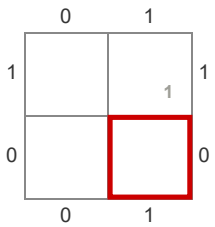
6. Poner al lado

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



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

¡Dame una pista!

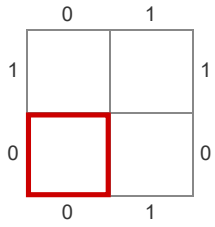
```
procedure PonerAl(direccion, color) {
  Mover(direccion)
  Poner(color)
  Mover(opuesto(direccion))
}

```

Enviar

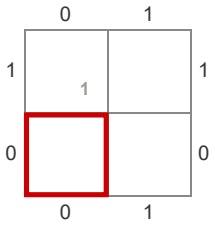
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

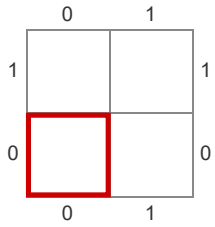
Tablero inicial



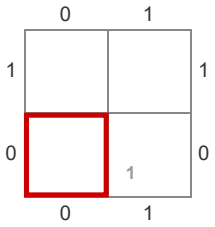
Tablero final



Tablero inicial



Tablero final



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

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

La línea que vuelve

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

La versión que sabíamos hacer hasta ahora era esta:

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

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

¡Dame una pista!

Solución | Biblioteca

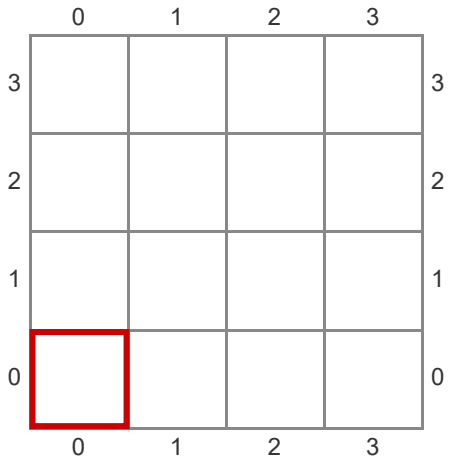
```
1 procedure Linea(direccion, color, longitud) {
2   repeat(longitud) {
3     Poner(color)
4     Mover(direccion)
5   }
6   MoverN(longitud,opuesto(direccion))
7 }
8
9
```

Enviar

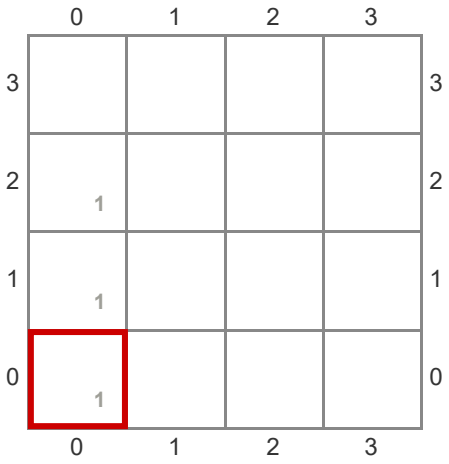
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

