



Introducción a la Programación

Clases teóricas

por Pablo E. “Fidel” Martínez López

7. Variables y funciones con procesamiento





Repaso



- **Programar es comunicar** (con máquinas y personas)
 - Estrategia de solución (división en subtarefas)
 - Legibilidad (elección de nombres, indentación)
 - **CONTRATOS:** Propósito, parámetros y precondiciones
- **Programas** (texto con diversos elementos)
 - **Comandos:** describen acciones
 - **Expresiones:** describen información
 - **Tipos:** clasifican expresiones



- **Comandos**

- Primitivos y secuencia
- PROCEDIMIENTOS (con y sin parámetros)
- Repetición simple
- Alternativa condicional
- Repetición condicional



- **Expresiones**

- Valores literales y expresiones primitivas
- Operadores
 - numéricos, de enumeración, de comparación, lógicos
- FUNCIONES (con y sin parámetros)
- Parámetros (como datos)



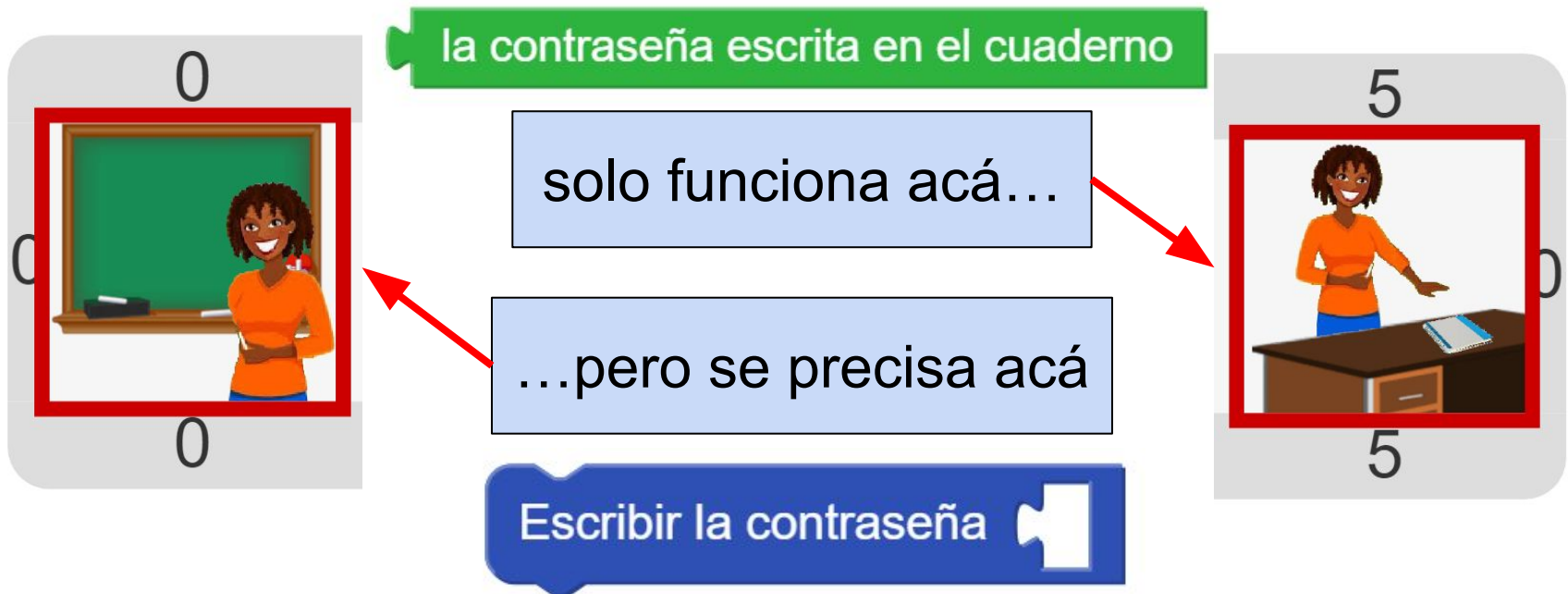
- **Tipos de datos**

- permiten clasificar expresiones
- en Gobstones, por ahora, son cuatro
 - colores, direcciones, números y valores de verdad
- toda expresión tiene un tipo
- los parámetros deben especificar qué tipo de expresiones aceptan



Variables

- Las expresiones primitivas solo dan información de la celda actual
- ¿Y si necesitamos esa información en otra celda?
¿Cómo recordamos información en un programa?
- Hace falta otra **herramienta del lenguaje**





- Una **variable** es un **nombre** que permite recordar un valor durante la ejecución de un procedimiento
 - La acción de recordar es un comando (**asignación**)
 - El **nombre** se puede usar como expresión

Recordar que contraseña vale



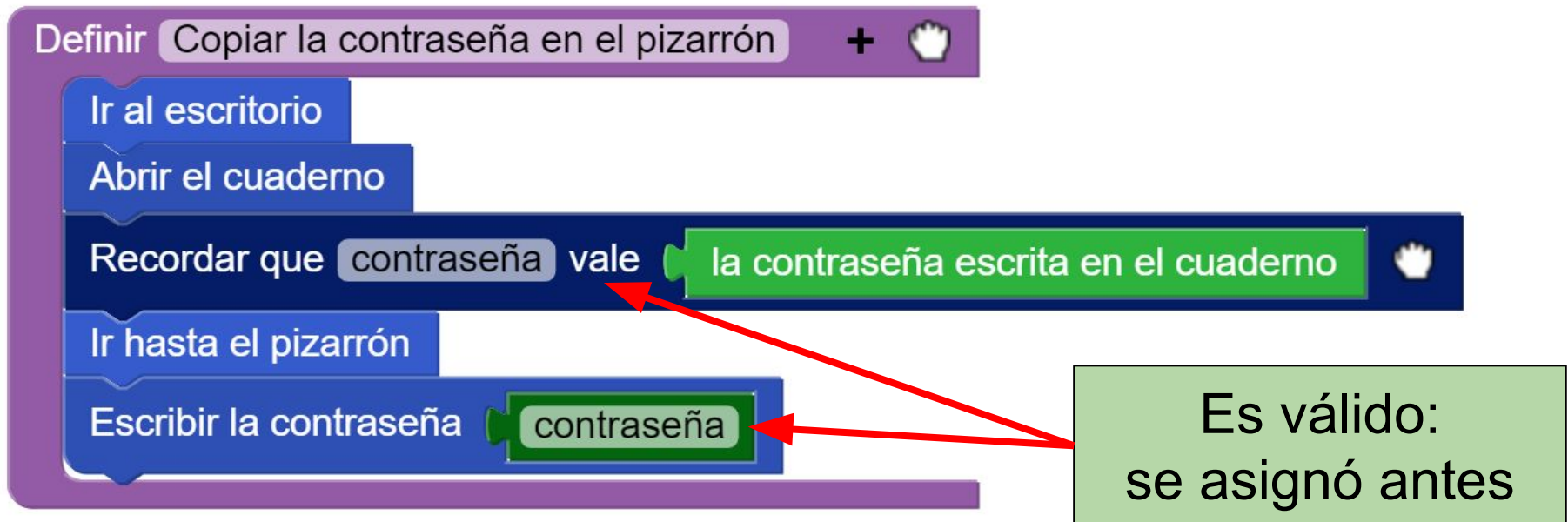
Asignación de la variable

Uso de la variable

contraseña



- Una variable recuerda un valor
 - solamente en el procedimiento que la asigna y
 - solo el último valor asignado
(si no fue asignada, no tiene valor y da BOOM)





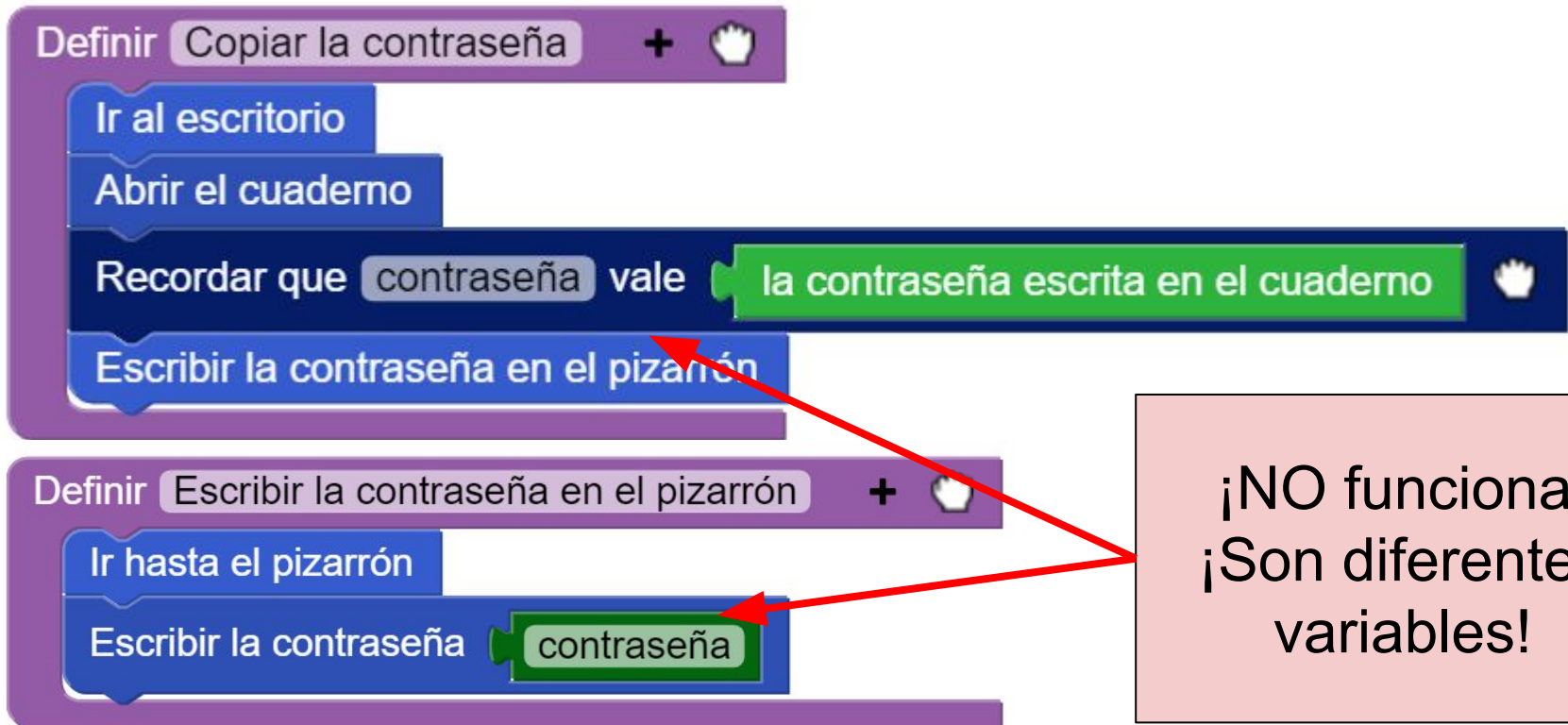
- Una variable recuerda un valor
 - solamente en el procedimiento que la asigna y
 - solo el último valor asignado
(si no fue asignada, no tiene valor y da BOOM)



NO es válido:
el procedimiento
la usa pero no la
asigna



- Una variable recuerda un valor
 - solamente en el procedimiento que la asigna y
 - solo el último valor asignado
(si no fue asignada, no tiene valor y da BOOM)



¡NO funciona!
¡Son diferentes
variables!



