# Tipos, Alternativa condicional y Funciones

Introducción a la programación

Tecnicatura en programación informática,
Licenciatura en Desarrollo de Software,
Universidad Nacional de Quilmes

# Lo que vimos hasta ahora

- Programas Gobstones: tablero, celdas, cabezal, comandos, colores, direcciones
- Procedimientos simples: reutilización de código, abstracción
- Procedimientos compuestos: parámetros, argumentos
- Repetición simple: valores numéricos
- Repetición indexada: índices

# Valores

- Colores
  - ●, ●, ●, ●.
- Direcciones
  - ↑, →, ↓, ←.
- Números
  - . . . , **-2**, **-1**, **0**, **1**, **2**, . . . .
  - Verdadero, Falso.

# Valores

- Colores
  - ●, ●, ●, ●.
- Direcciones
  - ↑, →, ↓, ←.
- Números
  - . . . , **-2**, **-1**, **0**, **1**, **2**, . . . .
  - Verdadero, Falso.

- Cada valor es igual a sí mismo, y diferente del resto de los valores.

# Expresiones

- Una expresión es una cadena de símbolos que denota un valor.
- Evaluar significa establecer qué valor denota una expresión.
  - `1 + 2`, `2 / 4`, `3`, `nroBolitas(Azul)`, `nroBolitas(Rojo)`;

# Expresiones

- Una expresión es una cadena de símbolos que denota un valor.
- Evaluar significa establecer qué valor denota una expresión.
  - `1 + 2`, `2 / 4`, `3`, `nroBolitas(Azul)`, `nroBolitas(Rojo)`;

- Expresiones distintas pueden denotar el mismo valor.
- En distintos "instantes", una expresión puede denotar valores distintos.

# Tipos

- Un tipo es un conjunto de valores que comparten ciertas propiedades.
- En un lenguaje de programación, a valores del mismo tipo se le pueden aplicar las mismas funciones.

# Tipos

- Un tipo es un conjunto de valores que comparten ciertas propiedades.
- En un lenguaje de programación, a valores del mismo tipo se le pueden aplicar las mismas funciones.
- Hay cuatro tipos básicos en Gobstones:
  - Direcciones, Colores, Números y ... en minutos veremos el cuarto
- Vamos a decir que el tipo de una expresión es el tipo del valor que denota.
  - El tipo de `nroBolitas(Azul)` es número.

# Tipos

- Un tipo es un conjunto de valores que comparten ciertas propiedades.
- En un lenguaje de programación, a valores del mismo tipo se le pueden aplicar las mismas funciones.
- Hay cuatro tipos básicos en Gobstones:
  - Direcciones, Colores, Números y ... en minutos veremos el cuarto
- Vamos a decir que el tipo de una expresión es el tipo del valor que denota.
  - El tipo de `nroBolitas(Azul)` es número.
- ¿Qué ocurre si se intenta ejecutar `1 + Verde`, o `Poner(True)`?

# Tipos

- Un tipo es un conjunto de valores que comparten ciertas propiedades.
- En un lenguaje de programación, a valores del mismo tipo se le pueden aplicar las mismas funciones.
- Hay cuatro tipos básicos en Gobstones:
  - Direcciones, Colores, Números y ... en minutos veremos el cuarto
- Vamos a decir que el tipo de una expresión es el tipo del valor que denota.
  - El tipo de `nroBolitas(Azul)` es número.
- ¿Qué ocurre si se intenta ejecutar `1 + Verde`, o `Poner(True)`?
- Error de tipado vs. error de ejecución
  - procedimientos totales vs. procedimientos parciales

# Operaciones sobre el cabezal

- Si `<c>` denota un color $c$ y `<d>` una dirección $d$, entonces

- `nroBolitas(<c>)` es número: bolitas de color $c$ que hay en la celda actual

# Operaciones con colores

- Si <c> denota un color $c$, entonces

- `minColor()` es color: mínimo en orden alfabético (●)
- `maxColor()` es color: máximo en orden alfabético (●)
- `siguiente(<c>)` es color: siguiente a $c$ en orden alfabético. Cuando $c$ es el máximo color, la operación
- `siguiente(<c>)` denota el color mínimo.
- `previo(<c>)` es color: color previo a $c$ en orden alfabético. Cuando $c$ es el mínimo color, la operación
- `previo(<c>)` denota el color máximo.

# Operaciones con direcciones

- Si `<d>` denota una dirección $d$, entonces
- `minDir()` es dirección: mínima desde $\uparrow$ en orden de las agujas del reloj ($\uparrow$)
- `maxDir()` es dirección: máxima desde $\uparrow$ en orden de las agujas del reloj ($\leftarrow$)
- `siguiente(<d>)` es dirección: siguiente a $d$ en el orden de las agujas del reloj. En caso que $d$ sea la dirección máxima, `siguiente(<d>)` denota la dirección mínima
- `previo(<d>)` es dirección: anterior a $d$ en el orden de las agujas del reloj. En caso que $d$ sea la dirección mínima, `previo(<d>)` denota la dirección máxima
- `opuesto(<d>)` es dirección: cardinal opuesta a $d$
- `-<d>` es dirección: idem opuesto

