Tutorial CGobstones
Versión 1.0

Nicolás Passerini
npasserini@unq.edu.ar

Pablo E. Martínez López
fidel@unq.edu.ar

Eduardo Bonelli
ebonelli@unq.edu.ar

22 de agosto de 2011

1. Introducción

El objetivo de este documento es brindar una introducción al manejo de estructuras de datos básicas para estudiantes que hayan hecho sus primeros pasos en la programación utilizando el lenguaje Gobstones. Las estructuras de datos que nos interesarán serán *registros* y *listas*.

Debido a que el lenguaje Gobstones no posee (aún) la capacidad para manejar este tipo de estructuras, utilizaremos una versión llamada CGobstones. Este lenguaje está basado en el lenguaje C++, agregando el dominio de Gobstones: tablero, celdas, cabezal, bolitas, colores, direcciones, etc., y todas las operaciones asociadas. Cabe aclarar que, aunque CGobstones utiliza C++ como base, no es nuestro propósito brindar una introducción a todas las características del lenguaje C++, sino utilizarlo únicamente como soporte para realizar programas que utilicen registros y listas. Es por ello que habrá muchas características de C++ que no serán utilizadas en CGobstones, y debido a su complejidad, recomendamos que su estudio sea hecho con posterioridad (para lo cual pueden consultar la bibliografía [**?**]). Remarcamos: NO es objetivo de este documento el que aprendan C++.

A los efectos de este documento, no nos interesa diferenciar las características generales del lenguaje C++ de las específicas de CGobstones. Presentaremos todas las características como parte del lenguaje CGobstones.

La primera parte de este documento se ocupará de describir las operaciones básicas de CGobstones, siguiendo un camino similar al del apunte de Gobstones [**?**] y concentrándose en las diferencias entre ambos lenguajes. Luego se introducirán los conceptos de registro y lista, para finalizar realizando programas que combinen dichas estructuras de datos para representar información.

Índice

2. Elementos Básicos

En este capítulo se repasan los comandos básicos de GOBSTONES y se los compara con su versión en CGOBSTONES, estableciendo las diferencias más significativas entre ambos lenguajes.

2.1. Comandos Simples

Un programa GOBSTONES está formado por *comandos*, que son descripciones de acciones individuales. Los comandos más simples de GOBSTONES son `Poner`, `Sacar` y `Mover`. En CGOBSTONES existen los mismos tres comandos; sin embargo hay una diferencia: todos los comandos simples en CGOBSTONES deben terminar en un punto y coma (`;`). Los comandos básicos de CGOBSTONES son:

- `Poner(`<*color*>`);`
- `Sacar(`<*color*>`);`
- `Mover(`<*dir*>`);`

Los valores posibles para <*color*> son los ya conocidos de GOBSTONES, `Verde`, `Rojo`, `Azul` y `Negro`, y para <*dir*>, `Norte`, `Sur`, `Este` y `Oeste`.

Al igual que en GOBSTONES, el comando `Poner` es una operación total, mientras que `Sacar` y `Mover` son operaciones parciales.

2.2. Secuenciación y bloques

Tanto en GOBSTONES como en CGOBSTONES la secuenciación de comandos se logra poniendo uno a continuación del otro, en la misma línea o en líneas separadas. Por ejemplo:

```
Poner(Verde); Poner(Verde);
```

y

```
Poner(Verde);
Poner(Verde);
```

son programas válidos.

Vale la pena mencionar que estos dos programas no son válidos:

```
Poner(Verde) Poner(Verde)     // Código inválido!
```

y

```
Poner(Verde)                  // Código inválido!
Poner(Verde)
```

Esto se debe a que ambos utilizan el comando `Poner(Verde)`, que no es un comando válido en CGOBSTONES, por no terminar con un punto y coma.

Al igual que en GOBSTONES, podemos delimitar un grupo de comandos en un comando compuesto denominado bloque. Para esto lo encerramos entre llaves. Entonces, es válido escribir

```
Poner(Verde);
Poner(Verde);
{ Poner(Rojo);  Poner(Rojo); }
{
  Poner(Azul);  Poner(Azul);
  Poner(Negro); Poner(Negro);
}
```

Observar que un comando de bloque **no** finaliza con punto y coma.

2.3. Procedimientos Simples

Un procedimiento GOBSTONES es un bloque al que se le asigna un nombre. En CGOBSTONES utilizaremos un mecanismo similar aunque la sintaxis es diferente; se reemplaza la palabra `procedure` por `void`. La forma entonces es:

`void` <*procName*>`()`
<*bloque*>

siendo <*procName*> un identificador que comienza en mayúscula[1], y <*bloque*> un bloque cualquiera.

Por ejemplo:

```
void PonerUnaBolitaDeCadaColor ()
    {
        Poner(Azul);
        Poner(Negro);
        Poner(Rojo);
        Poner(Verde);
    }
```

Al igual que en GOBSTONES, la invocación a un procedimiento seguida de paréntesis puede ser utilizada como un comando:

```
void PonerTresBolitasDeCadaColor ()
    {
      PonerUnaBolitaDeCadaColor();
      PonerUnaBolitaDeCadaColor();
      PonerUnaBolitaDeCadaColor();
    }
```

Como todos los comandos simples, la invocación a un procedimiento debe terminar en un punto y coma.