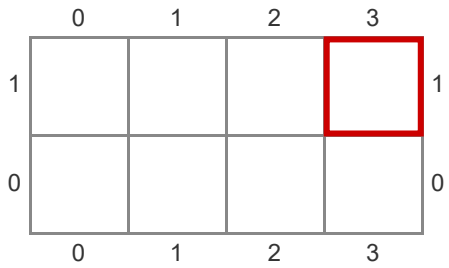
Tablero inicial



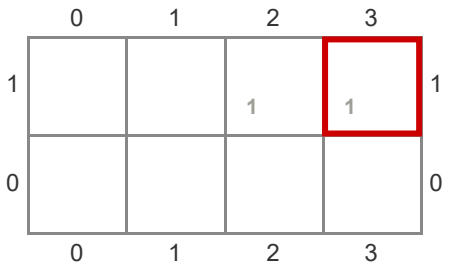
Tablero final



Tablero inicial



Tablero final



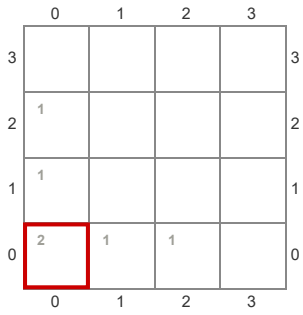
Términos y Condiciones

Reglas del Espacio de Consultas

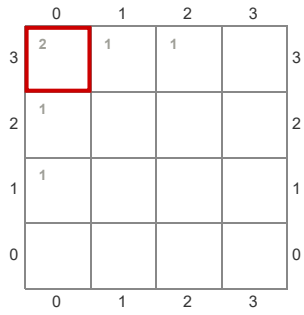
Dibujando una L

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

Ele(Norte)

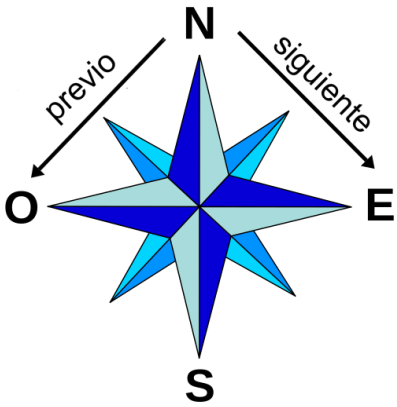


Ele(Este)



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 como argumento... ¿pero la segunda? Bueno, ahí viene lo interesante: además de `opuesto`, Gobstones nos provee dos funciones más para operar sobre las direcciones, `siguiente` y `previo`. `siguiente(direccion)` retorna la dirección siguiente a la especificada, mientras que `previo(direccion)` retorna la anterior, siempre pensándolo en el sentido de las agujas del reloj:



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

¡Dame una pista!

Solución | Biblioteca

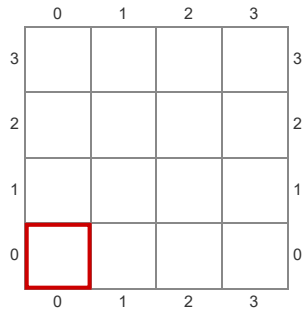
```
procedure Ele(direccion) {
  Linea(direccion, Azul, 3)
  Linea(siguiente(direccion), Azul ,3)
}
```

Enviar

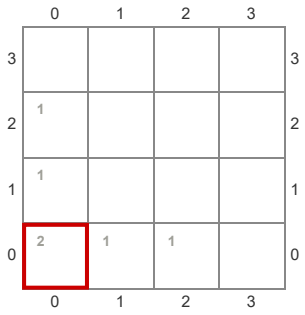
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

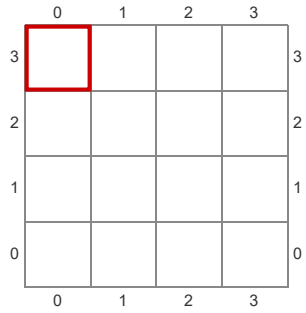
Tablero inicial



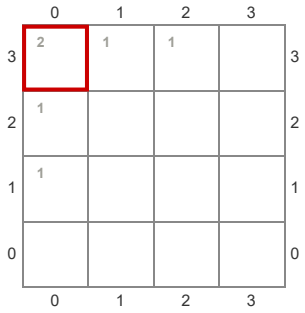
Tablero final



Tablero inicial



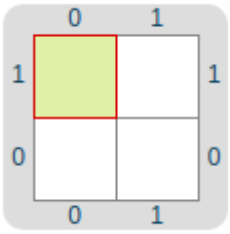
Tablero final



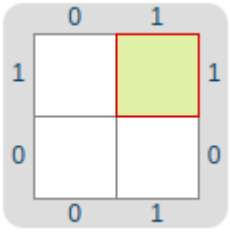
Previo a lo siguiente

Ya te presentamos las funciones `opuesto`, `previo` y `siguiente` que nos permiten desplazarnos en forma relativa a alguna dirección . Antes de seguir utilizándolas, vamos a conocerlas un poco mejor.