# Operaciones con números

- Si `<n>` y `<m>` denotan los números **n** y **m**, entonces
- `<n> + <m>` es número: denota $\mathbf{n}+\mathbf{m}$
- `<n> - <m>` es número: denota $\mathbf{n}-\mathbf{m}$
- `<n> * <m>` es número: denota $\mathbf{n}\times\mathbf{m}$
- `<n> div <m>` es número: denota $\mathbf{n}/\mathbf{m}$
- `<n> mod <m>` es número: denota el resto de $\mathbf{n}/\mathbf{m}$
- `<n> ^ <m>` es número: denota $\mathbf{n}^{\mathbf{m}}$

  opuesto(`<n>`) es número: denota $-\mathbf{n}$
- `-<n>` es número: idem opuesto

# El tipo Booleano: valores de verdad y condiciones

- Valores de verdad: verdadero y falso
  - Se agregan a los colores, direcciones y números
  - Se llaman booleanos

# El tipo Booleano: valores de verdad y condiciones

- Valores de verdad: verdadero y falso
  - Se agregan a los colores, direcciones y números
  - Se llaman booleanos
- Literales booleanos: Verdadero, Falso.

# El tipo Booleano: valores de verdad y condiciones

- Valores de verdad: verdadero y falso
  - Se agregan a los colores, direcciones y números
  - Se llaman booleanos
- Literales booleanos: Verdadero, Falso.
- Las expresiones pueden denotar booleanos

# El tipo Booleano: valores de verdad y condiciones

- Valores de verdad: verdadero y falso
  - Se agregan a los colores, direcciones y números
  - Se llaman booleanos
- Literales booleanos: Verdadero, Falso.
- Las expresiones pueden denotar booleanos
- Condición: expresión que denota un valor booleano

# El tipo Booleano: operaciones

**Operaciones que denotan booleanos: *c* color, *d* dirección**

- `hayBolitas`(⟨color⟩): Verdaderosi en la celda actual hay una bolita de ⟨color⟩
- `puedeMover`(⟨dir⟩): Verdaderosi existe una celda contigua al cabezal en ⟨dir⟩

# El tipo Booleano: operaciones

**Operaciones que denotan booleanos: *c* color, *d* dirección**

- `hayBolitas`(⟨color⟩): Verdaderosi en la celda actual hay una bolita de ⟨color⟩
- `puedeMover`(⟨dir⟩): Verdaderosi existe una celda contigua al cabezal en ⟨dir⟩

**Operaciones entre booleanos: *b*, *w* booleanos**

- `minBool`(): denota Falso
- `maxBool`(): denota Verdadero
- `not` *b*: denota la negación de *b*
- *b* `&&` *w*: Verdadero cuando ambos *b* y *w* denotan Verdadero
- *b* `||` *w*: Verdadero cuando alguno de *b* y *w* denota Verdadero

# Operaciones relacionales (para todos los tipos)

- Todas estas operaciones denotan valores booleanos.
- Si `<a>` y `<b>` denotan valores $a$ y $b$ del mismo tipo, entonces

- `<a> == <b>` es booleano: Verdadero para $a = b$
- `<a> /= <b>` es booleano: Verdadero para $a \neq b$
- `<a> > <b>` es booleano: Verdadero para $a > b$ ($a$ mayor que $b$)
- `<a> >= <b>` es booleano: Verdadero para $a \geq b$ ($a$ mayor o igual que $b$)
- `<a> < <b>` es booleano: Verdadero para $a < b$ ($a$ menor que $b$)
- `<a> <= <b>` es booleano: Verdadero para $a \leq b$ ($a$ menor o igual que $b$)

# Calculo de tipos

- El tipo de una expresión se puede calcular sin necesidad de ejecutar el programa

# Calculo de tipos

- El tipo de una expresión se puede calcular sin necesidad de ejecutar el programa
- El tipo de las expresiones literales es obvio

# Calculo de tipos

- El tipo de una expresión se puede calcular sin necesidad de ejecutar el programa
- El tipo de las expresiones literales es obvio
- Para saber el tipo de una expresión *<func>*(*<args>*) es necesario saber el tipo de sus argumentos
- Luego, mirando las tablas previas, se sabe el tipo de *<func>*(*<args>*)

# Calculo de tipos

- El tipo de una expresión se puede calcular sin necesidad de ejecutar el programa
- El tipo de las expresiones literales es obvio
- Para saber el tipo de una expresión *<func>*(*<args>*) es necesario saber el tipo de sus argumentos
- Luego, mirando las tablas previas, se sabe el tipo de *<func>*(*<args>*)
- Ejemplos:
  - `Azul` $\rightarrow$ color

# Calculo de tipos

- El tipo de una expresión se puede calcular sin necesidad de ejecutar el programa
- El tipo de las expresiones literales es obvio
- Para saber el tipo de una expresión *<func>*(*<args>*) es necesario saber el tipo de sus argumentos
- Luego, mirando las tablas previas, se sabe el tipo de *<func>*(*<args>*)
- Ejemplos:
  - `Azul` → color
  - `nroBolitas(Azul)` → numero

# Calculo de tipos

- El tipo de una expresión se puede calcular sin necesidad de ejecutar el programa
- El tipo de las expresiones literales es obvio
- Para saber el tipo de una expresión *<func>*(*<args>*) es necesario saber el tipo de sus argumentos
- Luego, mirando las tablas previas, se sabe el tipo de *<func>*(*<args>*)
- Ejemplos:
  - `Azul` → color
  - `nroBolitas(Azul)` → numero
  - `nroBolitas(Azul) + nroBolitas(Verde)` → numero.