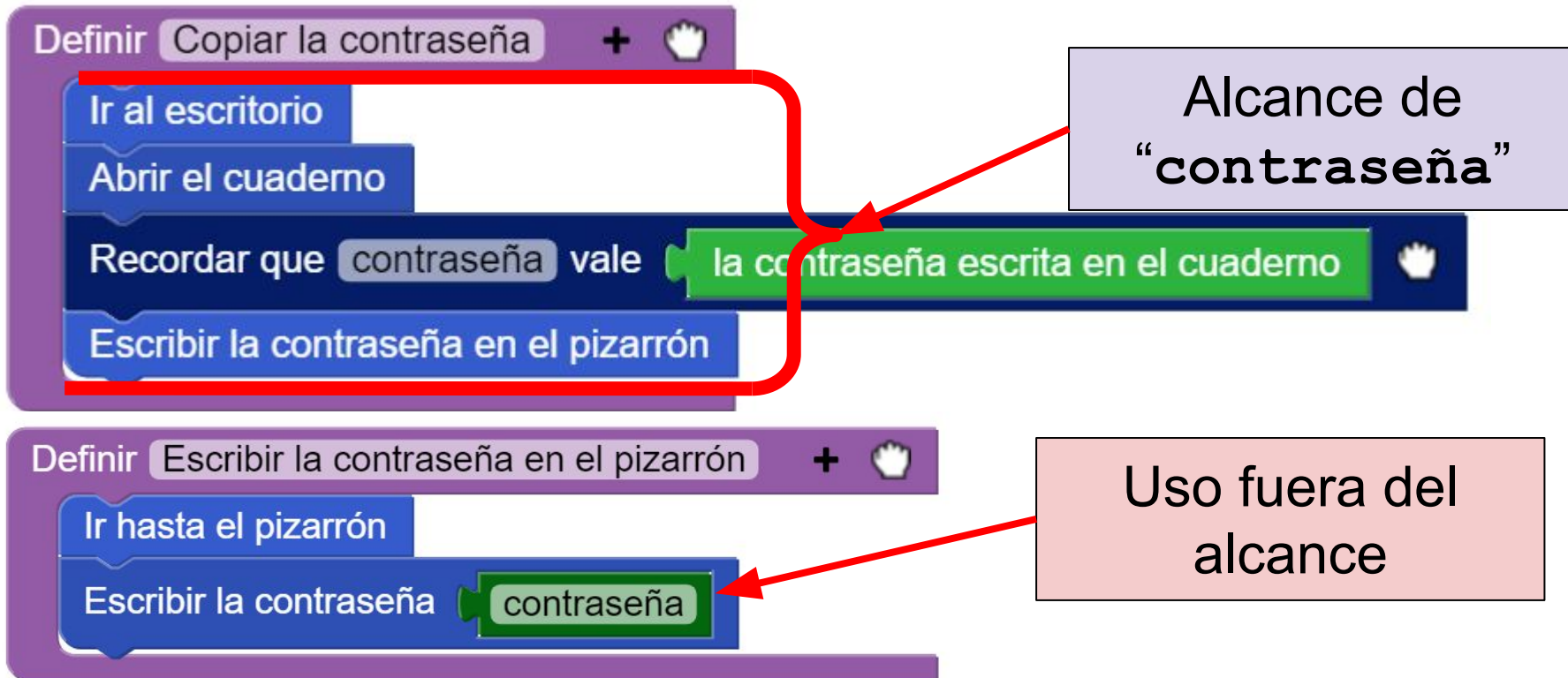
- Una variable recuerda un valor
 - solamente en el procedimiento que la asigna y
 - solo el último valor asignado
(si no fue asignada, no tiene valor y da BOOM)



Válido pero inútil:
el procedimiento
la asigna pero no
la usa

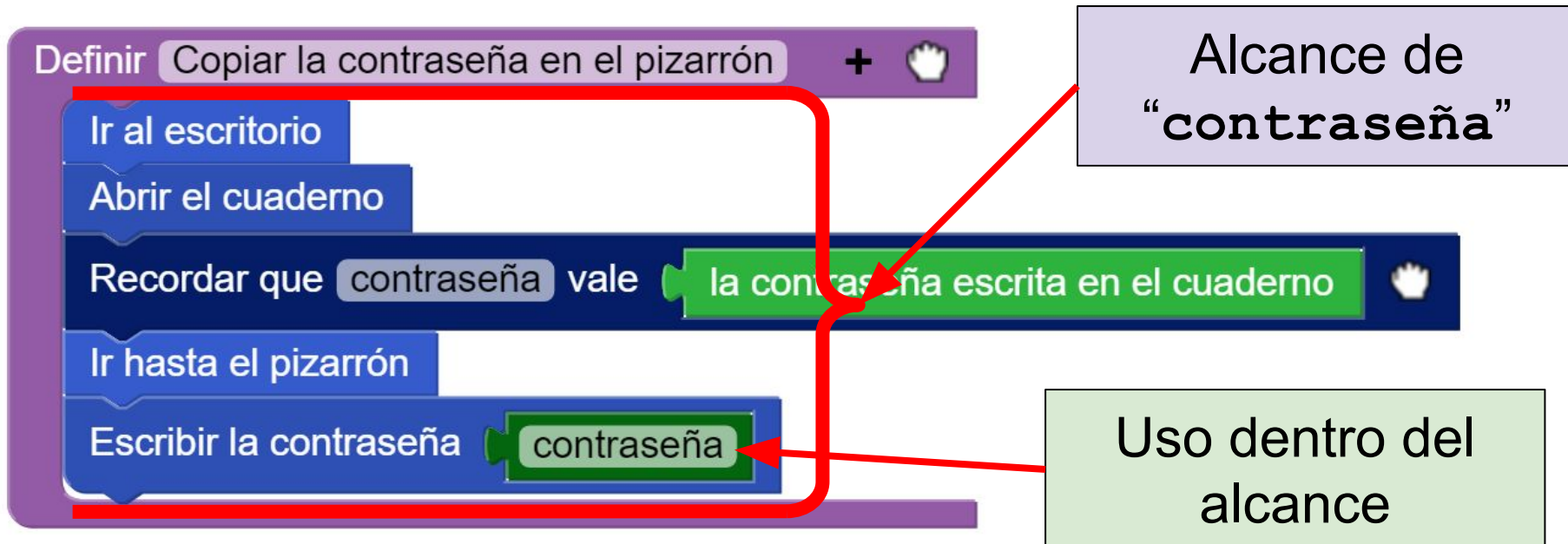


- El **alcance** (*scope*) de una variable es la parte del código donde el valor sigue siendo recordado por la variable
 - En Gobstones, el alcance es **local**
 - O sea, solo en el cuerpo del procedimiento que asigna






- El **alcance** (*scope*) de una variable es la parte del código donde el valor sigue siendo recordado por la variable
 - En Gobstones, el alcance es **local**
 - O sea, solo en el cuerpo del procedimiento que asigna






- En **texto**, la asignación se escribe infija, con el símbolo `:=`
 - `<nombreDeVariable> := <expresión>`
 - El **nombre** de la variable empieza con minúscula
 - Y sigue las mismas reglas que otros nombres


```
procedure CopiarLaContraseñaEnElPizarrón() {  
    /*  */  
    IrAlEscritorio()  
    AbrirElCuaderno()  
    contraseña := laContraseñaEscritaEnElCuaderno()  
    IrHastaElPizarrón()  
    EscribirLaContraseña_(contraseña)  
}
```

Asignación

- En este caso, la variable puede evitarse
 - Usar parámetros para comunicar dos procedimientos
 - ¿Y entonces para qué tener variables?

```
procedure CopiarLaContraseñaEnElPizarrón() {  
    /**/  
    IrAlEscritorio()  
    AbrirElCuaderno()  
    EscribirEnElPizarrón(laContraseñaEscritaEnElCuaderno())  
}
```

Argumento

```
procedure EscribirEnElPizarrón(contraseña) {  
    /**/  
    IrHastaElPizarrón()  
    EscribirLaContraseña_(contraseña)  
}
```

Parámetro

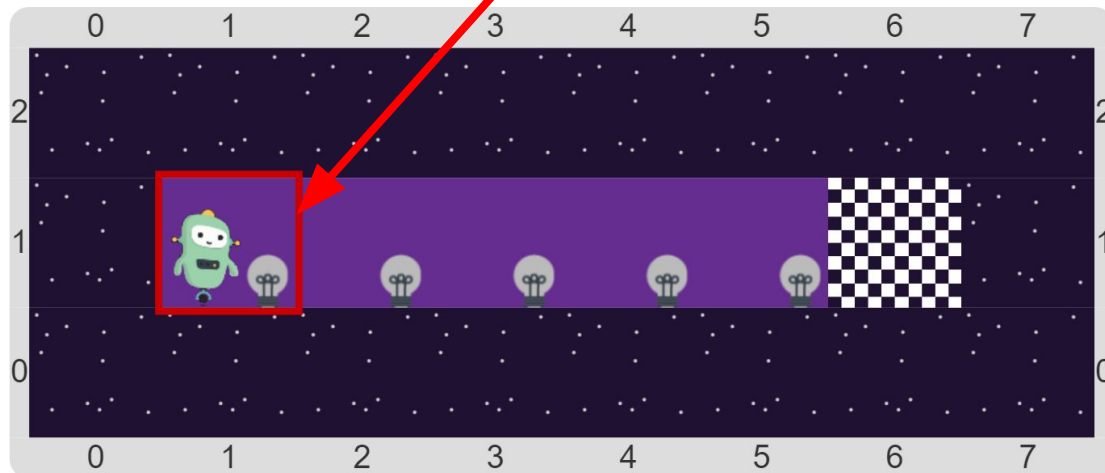


Acumuladores y recorridos de acumulación



- Las variables son útiles para realizar un recorrido que permita contar
 - Recuerda la cantidad ya procesada
 - El recuerdo cambia en cada iteración