En GOBSTONES no hay restricciones en cuanto al orden de los procedimientos, salvo para el procedimiento `Main`, que debe ser el último. En CGOBSTONES el orden es más estricto y cada procedimiento sólo puede realizar invocaciones a los procedimientos definidos previamente en ese archivo. Por lo tanto es necesario ordenar los procedimientos siguiendo ese criterio. Por ejemplo, el siguiente programa es incorrecto:

```
void PonerNegroAlEste()
  {
    Mover(Este);
```

[1] Si bien el CGOBSTONES toma como válidos los identificadores comenzando por una letra minúscula, se considera una buena práctica el respetar la convención de nombres de GOBSTONES

```
    Poner(Negro);
    Mover(Oeste);
  }

void PonerNegrasAAmbosLados()
  {
    PonerNegroAlEste()
    PonerNegroAlOeste()       // Error!! No se encuentra el
                              // procedimiento PonerNegroAlOeste!!
  }

void PonerNegroAlOeste()
  {
    Mover(Oeste);
    Poner(Negro);
    Mover(Este);
  }
```

La forma correcta de hacerlo es la siguiente:

```
void PonerNegroAlEste()
  {
    Mover(Este);
    Poner(Negro);
    Mover(Oeste);
  }

void PonerNegroAlOeste()
  {
    Mover(Oeste);
    Poner(Negro);
    Mover(Este);
  }

void PonerNegrasAAmbosLados()
  {
    PonerNegroAlEste()     // Ambos procedimientos son definidos
    PonerNegroAlOeste()    // antes de ser invocados
  }
```

2.4. El procedimiento main

El procedimiento `Main` de GOBSTONES indica cuáles son todas las acciones a realizar por el cabezal. En CGOBSTONES existe un procedimiento análogo. Aunque su sintaxis presenta varias diferencias notables:

- La palabra `int` reemplaza a la palabra `procedure`
- Debe nombrarse comenzando con minúscula: `main`
- Debe finalizar con el comando `return 0;`[2].

[2]Esto indica la finalización correcta del programa (por ejemplo sin caerse del tablero). Utilizando valores distintos de `0` es posible indicar un error en la ejecución. El aprovechamiento de esta característica excede los límites de este documento.

La forma del procedimiento `main` es:

```
int main() {
    <comandos>
    return 0;
}
```

Es, por supuesto, el más importante de los procedimientos declarados: siempre debe estar presente y ser el último de todos. Un ejemplo podría ser:

```
int main() {
    Poner(Verde);
    Poner(Verde);
    return 0;
}
```

2.5. Comentarios

De las cuatro formas posibles de comentario presentes en GOBSTONES, en CGOBSTONES hay disponibles sólo dos:

- Comentarios de línea indicados con `//`:

  ```
  // Este es un comentario de línea.
  ```

- Comentarios de párrafo encerrados entre `/*` y `*/`:

  ```
  /* Este es un
  comentario de
  varias líneas. */
  ```

Lamentablemente el lenguaje CGOBSTONES no acepta los comentarios indicados con `--` ni encerrados entre {- y -}.

2.6. El tablero y las bolitas no son nativos en CGobstones

En GOBSTONES todo programa puede darle instrucciones al cabezal y de esa manera manipular las bolitas y el tablero. También, al finalizar el programa se muestra automáticamente el estado final del tablero. Se dice entonces que el tablero, las bolitas y los comandos básicos de `Poner`, `Sacar`, etc son *nativos* en el lenguaje GOBSTONES. Esto significa que son parte del lenguaje y que no se requiere de ninguna acción especial para que el programa pueda manipular el cabezal.

En cambio el lenguaje CGOBSTONES requiere de algunas indicaciones especiales para poder utilizar el tablero, el cabezal y los demás elementos del dominio ya conocidos. Los comandos que manipulan al cabezal están definidos en una *biblioteca* y se requiere que todo programa que necesite de esos comandos indique explícitamente la intención de utilizar dicha biblioteca.

Definición 1 *Una* biblioteca *es una herramienta para definir procedimientos que podrán ser utilizados en múltiples programas sin tener que reescribirlos.*

Mediante la utilización de bibliotecas es posible *partir* un programa CGOBSTONES en muchos archivos. Esto permite ordenar el código del programa en unidades más pequeñas, y por ende más manipulables[3].

[3] Es también deseable la creación de bibliotecas propias para evitar la repetición de procedimientos que son útiles en más de un programa. Los conceptos necesarios para la creación de bibliotecas de procedimientos en CGOBSTONES se introducirán más adelante en este documento.

La biblioteca CGobstones consta de dos archivos: `Gobstones.h` y `Gobstones.cpp`, por lo tanto todo programa CGobstones constará al menos de tres archivos:

- `Gobstones.h`
- `Gobstones.cpp`
- El archivo en el que se defina el procedimiento `main` y otros procedimientos específicos del programa. Este archivo suele denominarse `main.cpp`.

Para poder utilizar los comandos de CGobstones desde cualquier procedimiento definido en el archivo `main.cpp` es necesario indicarlo al principio del archivo de la siguiente manera[4]:

```
#include "Gobstones.h"
```

A la conjunción de archivos que conforman un programa se la suele denominar *proyecto*. En todo el proyecto debe haber un *único* procedimiento de nombre `main`.

Por otro lado, en el Gobstones original todos los programas imprimen el tablero al terminar. Esto no es así en CGobstones y en cambio es necesario hacerlo explícitamente. Para ello existe una primitiva especial denominada `ImprimirTablero();`, que permite mostrar el estado del tablero en pantalla al terminar el programa, emulando el comportamiento del Gobstones original. La forma completa del procedimiento `main` entonces deberá ser la siguiente:

```
int main()
  {
    <comandos>
    ImprimirTablero();
    return 0;
  }
```

Como todo comando simple `ImprimirTablero();` debe terminar en punto y coma.

2.7. Versión completa de un programa CGobstones

Finalmente todos los conceptos necesarios para ejecutar un programa CGobstones han sido definidos. En el siguiente ejemplo se comparan las versiones en Gobstones y CGobstones de un programa que pone dos bolitas verdes en la celda actual. El programa en Gobstones tendría la siguiente forma:

```
procedure Main()
  {
    Poner(Verde)
    Poner(Verde)
  }
```

Un programa CGobstones que tuviera el mismo comportamiento podría ser:

```
#include "Gobstones.h"
int main()
  {
    Poner(Verde);
```

[4]Esta indicación asume que el archivo `Gobstones.h` se encuentra en la misma carpeta que el archivo `main.cpp`. En caso de preferirse otra organización de los archivos deberá indicarse el *path relativo* del archivo, por ejemplo: `#include "../CGobstones/Gobstones.h"`

```
  Poner(Verde);
  ImprimirTablero();
  return 0;
}
```

A modo de resumen, en el ejemplo se pueden destacar las siguientes diferencias:

- La inclusión de la indicación `#include` para habilitar la utilización de los comandos básicos de CGobstones.
- La definición del procedimento main como `int Main` en lugar de `procedure Main`.
- La finalización de cada comando simple con un punto y coma.
- La utilización de la primitiva `ImprimirTablero();` para mostrar el estado del tablero al finalizar el programa, emulando el comportamiento habitual en Gobstones.