# Calculo de tipos

- El tipo de una expresión se puede calcular sin necesidad de ejecutar el programa
- El tipo de las expresiones literales es obvio
- Para saber el tipo de una expresión *<func>*(*<args>*) es necesario saber el tipo de sus argumentos
- Luego, mirando las tablas previas, se sabe el tipo de *<func>*(*<args>*)
- Ejemplos:
  - `Azul` → color
  - `nroBolitas(Azul)` → numero
  - `nroBolitas(Azul) + nroBolitas(Verde)` → numero.
  - Si c denota un color, entonces `siguiente(c)` → color

# Calculo de tipos

- El tipo de una expresión se puede calcular sin necesidad de ejecutar el programa
- El tipo de las expresiones literales es obvio
- Para saber el tipo de una expresión *<func>*(*<args>*) es necesario saber el tipo de sus argumentos
- Luego, mirando las tablas previas, se sabe el tipo de *<func>*(*<args>*)
- Ejemplos:
  - `Azul` → color
  - `nroBolitas(Azul)` → numero
  - `nroBolitas(Azul) + nroBolitas(Verde)` → numero.
  - Si c denota un color, entonces `siguiente(c)` → color
- Recordar: los argumentos deben ser de tipo correcto
  - `nroBolitas(Azul) + (nroBolitas(Verde) == nroBolitas(Negro))` NO es una expresión correcta.

# Expresiones (Resumen)

- Valor: igual a sí mismo y distinto del resto.
- Expresión: forma de denotar un valor.
  - Distintas expresiones pueden denotar el mismo valor.
  - La misma expresión puede denotar distintos valores.
- Tipo: conjunto de valores que comparten propiedades (en particular las operaciones).
  - Cálculo de tipos
  - Error de tipado vs. error de ejecución
- Procedimiento parcial vs. Procedimiento total

# Repaso: repeat

## Repetición simple (repeat)

```
repeat(<numero>)
<bloque a repetir>
```

- *<numero>* debe ser una expresión numérica
- *<bloque a repetir>* se repite tantas veces como el valor denotado por *<numero>*

# ¿Qué hace `IgualarParaAbajo(Azul, Rojo)`?

```
1  /* Proposito: ??? */
2  /* Precondicion: ??? */
3  procedure IgualarParaAbajo(color1, color2) {
4    repeat(nroBolitas(color1) - nroBolitas(color2)) {
5      Sacar(color1)
6    }
7    repeat(nroBolitas(color2) - nroBolitas(color1)) {
8      Sacar(color2)
9    }
10 }
```

# ¿Qué hace `IgualarParaAbajo(Azul, Rojo)`?

```
1  /* Proposito: ??? */
2  /* Precondicion: ??? */
3  procedure IgualarParaAbajo(color1, color2) {
4    repeat(nroBolitas(color1) - nroBolitas(color2)) {
5      Sacar(color1)
6    }
7    repeat(nroBolitas(color2) - nroBolitas(color1)) {
8      Sacar(color2)
9    }
10 }
```

- ¿Cómo se puede mejorar la implementacion de IgualarParaAbajo?
  - Pensar en reutilizar procedimientos de la biblioteca.

# IgualarParaAbajo(Azul, Rojo) reloaded

```
1  /* Proposito: Iguala la cantidad de bolitas de color1 y color2
2   * de la celda actual, sacando de la que m s hubiera
3   */
4  procedure IgualarParaAbajo(color1, color2) {
5    NoSuperar(color1, color2)
6    NoSuperar(color2, color1)
7  }
8
9
10 /* Proposito: Saca tantas bolitas de color1 como sean necesarias
11  * para que no hayan m s bolitas de color1 que de color2;
12  * si nroBolitas(color1) <= nroBolitas(color2) no hace nada.
13  */
14 procedure NoSuperar(color1, color2) {
15   SacarN(nroBolitas(color1) - nroBolitas(color2), color1)
16 }
```

# IgualarParaAbajo(Azul, Rojo) reloaded

```
1  /* Proposito: Iguala la cantidad de bolitas de color1 y color2
2   * de la celda actual, sacando de la que m s hubiera
3   */
4  procedure IgualarParaAbajo(color1, color2) {
5    NoSuperar(color1, color2)
6    NoSuperar(color2, color1)
7  }
8
9
10 /* Proposito: Saca tantas bolitas de color1 como sean necesarias
11  * para que no hayan m s bolitas de color1 que de color2;
12  * si nroBolitas(color1) <= nroBolitas(color2) no hace nada.
13  */
14 procedure NoSuperar(color1, color2) {
15   SacarN(nroBolitas(color1) - nroBolitas(color2), color1)
16 }
```

- ¿Por qué es mejor esta implementación?