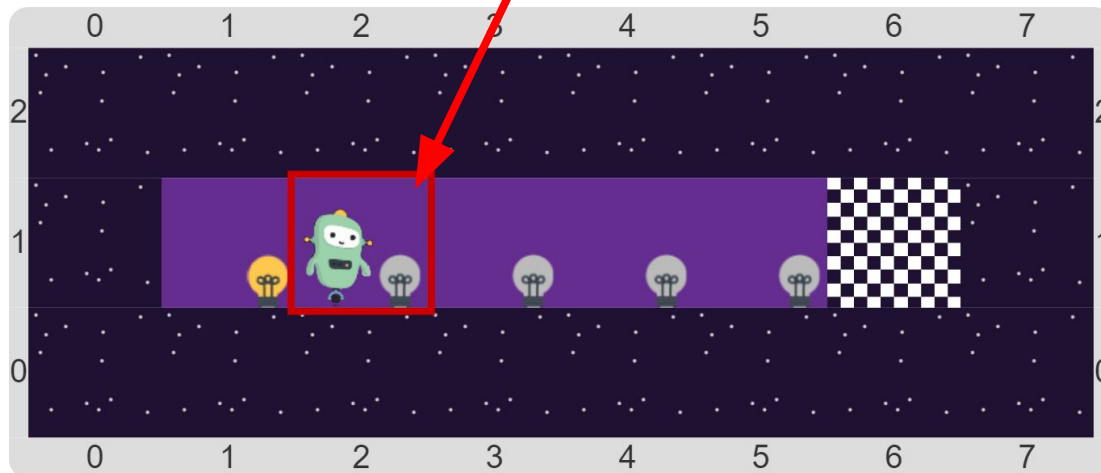
La cantidad de luces que ya prendí es 0





- Las variables son útiles para realizar un recorrido que permita contar
 - Recuerda la cantidad ya procesada
 - El recuerdo cambia en cada iteración

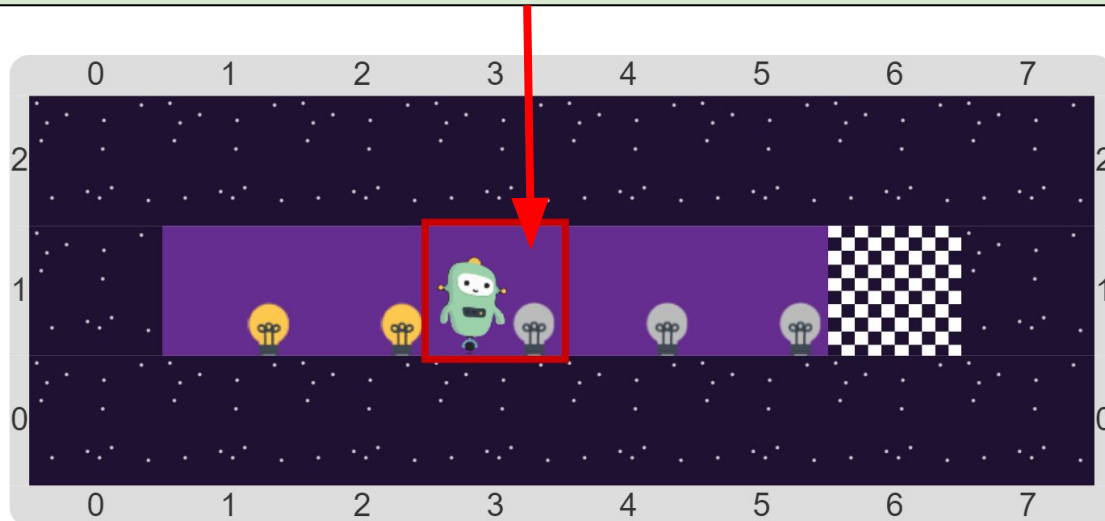
La cantidad de luces que ya prendí es 1





- Las variables son útiles para realizar un recorrido que permita contar
 - Recuerda la cantidad ya procesada
 - El recuerdo cambia en cada iteración

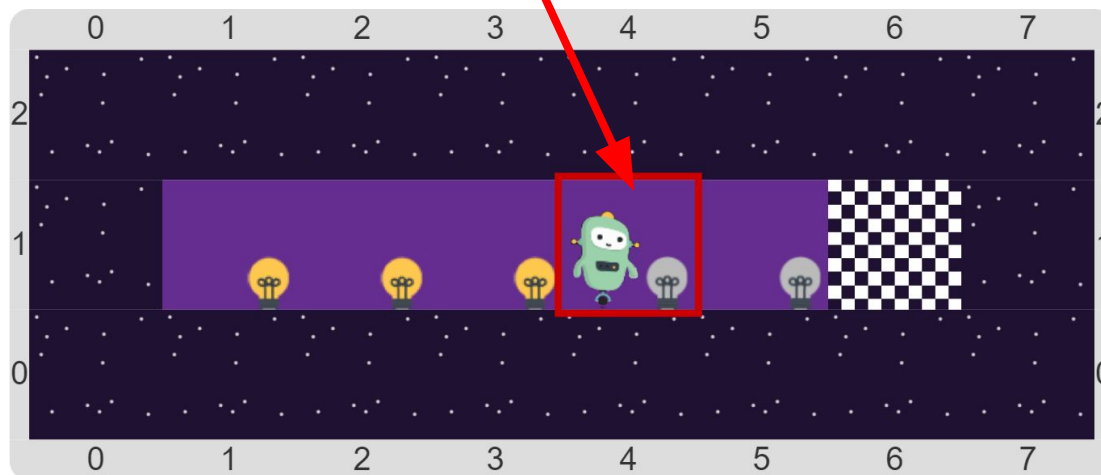
La cantidad de luces que ya prendí es 2 (1+1)





- Las variables son útiles para realizar un recorrido que permita contar
 - Recuerda la cantidad ya procesada
 - El recuerdo cambia en cada iteración

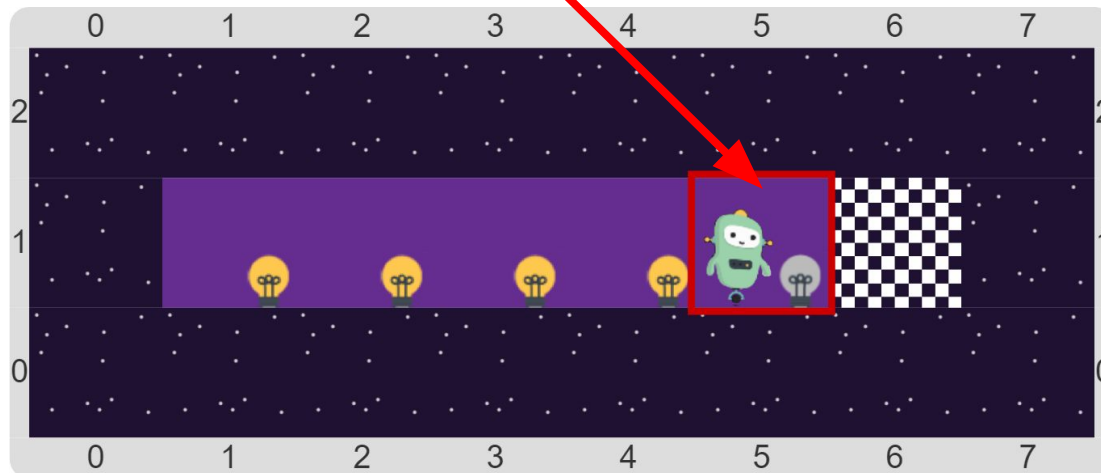
La cantidad de luces que ya prendí es 3 (2+1)





- Las variables son útiles para realizar un recorrido que permita contar
 - Recuerda la cantidad ya procesada
 - El recuerdo cambia en cada iteración

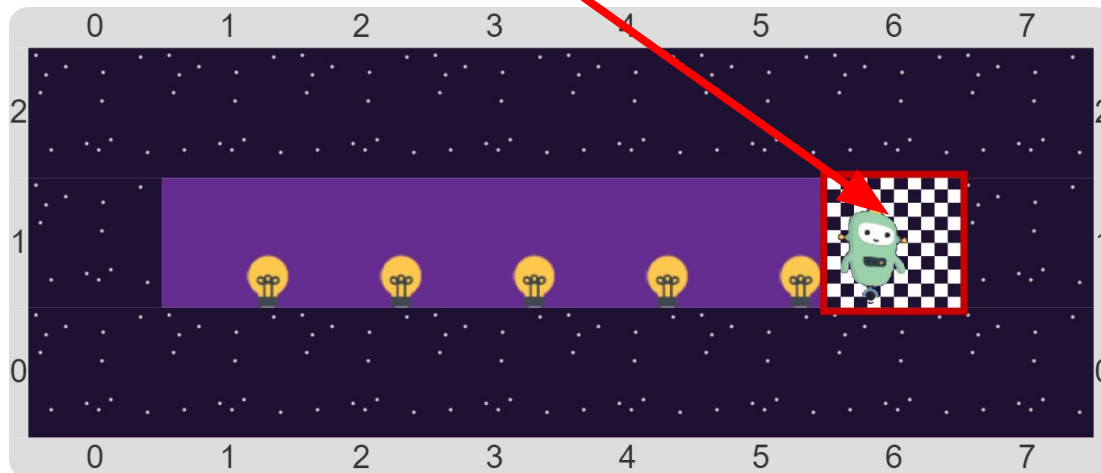
La cantidad de luces que ya prendí es 4 (3+1)





- Las variables son útiles para realizar un recorrido que permita contar
 - Recuerda la cantidad ya procesada
 - El recuerdo cambia en cada iteración