2.8. El proceso de ejecución

En cualquier lenguaje de programación hay una forma de tomar un programa y obtener los resultados de ese programa, que se suele denominar *proceso de ejecución*. Al programa que escribe el programador se lo denomina *código fuente*.

En muchos lenguajes el proceso de ejecución tiene sólo dos partes que se ejecutan en conjunto: *validación* del código fuente y *ejecución* propiamente dicha. Estos lenguajes se denominan *interpretados*. En la figura 1 puede observarse un gráfico que ilustra el proceso de ejecución de lenguajes interpretados

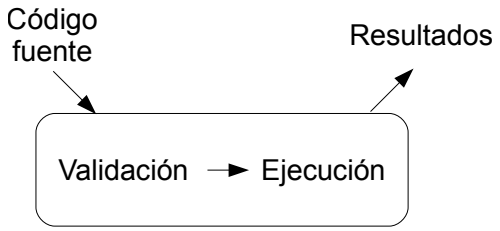

Figura 1: Proceso de ejecución de lenguajes interpretados.

En otros lenguajes el proceso de ejecución se realiza en varias etapas diferenciadas. En la primera de las etapas, denominada *compilación*, se valida el código fuente, y se lo traduce a un formato especial, denominado *código ejecutable*, que es el que se utiliza para obtener los resultados. Los lenguajes que realizan este tipo de transformación se denominan *compilados*. En la figura 2 se puede observar un gráfico que ilustra el proceso de compilación.

Además, muchos lenguajes permiten repartir el código fuente en varios archivos distintos, que se compilan individualmente, lo cual se denomina *compilación separada*. El resultado de compilar cada archivo de código fuente por separado se denomina *código objeto*. Es el código objeto el que luego se transforma en código ejecutable. Esto introduce la necesidad de una etapa adicional, denominada *linking*, que consiste en combinar todos los archivos con código objeto resultantes de la compilación en un único código ejecutable. Al proceso que combina compilación de varios archivos fuente para luego

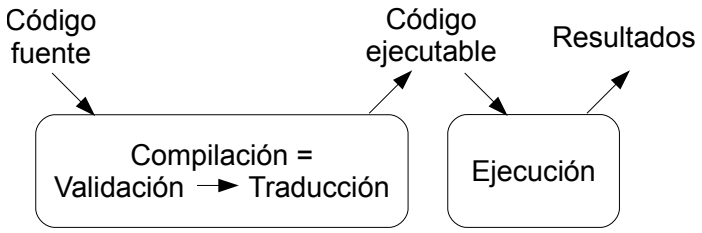

Figura 2: Proceso de ejecución de lenguajes compilados.

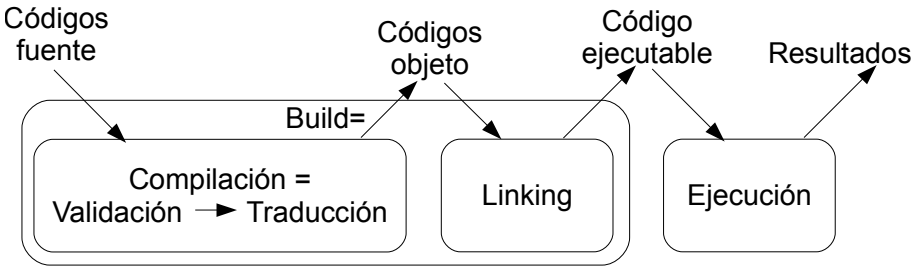

Figura 3: Proceso de ejecución de lenguajes compilados con compilación separada.

linkear los programas objeto resultantes se lo denomina *build*. En la figura 3 se puede observar un gráfico que ilustra el proceso de compilación separada.

Mientras que GOBSTONES es un lenguaje interpretado, CGOBSTONES es un lenguaje compilado que permite compilación separada. En resumen, para que un programa CGOBSTONES pueda ser ejecutado son necesarios tres pasos:

Compilación Validar el código fuente y traducirlo a código objeto.

Linking Combinar varios programas objeto a un único ejecutable.

Ejecución Llevar a cabo las acciones descritas por el programa.

3. Expresiones, valores y tipos

3.1. Valores

Los valores de CGOBSTONES son equivalentes a los de GOBSTONES[5]:

- Números: `1`, `2`, `3`, ...
- Booleanos: `true` y `false`
- Colores: `Verde`, `Rojo`, `Azul` y `Negro`

[5]En CGOBSTONES es posible también crear nuevos tipos de datos y valores, más adelante se introducirán las construcciones del lenguaje necesarias para ello.

- Direcciones: `Norte`, `Sur`, `Este` y `Oeste`

La única diferencia es que en CGobstones los valores booleanos se escriben en minúscula (`true` y `false`).

3.2. Parámetros

En Gobstones un parámetro es un identificador que denota a un valor que puede ser diferente en cada invocación de un procedimiento. Este comportamiento se mantiene en CGobstones aunque con una diferencia importante: se requiere la indicación *explícita* del tipo de valores que puede tomar cada uno de los parámetros. Los tipos de datos se indican de la siguiente manera:

- `Color`: Para un parámetro que puede tomar los valores `Azul`, `Negro`, `Rojoo Verde`.
- `Dir`: Para un parámetro que puede tomar los valores `Norte`, `Este`, `Suro Oeste`.
- `int`: Para un parámetro que puede tomar valores enteros (`1`, `2`, etc).
- `bool`: Para un parámetro que puede tomar los valores `true` y `false`.