# Repaso: foreach

## Repetición indexada (foreach)

```
foreach <indice>in [<primero>..<ultimo>]
<bloque a repetir>
```

- Se repite *<bloque a repetir>* tantas veces como elementos tenga el rango
- *<primero>* y *<ultimo>* DEBEN ser expresiones del mismo tipo
- En la primer repetición, *<indice>* denota *<primero>*
- En la segunda repetición, *<indice>* denota `siguiente(`*<primero>*`)`
- ...
- En la última repetición, *<indice>* denota *<ultimo>*

# Repaso: foreach

## Repetición indexada (foreach)

```
foreach <indice>in [<primero>..<ultimo>]
<bloque a repetir>
```

- Se repite *<bloque a repetir>* tantas veces como elementos tenga el rango
- *<primero>* y *<ultimo>* DEBEN ser expresiones del mismo tipo
- En la primer repetición, *<indice>* denota *<primero>*
- En la segunda repetición, *<indice>* denota `siguiente(`*<primero>*`)`
- ...
- En la última repetición, *<indice>* denota *<ultimo>*
- ¿De qué tipo es *<indice>*?

# ¿Qué hace `Progresion(Azul, Sur, 4)`?

```
1  /* Proposito: ??? */
2  procedure Progresion(c, d, n) {
3    foreach i in [1..n] {
4      if(nroBolitas(c) > i) {
5        SacarN(c, nroBolitas(c) -i)
6      } else {
7        PonerN(c, i - nroBolitas(c))
8      }
9      Mover(d)
10   }
11   SacarN(c, nroBolitas(c))
12   repeat(n) {
13     Mover(opuesto(d))
14   }
15 }
```

# ¿Qué hace `Progresion(Azul, Sur, 4)`?

```
1  /* Proposito: ??? */
2  procedure Progresion(c, d, n) {
3    foreach i in [1..n] {
4      if(nroBolitas(c) > i) {
5        SacarN(c, nroBolitas(c) -i)
6      } else {
7        PonerN(c, i - nroBolitas(c))
8      }
9      Mover(d)
10   }
11   SacarN(c, nroBolitas(c))
12   repeat(n) {
13     Mover(opuesto(d))
14   }
15 }
```

- ¿Cómo se puede mejorar la implementacion de Progresion?

# Progresion(Azul, Sur, 4) reloaded

```
1  /* Proposito: ??? */
2  procedure Progresion(c, d, n) {
3    foreach i in [1..n] {
4      DejarN(c, i)
5      Mover(d)
6    }
7    SacarTodas(c)
8    MoverN(opuesto(d), n)
9  }
10
11
12 procedure DejarN(c, n) {
13     SacarN(c, nroBolitas(c) - n)
14     PonerN(c, n - nroBolitas(c))
15 }
```

# Progresion(Azul, Sur, 4) reloaded

```
1  /* Proposito: ??? */
2  procedure Progresion(c, d, n) {
3    foreach i in [1..n] {
4      DejarN(c, i)
5      Mover(d)
6    }
7    SacarTodas(c)
8    MoverN(opuesto(d), n)
9  }
10
11
12 procedure DejarN(c, n) {
13     SacarN(c, nroBolitas(c) - n)
14     PonerN(c, n - nroBolitas(c))
15 }
```

- ¿Por qué es mejor esta implementación?

# Alternativa condicional: motivación

- ¿Cómo hacemos para eliminar una bolita roja sin que se produzca BOOM cuando no hay bolitas?

# Alternativa condicional: motivación

- ¿Cómo hacemos para eliminar una bolita roja sin que se produzca BOOM cuando no hay bolitas?

# Alternativa condicional: forma general

## Alternativa condicional `if-else`

**if** ⟨condicion⟩
    ⟨bloque caso verdadero⟩
**else**
    ⟨bloque caso falso⟩

donde,

- ⟨condicion⟩ es una expresión que puede ser verdadera o falsa

# Alternativa condicional: forma general

## Alternativa condicional `if-else`

**if** ⟨condicion⟩
    ⟨bloque caso verdadero⟩
**else**
    ⟨bloque caso falso⟩

donde,

- ⟨condicion⟩ es una expresión que puede ser verdadera o falsa

- El `if-else` se puede escribir en los mismos lugares que los comandos
    - Bloque: secuencia de comandos, repeticiones y alternativas

# Alternativa condicional: forma general

## Alternativa condicional `if-else`

**if** ⟨condicion⟩
    ⟨bloque caso verdadero⟩
**else**
    ⟨bloque caso falso⟩

donde,

- ⟨condicion⟩ es una expresión que puede ser verdadera o falsa

- El `if-else` se puede escribir en los mismos lugares que los comandos
  - Bloque: secuencia de comandos, repeticiones y alternativas
- Funcionamiento: si ⟨condicion⟩ denota verdadero, se ejecuta ⟨bloque caso verdadero⟩; caso contrario, se ejecuta ⟨bloque caso falso⟩

