



Introducción a la Programación

Clases teóricas

por Pablo E. “Fidel” Martínez López

6. Repetición condicional



Repaso test



- **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



- **Expresiones**

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



- **Tipos de datos**

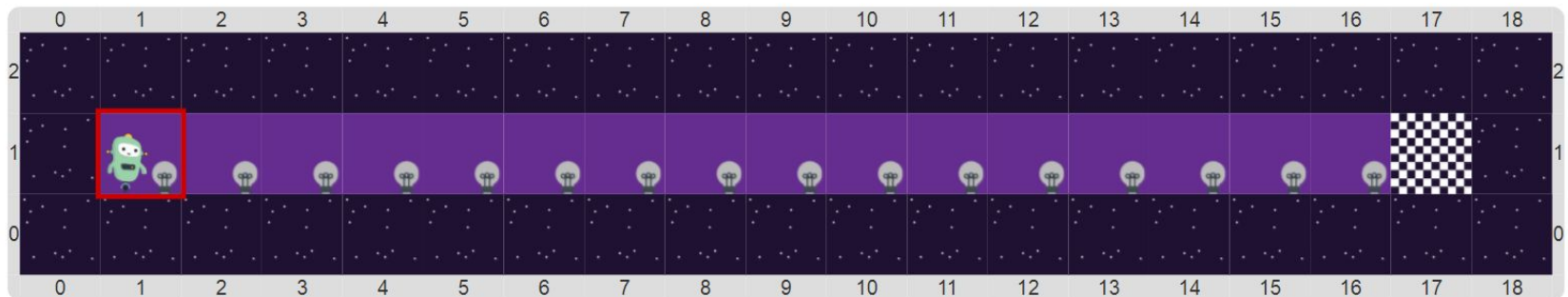
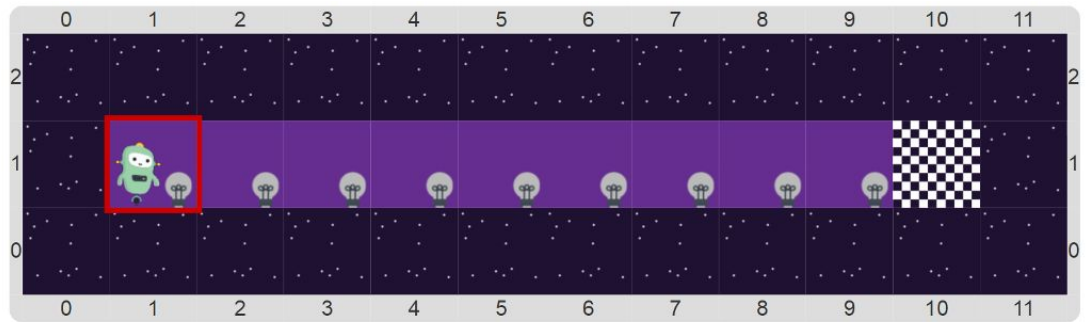
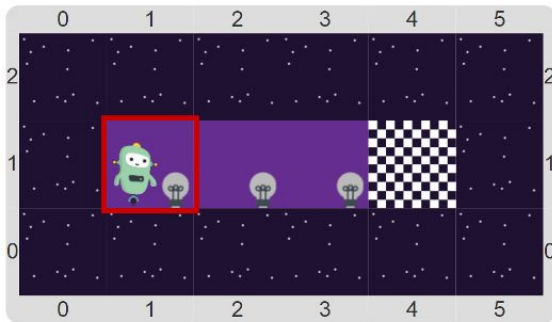
- permiten clasificar expresiones
- en Gobstones, por ahora, son cuatro
 - colores, direcciones, números y valores de verdad
- toda expresión tiene un tipo
- los parámetros deben especificar qué tipo de expresiones aceptan



Repeticiones condicionales

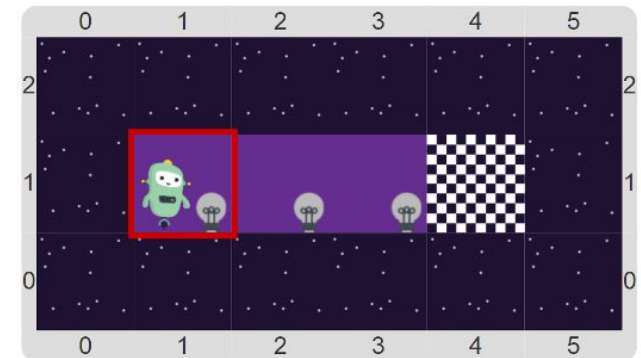
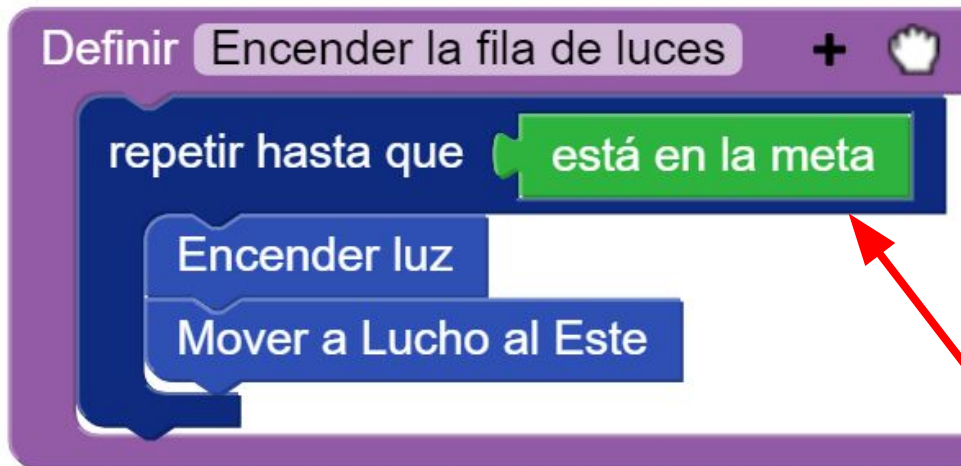


- ¿Cómo manejamos escenarios cambiantes, cuando lo que cambia es la distancia a la que está cierto elemento?
 - Se debe **repetir** la acción **hasta que** lleguemos al elemento
- Precisamos una herramienta nueva...





- La **repetición condicional** es una forma de armar comandos que permite hacer eso
 - La **condición** establece cuándo debe dejar de repetirse la acción indicada



¿Cuándo es verdadera esta condición?