Debe notarse que los tipos `Color` y `Dir` se escriben comenzando con mayúscula, a diferencia de `int` y `bool` que van en minúscula.

La indicación del tipo de un parámetro se realiza anteponiendo al nombre del parámetro el tipo de datos correspondiente. Por ejemplo para un parámetro de nombre `c` que espera un color como valor posible se escribe: `Color c`. Esta indicación se realiza únicamente en la declaración del procedimiento.

Resumiendo, el siguiente procedimiento Gobstones:

```
procedure Poner_Al_(colorAPoner, hacia)
  {
    Mover(hacia)
    Poner(colorAPoner)
    Mover(opuesto(hacia))
  }
```

puede escribirse en CGobstones de la siguiente manera:

```
void Poner_Al_(Color colorAPoner, Dir hacia)
  {
    Mover(hacia);
    Poner(colorAPoner);
    Mover(opuesto(hacia));
  }
```

En el ejemplo es posible verificar que la utilización del parámetro en expresiones no se ve afectado por la indicación de tipos, sólo la declaración del procedimiento tiene la indicación de tipos.

Resumiendo, la definición del concepto de parámetro se modifica de la siguiente manera:

Definición 2 *Un* parámetro *es un identificador que denota a un valor que puede ser diferente en cada invocación de un procedimiento. La forma de definir procedimientos con parámetros es*

`void` <*procName*>`(`<*params*>`)`
<*bloque*>

donde <params> es una lista de <tipo> <identificador> separados por comas. Los identificadores usados como parámetros deben comenzar con minúscula[6].

La invocación a un procedimiento con parámetros es igual en GOBSTONES y CGOBSTONES. Por ejemplo:

```
Poner_Al_(Verde, Oeste)
```

Se puede notar que tampoco aquí es necesario indicar el tipo de datos, solamente se debe proveer una expresión que denote un valor del tipo especificado por el parámetro.

Naturalmente esa información de tipos disponible ahora permite que el validador sea más estricto al controlar si un programa es válido. Por ejemplo el siguiente programa sería inválido teniendo en cuenta la anterior definición del procedimiento `Poner_Al_`

```
Poner_Al_(3, Oeste)       // Error de tipos
```

ya que `3` no es una expresión que denote un `Color`.

El lenguaje CGOBSTONES incorpora la capacidad de realizar conversiones automáticas entre valores de distinto tipo. Esto puede hacer que un programa como el anterior pueda ser considerado válido en algunos entornos de trabajo. Se recomienda sin embargo minimizar la utilización de esta capacidad ya que es potencial fuente de muchos errores difíciles de detectar.

En el resto de ese documento se considerará a dichas conversiones como inexistentes y, bajo esa premisa, un programa como el anterior es inválido por inconsistencia de tipos entre la expresión `3` y la definición del parámetro `colorAPoner`.

Algunas consideraciones finales:

- Al igual que en GOBSTONES, los parámetros de un procedimiento sólo pueden ser utilizados *dentro* de ese procedimiento.
- En CGOBSTONES los parámetros pueden alterarse al igual que una variable. Sin embargo esta práctica es poco recomendable; un parámetro se define como un valor que puede ser diferente en *cada invocación* de un procedimiento, pero se mantiene inalterada durante toda la invocación a ese procedimiento.
- Es importante notar que el tipo del parámetro se indica **únicamente** en la definición del procedimiento, los demás usos del parámetro *no se ven afectados*. Tanto la invocación de un procedimiento con parámetros como el uso del parámetro dentro del procedimiento en expresiones tienen la misma forma que en GOBSTONES.

3.3. Variables y asignación

En GOBSTONES una variable es un identificador que se utiliza para denotar algún valor en cierto momento de la ejecución de un programa. En CGOBSTONES también es posible utilizar variables, aunque se agrega la obligación de explicitar el tipo de los valores que puede denotar una variable. Esto es naturalmente similar a la indicación de tipos utilizada en el capítulo anterior para los parámetros de un procedimiento. La indicación de tipos se realiza mediante una construcción especial del lenguaje llamada *declaración de variable*. En CGOBSTONES sólo es posible utilizar un identificador como variable si fue previamente declarado como variable.

[6] Aunque en realidad esto no es obligatorio en CGOBSTONES, se recomienda mantener la convención usual de GOBSTONES por cuestiones de legibilidad del código

Definición 3 *Una* declaración de variable *es el mecanismo utilizado para disponer de una nueva variable en un procedimiento. La forma de declarar variables es la siguiente*

<tipo> <identificador>;

Los identificadores usados como variables deben comenzar con minúscula[7] *y deben ser únicos dentro del procedimiento.*

Los tipos que podemos utilizar al declarar una variable son los mismos que para los parámetros: `int`, `bool`, `Color` y `Dir`. Por ejemplo,

```
int contador;
```

declara una variable de nombre `contador` que tomará valores numéricos.

Al igual que en GOBSTONES para establecer la correspondencia entre una variable y el valor que denota se utiliza una asignación. En CGOBSTONES la asignación se realiza escribiendo la variable, luego el signo `=`, y luego la expresión que la variable nombra. Se puede ver que la funcionalidad es similar a la de Gobstones, utilizando el signo `=` en lugar de `:=`. Por ejemplo:

```
int cantidadRojas;
cantidadRojas = nroBolitas(Rojo);
```

establece que la variable llamada `cantidadRojas` nombre al número de bolitas de la celda en la que se encuentra el cabezal al ejecutar este comando.

> **Observación:** la asignación (`=`) y la comparación (`==`) son muy fáciles de confundir y son una fuente importante de errores. A diferencia de lo que hubiera ocurrido en GOBSTONES, el compilador CGOBSTONES en muchos casos no es capaz de detectar este error y puede suceder que un programa con este error sea aceptado como válido.
> Los errores de este tipo hacen que el programa tenga un comportamiento inesperado y son difíciles de detectar. Por eso es **muy importante** asegurarse de escribir correctamente las asignaciones y comparaciones, para no encontrarse con este tipo de problemas luego.