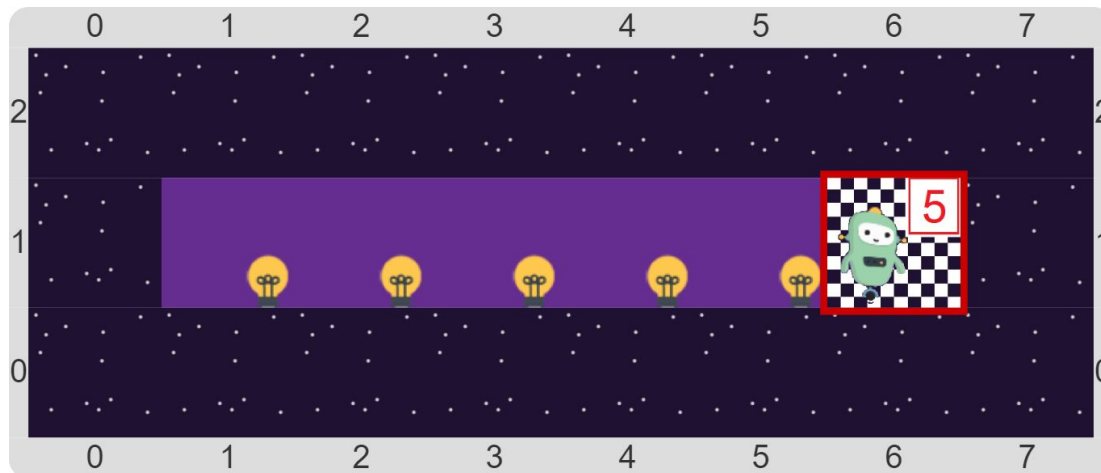
La cantidad de luces que ya prendí es 5 (4+1)





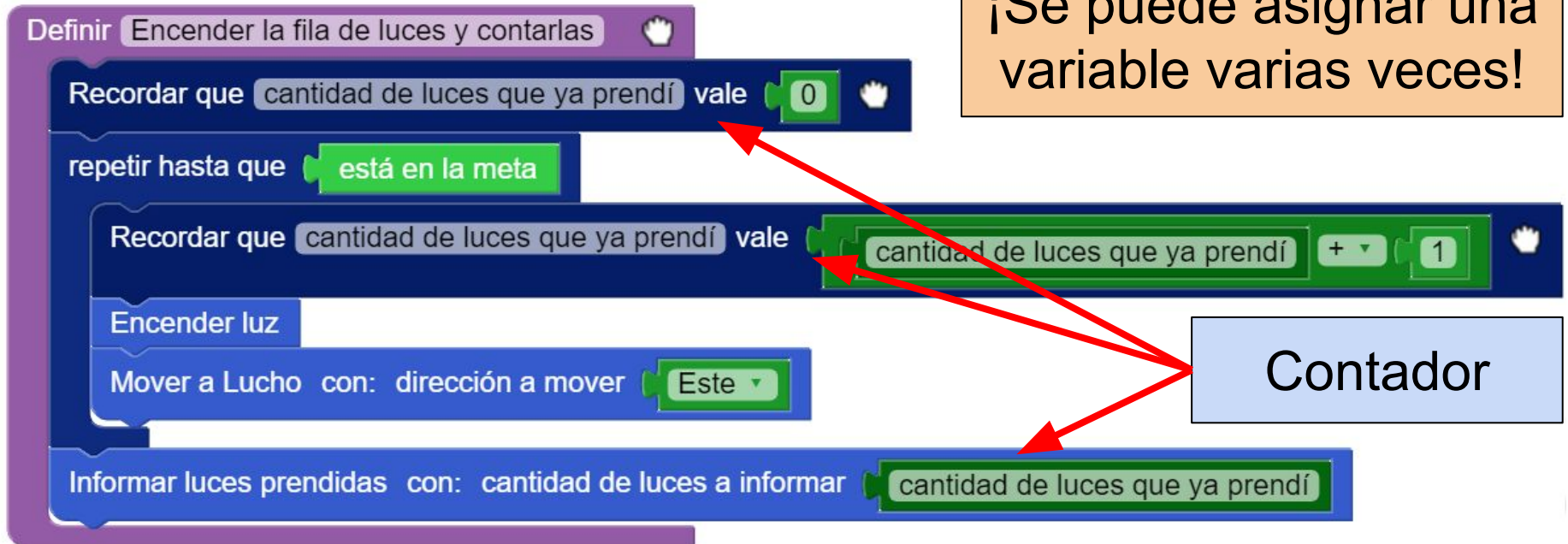
- Las variables son útiles para realizar un recorrido que permita contar
 - Recuerda la cantidad ya procesada
 - El recuerdo cambia en cada iteración

¡La cantidad recordada cambia en cada iteración!



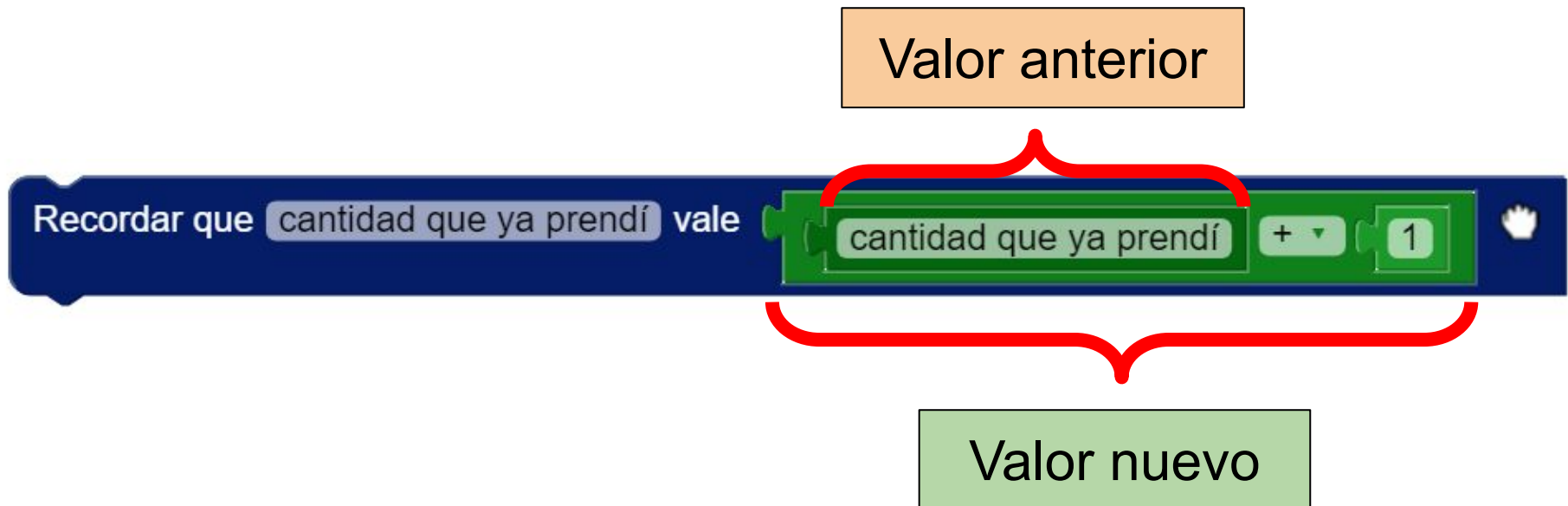


- Las variables son útiles para realizar un recorrido que permita contar, un **recorrido de acumulación**
 - En ese caso, se la llama **contador**, o **acumulador**
 - El contador recuerda la cantidad ya procesada





- ¿Cómo funciona la asignación de contador?
 - Se toma el valor anterior, se lo incrementa, y se recuerda el nuevo valor
 - Esto recibe el nombre de **incrementar** el contador





- ¿Cual es el “valor anterior” la primera vez?
 - El contador debe tomar un **valor inicial**
 - Esto se conoce como ***inicializar*** el contador
 - ¿Qué valor inicial usar para contar?


Inicializar es recordar el valor inicial



¿Por qué usar 0?



- Un **recorrido de acumulación** es un recorrido que utiliza un **contador** o **acumulador** para calcular una cantidad

```
procedure EncenderLaFilaDeLucesYContarlas() {  
    /*  */  
    cantidadDeLucesQueYaPrendí := 0  
    while (not (estáEnLaMeta())) {  
        EncenderLuz()  
        cantidadDeLucesQueYaPrendí :=  
            cantidadDeLucesQueYaPrendí + 1  
        MoverALucho(Este)  
    }  
    InformarLucesPrendidas(cantidadDeLucesQueYaPrendí) // Finalizar recorrido  
}
```

¡No se puede poner la asignación en otro procedimiento! ¿Por qué?

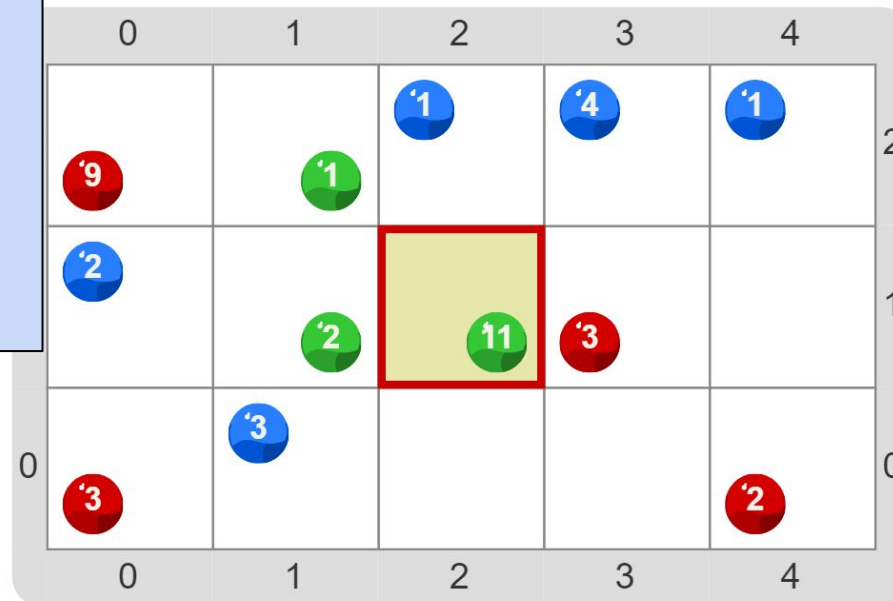


Funciones con procesamiento



- ¿Cómo conseguir información de una celda que no sea la actual? Haría falta una función...
 - ...¡pero las funciones no se pueden mover!
 - Precisamos una **herramienta** nueva