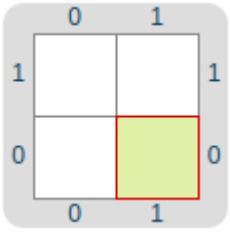
Si partimos de este tablero inicial:



Y luego ejecutamos `Mover(siguiente(Norte))` el tablero obtenido será así:



Porque `siguiente(Norte)` es `Este` . Si luego ejecutamos `Mover(previo(Oeste))` lograremos el siguiente tablero:



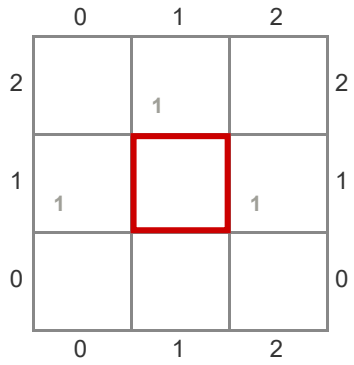
Porque `previo(Oeste)` es `Sur` . ¡Veamos si se entendió en el siguiente ejercicio!

Siga la flecha

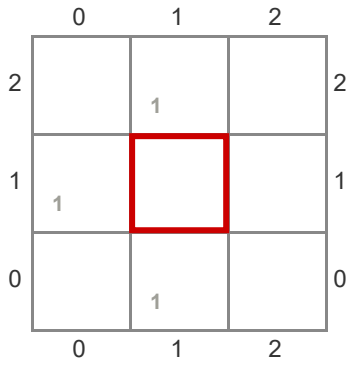
Ya vimos distintas funciones que a partir de una dirección nos permiten obtener otra distintas.

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

Flecha(Norte)



Flecha(Oeste)



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

¡Dame una pista!

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

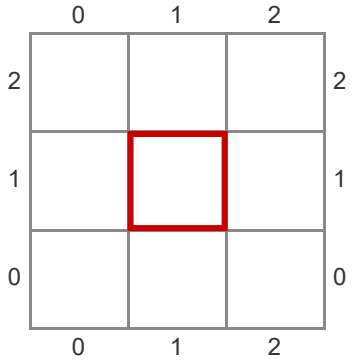
```
12
13
```

Enviar

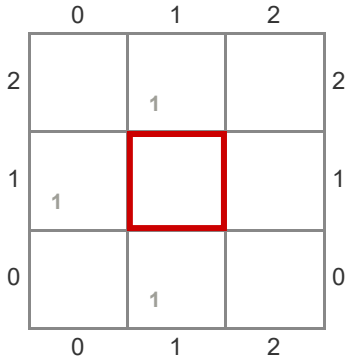
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

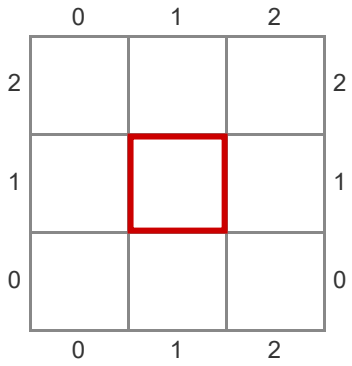
Tablero inicial



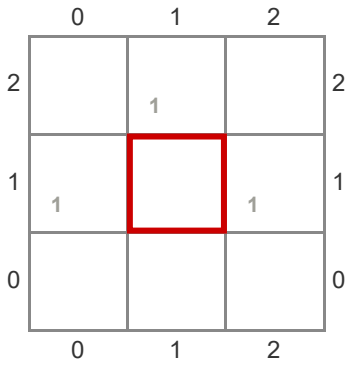
Tablero final



Tablero inicial



Tablero final



Información importante

Términos y Condiciones

Reglas del Espacio de Consultas

Copiando bolitas

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

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

Miremos algunos ejemplos:

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

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

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

¡Dame una pista!