- La **repetición condicional** se arma con
 - una expresión de tipo Bool (la **condición** de finalización)
 - en texto, entre paréntesis
 - un grupo de comandos (el **cuerpo** de la repetición)
 - en texto, entre llaves

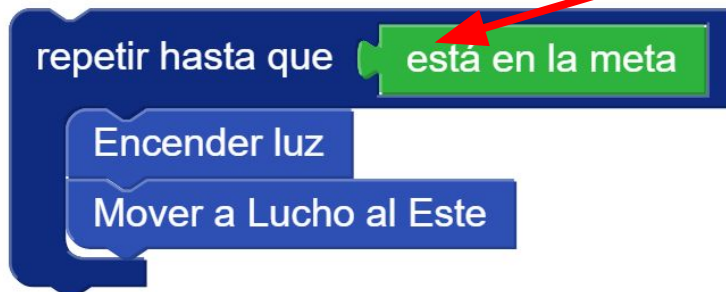


El cuerpo indica la acción a repetir
La condición indica cuándo debe finalizar la repetición



- En bloques, usamos la variante ***repetir-hasta-que***
- En texto, usamos la variante ***while*** (*mientras*)
 - Una pregunta cuándo terminar y la otra cuándo seguir
 - Por eso la condición de una aparece negada en la otra (repite *hasta que* terminó, o *mientras NO* terminó)


Comparar repetir
hasta que está en la meta
vs.
mientras no está en la meta



```
while (not estáEnLaMeta()) {
  EncenderLuz()
  MoverALuchoAlEste()
}
```



- La condición se vuelve a evaluar luego de cada repetición
 - Esto puede llevar a situaciones donde el programa NO TERMINA nunca pero es difícil darse cuenta
 - Es una situación de falla nueva

```
procedure LaBuenaPipa() {  
    /*  */  
    QuerésQueTeCuentoElCuentoDeLaBuenaPipa?()  
    while (respuesta()==respuesta()) {  
        YoNoTeDije-respuesta-TeDijeSiQuerésQueTeCuentoElCuentoDeLaBuenaPipa?()  
    }  
}
```

¿Cuándo termina este juego?
¡La condición siempre es verdadera!



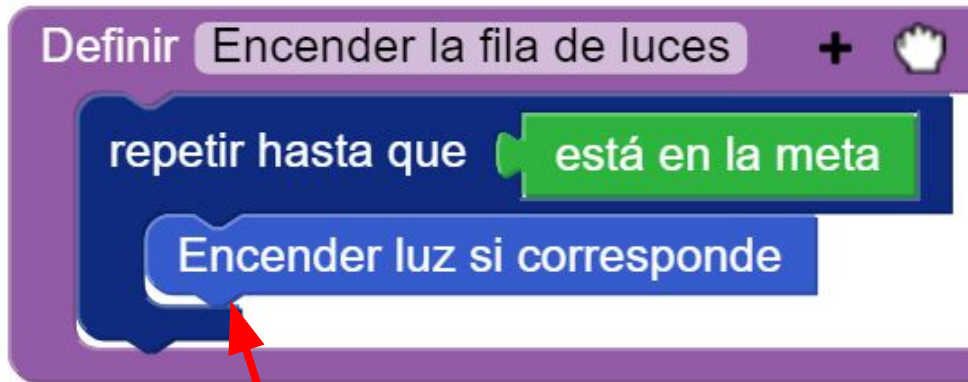
- A esta forma de falla se la considera similar a un BOOM
- La precondition **debe** tener en cuenta estos casos
- Puede darse en condiciones sutiles

¿Cuándo termina `IrALaEsquina__(Norte, Sur)`?
¿Existe la esquina Norte-Sur?

```
procedure IrALaEsquina__(dirección1, dirección2) {  
    /*  
        PROPÓSITO: Ubicar el cabezal en la esquina descrita por las  
                   dos direcciones dadas.  
        PRECONDICIONES: Las direcciones dadas no son opuestas ni iguales.  
        PARÁMETROS: dirección1 y dirección2 son direcciones.  
        OBSERVACIÓN: Si las direcciones son opuestas, NUNCA TERMINA.  
    */  
    while (puedeMover(dirección1) || puedeMover(dirección2)) {  
        Mover_SiPuede(dirección1) Mover_SiPuede(dirección2)  
    }  
}
```



- La no-terminación puede aparecer porque olvidamos hacer algo, o porque no consideramos todos los casos



¡Olvidamos
mover a
Lucho!



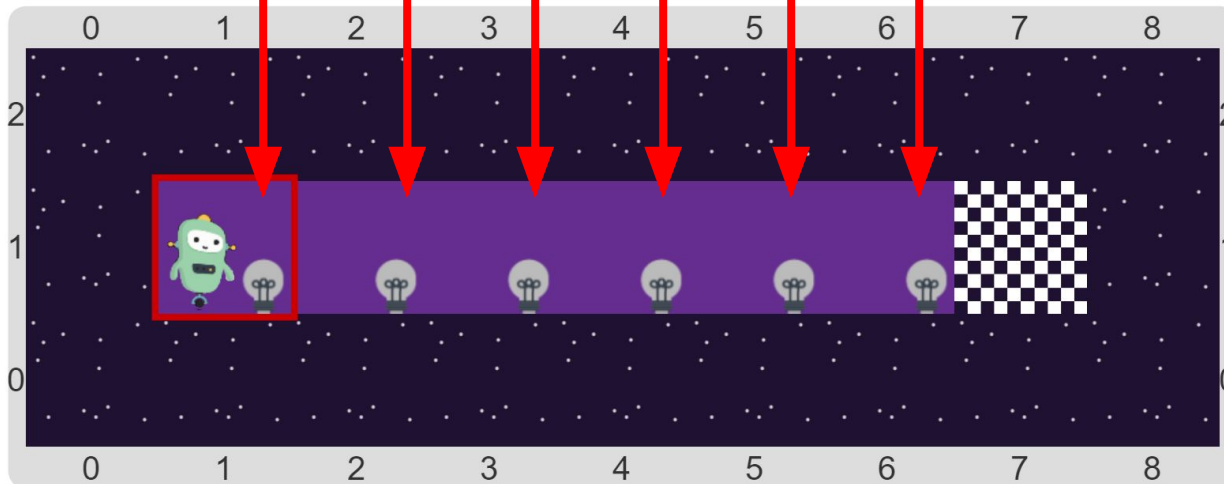
BOOM

La ejecución del programa
demoró más de 3000ms.