Naturalmente los valores que se pueden asociar a una variable deberán coincidir con el tipo con el que fue declarada. Por lo tanto el siguiente código es considerado inválido:

```
Color c;
c = Norte;    // Error de tipos
```

También es posible inicializar el valor de una variable en el mismo momento de la declaración, por lo tanto las formas:

```
Color c = Azul;
```

y

```
Color c;
c = Azul;
```

son equivalentes.

Nuevamente se destaca que el tipo de la variable sólo es necesario indicarlo en la declaración. Tanto las posteriores asignaciones a esa variable como su uso para denotar un valor en otras expresiones no deben incluir la información de tipos. Por ejemplo:

[7] Se recomienda respetar esta convención.

```
Color c;
c = siguiente(Azul);

Direccion d;
if (hayBolitas(c)) {
   d = Norte;
}
else {
   d = Sur;
}
```

3.4. Operaciones predefinidas

La amplia mayoría de las operaciones presentes en GOBSTONES estarán presentes en CGOBSTONES. Las diferencias son las siguientes:

- Para la división entera debe utilizarse el símbolo `/` en lugar de `div`.
- Para el resto de la división entera se usa `%` en lugar de `mod`.
- No existe una operación primitiva para la exponenciación (equivalente al `^` de GOBSTONES).
- Para la operación distinto se utiliza `!=` en lugar de `/=`.
- La operación `opuesto` sólo es válida para ser utilizada con direcciones, mientras que `-` sólo es válida para valores numéricos

El resto de las operaciones son iguales a GOBSTONES, a saber:

- `not`, `&&`, `||`, `minBool` y `maxBool` para valores booleanos
- `+`, `-` y `*` para valores numéricos
- `minColor` y `maxColor` para los colores
- `minDir` y `maxDir` para direcciones
- la igualdad `==` para todos los tipos básicos
- las comparaciones de orden `<`, `<=`, `>` y `>=`
- las operaciones de movimiento en el orden `siguiente` y `previo`
- las expresiones relativas a la operatoria del cabezal `hayBolitas`, `puedeMover` y `nroBolitas`

4. Estructuras de control

Las estructuras de control de CGOBSTONES son equivalentes a las de GOBSTONES, siendo la principal diferencia la ausencia de `repeatWith`.

A modo de repaso, la alternativa condicional tiene la siguiente forma:

`if (`<*condición*>`)`
<*unBloque*>
`else`
<*otroBloque*>

siendo <*condicion*> una expresión que describe un valor de verdad (booleano), y donde los bloques <*unBloque*> y <*otroBloque*> son bloques de código cualesquiera.

También podemos utilizar la variante en la que no existe alternativa para la rama del `else`:

`if (`<*condición*>`)`
<*unBloque*>

La repetición condicional también es equivalente a la de GOBSTONES:

`while (`<*condición*>`)`
<*bloque*>

donde <*condición*> es una expresión booleana y <*bloque*> un bloque cualquiera.

5. Procedimientos que devuelven valores

5.1. Definición

El rol que en GOBSTONES cubrían las funciones, en CGOBSTONES es cubierto por procedimientos que cuentan una característica adicional: pueden devolver un valor. Al igual que en GOBSTONES, el valor a devolver se indica con `return`.

En la declaración de un procedimiento que devuelve valores se requiere de la especificación del tipo de valores que puede devolver. Esta indicación del tipo retorno reemplazará a la palabra `void` utilizada para los demás procedimientos. En resumen, un procedimiento CGOBSTONES que devuelve valores se define de la siguiente manera:

<*tipo*> <*procName*>`(`<*params*>`)`
`{`
    <*comando*>
    `return(`<*expr*>`)`
`}`

siendo <*tipo*> una indicación de un tipo de CGOBSTONES, <*procName*> un identificador, <*params*> una lista de <*tipo*> <*identificador*> separados por comas, <*expr*> una expresión que denota un valor cuyo tipo coincida con el tipo especificado para la función y <*comando*> un comando (que puede no aparecer).

Mientras que en GOBSTONES el `return` es el punto de salida de una función, en CGOBSTONES es un comando que se utiliza para finalizar un procedimiento que retorna valores. Si bien el lenguaje CGOBSTONES no exige que el comando `return` sea el último del procedimiento, se recomienda respetar esta forma para incrementar la claridad del código. Lamentablemente en CGOBSTONES no es posible devolver más de un valor. Para emular este comportamiento existen los valores compuestos, que serán introducidos más adelante.

Por ejemplo, el siguiente procedimiento devuelve el color con mayor cantidad de bolitas en la celda actual:

```
Color colorMaximo()
  /* PROPÓSITO:
      * calcula el color de las bolitas de las que
        hay más en la celda actual, con repetición
     PRECONDICIÓN:
      * ninguna, es una operación total
     OBSERVACIÓN:
      * notar que el código es independiente de la cantidad de colores
```

```
      y del orden de los mismos
    * organizado como recorrido sobre los colores
*/
{
  // Iniciar recorrido
  Color colorAPoner = minColor();
  Color c           = minColor();
  int nroMaximo     = nroBolitas(minColor());
  int finRecorrido  = (colorAPoner == maxColor());

  while (not finRecorrido)
    {
      // Cambiar de color
      Color c = siguiente(c);

      // Procesar un color
      if (nroBolitas(c) > nroMaximo)
        { // Hay más bolitas de color c que de las
          // que se tenían como máximo hasta el momento
          colorAPoner = c;
          nroMaximo = nroBolitas(c);
        }

      // Si se llegó al último color, terminó el recorrido
      finRecorrido = (c == maxColor());
    }

    // No es necesario procesar el último elemento por separado.
    // No hay acciones para finalizar recorrido

  return (colorAPoner);
}
```

5.2. Modificación del tablero

Naturalmente los procedimientos que devuelven valores pueden contener comandos que modifiquen el tablero. Entonces, si bien es factible utilizar este tipo de procedimientos en un rol similar a las funciones de GOBSTONES, los procedimientos pueden alterar el estado en forma permanente. Esto deberá ser tenido en cuenta al considerar el efecto del procedimiento.