# Alternativa condicional: ejemplo

- ¿Qué ocurre si ejecutamos el el siguiente procedimiento varias veces?
- ¿Cuál es su precondición?

```
//Propo: Adivinar: c es un color
//Precondici?
procedure InvertirBolita(c) {
1. if(hayBolitas(c))
2. {
3.       Sacar(c)
4. }
5. else
6. {
7.       Poner(c)
8. }
}
```

# Alternativa condicional: else omitido

- El **else** puede omitirse. En ese caso, no se hace nada cuando la condición es `False`.

```
//Propo: ??
//Precondici?
procedure MoverSiPuede(d) {
1. if(puedeMover(d))
2. {
3.       Mover(d)
4. }
}
```

# Alternativa condicional: anidación

- ¿Qué hace el siguiente procedimiento?

```
//Propo: ??
//Precondici?
procedure MoverSiNoAnteultima(d) {
1. if(puedeMover(d))
2. {
3.       Mover(d)
4.       if(not puedeMover(d))
5.       {
6.            Mover(opuesto(d))
7.       }
8. }
}
```

- Un **if** puede aparecer dentro de cualquier bloque
  - En consecuencia pueden anidarse
  - Pero no conviene anidar; mejor dividir en subtareas

# Problemas de anidación: código repetido

- ¿Qué hace el siguiente procedimiento?

```
procedure Adivinar() {
 1. if(hayBolitas(Rojo))
 2. {
 3.       if(hayBolitas(Azul))
 4.       {
 5.            Poner(Azul)
 6.       }
 7. }
 8. else
 9. {
10.       if(hayBolitas(Negro))
11.       {
12.            Poner(Negro)
13.       }
14. }
}
```

# Problemas de anidación: código repetido

- ¿Qué hace el siguiente procedimiento?

```
procedure Adivinar() {
 1. if(hayBolitas(Rojo))
 2. {
 3.       if(hayBolitas(Azul))
 4.       {
 5.             Poner(Azul)
 6.       }
 7. }
 8. else
 9. {
10.       if(hayBolitas(Negro))
11.       {
12.             Poner(Negro)
13.       }
14. }
 }
```

- ¿Cómo se puede mejorar?
- Anidar mucho lleva a una explosión combinatoria de casos

# Alternativa condicional + repetición (ejercicio)

- Hacer un procedimiento que mueva el cabezal a lo sumo $n$ posiciones en dirección $d$ sin hacer BOOM.

# Alternativa condicional + repetición (ejercicio)

- Hacer un procedimiento que mueva el cabezal a lo sumo $n$ posiciones en dirección $d$ sin hacer BOOM.

```
procedure MoverALoSumoN(d, n) {
1. repeat(n)
2. {
3.       MoverSiPuede(d)
4. }
}
```

- Evitar la anidación sirve además para ejercitar división en subtareas

# Alternativa condicional (Resumen)

- La alternativa condicional permite condicionar la ejecución de bloques

# Alternativa condicional (Resumen)

- La alternativa condicional permite condicionar la ejecución de bloques
- Se escriben dentro de la secuencia de un bloque $\rightarrow$ anidación

# Alternativa condicional (Resumen)

- La alternativa condicional permite condicionar la ejecución de bloques
- Se escriben dentro de la secuencia de un bloque $\rightarrow$ anidación
- Útiles para verificar las precondiciones ANTES de invocar un procedimiento

# Alternativa condicional (Resumen)

- La alternativa condicional permite condicionar la ejecución de bloques
- Se escriben dentro de la secuencia de un bloque $\rightarrow$ anidación
- Útiles para verificar las precondiciones ANTES de invocar un procedimiento
- Cuidado con la anidación y el miedo a la condición

# Alternativa condicional (Resumen)

- La alternativa condicional permite condicionar la ejecución de bloques
- Se escriben dentro de la secuencia de un bloque $\rightarrow$ anidación
- Útiles para verificar las precondiciones ANTES de invocar un procedimiento
- Cuidado con la anidación y el miedo a la condición
- El tema de booleanos y condiciones lo formalizamos mejor la clase que viene

# Funciones: motivación

- Escribir un procedimiento que agregue una bolita roja en la celda actual si la celda lindante al oeste tiene más bolitas azules que la celda lindante al este. Suponer que en la celda actual no hay bolitas.

# Funciones: motivación

- Escribir un procedimiento que agregue una bolita roja en la celda actual si la celda lindante al oeste tiene más bolitas azules que la celda lindante al este. Suponer que en la celda actual no hay bolitas.

```
1  /* Precondicion: el cabezal no esta ni en
2   * la primer ni en la ultima columna.
3   */
4  procedure Motivacional() {
5    Mover(Oeste)
6    PonerNAl(Negro, nroBolitas(Azul), Este)
7    MoverN(Este, 2)
8    PonerNAl(Verde, nroBolitas(Azul), Oeste)
9    Mover(Oeste)
10   if(nroBolitas(Negro) > nroBolitas(Verde)) {
11     Poner(Rojo)
12   }
13   SacarN(Verde, nroBolitas(Verde))
14   SacarN(Negro, nroBolitas(Negro))
15 }
```