- ¿Cómo asegurar que una repetición condicional termina?
 - Hay muchas técnicas
 - Nosotros usaremos una simple: la idea de **recorrido**
 - Se basa en una secuencia finita de “elementos”

Secuencia de 6
celdas con luz

“No pregunto cuántos son,
sino que vayan saliendo”





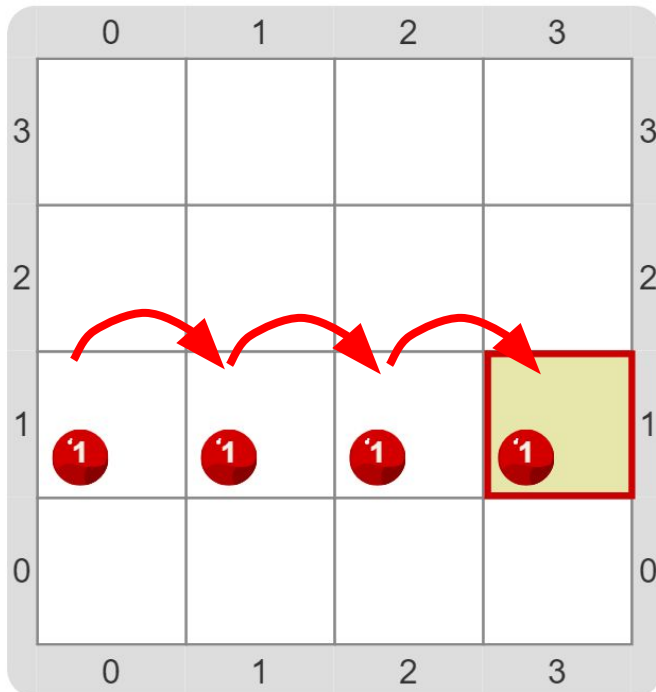
Recorridos

- Un **recorrido** es una forma de dividir en subtarear
 - en problemas con una secuencia finita de “elementos”
 - para asegurar que todos los elementos son procesados
- Involucra definir 5 subtarear (los nombres pueden variar)

```
procedure RecorridoGenérico() {  
    /*  
        PROPÓSITO:  
        * procesar todos los elementos de una  
        secuencia de elementos determinada  
        PRECONDICIÓN: según el problema  
    */  
    IniciarRecorrido()  
    while (quedanElementosParaProcesar()) {  
        ProcesarElementoActual()  
        PasarAlSiguienteElemento()  
    }  
    FinalizarRecorrido()  
}
```

Muchos
problemas
involucran
secuencias de
elementos

- El recorrido más simple es procesar las celdas de una fila (o una columna)
 - los “elementos” son las celdas, que están una al lado de otra
 - **PasarAlSiguienteElemento** es simplemente moverse a la celda lindante en la dirección dada





- El recorrido más simple es procesar las celdas de una fila (o una columna)
 - los “elementos” son las celdas, que están una al lado de otra
 - **PasarAlSiguienteElemento** es simplemente moverse a la celda lindante en la dirección dada

```
procedure PonerBolitasEnTodaLaFila() {  
  /* PROPÓSITO:  
    * Poner un bolita roja en cada celda de la  
      fila actual.  
    * Ubicar el cabezal en la celda más al Este  
      de la fila actual.  
    PRECONDICIONES: Ninguna (es una operación total).  
  */  
  IrAlBorde(Oeste)           // IniciarRecorrido()  
  while (puedeMover(Este)) { // quedanElementosParaProcesar()  
    Poner(Rojo)               // ProcesarElementoActual()  
    Mover(Este)               // PasarAlSiguienteElemento()  
  }  
  Poner(Rojo)                // FinalizarRecorrido()  
}
```

Observar la
estructura de
recorrido



- Otro recorrido sencillo es procesar un camino simple
 - cada parte del camino tiene dos partes del camino lindantes
 - **PasarAlSiguienteElemento** es moverse a la celda lindante que es parte del camino y no se visitó

Definir Salir del laberinto comiendo el queso + 🐾

repetir hasta que **estoy en la salida**

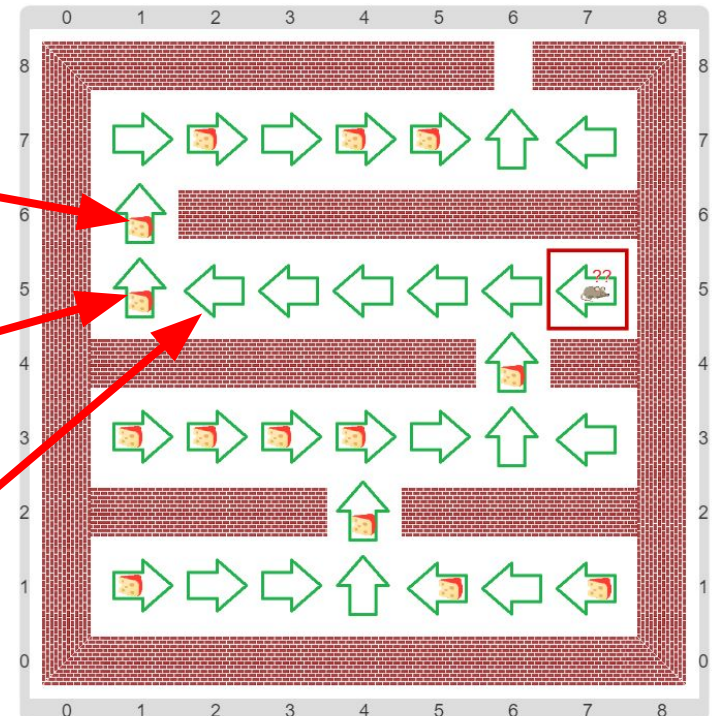
Comer el queso si hay

Avanzar un paso siguiendo la flecha

Después

Cada posición del camino tiene dos partes lindantes

Antes





- Otro recorrido sencillo es procesar un camino simple
 - cada parte del camino tiene dos partes del camino lindantes
 - **PasarAlSiguienteElemento** es moverse a la celda lindante que es parte del camino y no se visitó


```
procedure SalirDelLaberintoComiendoElQueso() {  
  /* PROPÓSITO: Sacar al ratón del laberinto, comiendo  
    el queso que encuentre por el camino.  
    PRECONDICIONES: Hay un escenario correctamente  
      representado { ... }  
    OBSERVACIÓN: Los elementos a recorrer son las  
      posiciones del camino hasta la salida, indicadas  
      por las flechas.  
  */
```

```
while (not estoyEnLaSalida()) {  
  ComerElQuesoSiHay()  
  AvanzarUnPasoSiguiendoLaFlecha  
}  
}
```

```
// NO HAY IniciarRecorrido()  
// quedanElementosParaProcesar()  
// ProcesarElementoActual()  
// PasarAlSiguienteElemento()  
// NO HAY FinalizarRecorrido()
```

