



# Introducción a la Programación

Clases teóricas

por Pablo E. “Fidel” Martínez López

## 5. Alternativas y funciones



# Repaso



- **Programar es comunicar** (con máquinas y personas)
- **Programas** (texto con diversos elementos)
  - **Comandos:** describen acciones
  - **Expresiones:** describen información
- **Procedimientos (con y sin parámetros)**
  - Para definir nuevos comandos
  - Permiten expresar **estrategia y representación de información**
  - Aportan legibilidad, reutilización, modificabilidad, generalidad
  - **CONTRATO: Propósito, parámetros y precondiciones**



- **Repetición simple**

- herramienta para evitar la repetición de código
- repite un número fijo de veces
- deben considerarse condiciones de borde

- **Parámetros**

- son agujeros en un procedimiento
- por cada uno hay que poner un argumento
- llevan un nombre y tienen un alcance
- proveen generalidad y abstracción



- **Tipos de datos**

- formas de clasificar las expresiones
- permite verificar usos incorrectos de expresiones
- al definir parámetros, debe establecerse su tipo

- **Expresiones primitivas**

- permiten sensar el tablero y obtener información
- describen un dato que depende de la celda actual



- **Operadores**

- permiten calcular nuevos valores en base a otros dados
- pueden ser numéricos, de enumeración, etc.

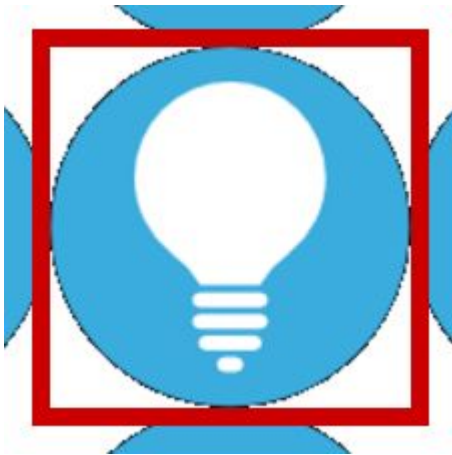
- **Funciones primitivas**

- son parecidas a las expresiones primitivas
- pero las construye el que diseña la actividad

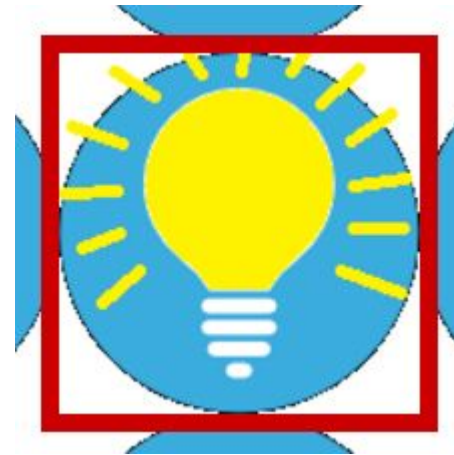


# Alternativas condicionales

- ¿Cómo tomamos decisiones ante escenarios cambiantes?
  - El programa no debe seguir *siempre las mismas* instrucciones, sino elegir entre diferentes **alternativas**
  - Es necesaria una nueva herramienta del lenguaje



En este escenario hay  
que prender la luz



En este escenario hay  
que apagar la luz

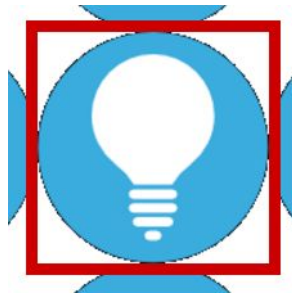
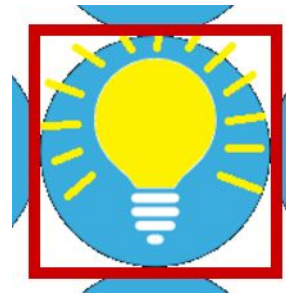




- ¿Cómo saber cuál es el estado del tablero?
- ¿Cómo describir esa información?
  - Es necesaria una **condición**
  - ¿Qué elemento del lenguaje describe información?

Hay 2 situaciones posibles:

- la luz está apagada o
- la luz está prendida

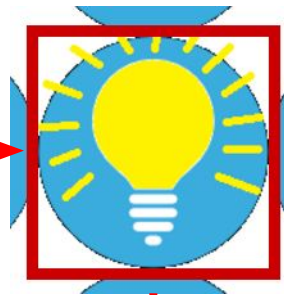
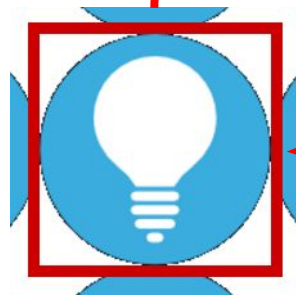




- ¿Para qué necesitamos saber el estado del tablero?
  - Para tomar **decisiones**
    - O sea, una opción elegida entre varias alternativas
  - Las condiciones se usan, entonces, para decidir

Hay 2 situaciones posibles:

- la luz está apagada o
- la luz está prendida

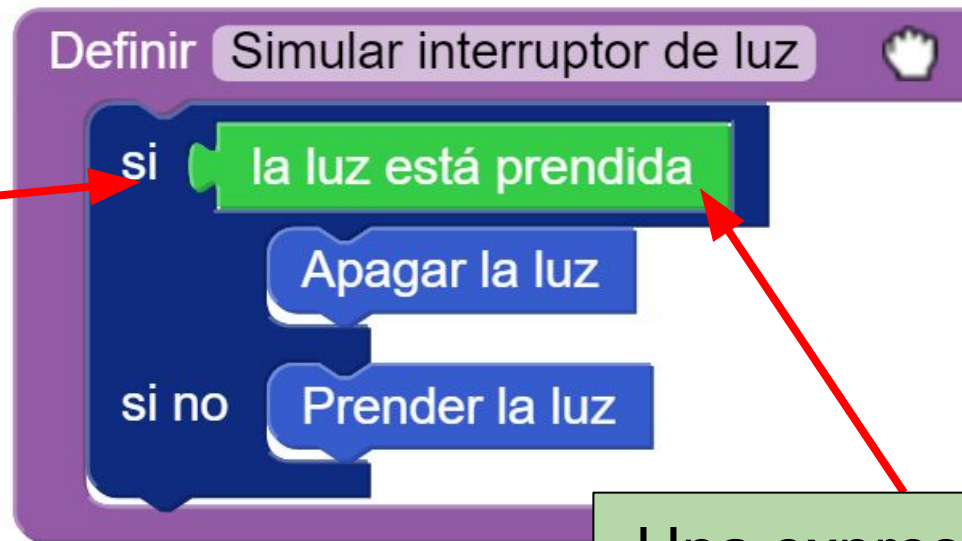


**Si la luz está prendida entonces apagar la luz si no prender la luz**



- Las **alternativas condicionales** permiten que el programa tome decisiones entre diferentes situaciones
  - La **condición** se describe con una expresión
    - ¿Qué valores puede tomar la condición?

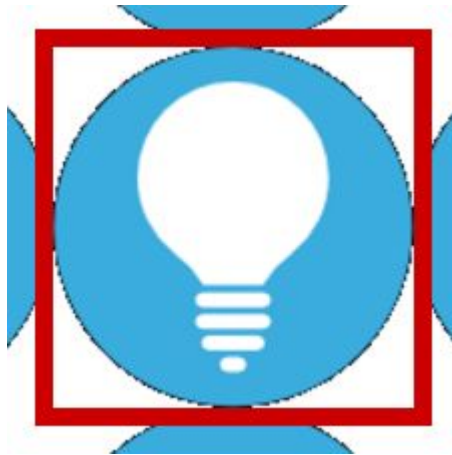
Un comando  
que elige  
entre  
alternativas



Una expresión que  
describe  
**verdadero o falso**

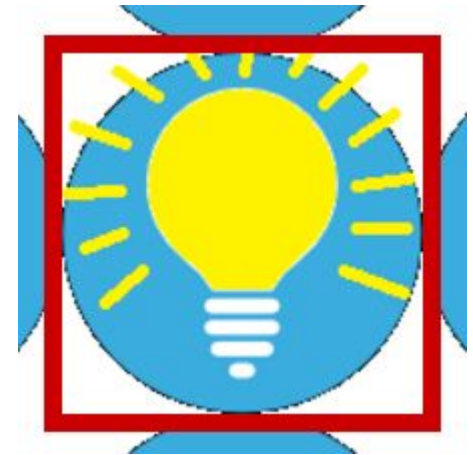
- La **condición** puede ser *verdadera* o *falsa*
- ¡Un nuevo tipo de datos!
  - Los **Valores de verdad**
  - También llamado **Booleanos** (abreviado **Bool**)

o



La expresión describe  
al valor ***falso***

la luz está prendida



La expresión describe  
al valor ***verdadero***



- ¿Qué expresiones tienen tipo Bool?
- Conozcamos primero 2 **expresiones primitivas**
  - Permiten sensor el tablero y dar información sobre él
  - Están vinculadas a los comandos primitivos parciales



¿Cuándo describen verdadero y cuándo falso?



- ¿Cómo serían en texto? ¿Qué argumentos esperan?
  - **hayBolitas** (<color>)
  - **puedeMover** (<dirección>)

`hayBolitas(previo(colorDelCuadrado))`    `hayBolitas(Rojo)`

`puedeMover(siguiente(direcciónInicial))`

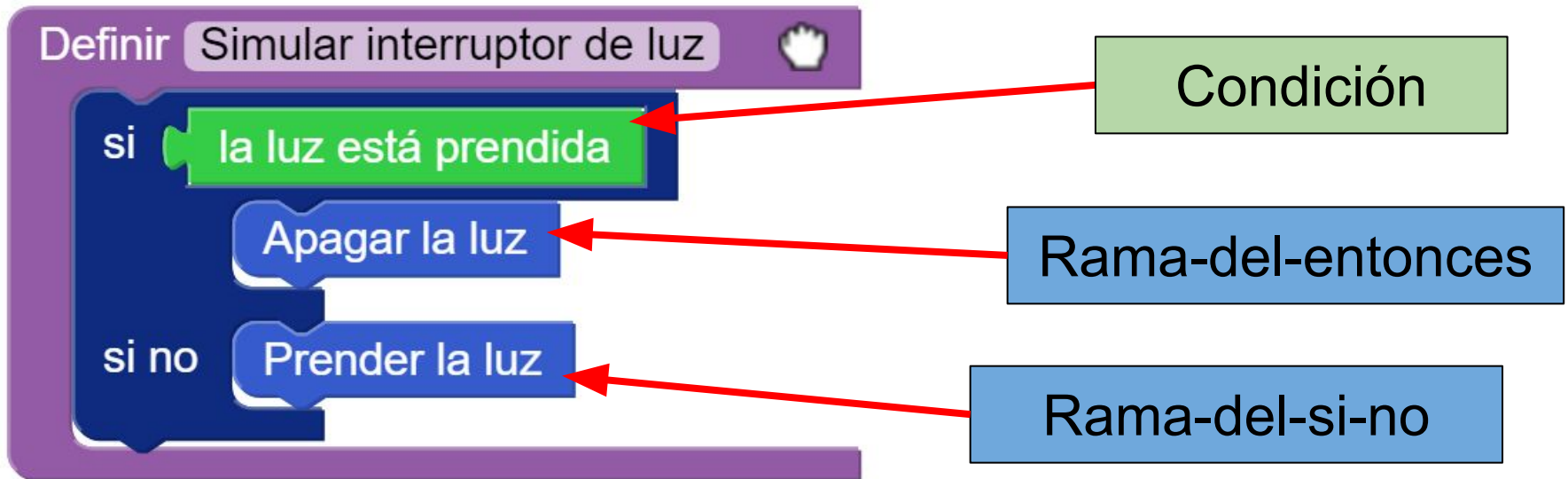
`hayBolitas(siguiente(Azul))`    `puedeMover(Norte)`