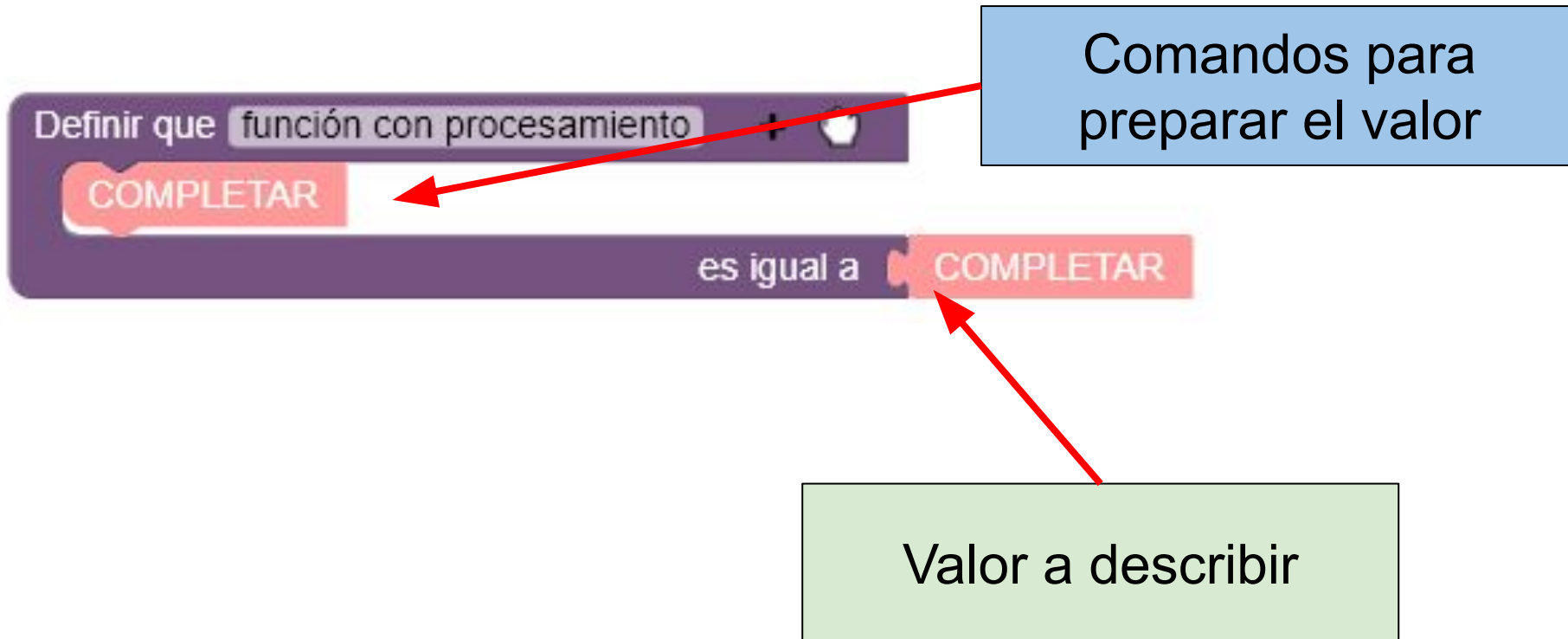
¿Hay alguna bolita en la celda lindante al Oeste?



¿Cuántas bolitas hay en el tablero?

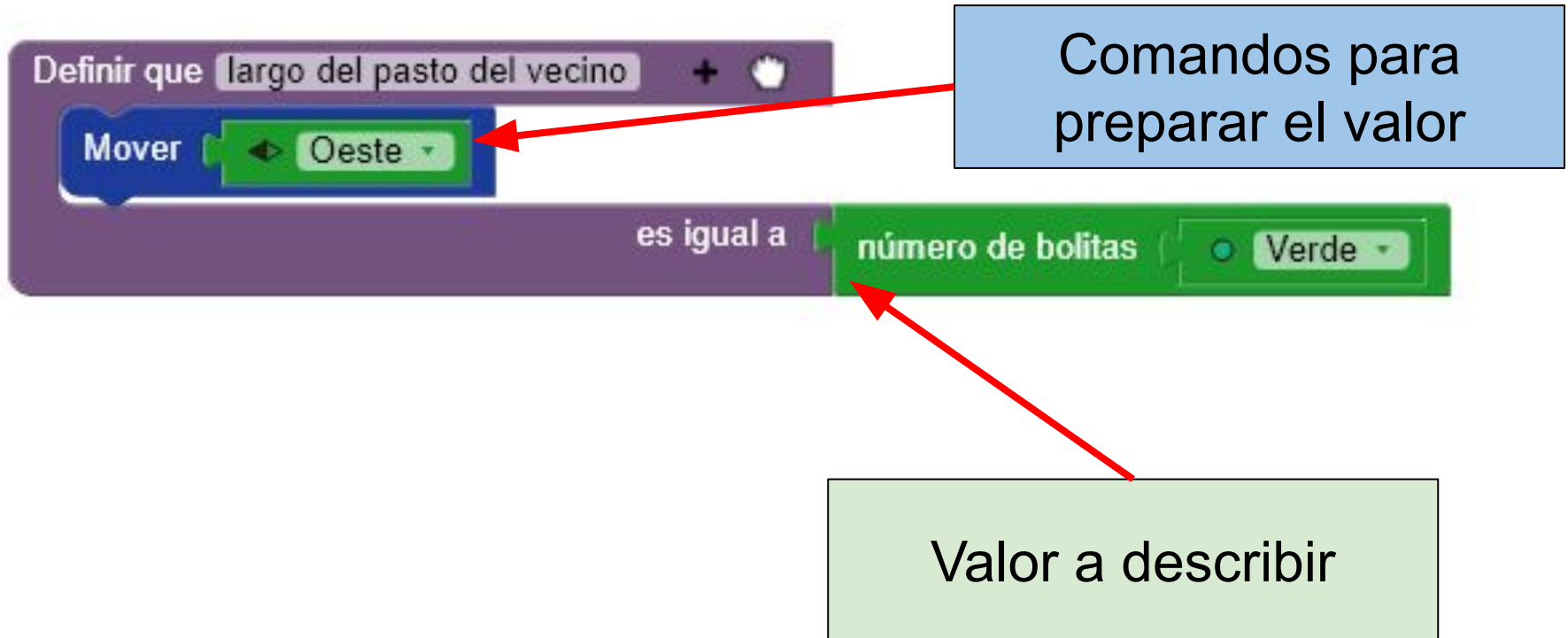


- Una ***función con procesamiento***
 - es una *función*, porque describe un valor
 - pero puede hacer *acciones* para calcularlo





- Una ***función con procesamiento***
 - es una *función*, porque describe un valor
 - pero puede hacer *acciones* para calcularlo





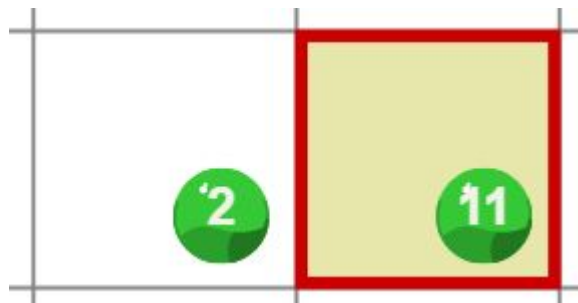
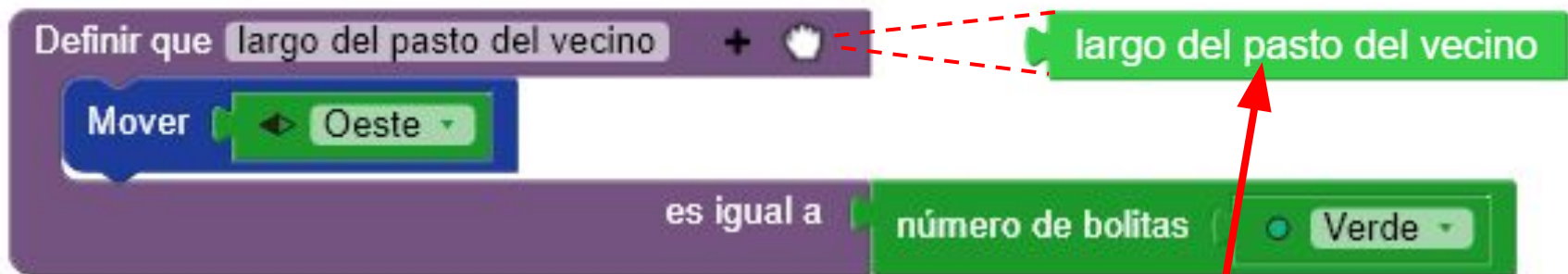
- Una ***función con procesamiento***
 - es una *función*, porque describe un valor
 - pero puede hacer *acciones* para calcularlo



Valor descrito




- Una **función con procesamiento**
 - es una *función*, porque describe un valor
 - pero puede hacer *acciones* para calcularlo



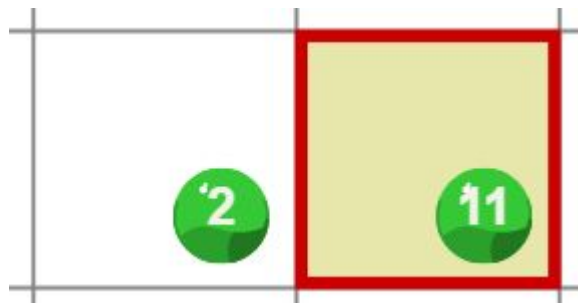
¿Cuánto vale la expresión?
¡Vale 2!