El código
para pasar al
siguiente
elemento
requiere más
trabajo

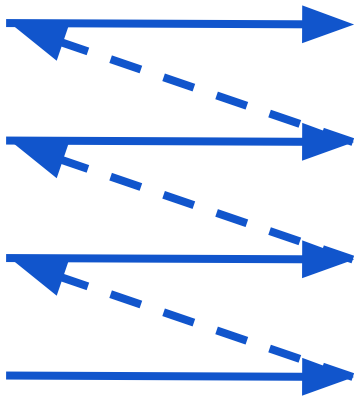
- Otro recorrido sencillo es procesar un camino simple
 - cada celda tiene una antes y una después
 - **PasarAlSiguienteElemento** es moverse a la celda lindante que es parte del camino y no se visitó

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

El código
para pasar al
siguiente
elemento
requiere más
trabajo

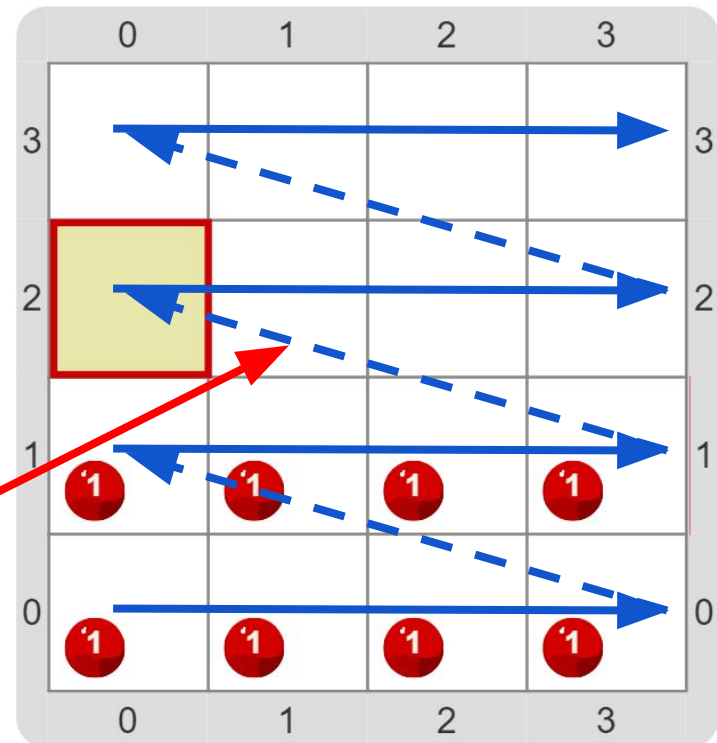


- Un recorrido más complicado es recorrer todas las celdas del tablero, en cierto orden
 - **PasarAlSiguienteElemento** debe determinar si sigue en la misma fila, o empieza con otra



Recorrido por celdas,
al Este y al Norte

¡La celda siguiente a la última de una fila, es la primera de la fila siguiente!





- Un recorrido más complicado es recorrer todas las celdas del tablero, en cierto orden
 - **PasarAlSiguienteElemento** debe determinar si sigue en la misma fila, o empieza con otra

```
procedure PonerUnaBolitaEnCadaCeldaDelTablero() {  
  /* PROPÓSITO:  
   * Poner un bolita roja en cada celda del tablero.  
   * Ubicar el cabezal en la esquina NorEste.  
   PRECONDICIONES: Ninguna (es una operación total).  
   OBSERVACIÓN: Es un recorrido sobre todas las celdas  
     del tablero, en dirección Este y Norte.  
  */  
  IrAlBorde(Sur) IrAlBorde(Oeste)    // IniciarRecorrido()  
  while (puedeMover(Este)  
    || puedeMover(Norte)) {          // quedanElementosParaProcesar()  
    Poner(Rojo)                       // ProcesarElementoActual()  
    PasarASiguienteCeldaDelTablero() // PasarAlSiguienteElemento()  
  }  
  Poner(Rojo)                        // FinalizarRecorrido()  
}
```

Acá también
pasar al siguiente
elemento requiere
algo de trabajo



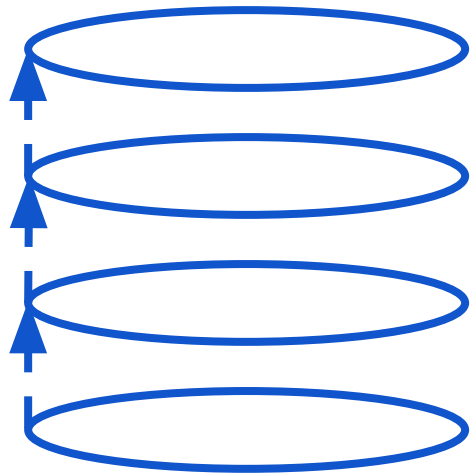
- Un recorrido más complicado es recorrer todas las celdas del tablero, en cierto orden
 - **PasarAlSiguienteElemento** debe determinar si sigue en la misma fila, o empieza con otra

```
procedure PasarASiguienteCeldaDelTablero() {  
  /* PROPÓSITO: Ubicar el cabezal en la celda  
    siguiente a la actual en un recorrido  
    por celdas en dirección Este y Norte.  
    PRECONDICIONES: La celda actual no es  
    la esquina NorEste.  
    OBSERVACIÓN: Es una de las operaciones del  
    recorrido.  
  */  
  if (puedeMover(Este))  
    then { Mover(Este) }  
    else {  
      // Puede mover al Norte, por la precondición  
      Mover(Norte) IrAlBorde(Oeste)  
    }  
}
```

Acá también
pasar al siguiente
elemento requiere
algo de trabajo

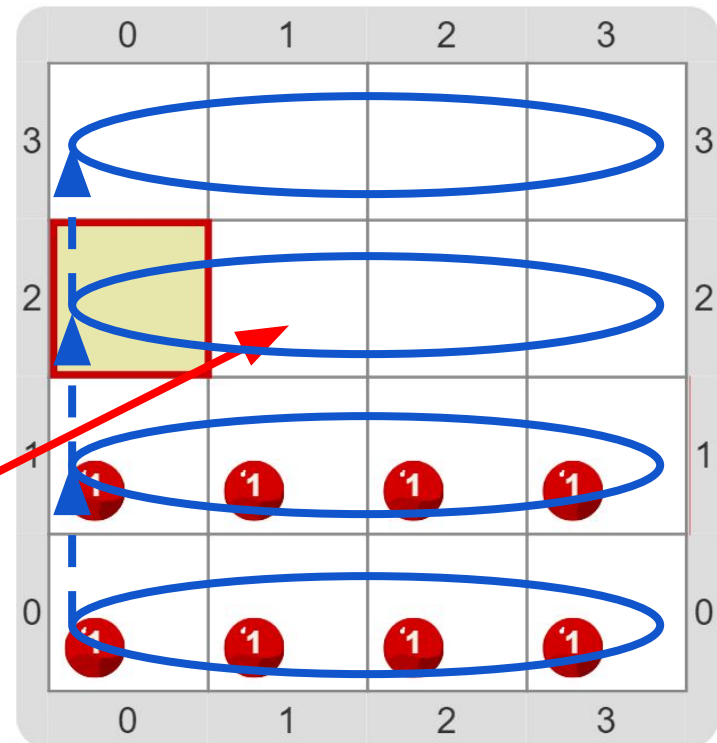


- Los elementos a recorrer no tienen por qué ser celdas
 - Podemos hacer un recorrido por filas, o por columnas



Recorrido por filas
al Norte


¡Cada elemento es
TODA una fila!