`puedeMover(opuesto(Norte))`

Cualquier expresión del tipo adecuado sirve como argumento



- Las **alternativas condicionales** se forman con
  - una expresión booleana (la **condición**) y
  - dos grupos de comandos
    - la **rama-del-entonces**
    - la **rama-del-si-no**





- En texto
  - la condición va entre paréntesis
  - los comandos de las ramas van entre llaves
  - se usan las palabras clave **if**, **then** y **else**
    - por eso se suele hablar de la alternativa como el comando **if-then-else**

```
if (laLuzEstáPrendida())  
  then { ApagarLaLuz() }  
  else { PrenderLaLuz() }
```

Condición

Ramas

```
if (laLuzEstáPrendida())  
  { ApagarLaLuz() }  
else { PrenderLaLuz() }
```

La palabra clave  
**then** es opcional

¡Observar la indentación!



- Cuando la alternativa es no hacer nada, se puede omitir la rama-del-si-no (también llamada **rama-del-else**)
  - En ese caso, la ejecución de la rama-del-entonces (o **rama-del-then**) depende de la condición
  - La llamamos **alternativa simple** (o en ocasiones, simplemente **condicional**)

¿Qué hace la computadora al ejecutar si la condición no se cumple?



```
if (puedeMover(Este))  
{ Mover(Este) }
```



- Como *regla general*, no es buena práctica eliminar las precondiciones utilizando alternativas
  - Programación *a-la-defensiva*

```
// ¿Es correcto este procedimiento?  
procedure DibujarLíneaDeLargo10() {  
    /*  
        PROPÓSITO: Dibujar una línea roja al Este de largo  
                   10 y dejar el cabezal al final de la misma  
        PRECONDICIÓN: ninguna (es una operación total)  
    */  
    repeat (10) { PintarYAvanzarSiSePuede() }  
}  
  
procedure PintarYAvanzarSiSePuede() {  
    /*  
        PROPÓSITO: Pintar una celda de Rojo y avanzar a la  
                   siguiente posición al Este, si se puede  
        PRECONDICIÓN: ninguna (es una operación total)  
    */  
    if (puedeMover(Este)) { Poner(Rojo) Mover(Este) }  
}
```

¿La línea  
queda de  
largo 10?  
¿Por qué?



- Las funciones parciales hacen lo esperado si la precondición se cumple
  - Debemos confiar en la precondición

```
// ¿Es correcto este procedimiento?  
procedure DibujarLíneaDeLargo10() {  
  /*  
    PROPÓSITO: Dibujar una línea roja al Este de largo  
               10 y dejar el cabezal al final de la misma  
    PRECONDICIÓN: hay al menos 10 celdas al Este  
  */  
  repeat (10) { PintaYAvanzar() }  
}  
  
procedure PintaYAvanzar() {  
  /*  
    PROPÓSITO: Pinta una celda de Rojo y avanzar a la  
               siguiente posición al Este  
    PRECONDICIÓN: hay al menos una celda al Este  
  */  
  Poner(Rojo) Mover(Este)  
}
```

¡Dibuja línea  
de largo 10,  
o falla!



- Al cambiar de nivel puede ser necesario cambiar el mensaje de error
  - Para eso es necesaria una alternativa
  - ¡Pero no cambia la parcialidad de la función!



¿Cómo sería el mensaje de error de Comer el queso, si no ponemos el BOOM?



- No es buena práctica anidar repeticiones y alternativas
  - Para evitarlo, ¡hay que definir subtareas!
  - Comparar esto...

Definir Salir del laberinto comiendo el queso



repetir 20 veces

si hay queso

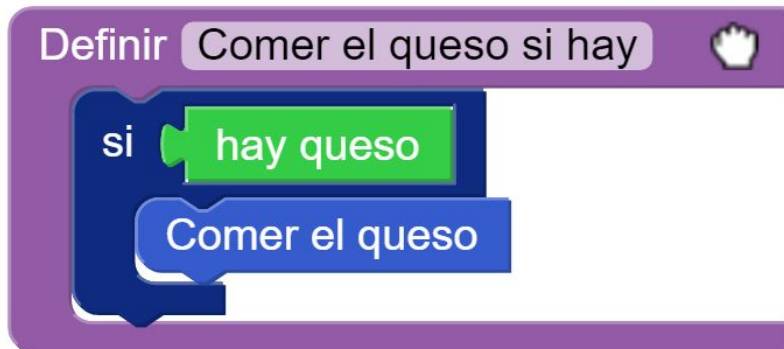
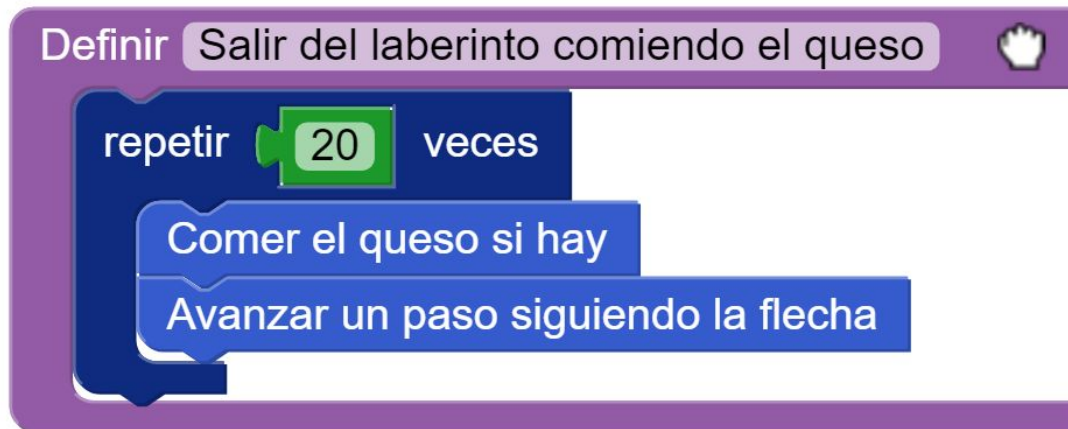
Comer el queso

Avanzar un paso siguiendo la flecha

¡Mucho código junto!