# Funciones: motivación

- Escribir un procedimiento que agregue una bolita roja en la celda actual si la celda lindante al oeste tiene más bolitas azules que la celda lindante al este. Suponer que en la celda actual no hay bolitas.

```
1  /* Precondicion: el cabezal no esta ni en
2   * la primer ni en la ultima columna.
3   */
4  procedure Motivacional() {
5    Mover(Oeste)
6    PonerNAl(Negro, nroBolitas(Azul), Este)
7    MoverN(Este, 2)
8    PonerNAl(Verde, nroBolitas(Azul), Oeste)
9    Mover(Oeste)
10   if(nroBolitas(Negro) > nroBolitas(Verde)) {
11     Poner(Rojo)
12   }
13   SacarN(Verde, nroBolitas(Verde))
14   SacarN(Negro, nroBolitas(Negro))
15 }
```

# Funciones: motivación

- Escribir un procedimiento que agregue una bolita roja en la celda actual si la celda lindante al oeste tiene más bolitas azules que la celda lindante al este. Suponer que existe una función `nroBolitasAl(c,d)` que denota la cantidad de bolitas de color `c` en la celda lindante en dirección `d`.

# Funciones: motivación

- Escribir un procedimiento que agregue una bolita roja en la celda actual si la celda lindante al oeste tiene más bolitas azules que la celda lindante al este. Suponer que existe una función `nroBolitasAl(c,d)` que denota la cantidad de bolitas de color `c` en la celda lindante en dirección `d`.

```
1  /* Precondicion: el cabezal no est  ni en
2   * la primer ni en la  ltima  columna.
3   */
4  procedure Motivacional() {
5    if(nroBolitasAl(Azul, Este) > nroBolitasAl(Azul, Oeste)) {
6      Poner(Rojo)
7    }
8  }
```

- Este código comunica mejor las ideas

# Funciones: motivación

- Escribir un procedimiento que agregue una bolita roja en la celda actual si la celda lindante al oeste tiene más bolitas azules que la celda lindante al este. Suponer que existe una función `nroBolitasAl(c,d)` que denota la cantidad de bolitas de color `c` en la celda lindante en dirección `d`.

```
1  /* Precondicion: el cabezal no est  ni en
2   * la primer ni en la  ltima  columna.
3   */
4  procedure Motivacional() {
5    if(nroBolitasAl(Azul, Este) > nroBolitasAl(Azul, Oeste)) {
6      Poner(Rojo)
7    }
8  }
```

- Este código comunica mejor las ideas
- Lamentablemente, Gobstones no tiene una función `nroBolitasAl`

# Funciones: motivación

- Escribir un procedimiento que agregue una bolita roja en la celda actual si la celda lindante al oeste tiene más bolitas azules que la celda lindante al este. Suponer que existe una función `nroBolitasAl(c,d)` que denota la cantidad de bolitas de color `c` en la celda lindante en dirección `d`.

```
1  /* Precondicion: el cabezal no est  ni en
2   * la primer ni en la  ltima  columna.
3   */
4  procedure Motivacional() {
5    if(nroBolitasAl(Azul, Este) > nroBolitasAl(Azul, Oeste)) {
6      Poner(Rojo)
7    }
8  }
```

- Este código comunica mejor las ideas
- Lamentablemente, Gobstones no tiene una función `nroBolitasAl`
- Afortunadamente, la podemos agregar usando funciones

# Funciones simples

- Una función simple es una manera de darle un nombre a una expresión.

**Definición de funciones (`function`)**

```
function <nombre>(<parametros>) {
  return(<expresion>)
}
```

- *<nombre>* es un identificador que empieza en minuscula.
- *<parametros>* es una lista de parámetros (como en los procedimientos)
- *<expresion>* es la expresión a la que se le da nombre.
- `return` es un comando especial que se utiliza para indicar la expresión denotada por la función.

# Funciones simples (ejemplo)

- ¿Cúal es el propósito de las siguientes funciones?

```
1  function bisiesto(anio) {
2    return(anio mod 4 == 0 &&
3           (anio mod 100 /= 0 || anio mod 400 == 0))
4  }
5
6  function puedeMoverDiag(dir) {
7    return(puedeMover(dir) && puedeMover(siguiente(dir)))
8  }
9
10 function totalBolitas() {
11   return(nroBolitas(Azul) + nroBolitas(Negro) +
12          nroBolitas(Rojo) + nroBolitas(Verde))
13 }
```

# Invocación de funciones

- Las funciones que programamos se invocan como las funciones primitivas
  - Se escribe el nombre de la función seguido por paréntesis y los argumentos
  - Recordar: misma cantidad de argumentos que parámetros tenga la función

# Invocación de funciones

- Las funciones que programamos se invocan como las funciones primitivas
  - Se escribe el nombre de la función seguido por paréntesis y los argumentos
  - Recordar: misma cantidad de argumentos que parámetros tenga la función
- Las invocaciones de funciones son expresiones, i.e. denotan valores y no alteran el estado del tablero.
  - Esta condición las diferencia de los procedimientos.

# Invocación de funciones

- Las funciones que programamos se invocan como las funciones primitivas
  - Se escribe el nombre de la función seguido por paréntesis y los argumentos
  - Recordar: misma cantidad de argumentos que parámetros tenga la función
- Las invocaciones de funciones son expresiones, i.e. denotan valores y no alteran el estado del tablero.
  - Esta condición las diferencia de los procedimientos.
- El tipo de la invocación de la función es el tipo del valor denotado.