- Los elementos a recorrer no tienen por qué ser celdas
 - Podemos hacer un recorrido por filas, o por columnas

```
procedure PintarCadaCeldaDelTablero() {  
  /* PROPÓSITO:  
    * Poner un bolita roja en cada celda del tablero.  
    * Ubicar el cabezal en la esquina NorEste.  
    PRECONDICIONES: Ninguna (es una operación total).  
    OBSERVACIÓN: Es un recorrido por filas  
                  del tablero, en dirección Este y Norte.  
  */  
  IrAlBorde(Sur)                // IniciarRecorrido()  
  while (puedeMover(Norte)) {   // quedanElementosParaProcesar()  
    PintarFilaActual()          // ProcesarElementoActual()  
    Mover(Norte)                // PasarAlSiguienteElemento()  
  }  
  PintarFilaActual()            // FinalizarRecorrido()  
}
```



¡Acá, **ProcesarElemento** también involucra un recorrido!



- Los elementos a recorrer no tienen por qué ser celdas
 - Podemos hacer un recorrido por filas, o por columnas

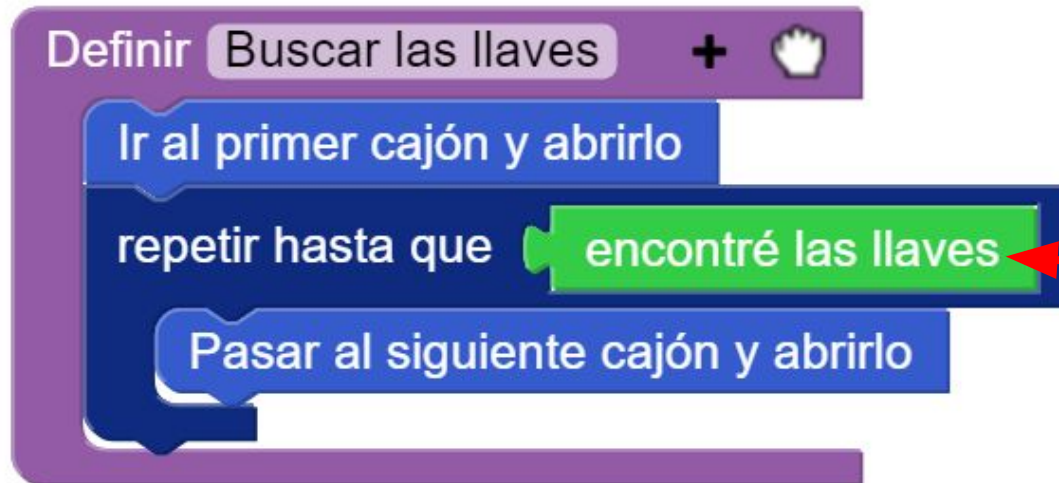
```
procedure PintarFilaActual() {  
  /* PROPÓSITO:  
    * Poner un bolita roja en cada celda de la fila actual.  
    * Ubicar el cabezal en la celda más al Este de la fila actual.  
    PRECONDICIONES: Ninguna (es una operación total).  
    OBSERVACIÓN: Es un recorrido por celdas de la fila,  
                 en dirección Este.  
  */  
  IrAlBorde(0este)  
  while (puedeMover(Este)) {  
    Poner(Rojo)  
    Mover(Este)  
  }  
  Poner(Rojo)  
}
```

```
  // IniciarRecorrido()  
  // quedanElementosParaProcesar()  
  // ProcesarElementoActual()  
  // PasarAlSiguienteElemento()  
  
  // FinalizarRecorrido()
```

¡Acá, **ProcesarElemento** también involucra un recorrido!



- En ocasiones, el recorrido debe terminar antes
 - Hablamos de un **recorrido de búsqueda**
 - Se detiene cuando se encontró lo que se buscaba
 - ¿Qué pasa si no está lo que buscamos?

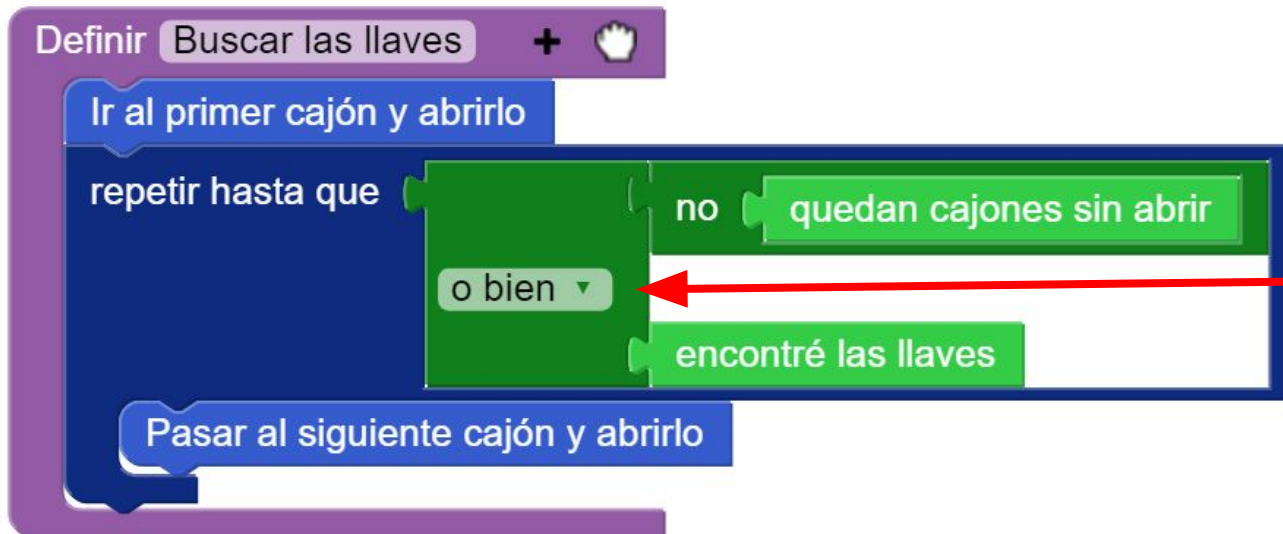


¡Si las llaves no están, va a explotar cuando no haya más cajones!