- No es buena práctica anidar repeticiones y alternativas
  - Para evitarlo, ¡hay que definir subtareas!
  - ...con esto

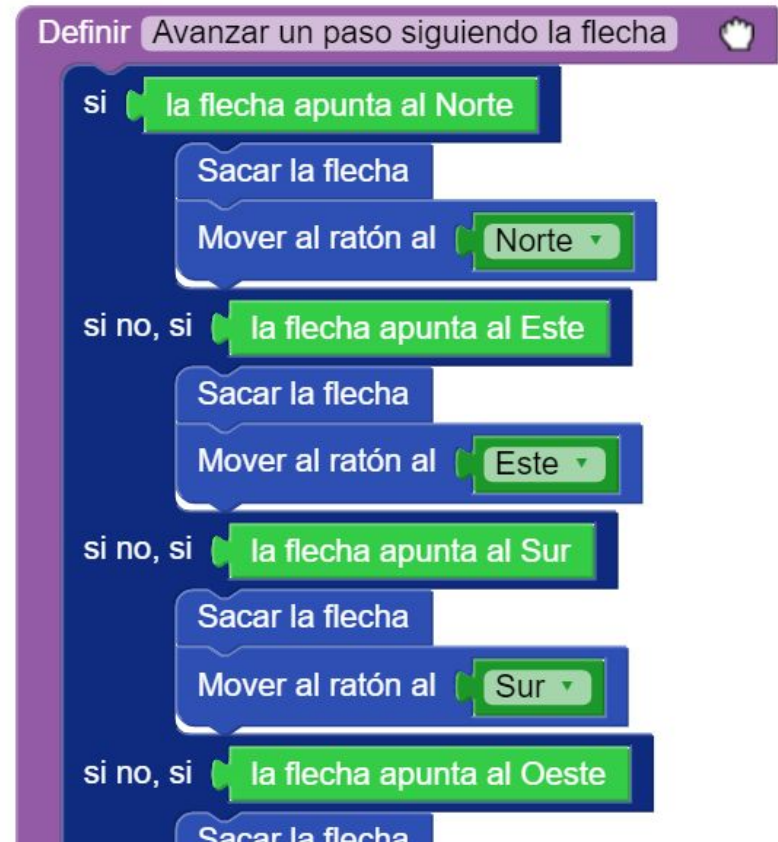
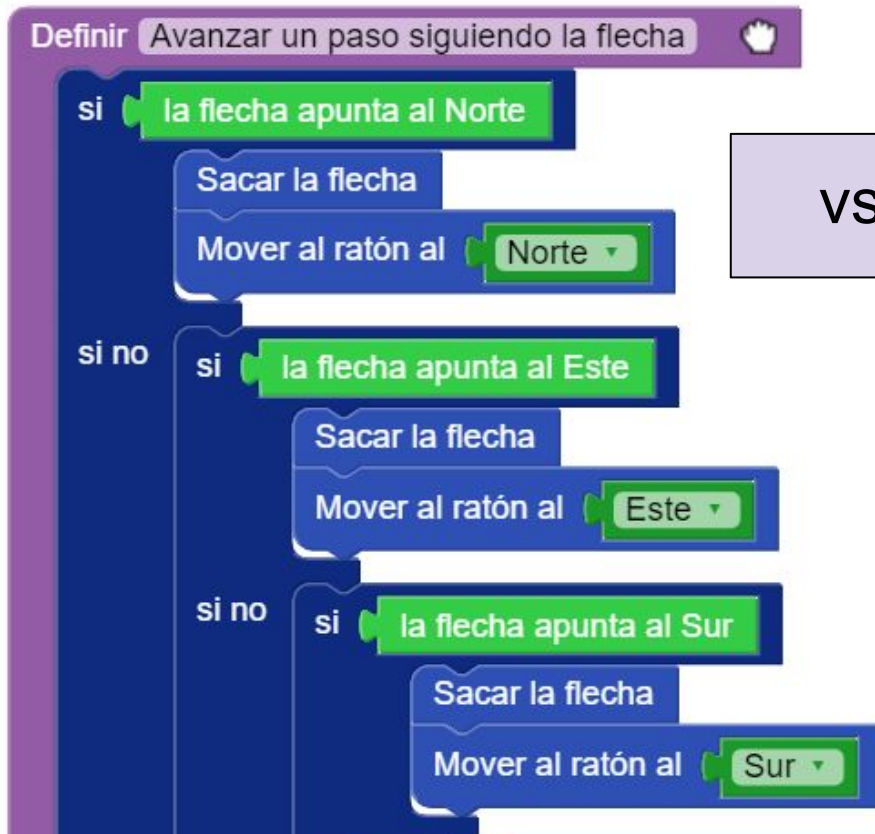


Se expresa  
adecuadamente la  
estrategia






- Tampoco es buena práctica anidar alternativas
  - Una opción es usar **multi-alternativas**...
  - ...y otra es usar condiciones más complejas