- En **texto** el procesamiento se escribe entre las llaves, ANTES del *return* de la función
 - En forma parecida al cuerpo de un procedimiento

```
function largoDelPastoDelVecino() {  
    /*  */  
    Mover(Oeste)  
    return (nroBolitas(Verde))  
}
```

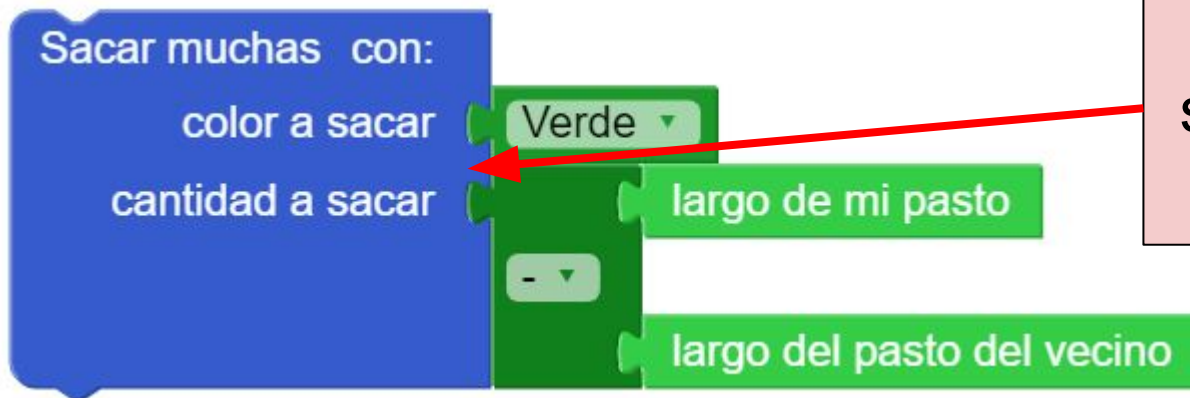
Comandos para preparar el valor



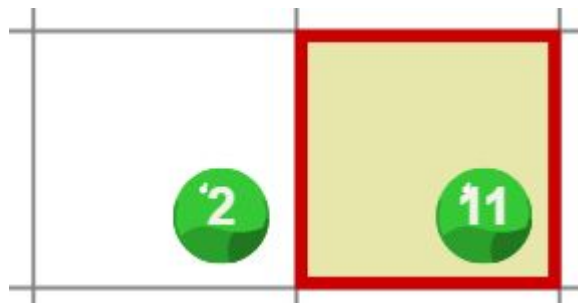
Valor a describir



- Una **función con procesamiento** NO CAMBIA el estado
 - SOLAMENTE describe un valor
 - Las acciones son *imaginarias*




¿En cuál celda
sucede esta acción?
¡En la celda actual!



¡Las funciones NO
MODIFICAN la celda actual!



- La acción de una función con procesamiento puede ser un recorrido de acumulación
 - ¡Pero es un recorrido imaginario!
 - Solo importa el valor final calculado

```
function cantidadDeBolitasEnElTablero() {  
    /**/  
    cantidadDeBolitasVistas := 0 // Iniciar recorrido  
    IniciarRecorridoDeCeldas(Este, Norte) // "  
    while (quedanCeldasDelRecorridoDeCeldas(Este, Norte)) { // mientras queden celdas  
        cantidadDeBolitasVistas := cantidadDeBolitasVistas // Procesar celda  
                                + nroTotalDeBolitas() // actual  
        PasarALaSiguienteCeldaDelRecorridoDeCeldas(Este, Norte) // Pasar a la siguiente  
    }  
    cantidadDeBolitasVistas := cantidadDeBolitasVistas // Procesar la última  
                                + nroTotalDeBolitas() // celda  
  
    return (cantidadDeBolitasVistas) // Finalizar recorrido  
}
```



- La acción puede ser un recorrido de acumulación
 - ¡Pero es un recorrido imaginario!
 - Solo importa el valor final calculado
 - ¡ATENCIÓN a no usar variables innecesariamente!

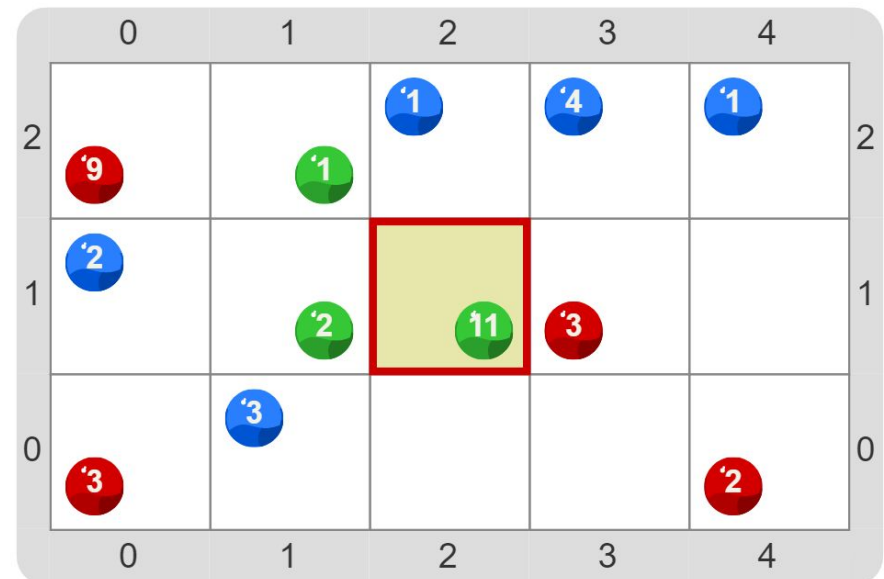
```
function cantidadDeBolitasEnElTablero() {  
    /*  */  
    cantidadDeBolitasVistas := 0 // Iniciar recorrido  
    IniciarRecorridoDeCeldas(Este, Norte) // "  
    while (quedanCeldasDelRecorridoDeCeldas(Este, Norte)) { // mientras queden celdas  
        cantidadDeBolitasVistas := cantidadDeBolitasVistas // Procesar celda  
                                + nroTotalDeBolitas() // actual  
        PasarALaSiguienteCeldaDelRecorridoDeCeldas(Este, Norte) // Pasar a la siguiente  
    }  
    // Finalizar el recorrido  
    return (cantidadDeBolitasVistas + nroTotalDeBolitas()) // y procesar el último  
    // ¡NO HACE FALTA UNA ASIGNACIÓN AL FINAL!  
}
```




- La acción de una función con procesamiento puede ser un recorrido de acumulación
 - ¡Pero es un recorrido imaginario!
 - Solo importa el valor final calculado

```
program { /* ... */  
    Poner__Veces(Negro, cantidadDeBolitasEnElTablero())  
}
```

¿Dónde se van a poner las 42 bolitas negras?

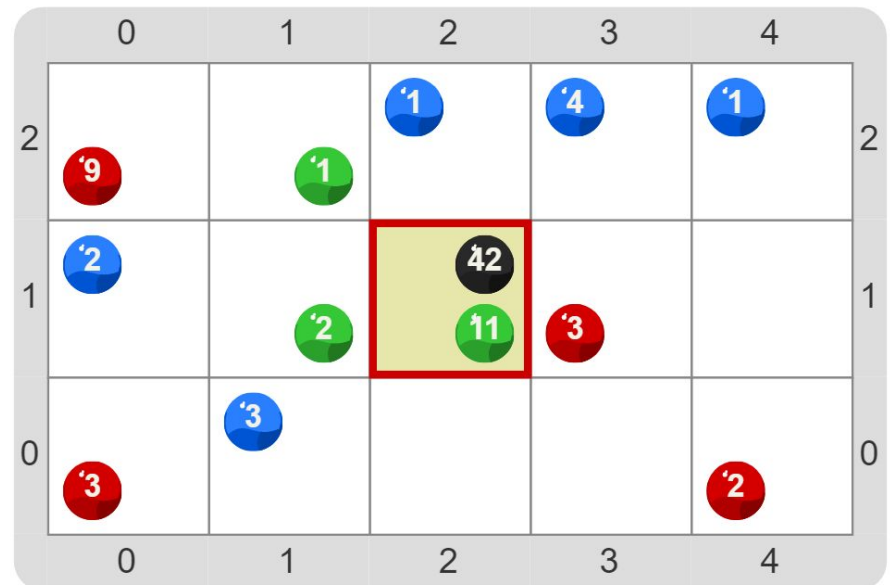




- La acción de una función con procesamiento puede ser un recorrido de acumulación
 - ¡Pero es un recorrido imaginario!
 - Solo importa el valor final calculado

```
program { /* ... */  
  Poner__Veces(Negro, cantidadDeBolitasEnElTablero())  
}
```

¡En la celda actual!

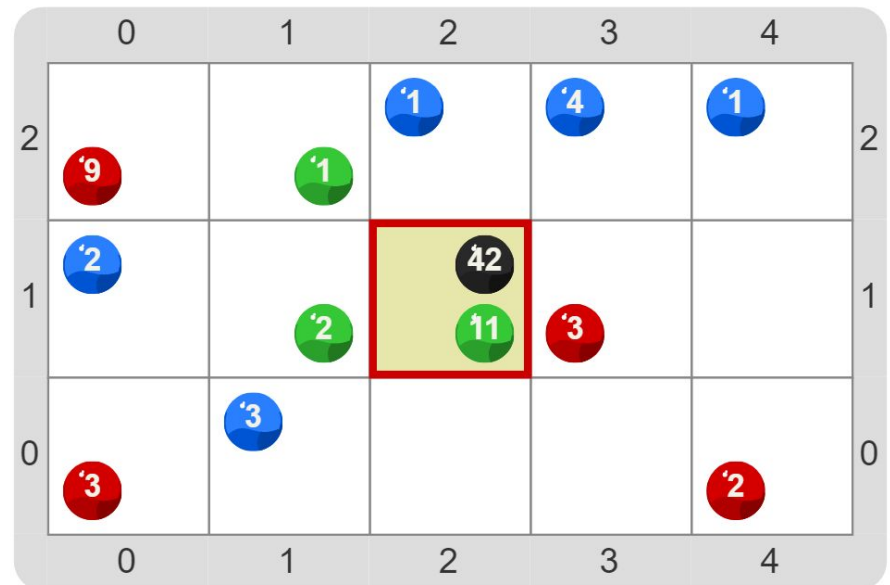




- La acción de una función con procesamiento puede ser un recorrido de acumulación
 - ¡Pero es un recorrido imaginario!
 - Solo importa el valor final calculado

```
program { /* ... */  
    Poner__Veces(Negro, cantidadDeBolitasEnElTablero())  
}
```

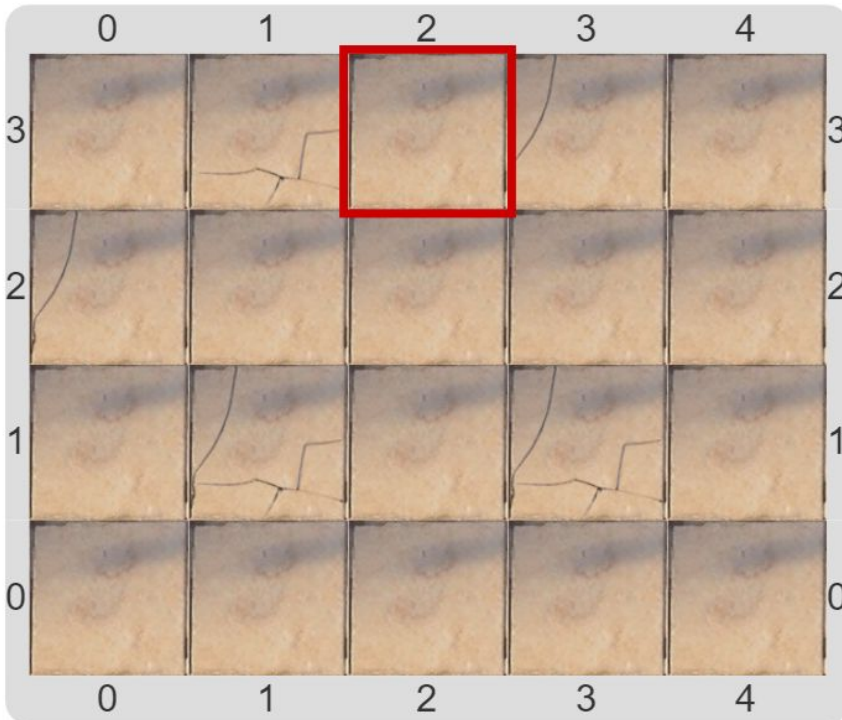
Durante el cálculo se mueve para contar, pero no es parte de la acción del programa





Alternativa condicional en expresiones

- El procesamiento de un recorrido de acumulación puede depender de una condición
 - Se precisa una alternativa condicional
 - ¿Cómo lograr modularizar?