La precondition debería pedir que las llaves estén...



- En ocasiones, el recorrido debe terminar antes
 - Hablamos de un **recorrido de búsqueda**
 - Se detiene cuando se encontró lo que se buscaba
 - ¿Qué pasa si no está lo que buscamos?



Ahora no falla, porque si no hay llaves, frena al final

...debe controlarse que queden cajones

- En ocasiones, el recorrido debe terminar antes
 - Hablamos de un **recorrido de búsqueda**
 - Se detiene cuando se encontró lo que se buscaba
 - ¿Qué pasa si no está lo que buscamos?

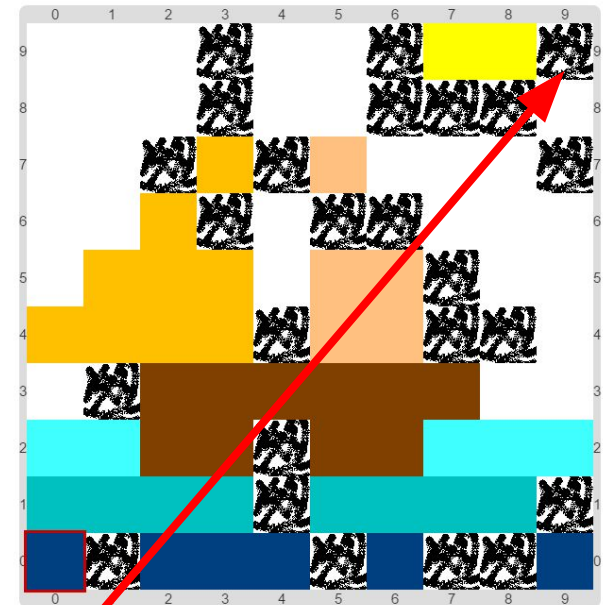
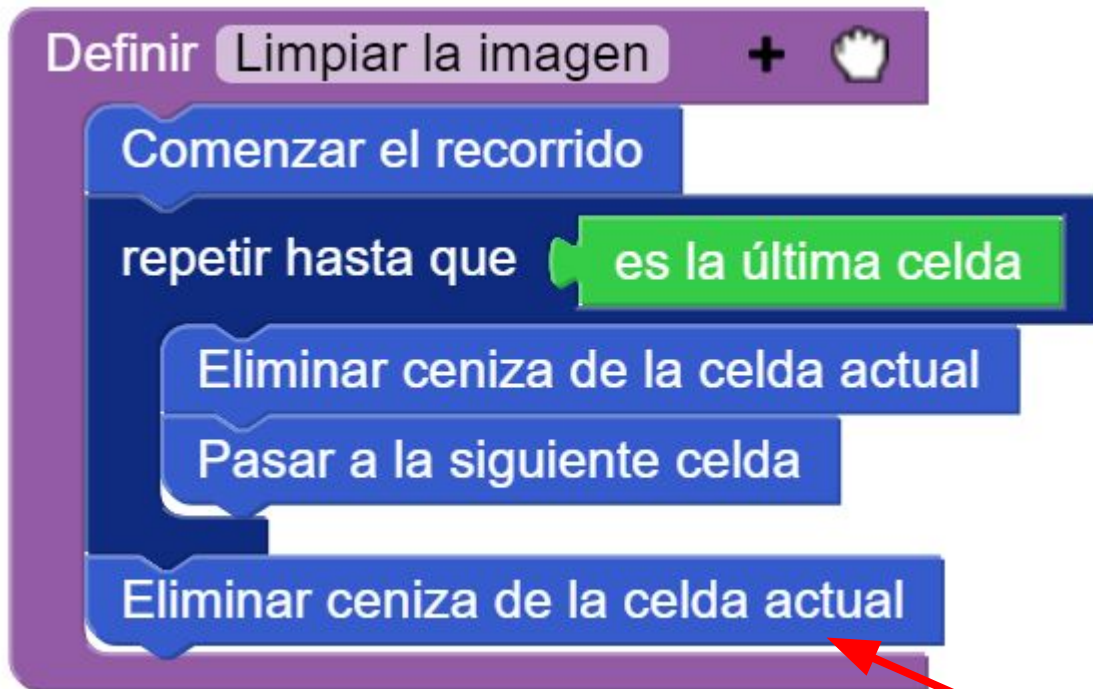
Ahora no falla, porque si no hay llaves, frena al final

```
procedure BuscarLasLlaves() {  
  /* ... */  
  IrAlPrimerCajónYAbrirlo()  
  while (quedanCajonesSinAbrir() && // IniciarRecorrido()  
        not encontréLasLlaves()) { // quedanElementos()  
    PasarAlSiguienteCajónYAbrirlo() // noEncontréLoQueBusco()  
  } // PasarAlSiguiente()  
}
```