- Tampoco es bueno anidar una alternativa con otra
  - Una opción es usar **multi-alternativas...**
  - ...y otra es usar condiciones más complejas

```
procedure AvanzarUnPasoSiguiendoLaFlecha() {  
    /*  */  
    if (laFlechaApuntaAlNorte())  
    { SacarLaFlecha()  
      MoverAlRatónAl_(Norte)  
    }  
    else { if (laFlechaApuntaAlEste())  
           { SacarLaFlecha()  
             MoverAlRatónAl_(Este)  
           }  
          else { if (laFlechaApuntaAlSur())  
                  { SacarLaFlecha()  
                    MoverAlRatónAl_(Sur)  
                  }  
                  else { if (laFlechaApuntaAlOeste())  
                          { SacarLaFlecha()  
                            MoverAlRatónAl_(Oeste)  
                          }  
                          else { BOOM("No hay flecha")  
                                }  
                        }  
                    }  
                }  
    }  
}
```

VS

```
procedure AvanzarUnPasoSiguiendoLaFlecha() {  
    /*  */  
    if (laFlechaApuntaAlNorte())  
    { SacarLaFlecha()  
      MoverAlRatónAl_(Norte)  
    }  
    elseif (laFlechaApuntaAlEste())  
    { SacarLaFlecha()  
      MoverAlRatónAl_(Este)  
    }  
    elseif (laFlechaApuntaAlSur())  
    { SacarLaFlecha()  
      MoverAlRatónAl_(Sur)  
    }  
    elseif (laFlechaApuntaAlOeste())  
    { SacarLaFlecha()  
      MoverAlRatónAl_(Oeste)  
    }  
}
```

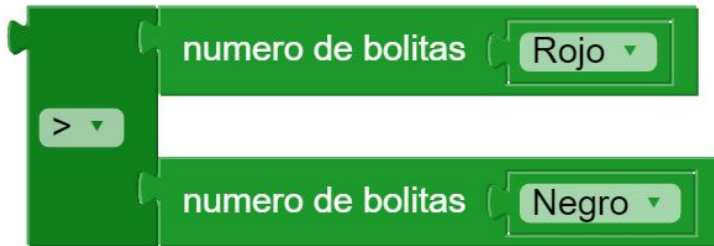




## Más expresiones booleanas



- ¿Cómo armar condiciones más complejas?
  - Una posibilidad es comparar otros valores
  - Para eso existen los **operadores de comparación**



```
nroBolitas(Rojo) > nroBolitas(Negro)
```

```
nroBolitas(Rojo) == 10
```



```
valorDelDominó <= 6
```



En texto se  
escriben infijos



- Los posibles operadores de comparación son
  - igual (`==`), distinto (`/=`),
  - menor o igual (`<=`), menor (`<`),
  - mayor o igual (`>=`), mayor (`>`)



✓ `==`  
`/=`  
`<=`  
`<`  
`>=`  
`>`

¿Cuándo es verdadero o falso cada uno?

¿Qué tipo de argumentos esperan?



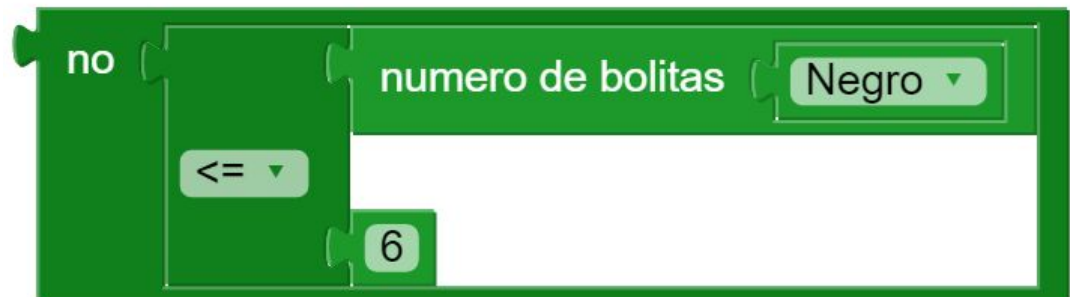
- ¿Cómo armar condiciones más complejas?
  - Otra posibilidad es negar la condición
  - Para eso existe el ***operador de negación***



`not hayBolitas(Negro)`

`not (nroBolitas(Negro) <= 6)`

En texto se  
escribe prefijo



- El operador de negación
  - Recibe una expresión de tipo Bool como argumento
  - ¡Puede usarse con cualquier condición!
  - Devuelve el valor de verdad contrario al dado
    - **Niega** el valor de verdad dado



Cualquier expresión de  
tipo Bool



- ¿Cómo armar condiciones más complejas?
  - Una tercera posibilidad es combinar dos condiciones
  - Para eso existen los **operadores lógicos**
    - la **conjunción** (y también, &&) y
    - la **disyunción** (o bien, ||)

Son operadores binarios con argumentos de tipo Booleano

¡Puede usarse cualquier condición como argumento!