Contar solamente las
baldosas rajadas

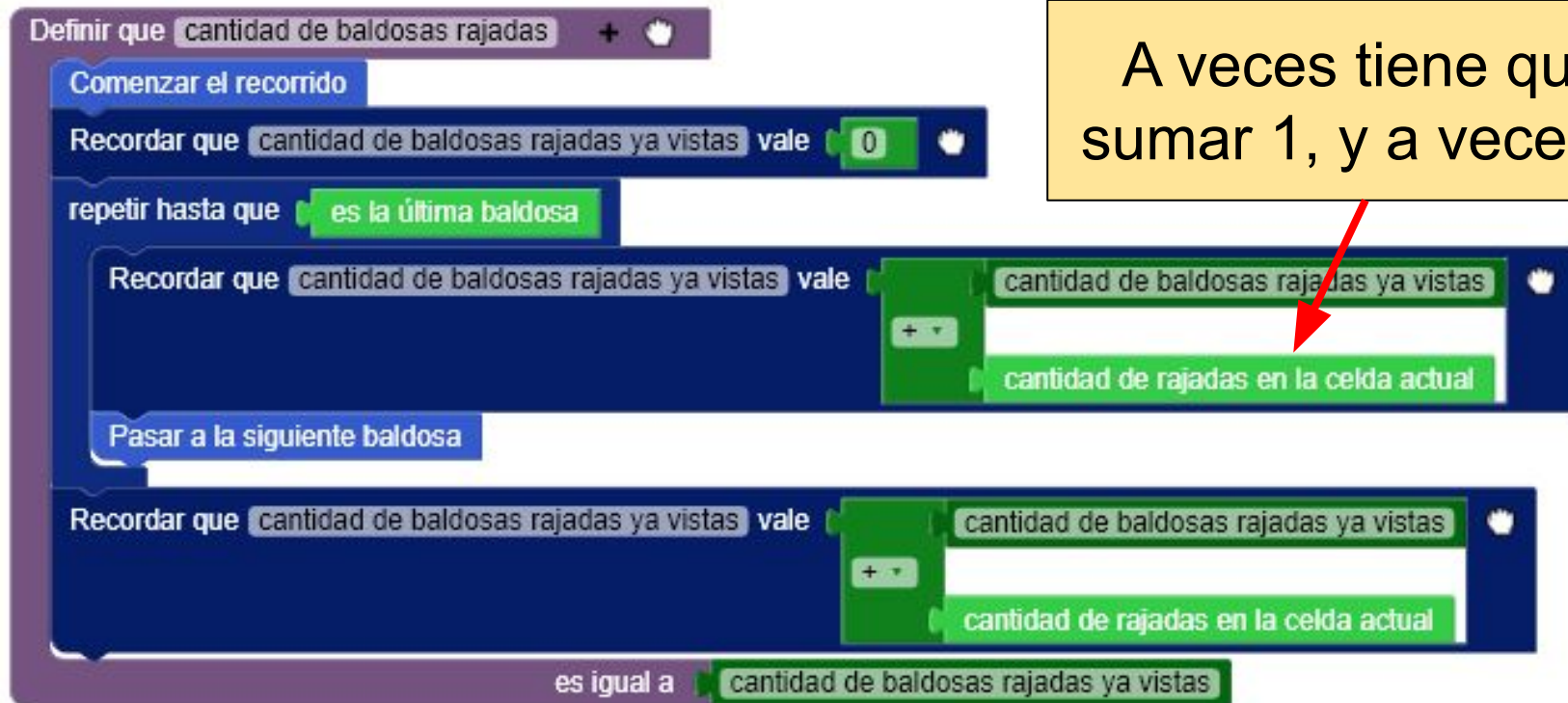


- El procesamiento de un recorrido de acumulación puede depender de una condición
 - Se precisa una alternativa condicional
 - ¿Cómo lograr modularizar?





- El procesamiento de un recorrido de acumulación puede depender de una condición
 - Se precisa una alternativa condicional
 - ¿Cómo lograr modularizar?



A veces tiene que sumar 1, y a veces 0



- Gobstones tiene ***alternativa condicional en expresiones***
 - Por ahora, solo en texto
 - Se usan las palabras clave **choose**, **when** y **otherwise**

```
function cantidadDeRajadasEnLaCeldaActual() {  
  /* PROPÓSITO: indica la cantidad de baldosas  
    rajadas en la celda actual  
    PRECONDICIÓN: ninguna  
    RESULTADO: un número (1 o 0)  
    OBSERVACIONES: usa una alternativa condicional  
    para elegir entre 0 y 1  
  */  
  return (choose 1 when (esBaldosaRajada())  
          0 otherwise)  
}
```

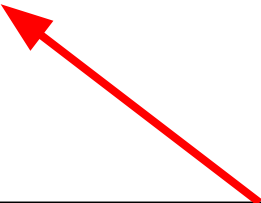
Alternativa
en
expresiones





- La **alternativa condicional en expresiones** se escribe
choose *<expresión1>* **when** (*<condición>*)
<expresión2> **otherwise**
 - Si la condición es verdadera, elige la *expresión1*, y en otro caso, elige la *expresión2*

```
choose 1 when (esBaldosaRajada())  
0 otherwise
```



Vale 1 si la baldosa está rajada,
y 0 en otro caso

- La **alternativa condicional en expresiones** puede tener múltiples ramas, al igual que la de comandos
 - Cada rama tiene su condición (excepto la última)
 - Es útil cuando hay más de 2 alternativas

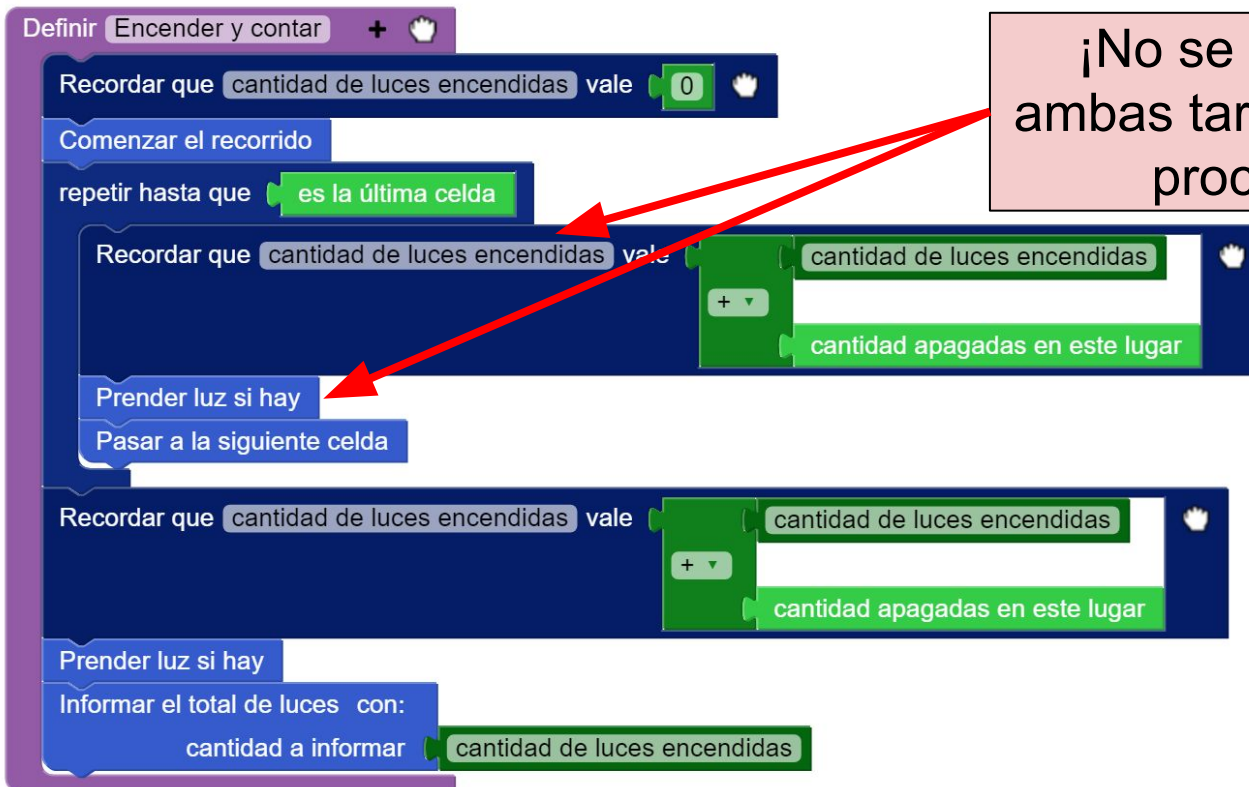
```
function direcciónParaElCódigo(código) {  
  /*  
    PROPÓSITO: describe la dirección correspondiente  
               al código dado  
    PRECONDICIONES: código está entre 1 y 4  
    PARÁMETROS: código es un número  
    RESULTADO: una dirección, la que corresponde al código  
  */  
  return (choose Norte when (código == 1)  
           Este   when (código == 2)  
           Sur    when (código == 3)  
           Oeste  when (código == 4)  
           boom("No es código válido") otherwise)  
}
```



Variables y modularización



- Las variables locales desafían a la *modularización*
 - ¿Cómo describir información y cambiar el estado al mismo tiempo?
 - En Gobstones comandos y expresiones están separados



¡No se pueden hacer ambas tareas en el mismo procedimiento!