...debe controlarse que queden cajones



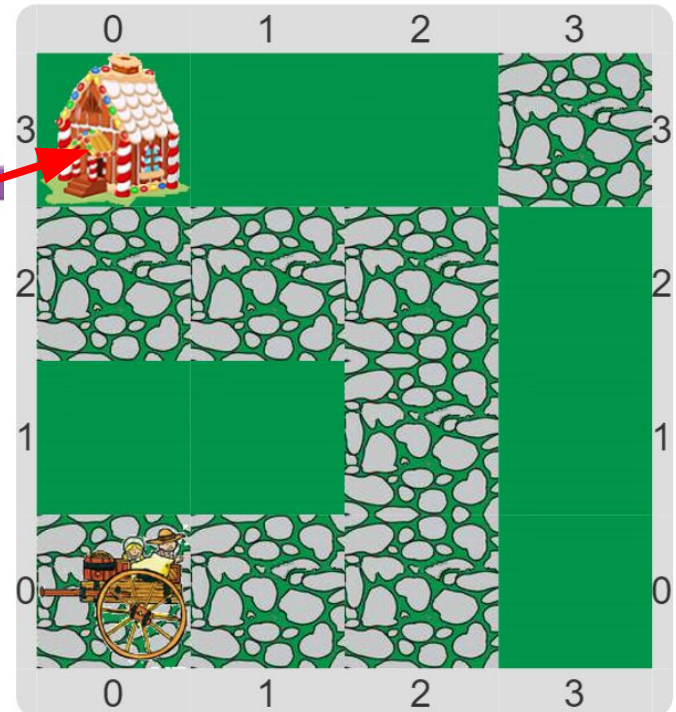
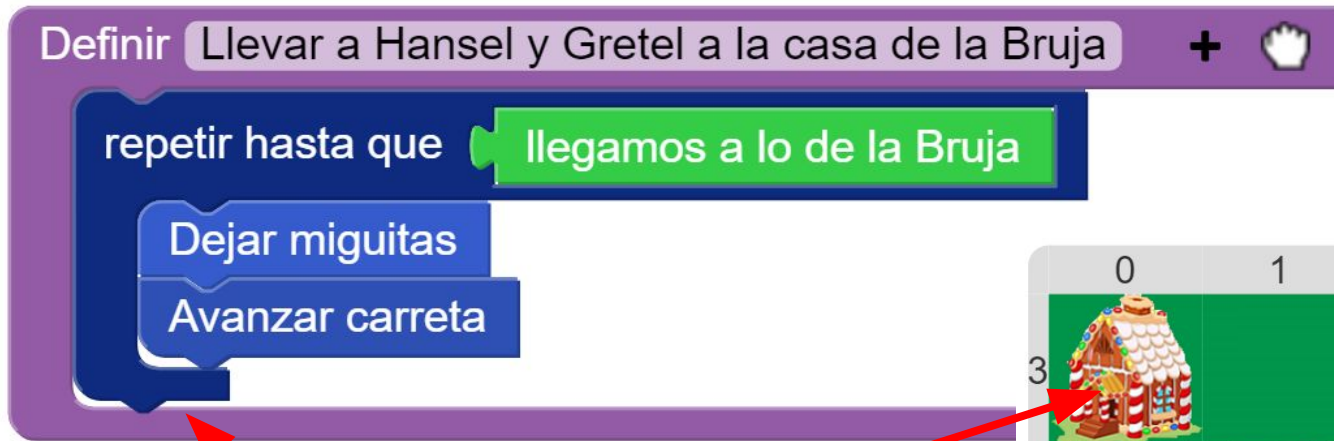
- En algunos recorridos hay que considerar casos de borde...



Hay que moverse una vez menos que la cantidad de celdas



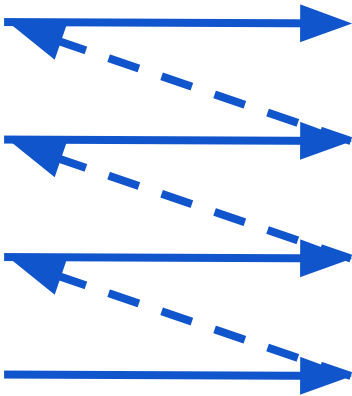
- ...y en otros recorridos, no hace falta ver casos de borde



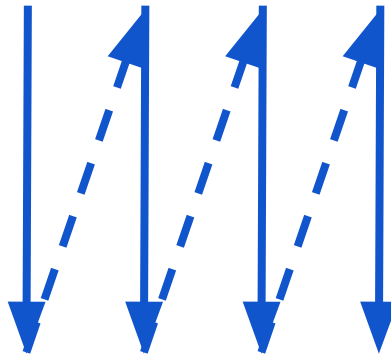
En la casa de la bruja no
hay que dejar miguitas



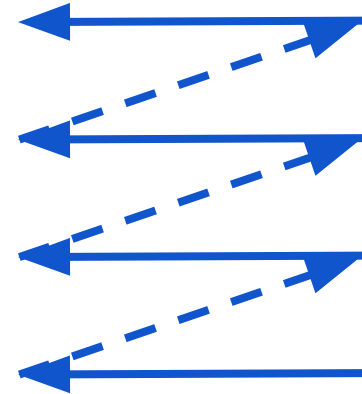
- El recorrido por celdas no tiene por qué ser solamente al Este y al Norte
 - ¿Cómo hacer que no sea siempre para el mismo lado?
 - ¡Parámetros!



Al Este y
al Norte



Al Sur y
al Este



Al Oeste
y al Norte



• ¿Cómo saber qué parametrizar? Técnica de los recuadros

```
procedure PintarTodasLasCeldas() {  
    /* ... */  
    IrAPrimeraCelda()  
    while (haySiguienteCelda()) {  
        Poner(Rojo)  
        IrASiguienteCelda()  
    }  
    Poner(Rojo)  
}
```

Primero mueve al Este,
y si no puede, al Norte

```
procedure IrAPrimeraCelda() {  
    /* ... */  
    IrAlBorde(Sur)  
    IrAlBorde(Oeste)  
}
```

```
procedure IrASiguienteCelda() {  
    /* ... */  
    if (puedeMover(Este)) { Mover(Este) }  
    else { Mover(Norte) IrAlBorde(Oeste) }  
}
```

```
function haySiguienteCelda() {  
    /* ... */  
    return (puedeMover(Este) || puedeMover(Norte))  
}
```



- ¿Cómo saber qué parametrizar? Técnica de los recuadros

```
procedure PintarTodasLasCeldas() {  
    /* ... */  
    IrAPrimeraCelda()  
    while (haySiguienteCelda()) {  
        Poner(Rojo)  
        IrASiguienteCelda()  
    }  
    Poner(Rojo)  
}
```

Primero mueve al Este,
y si no puede, al Norte

```
procedure IrAPrimeraCelda() {  
    /* ... */  
    IrAlBorde(opuesto(Norte))  
    IrAlBorde(opuesto(Este))  
}
```

```
procedure IrASiguienteCelda() {  
    /* ... */  
    if (puedeMover(Este)) { Mover(Este) }  
    else { Mover(Norte) IrAlBorde(opuesto(Este)) }  
}
```

```
function haySiguienteCelda() {  
    /* ... */  
    return (puedeMover(Este) || puedeMover(Norte))  
}
```



- ¿Cómo saber qué parametrizar? Técnica de los recuadros

```
procedure PintarTodasLasCeldas() {  
    /* ... */  
    IrAPrimeraCelda()  
    while (haySiguienteCelda()) {  
        Poner(Rojo)  
        IrASiguienteCelda()  
    }  
    Poner(Rojo)  
}
```

Primero mueve al Este,
y si no puede, al Norte

```
procedure IrAPrimeraCelda() {  
    /* ... */  
    IrAlBorde(opuesto(Norte))  
    IrAlBorde(opuesto(Este))  
}
```

```
procedure IrASiguienteCelda() {  
    /* ... */  
    if (puedeMover(Este)) { Mover(Este) }  
    else { Mover(Norte) IrAlBorde(opuesto(Este)) }  
}
```

```
function haySiguienteCelda() {  
    /* ... */  
    return (puedeMover(Este) || puedeMover(Norte))  
}
```