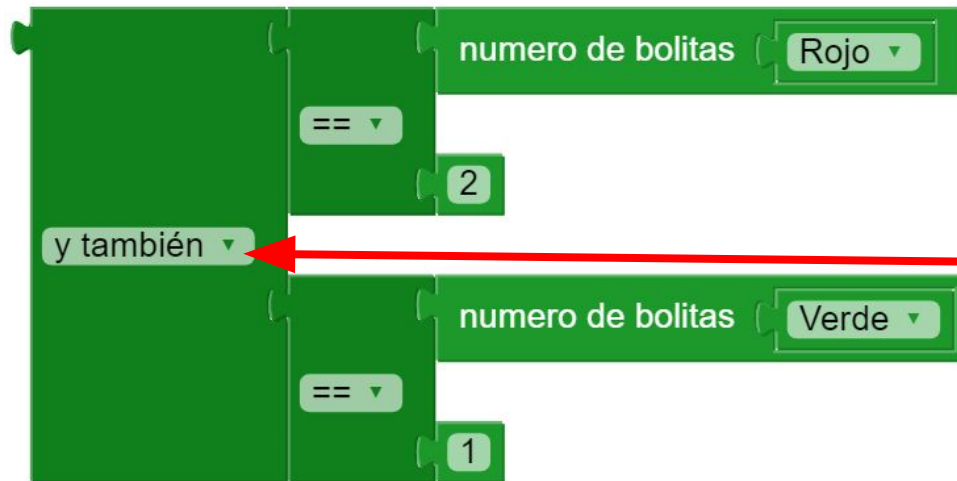
&&



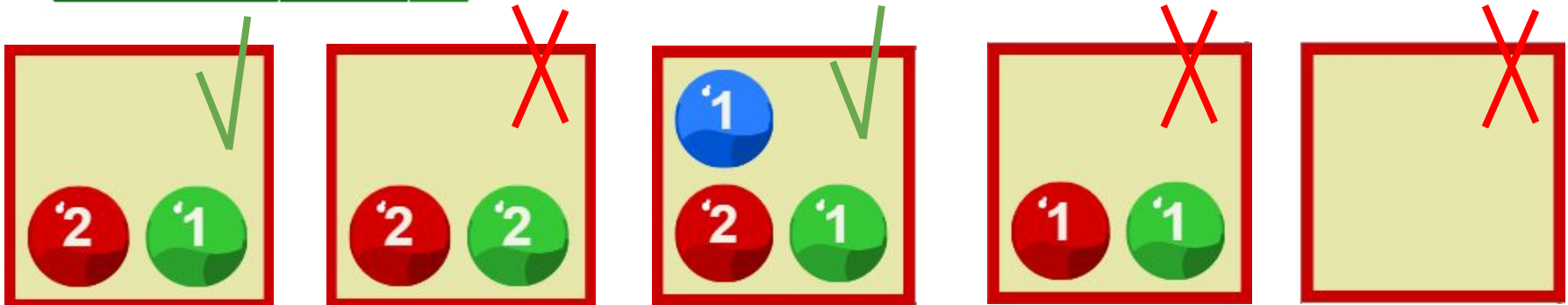
||



- Uno de los operadores lógicos es la **conjunción**
  - Ambas condiciones tienen que ser verdaderas
  - Si alguna es falsa, todo es falso



¿Cuándo es verdadera?

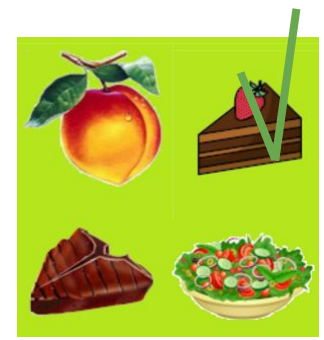
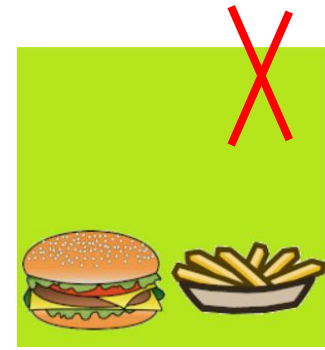




- El otro de los operadores lógicos es la **disyunción**
  - Al menos una tiene que ser verdadera
  - Solo es falso si ambas son falsas



¿Qué pasa si ambas son verdaderas?



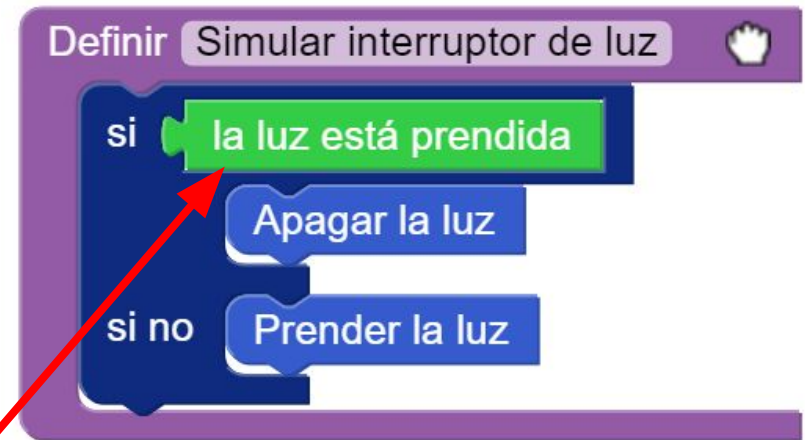
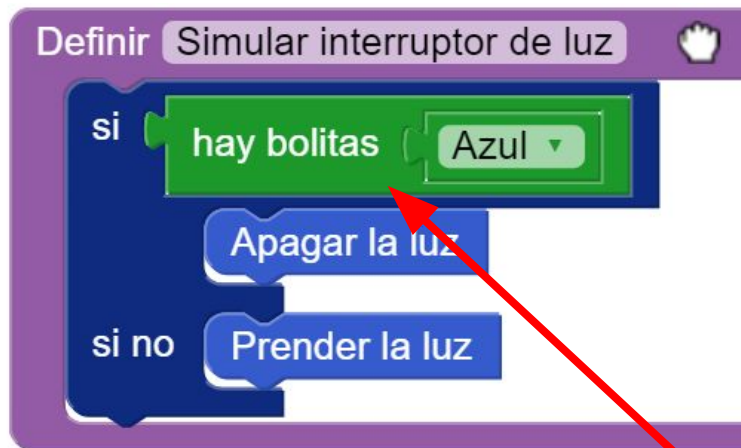