Por ejemplo, en GOBSTONES la función que calcula la distancia al borde del tablero en una dirección dada, podría programarse así:

```
function distanciaAlExtremo(dir)
  {- PROPÓSITO:
      * retorna la distancia al extremo en la dirección dada
     PRECONDICIÓN:
      * ninguna, es una operación total
     OBSERVACIONES:
      * es una función, el estado no se altera.
  -}
  {
    dist := 0               -- inicializar contador
```

```
  while (puedeMover(dir))
    {
      Mover(dir)           -- mover a la siguiente celda
      dist := dist + 1     -- contar los movimientos realizados
    }

  return (distancia)
}
```

Esta función aprovecha la característica de las funciones GOBSTONES de no afectar al estado global del tablero. Por lo tanto, se permite mover el cabezal hasta el extremo en la dirección indicada, sin que eso tenga efecto en el contexto desde el que se invoque a la función.

Si bien en CGOBSTONES es posible emular el comportamiento de dicha función con un procedimiento que devuelva un valor, la traducción no puede hacerse *literalmente*. Por ser un procedimiento, los comandos ejecutados por el cabezal dentro del mismo tendrán efectos permanentes sobre el tablero. Por lo tanto una traducción literal a CGOBSTONES de la función anterior tendrá el efecto lateral de cambiar el cabezal de lugar.

```
int DistanciaAlExtremo(Dir dir)
  /* PROPÓSITO:
      * retorna la distancia al extremo en la posición dada
     PRECONDICIÓN:
      * ninguna, es una operación total
     EFECTO:
      * el cabezal queda en el extremo correspondiente
        a la dirección dir
     OBSERVACIONES:
      * programado de esta forma, el nombre del procedimiento resulta
        confuso
  */
  {
    int dist := 0;           // inicializar contador
    while (puedeMover(dir))
      {
        Mover(dir);          // mover a la siguiente celda
        dist = dist + 1;     // contar los movimientos realizados
      }

    return (dist);
  }
```

Como se indica en el comentario, el nombre del procedimiento resulta confuso; esto se debe a que, además de cumplir el propósito de retornar la distancia al extremo, tiene el efecto de cambiar la posición del cabezal. Esta característica se puede interpretar de dos maneras:

- Considerar al efecto como parte del procedimiento. En ese caso bastará con cambiar el nombre del procedimiento por uno más representativo, un nombre posible sería `IrAlExtremoCalculandoDistancia`.
- Modificar el procedimiento de forma que su efecto sobre el tablero sea nulo.

A los procedimientos que devuelven valores y no producen efectos sobre el tablero se los denomina *funciones*. A diferencia de GOBSTONES, donde el lenguaje impide que las funciones tengan efectos sobre el tablero, en CGOBSTONES es el programador el que debe garantizar eso.

A veces es posible modificar un procedimiento que devuelve un valor para transformarlo en una función. Para ello se debe *deshacer* los comandos que haya ejecutado el cabezal. Por ejemplo, el procedimiento anterior puede transformarse en una función si se devuelve el cabezal a la posición original.

```
int distanciaAlExtremo(Dir dir)
  /* PROPÓSITO:
      * retorna la distancia al extremo en la posición dada
     PRECONDICIÓN:
      * ninguna, es una operación total
     EFECTO:
      * nulo, el cabezal termina en la posición en que empezó
  */
  {
    int dist := 0;           // inicializar contador
    while (puedeMover(dir))
      {
        Mover(dir);          // mover a la siguiente celda
        dist = dist + 1;     // contar los movimientos realizados
      }

    // devolver el cabezal a su posición original
    MoverN(opuesta(dir), distancia);

    return (distancia);
  }
```

Diferenciar a los procedimientos que devuelven valores de las funciones resulta de vital importancia al programar. Confundirlos puede llevar a utilizar un procedimiento como si fuera una función, es decir, desconocer el efecto del mismo; esto puede llevar a un comportamiento inesperado del programa, dependiendo de la naturaleza de ese efecto.

Para evitar esta confusión se recomienda mantener la nomenclatura de GOBSTONES, utilizando identificadores que comiencen con minúscula para los nombres de las funciones e identificadores que comiencen con mayúscula para los procedimientos (ya sea que devuelvan un valor o no).

5.3. Invocación

La forma más común de invocar a un procedimiento que devuelve valores es como parte de una asignación. La invocación es similar a la utilizada para los procedimientos `void`, almacenando el valor resultante en una variable. Por ejemplo:

```
Color c;
c = colorMaximo();

int distanciaAlNorte;
distanciaAlNorte = IrAlExtremoCalculandoDistancia(Norte);
```

también es posible realizar la asignación al mismo tiempo que la declaración de la variable:

```
Color c = colorMaximo();
```

Naturalmente el tipo de la variable deberá coincidir con el tipo de los valores que retorna el procedimiento Por ejemplo, la utilización del procedimiento anterior en la siguiente asignación producirá un error:

```
Dir d = maxColor();       // Error de tipos!!!
```

Al igual que sucede con las variables y parámetros, en muchos casos puede suceder que el compilador CGobstones no sea capaz de detectar algunos errores de tipos. Se recomienda tener especial precaución para escribir expresiones con los tipos correctos; como se dijo anteriormente, este tipo de errores pueden producir un comportamiento errático del programa, dificultando su corrección.