- ¿Cómo saber qué parametrizar? Técnica de los recuadros

```
procedure PintarTodasLasCeldas() {  
    /* ... */  
    IrAPrimeraCelda()  
    while (haySiguienteCelda()) {  
        Poner(Rojo)  
        IrASiguienteCelda()  
    }  
    Poner(Rojo)  
}
```

¡Las direcciones
pueden ser otras!

```
procedure IrAPrimeraCelda() {  
    /* ... */  
    IrAlBorde(opuesto([ ]))  
    IrAlBorde(opuesto([ ]))  
}
```

```
procedure IrASiguienteCelda() {  
    /* ... */  
    if (puedeMover([ ])) { Mover([ ]) }  
    else { Mover([ ]) IrAlBorde(opuesto([ ])) }  
}
```

```
function haySiguienteCelda() {  
    /* ... */  
    return (puedeMover([ ]) || puedeMover([ ]))  
}
```



- Un recorrido por celdas con parámetros para la dirección

```

procedure PintarTodasLasCeldas() {
  /* ... */
  IrAPrimeraCeldaEnUnRecorridoAl_YAl_( , )
  while (haySiguienteCeldaEnUnRecorridoAl_YAl_( , )) {
    Poner(Rojo)
    IrASiguienteCeldaEnUnRecorridoAl_YAl_( , )
  }
  Poner(Rojo)
}

```

¿Cómo pasar los parámetros a las subtarear?

```

procedure IrAPrimeraCeldaEnUnRecorridoAl_YAl_( , ) {
  /* ... */
  IrAlBorde(opuesto( , ))
  IrAlBorde(opuesto( , ))
}

procedure IrASiguienteCeldaEnUnRecorridoAl_YAl_(
  /* ... */
  if (puedeMover( , )) { Mover( , ) }
  else { Mover( , ) IrAlBorde(opuesto( , )) }
}

function haySiguienteCeldaEnUnRecorridoAl_YAl_( , ) {
  /* ... */
  return (puedeMover( , ) || puedeMover( , ))
}

```




- Un recorrido por celdas con parámetros para la dirección

```
procedure PintarTodasLasCeldas() {  
  /* ... */  
  IrAPrimeraCeldaEnUnRecorridoAl_YAl_( , )  
  while (haySiguienteCeldaEnUnRecorridoAl_YAl_( , )) {  
    Poner(Rojo)  
    IrASiguienteCeldaEnUnRecorridoAl_YAl_( , )  
  }  
  Poner(Rojo)  
}
```

¿Cómo pasar los parámetros a las subtarear?

```
procedure IrAPrimeraCeldaEnUnRecorridoAl_YAl_( , ) {  
  /* ... */  
  IrAlBorde(opuesto( , ))  
  IrAlBorde(opuesto( , ))  
}
```

```
procedure IrASiguienteCeldaEnUnRecorridoAl_YAl_( , ) {  
  /* ... */  
  if (puedeMover( , )) { Mover( , ) }  
  else { Mover( , ) IrAlBorde(opuesto( , )) }  
}
```

```
function haySiguienteCeldaEnUnRecorridoAl_YAl_( , ) {  
  /* ... */  
  return (puedeMover( , ) || puedeMover( , ))  
}
```



- Un recorrido por celdas con parámetros para la dirección

```
procedure PintarTodasLasCeldas() {  
  /* ... */  
  IrAPrimeraCeldaEnUnRecorridoAl_YAl_(  
  while (haySiguienteCeldaEnUnRecorridoAl_YAl_(  
    Poner(Rojo)  
    IrASiguienteCeldaEnUnRecorridoAl_YAl_  
  })  
  Poner(Rojo)  
}
```

¡Más parámetros!

```
procedure IrAPrimeraCeldaEnUnRecorridoAl_YAl_(dirPr,dirSc) {  
  /* ... */  
  IrAlBorde(opuesto(  
  IrAlBorde(opuesto(  
}  
  
procedure IrASiguienteCeldaEnUnRecorridoAl_YAl_(  
  /* ... */  
  if (puedeMover(  
  else { Mover(  
}  
  
function haySiguienteCeldaEnUnRecorridoAl_YAl_(dirPr,dirSc) {  
  /* ... */  
  return (puedeMover(  
}
```



- Un recorrido por celdas con parámetros para la dirección

```
procedure PintarTodasLasCeldas() {  
    /* ... */  
    IrAPrimeraCeldaEnUnRecorridoAl_YAl_(Este,Norte)  
    while (haySiguienteCeldaEnUnRecorridoAl_YAl_(Este,Norte)) {  
        Poner(Rojo)  
        IrASiguienteCeldaEnUnRecorridoAl_YAl_(Este,Norte)  
    }  
    Poner(Rojo)  
}
```

Nota: renombrar los parámetros

```
procedure IrAPrimeraCeldaEnUnRecorridoAl_YAl_(dirPr,dirSc) {  
    /* ... */  
    IrAlBorde(opuesto(dirPr))  
    IrAlBorde(opuesto(dirSc))  
}  
  
procedure IrASiguienteCeldaEnUnRecorridoAl_YAl_(  
    /* ... */  
    if (puedeMover(dirPr)){ Mover(dirPr) }  
    else { Mover(dirSc) IrAlBorde(opuesto(dirPr)) }  
}  
  
function haySiguienteCeldaEnUnRecorridoAl_YAl_(dirPr,dirSc) {  
    /* ... */  
    return (puedeMover(dirPr)|| puedeMover(dirSc))  
}
```



Cierre



- ***Repetición condicional***

- Una forma de repetición para cuándo no se sabe la cantidad de veces que hay que repetir
- Tiene un **cuerpo** de comandos a repetir, y una **condición** que establece cuándo terminar
- En bloques tiene la forma **repetir-hasta-que**
- En texto tiene la forma **mientras (while)**
- ¡Puede suceder que la repetición no termine nunca!
 - Esto es equivalente a hacer BOOM

- ***Recorridos***

- Una forma de controlar la repetición condicional sugiriendo cómo dividir en subtareas
- Se basa en una secuencia finita de “elementos”
- Sugiere 5 subtareas
 - `IniciarRecorrido()`
 - `quedanElementosParaProcesar()`
 - `ProcesarElementoActual()`
 - `PasarAlSiguienteElemento()`
 - `FinalizarRecorrido()`
- Las tareas pueden ponerse en procedimientos o definirse directamente con comandos sueltos