# Funciones simples



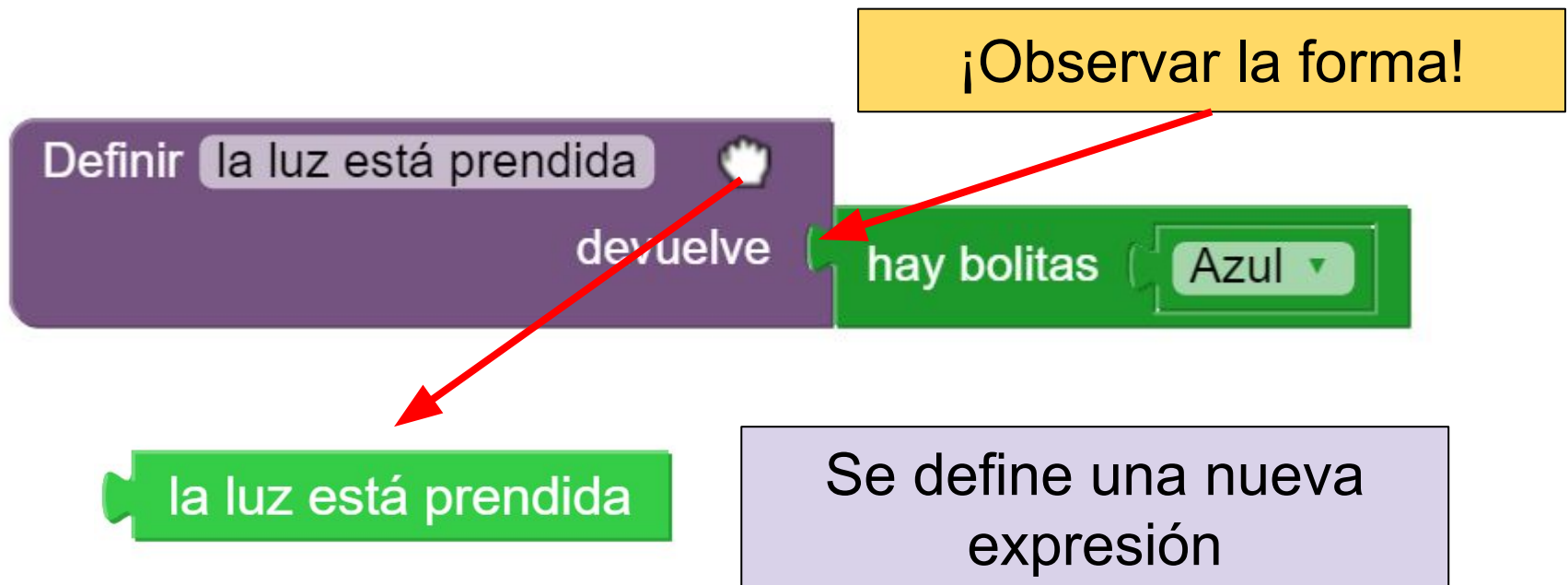
- ¿Cómo hacer condiciones que hablen del problema y no de la representación?
  - Hace falta una nueva **herramienta del lenguaje**



¿Cuál de las dos es más legible?

- Las ***funciones simples***

- Describen el valor de una expresión dada
- Las define el usuario, eligiendo su **nombre**
- Son el equivalente a los procedimientos, pero en el mundo de las expresiones



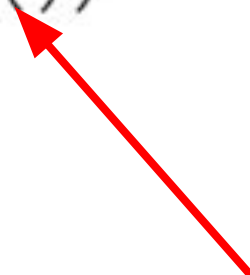


- En texto las funciones simples
  - Se indican con la palabra clave **function**
  - El nombre empieza con minúscula
  - Tienen un cuerpo que solo tiene la palabra clave **return** seguida de una expresión entre paréntesis
  - ¡Hay que escribir su contrato!

```
function laLuzEstáPrendida() {  
  /*  
    PROPÓSITO: indica si hay una luz prendida en la celda actual  
    PRECONDICIÓN: ninguna (es una operación total)  
  */  
  return (hayBolitas(Azul))  
}
```

- En texto las funciones simples
  - Se utilizan como cualquier otra expresión
  - Deben estar seguidas de paréntesis
  - Es decir, tiene sentido usarla como argumento (de comandos, procedimientos, funciones, operadores)
    - ¡Incluso puede ser argumento de un return!

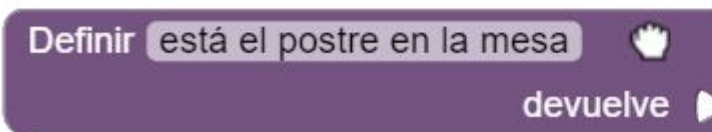
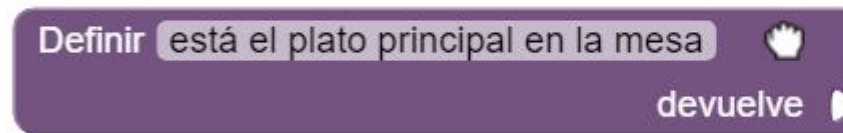
```
if (laLuzEstáPrendida())  
    { ApagarLuz() }  
else { PrenderLuz() }
```



El llamado es condición  
para la alternativa



- Las funciones simples
  - Permiten descomponer una expresión compleja en partes más simples
  - Aportan legibilidad



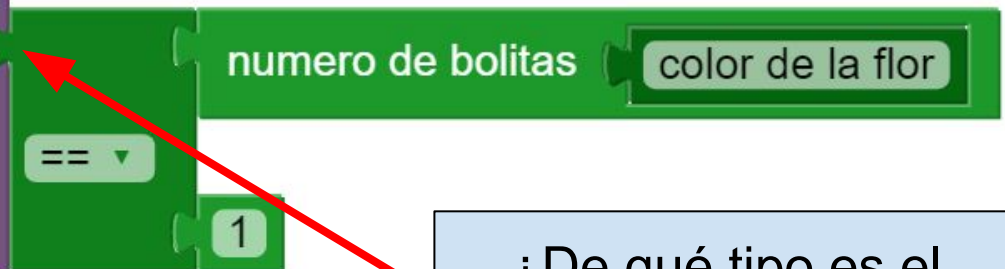
Acá se pueden definir condiciones tan complejas como haga falta