# Invocación de funciones

- Las funciones que programamos se invocan como las funciones primitivas
  - Se escribe el nombre de la función seguido por paréntesis y los argumentos
  - Recordar: misma cantidad de argumentos que parámetros tenga la función
- Las invocaciones de funciones son expresiones, i.e. denotan valores y no alteran el estado del tablero.
  - Esta condición las diferencia de los procedimientos.
- El tipo de la invocación de la función es el tipo del valor denotado.

- ¡Como cualquier otra expresión, las funciones se pueden usar como argumentos, y SOLO como argumentos!

# Invocación de funciones

- ¿Qué hacen los siguientes procedimientos?

```
1  procedure Bisiesto(anio) {
2    if(bisiesto(anio)) {
3      Poner(Rojo)
4    } else {
5      Poner(Azul)
6    }
7  }
```

```
1  procedure MoverSegunBolitas(d) {
2    repeat(totalBolitas()) {
3      Mover(d)
4    }
5  }
```

# Funciones con comandos

- Una función puede tener comandos antes del `return`.
- Las funciones DEBEN contener un UNICO `return`, y este DEBE ser el último comando.

## Función con comandos

```
function <nombre>(<params>) {
  <comandos>
  return(<expresion>)
}
```

# Funciones con comandos

- Una función puede tener comandos antes del `return`.
- Las funciones DEBEN contener un UNICO `return`, y este DEBE ser el último comando.

## Función con comandos

```
function <nombre>(<params>) {
  <comandos>
  return(<expresion>)
}
```

- Las funciones **no tienen efectos**, con lo cual...

# Funciones con comandos

- Una función puede tener comandos antes del `return`.
- Las funciones DEBEN contener un UNICO `return`, y este DEBE ser el último comando.

## Función con comandos

```
function <nombre>(<params>) {
  <comandos>
  return(<expresion>)
}
```

- Las funciones **no tienen efectos**, con lo cual...
- Los efectos de *<comandos>* desaparecen cuando la función termina

# Funciones y el tablero

- Los efectos de los comandos de una función desaparecen cuando la función termina
- Esto permite realizar una mejor abstracción

```
1  function nroBolitasAl(c, d) {
2    Mover(d)
3    return(nroBolitas(c))
4    //no hace falta regresar el cabezal
5  }
6
7
8  procedure Motivacional() {
9    if(nroBolitasAl(Azul, Este) > nroBolitasAl(Azul, Oeste)) {
10     Poner(Rojo)
11   }
12 }
```

# Funciones y el tablero

- Los efectos de los comandos de una función desaparecen cuando la función termina
- Esto permite realizar una mejor abstracción

```
1  function nroBolitasAl(c, d) {
2    Mover(d)
3    return(nroBolitas(c))
4    //no hace falta regresar el cabezal
5  }
6
7
8  procedure Motivacional() {
9    if(nroBolitasAl(Azul, Este) > nroBolitasAl(Azul, Oeste)) {
10     Poner(Rojo)
11   }
12 }
```

- ¡Notar como nroBolitasAl modifica la posición del cabezal y se despreocupa!

# Propósito y precondiciones

- Las funciones también tienen un propósito, que indica qué valor calcula

# Propósito y precondiciones

- Las funciones también tienen un propósito, que indica qué valor calcula
- Cuando una función tiene comandos, puede fallar
  - En ese caso, la función denota BOOM y el programa finaliza
  - Hay que incluir una precondición

# Propósito y precondiciones

- Las funciones también tienen un propósito, que indica qué valor calcula
- Cuando una función tiene comandos, puede fallar
  - En ese caso, la función denota BOOM y el programa finaliza
  - Hay que incluir una precondición
- ¿Hace falta indicar dónde quedará el cabezal?
  - No, ya que la función no tiene efectos

# Propósito y precondiciones

- Las funciones también tienen un propósito, que indica qué valor calcula
- Cuando una función tiene comandos, puede fallar
  - En ese caso, la función denota BOOM y el programa finaliza
  - Hay que incluir una precondición
- ¿Hace falta indicar dónde quedará el cabezal?
  - No, ya que la función no tiene efectos

```
1  /* Proposito: Denota la cantidad de bolitas de
2   * color c en direccion d
3   * Precondicion: puedeMover(d)
4   */
5  function nroBolitasAl(c, d) {
6    Mover(d)
7    return(nroBolitas(c))
8  }
```

# Cortocircuitos

- Escribir una función `celdaConBolitasAl(c, d)` que denote Verdadero cuando la celda lindante en dirección `d` existe y tiene bolitas de color `c`. Notar que en caso en que la celda actual este al extremo `d`, la función debe denotar Falso.