Dado que un procedimiento que devuelve un valor produce a su vez un efecto sobre el tablero, es posible que a veces resulte más interesante ese efecto que el valor calculado. En esos casos se puede obviar la asignación e invocar al procedimiento en forma de comando, siguiendo el mismo mecanismo utilizado para los procedimientos `void`. Por ejemplo, el procedimiento anterior puede ser utilizado para mover el cabezal al un extremo, de la siguiente manera:

```
IrAlExtremoCalculandoDistancia(Norte);
          // Se busca el efecto de mover el cabezal al extremo Norte
          // Se ignora la distancia devuelta
```

Si bien el lenguaje CGobstones permite hacer lo mismo con funciones, las funciones no tienen efectos y no tienen utilidad si son invocadas como comandos. Por lo tanto el siguiente código se considera incorrecto:

```
colorMaximo();  // No tiene sentido usar una función como comando!!!
```

Al igual que en Gobstones, las funciones pueden ser utilizadas para denotar valores dentro de cualquier expresión. Por ejemplo:

```
// Poner bolitas rojas hasta igualar al color con más bolitas
PonerN(Rojo, nroBolitas(colorMaximo()) - nroBolitas(Rojo));

// Moverse al extremo lateral más cercano
if (distanciaAlExtremo(Este) <= distanciaAlExtremo(Oeste))
  {
    IrAlExtremoCalculandoDistancia(Este);
  }
else
  {
    IrAlExtremoCalculandoDistancia(Este);
  }
```

Se considera un error utilizar procedimientos que devuelven valores de la forma anterior, reservándose su uso únicamente para asignaciones. Si bien el lenguaje lo permite, puede resultar complejo comprender los posibles efectos de un procedimiento invocado dentro de una expresión, por ejemplo la siguiente porción de código

```
if (maxColor() == Rojo || IrAlExtremoCalculandoDistancia(Este) > 4)
                          // ¡Peligro! ¡Se usa un procedimento
                          // en una expresión!
  {
    <comandos> // ¿Dónde está ubicado el cabezal
               // al ejecutar estos comandos?
  }
```

no es posible saber si se invocó al procedimiento `IrAlExtremoCalculandoDistancia` o no[8]. Entonces se desconoce la posición del cabezal al ejecutar los comandos dentro del `if` y no se puede asegurar que se cumplan las precondiciones de dichos comandos.

Una forma de corregir el código anterior es utilizar una variable intermedia. Al utilizar el procedimiento en una asignación, se garantiza que siempre el procedimiento es ejecutado y se evita la situación anterior.

```
                // El procedimiento se ejecuta siempre
int distancia = IrAlExtremoCalculandoDistancia(Este);
if (maxColor() == Rojo || distancia > 4)
  {
    <comandos> // El cabezal está en el extremo Este
  }
```

Como la subexpresión `distancia >4` no tiene efecto alguno es indiferente que se evalúe o no, y por lo tanto es seguro utilizarla de esa manera. Se recomienda siempre utilizar los procedimientos que devuelven valores en asignaciones y nunca en otro tipo de expresiones.

Otra forma de corregir el mismo código es utilizando una función en lugar del procedimiento. Al tener las funciones efecto nulo, es seguro utilizarlas en expresiones de todo tipo.

```
                          // La función tiene efecto nulo, por lo
                          // tanto este código es seguro
if (maxColor() == Rojo || distanciaAlExtremo(Este) > 4)
  {
    <comandos> // El cabezal no se movió de su posición original
  }
```

Cabe mencionar que ambas soluciones no son equivalentes. En la primera el cabezal se moverá al extremo `Este` del tablero, mientras que en la segunda solución el cabezal se mantiene en su posición original. Sin embargo, en ambos casos es posible conocer con seguridad si el cabezal se movió o no.

En resumen, existen tres formas básicas de invocar procedimientos o funciones, que no se deben confundir:

- Una invocación a procedimiento puede ser utilizada como un comando, tanto si devuelve un valor como si no lo hace. No tiene sentido utilizar funciones como comandos.
- Una función o un procedimiento que devuelve un valor puede ser invocado en una asignación. No tiene sentido hacerlo con procedimientos que no devuelven valores.
- Una función puede ser utilizada en expresiones arbitrarias. Hacerlo con procedimientos no es recomendable y se considera un error, ya que el código resultante suele ser propenso a errores.

6. CONTINUARÁ...

[8] Tanto GOBSTONES como CGOBSTONES evalúan las expresiones booleanas de izquierda a derecha, por lo tanto en caso de que `maxColor() == Rojo` verdadero el segundo parámetro del `||` no se evalúa.

A. El entorno de trabajo CodeBlocks

A.1. Instalación el entorno

El entorno de trabajo se puede descargar de `www.codeblocks.org`. En la página ingresar al menú `downloads` y luego seleccionar la opción `Download the binary release`. Una vez allí seleccionar la versión más nueva disponible (al momento de escribir este documento la última versión disponible es la 8.02); seleccionar la opción completa (la que incluye `mingw` en el nombre).

La instalación es directa al ejecutar el programa descargado.

A.2. Comenzar a programar CGobstones

Como un programa en CGobstones puede consistir de muchos archivos, se necesita una estructura más grande que defina todos los archivos que conforman un programa. En CodeBlocks esa estructura se denomina *proyecto*.

De todos los archivos que se incluyan en un proyecto, exactamente uno de esos archivos debe contener un procedimiento `main`. Por eso, cada programa a realizar deberá construirse en un proyecto nuevo, que contendrá como mínimo el archivo conteniendo al procedimiento `main`; normalmente a este archivo se lo denominará `main.cpp`.

La forma más simple de organizar los proyectos en CodeBlocks es utilizando un directorio para cada proyecto. En ese directorio se encontrará el archivo que describe al proyecto (con extensión `.cbp`), el `main.cpp` y los archivos correspondientes a las bibliotecas que esté utilizando Por lo general, para cada biblioteca utilizada se tendrá un archivo `.cpp` y uno `.h`.

La forma de organizar cada proyecto en un directorio tiene como desventaja que obliga a copiar los archivos de las bibliotecas utilizadas en el directorio de cada proyecto. Por otro lado simplifica tanto el manejo de archivos como la posibilidad de compartir proyectos, entonces resulta una forma adecuada para una etapa inicial. Se deja para una etapa más avanzada del aprendizaje la utilización de esquemas de organización más adecuados para un entorno productivo.