- La definición de una función puede tener parámetros
  - Funcionan exactamente igual que en procedimientos
    - Se deben nombrar de la misma forma
    - Deben proveerse argumentos al llamarla
    - Debe haber concordancia en cantidad, orden y tipo

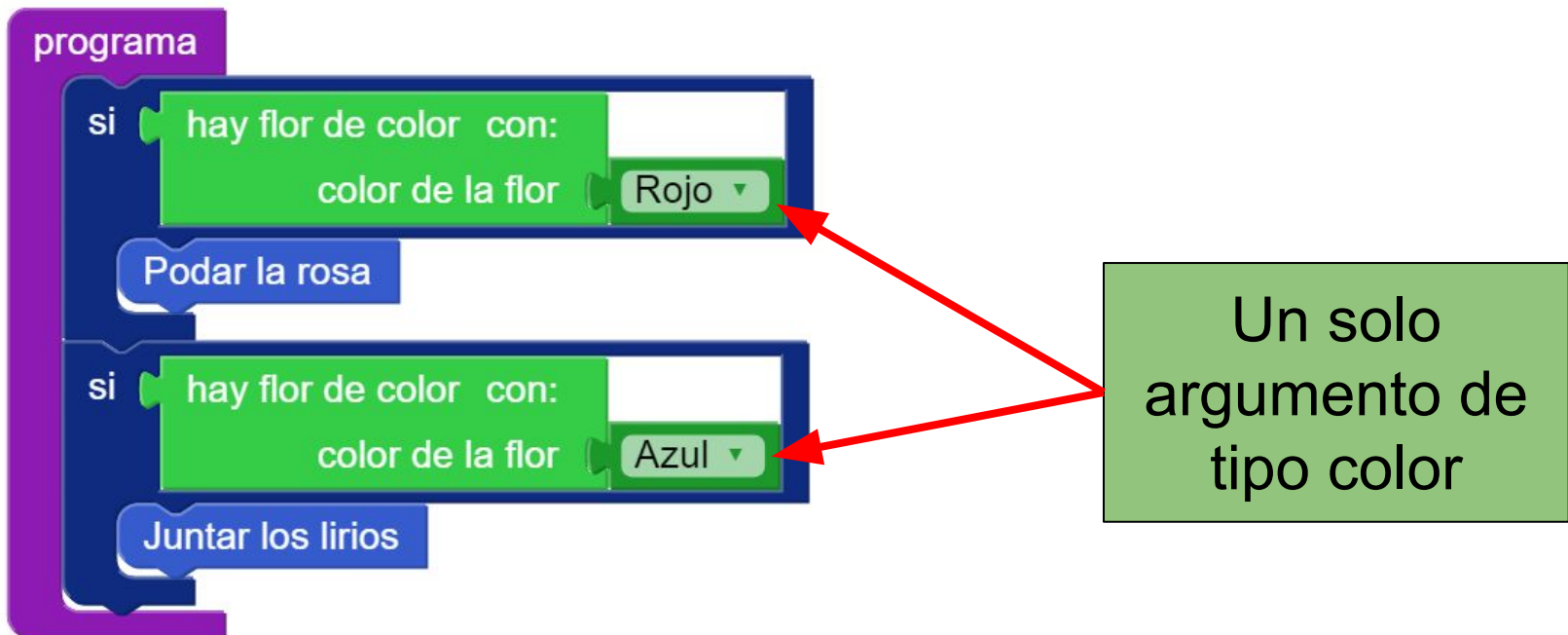


¿De qué tipo es el argumento que se espera?



¿De qué tipo es el resultado?

- La definición de una función puede tener parámetros
  - Funcionan exactamente igual que en procedimientos
    - Se deben nombrar de la misma forma
    - Deben proveerse argumentos al llamarla
    - Debe haber concordancia en cantidad, orden y tipo







- Las funciones simples pueden ayudar a la legibilidad
  - Expresando elementos
  - Expresando condiciones en términos del problema
  - Expresando otras expresiones complejas

Definir manzana devuelve Rojo

Definir arándano devuelve Azul

Definir uva devuelve Verde

Definir nuez devuelve Negro

Definir no hay +  
con fruta -  
devuelve

Definir hay solamente una manzana devuelve

¿Cómo definir las?



# Cierre



- ***Alternativa condicional***
  - Herramienta del lenguaje para elegir entre 2 posibles comportamientos del programa
  - Se basa en condiciones dadas por ***expresiones booleanas***
  - Tiene 2 grupos de comandos, las ***ramas*** (la *rama-del-si-no* puede omitirse)
  - Pueden agregarse más condiciones y más ramas, para tener ***multialternativas***
  - En texto se usan las palabras claves ***if***, ***then***, ***else*** y ***elseif*** para construirlas

- **Expresiones booleanas**

- Describen valores de un nuevo tipo de datos: los ***Valores de verdad***, o ***Booleanos***
- Pueden describir **verdadero** o **falso**
- Hay expresiones primitivas de este tipo
  - **hayBolitas** (*<color>*),  
describe la precondition de **Sacar**
  - **puedeMover** (*<dirección>*),  
describe la precondition de **Mover**



- **Expresiones booleanas**

- Se pueden hacer operaciones booleanas más complejas usando operadores
- ***Operadores de comparación***
  - para comparar números, direcciones y colores por igualdad (`==`, `\=`), mayor (`>`, `>=`) y menor (`<`, `<=`)
- ***Operador de negación***
  - para negar una condición (`not`)
- ***Operadores lógicos***
  - para combinar dos condiciones con conjunción (y también, `&&`) o disyunción (o bien, `||`)



- ***Funciones***

- Permiten nombrar expresiones complejas
- Son el equivalente a los procedimientos en el mundo de las expresiones
- Pueden tener parámetros
  - los argumentos deben coincidir en cantidad, orden y tipo con los parámetros
- Se clasifican según el tipo de su resultado
- Proveen abstracción y legibilidad para expresiones