# Cortocircuitos

- Escribir una función `celdaConBolitasAl(c, d)` que denote Verdadero cuando la celda lindante en dirección `d` existe y tiene bolitas de color `c`. Notar que en caso en que la celda actual este al extremo `d`, la función debe denotar Falso.

```
1  function Incorrecta(c,d) {
2    if(puedeMover(d)) {
3      Mover(d)
4    }
5    return(hayBolitas(c))
6  }
```

# Cortocircuitos

- Escribir una función `celdaConBolitasAl(c, d)` que denote Verdadero cuando la celda lindante en dirección `d` existe y tiene bolitas de color `c`. Notar que en caso en que la celda actual este al extremo `d`, la función debe denotar Falso.

```
1  function Incorrecta(c,d) {
2    if(puedeMover(d)) {
3      Mover(d)
4    }
5    return(hayBolitas(c))
6  }
```

```
1  function celdaConBolitasAl(c,d) {
2    return(puedeMover(d) &&
3           hayBolitasAl(c, d))
4  }
```

# Cortocircuitos

- Escribir una función `celdaConBolitasAl(c, d)` que denote Verdadero cuando la celda lindante en dirección `d` existe y tiene bolitas de color `c`. Notar que en caso en que la celda actual este al extremo `d`, la función debe denotar Falso.

```
1  function Incorrecta(c,d) {
2    if(puedeMover(d)) {
3      Mover(d)
4    }
5    return(hayBolitas(c))
6  }
```

```
1  function celdaConBolitasAl(c,d) {
2    return(puedeMover(d) &&
3           hayBolitasAl(c, d))
4  }
```

- En caso en que `puedeMover(d)` denota Falso, se produce un cortocircuito...

# Cortocircuitos

- Escribir una función `celdaConBolitasAl(c, d)` que denote Verdadero cuando la celda lindante en dirección `d` existe y tiene bolitas de color `c`. Notar que en caso en que la celda actual este al extremo `d`, la función debe denotar Falso.

```
1  function Incorrecta(c,d) {
2    if(puedeMover(d)) {
3      Mover(d)
4    }
5    return(hayBolitas(c))
6  }
```

```
1  function celdaConBolitasAl(c,d) {
2    return(puedeMover(d) &&
3           hayBolitasAl(c, d))
4  }
```

- En caso en que `puedeMover(d)` denota Falso, se produce un cortocircuito...
- que significa que `hayBolitasAl(c, d)` no se evalúa.

# Cortocircuitos

- Los cortocircuitos se producen cuando se evalúa
  - `<a> && <b>` y `<a>` denota Falso, o
  - `<a> || <b>` y `<a>` denota Verdadero

# Cortocircuitos

- Los cortocircuitos se producen cuando se evalúa
  - `<a> && <b>` y `<a>` denota Falso, o
  - `<a> || <b>` y `<a>` denota Verdadero
- En estos casos, la expresión `<b>` no se evalúa

# Cortocircuitos

- Los cortocircuitos se producen cuando se evalúa
  - `<a> && <b>` y `<a>` denota Falso, o
  - `<a> || <b>` y `<a>` denota Verdadero
- En estos casos, la expresión `<b>` no se evalúa
- En particular, la expresión es correcta aunque `<b>` denote BOOM
  - `<a> && <b>` denota Falso cuando `<a>` denota Falso
  - `<a> || <b>` denota Verdadero cuando `<a>` denota Verdadero

# Cortocircuitos

- Los cortocircuitos se producen cuando se evalúa
  - <a> && <b> y <a> denota Falso, o
  - <a> || <b> y <a> denota Verdadero
- En estos casos, la expresión <b> no se evalúa
- En particular, la expresión es correcta aunque <b> denote BOOM
  - <a> && <b> denota Falso cuando <a> denota Falso
  - <a> || <b> denota Verdadero cuando <a> denota Verdadero

```
1 function celdaConBolitasAl(c,d) {
2   return(puedeMover(d) && hayBolitasAl(c, d))
3   //Si no puedeMover(d), cortocircuito
4 }
```

# Funciones – Resumen

- Forma de programar expresiones
  - Se pueden invocar como argumentos y SOLO como argumentos

# Funciones – Resumen

- Forma de programar expresiones
  - Se pueden invocar como argumentos y SOLO como argumentos
- Comando return al final
  - Indica qué denota la función cuando se invoca

# Funciones – Resumen

- Forma de programar expresiones
  - Se pueden invocar como argumentos y SOLO como argumentos
- Comando return al final
  - Indica qué denota la función cuando se invoca
- Pueden tener comandos pero no producen efectos
  - Los cambios al tablero "desaparecen" al finalizar la ejecución

# Funciones – Resumen

- Forma de programar expresiones
  - Se pueden invocar como argumentos y SOLO como argumentos
- Comando return al final
  - Indica qué denota la función cuando se invoca
- Pueden tener comandos pero no producen efectos
  - Los cambios al tablero "desaparecen" al finalizar la ejecución
- Programas más expresivos, cortos y abstractos

# Funciones – Resumen

- Forma de programar expresiones
  - Se pueden invocar como argumentos y SOLO como argumentos
- Comando return al final
  - Indica qué denota la función cuando se invoca
- Pueden tener comandos pero no producen efectos
  - Los cambios al tablero "desaparecen" al finalizar la ejecución
- Programas más expresivos, cortos y abstractos
- Palabras nuevas para conceptos complejos