Solución | Biblioteca

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

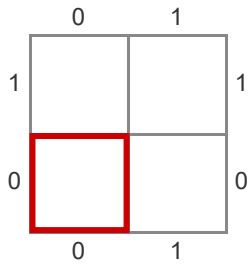
Enviar

¡Muy bien! Tu solución pasó todas las pruebas

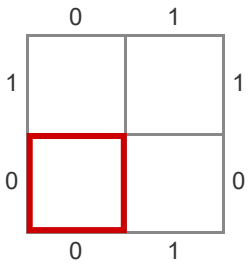
Resultados de las pruebas:

Si no hay verdes, no hace nada

Tablero inicial

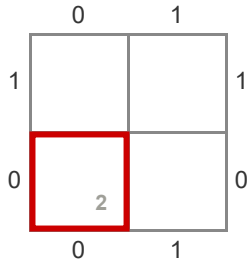


Tablero final

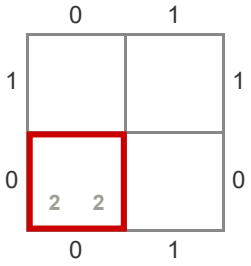


Si hay 2 verdes, pone 2 rojas

Tablero inicial

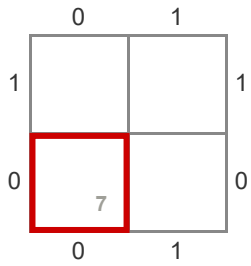


Tablero final

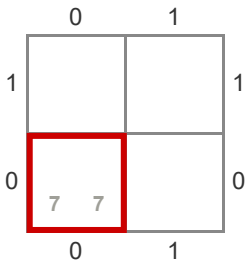


Si hay 7 verdes, pone 7 rojas

Tablero inicial



Tablero final



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

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

Sacando bolitas

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

Pensemos las subtareas necesarias:

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

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

¡Dame una pista!

Solución | Biblioteca

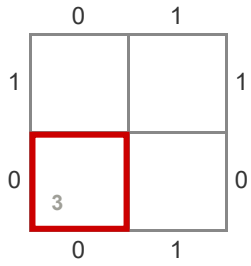
```
1 procedure SacarTodas(color) {
2   SacarN(nroBolitas(color),color)
3 }
4
5
```

Enviar

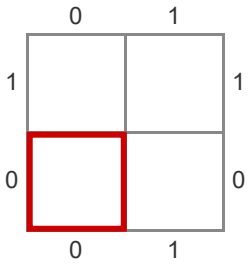
¡Muy bien! Tu solución pasó todas las pruebas

Resultados de las pruebas:

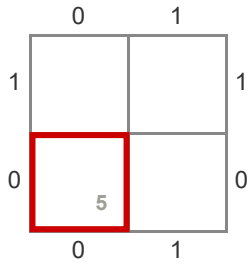
Tablero inicial



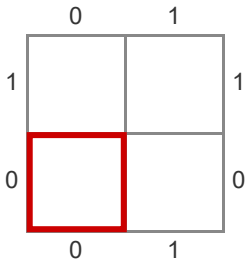
Tablero final



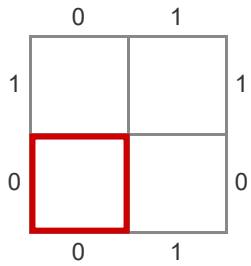
Tablero inicial



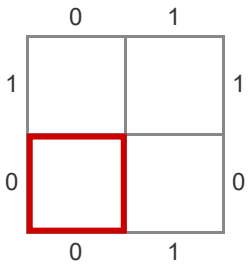
Tablero final



Tablero inicial



Tablero final



Información importante

Términos y Condiciones

Reglas del Espacio de Consultas