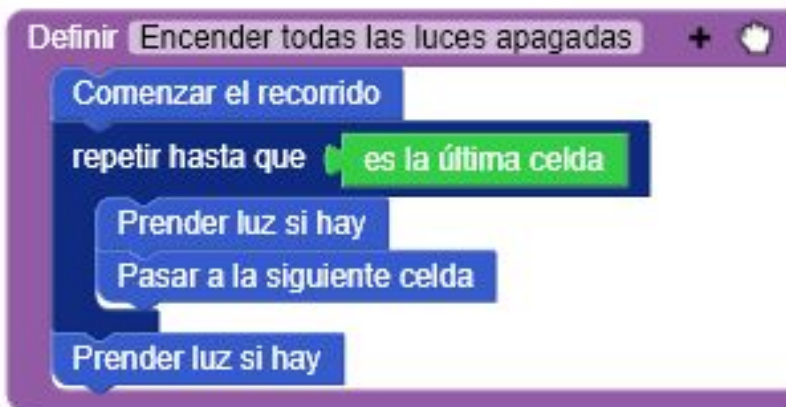
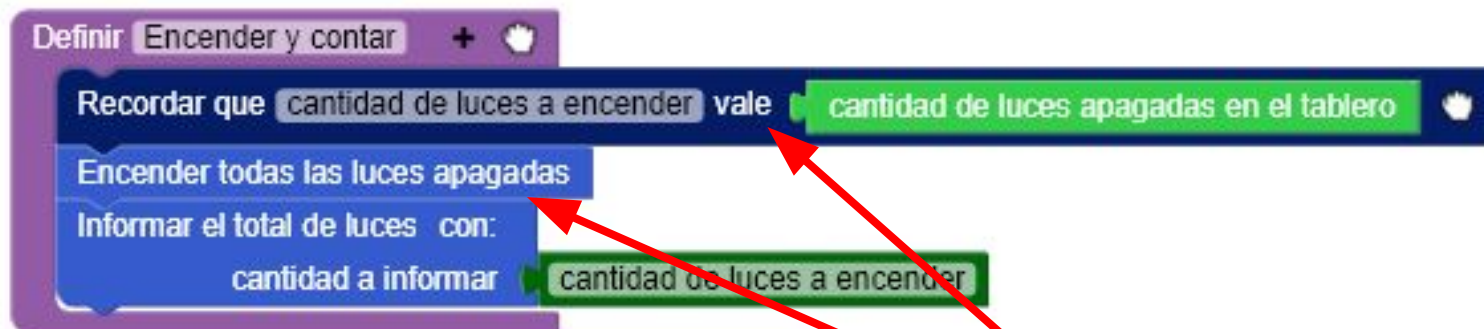
- Las variables locales desafían a la *modularización*
 - ¿Cómo describir información y cambiar el estado al mismo tiempo?
 - En Gobstones comandos y expresiones están separados

```
procedure EncenderYContar() {  
  ComenzarElRecorrido() // IniciarE  
  cantidadDeLucesEncendidas := 0 // (y la  
  while (not (esLaÚltimaCelda())) { // while (h  
    cantidadDeLucesEncendidas :=  
      cantidadDeLucesEncendidas // ProcesarElemento()  
      + cantidadDeLucesApagadasEnEsteLugar() // (¡¡y contarlos!!)  
    PrenderLuzSiHay() //  
    PasarALaSiguienteCelda() // PasarAlSiguienteElemento()  
  }  
  cantidadDeLucesEncendidas :=  
    cantidadDeLucesEncendidas // ProcesarÚltimoElemento()  
    + cantidadDeLucesApagadasEnEsteLugar() // (¡¡y contarlos!!)  
  PrenderLuzSiHay() // Cuenta primero, porque cuenta luces apagadas  
  // FinalizarRecorrido()  
  InformarElTotalDeLuces(cantidadDeLucesEncendidas)  
}
```

¡No se pueden hacer ambas tareas en el mismo procedimiento!




- Las variables locales desafían a la *modularización*
 - ¿Cómo describir información y cambiar el estado al mismo tiempo?
 - ¡Dos recorridos diferentes, uno *imaginario*!




¡Primero cuenta y después recorre encendiendo!



- Las variables locales desafían a la *modularización*
 - ¿Cómo describir información y cambiar el estado al mismo tiempo?
 - ¡Dos recorridos diferentes, uno *imaginario*!

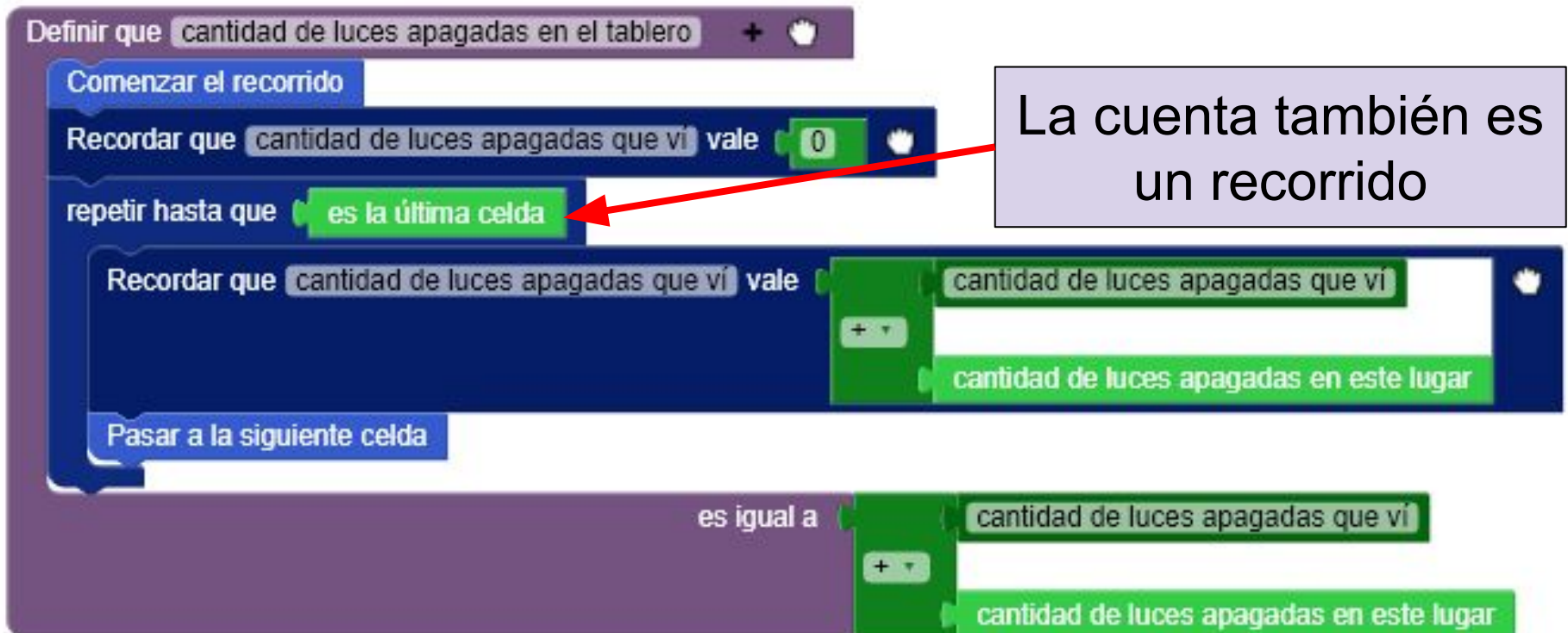
```
procedure EncenderYContar() {  
    /*  */  
    cantidadDeLucesAEncender := cantidadDeLucesApagadasEnElTablero()  
    EncenderTodasLasLucesApagadasDelTablero()  
    InformarElTotalDeLuces(cantidadDeLucesAEncender)  
}
```

```
procedure EncenderTodasLasLucesApagadasDelTablero() {  
    /*  */  
    ComenzarElRecorrido()           // IniciarRecorrido()  
    while (not esLaÚltimaCelda()) { // mientras (quedanElementos())  
        PrenderLuzSiHay()           // ProcesarElementoActual()  
        PasarALaSiguienteCelda()    // PasarASiguienteElemento()  
    }  
    PrenderLuzSiHay()               // ProcesarÚltimoElemento()  
}
```

¡Primero cuenta y después recorre encendiendo!




- Las variables locales desafían a la *modularización*
 - ¿Cómo describir información y cambiar el estado al mismo tiempo?
 - ¡Dos recorridos diferentes, uno *imaginario*!






- Las variables locales desafían a la *modularización*
 - ¿Cómo describir información y cambiar el estado al mismo tiempo?
 - ¡Dos recorridos diferentes, uno *imaginario*!

```
function cantidadDeLucesApagadasEnElTablero
/*  */
    cantidadDeLucesApagadas := 0
    ComenzarElRecorrido()
    while (not esLaÚltimaCelda()) {
        cantidadDeLucesApagadas :=
            cantidadDeLucesApagadas
            + cantidadDeLucesApagadasAcá()
        PasarALaSiguienteCelda()
    }
    return (cantidadDeLucesApagadas
            + cantidadDeLucesApagadasAcá())
}
```

La cuenta también es un recorrido

```
// IniciarRecorrido()
// "
// mientras (quedanElementos())
//   ProcesarElementoActual()
//   PasarASiguienteElemento()
//   ProcesarÚltimoElemento()
// FinalizarRecorrido()
```





Cierre



- **Variables**

- Forma de recordar un valor a través de un nombre
- Se recuerda con un comando de **asignación**
- El nombre solo vale dentro del procedimiento que realiza la asignación
 - **Alcance** de la variable
 - O sea, en Gobstones las variables son **locales**
- El nombre se puede usar como expresión para describir al valor recordado



- ***Funciones con procesamiento***
 - Son funciones, porque describen un valor
 - Pero pueden realizar una acción para calcular o descubrir el valor correspondiente
 - Sin embargo, la acción es *imaginaria*
 - O sea, la función no tiene ningún efecto sobre el tablero o las variables locales donde se usa
 - Son útiles para calcular información distante o compleja



- ***Alternativa condicional en expresiones***
 - Permite elegir entre varios valores, en base a condiciones
 - Por ahora, solo disponible en texto (sin coloreo)
- ***Modularización***
 - La separación entre comandos y expresiones, junto con las variables dificulta la modularización
 - Preferimos modularización sobre eficiencia