En resumen, los pasos necesarios para crear un programa CGobstones son:

Creación del proyecto Para eso se utiliza el menú `file -> new -> project` y se siguen los pasos descriptos a continuación:

- En la primer pantalla aparecen un montón de posibilidades de proyecto; elegir `Console Application`.
- A continuación se permite elegir el lenguaje, el lenguaje sugerido por defecto es correcto (`cpp`), presionar `next`.
- En la siguiente pantalla se debe definir el nombre del proyecto. También es posible indicar en qué lugar del disco rígido crear el proyecto. En ese lugar se creará el directorio del proyecto mencionado anteriormente y dentro de ese directorio se crearán los archivos `.cbp` y `main.cpp`.

Muchas veces es necesario enviar un programa por mail. Siguiendo la organización antes descripta, todos los archivos del proyecto están en un mismo directorio; entonces una forma fácil es enviar el directorio *zippeado*. Al hacer eso se debe tener en cuenta que dentro de ese directorio (en los subdirectorios `obj` y `bin`) se guardan los resultados de la compilación. Los clientes de mail suelen restringir el envío archivos ejecutables, por lo tanto los subdirectorios `obj` y `bin` deben ser excluidos al armar el `zip`. Como estos archivos fueron construidos a partir del código fuente, podrán ser reconstruidos posteriormente por quien reciba el mail.

A.3. Compilación y ejecución

El entorno de trabajo CodeBlocks provee las siguiente opciones para ejecutar un programa CGobstones:

Construcción (build) construir el programa ejecutable. Esto combina la compilación y el linking.

Ejecución (run) tomar un programa previamente compilado y hacer que el cabezal ejecute cada una de las acciones descritas por el programa.

Construcción y ejecución (build + run) realizar las dos operaciones anteriores secuencialmente. Obviamente en el caso en que el build falle la ejecución no se puede llevar a cabo.

Cuando un programa no puede ser traducido a código objeto, se denomina *error de compilación*. Estos errores no son realmente nuevos, de hecho son similares a los errores de sintaxis que pueden ocurrir en Gobstones se intenta ejecutar un programa inválido. Un error nuevo que sí puede ocurrir es utilizar la operación "ejecutar" sin haber compilado previamente. Esto puede llevar a dos situaciones diferentes:

- El programa nunca fue compilado anteriormente y por lo tanto no se puede ejecutar.
- El programa fue compilado anteriormente, pero modificado después de la última compilación, con lo cual se corre el riesgo de estar ejecutando una versión vieja y suele pasar que no refleja los cambios realizados.

Para evitar ambas situaciones lo más fácil es utilizar mayormente la opción de *build + execute*, al menos en la etapa de aprendizaje.

Como el proceso de compilación suele ser costoso en tiempo, muchos entornos de trabajo intentan minimizar la cantidad de compilaciones necesarias. En estos casos suele pasar que el entorno de trabajo compila únicamente los archivos que han sido modificados desde la última compilación realizada. Si bien este proceso optimiza el tiempo necesario para realizar el build del programa, en algunos casos las dependencias entre los distintos archivos hacen necesario realizar un build completo. Para esos casos se agrega una opción más a las anteriores:

Rebuild compilar y linkear nuevamente todos los archivos del programa. Para esto previamente se eliminan todos los resultados intermedios de compilaciones previas.

Se destaca que los dos archivos de la biblioteca CGobstones entrarán en juego en dos etapas distintas del proceso. El `Gobstones.h` contiene la información de *qué procedimientos* forman parte de la biblioteca y es necesario para compilar. En cambio el `Gobstones.cpp` se incluirá en la etapa de linking. Los errores que pueden ocurrir en cada etapa serán distintos y es importante comprender estas etapas para poder encontrar soluciones a los mismos.

A.4. Estructura de carpetas

En un programa CGobstones es importante entender la estructura de carpetas. Cada carpeta del proyecto tiene un lugar y un papel que jugar en el mismo, y debe saberse cuál es, para decirle al CodeBlocks de manera correcta dónde está cada cosa. La estructura de carpetas tiene que tener una carpeta que es el lugar donde están todos los demás; por ejemplo llamémosla "MisProyectos" (cuál nombre no es tan importante, pero

una vez que se elige uno tiene que ser ese y no otro...). En la carpeta "MisProyectos" tiene que estar la carpeta "CGobstones" con todos los archivos que se bajan de la página de la materia, y una carpeta por cada proyecto que se defina. Supongamos que definimos un proyecto "Proy1". Entonces la estructura de carpetas quedaría así:

- MisProyectos
  - CGobstones
  - Proy1

Dentro del proyecto "Proy1" (en CodeBlocks) hay que decirle que use CGobstones, agregando su carpeta con "Add files recursively". Si es llega a borrar o mover esta carpeta ("CGobstones"), de lugar a otra carperta, el proyecto deja de encontrarlo. Por otro lado, en cada "`#include`" que se haga de archivos de CGobstones (como "List.h" o "CGobstones.h"), se debe indicarle al comando include dónde encontrar ese archivo. Eso se hace con un *path*. El *path* empieza a contar desde la ubicación donde se guarda el archivo ejecutable (el .exe), que en el CodeBlocks es la carpeta "bin" dentro de la raíz del proyecto (en el ejemplo, "Proy1"). Por eso, los includes de "List.h"deben decir

```
#include "../../CGobstones/List.h"
```

que dice que se deben subir dos lugares (uno a "Proy1" y otro a "MisProyectos", indicado por los dos "..") y después desde ahí bajar a "CGobstones" (que ya vimos, tiene que estar ahí).

Un detalle más al que en una primera aproximación puede no prestársele atención es que en realidad nada es fijo. O sea, ni que haya una carpeta raíz, ni que CGobstones esté donde esté, ni nada. Esas son cosas que el programador decide. Pero cuando uno trabaja con otros (y acá. como todos los proyectos usan CGobstones, es como trabajar con otros), se deciden lugares fijos para ciertas cosas y se respetan (normalmente esas decisiones se toman por lo que es más cómodo o más simple o más modificable). Es quizás lo más confuso el hecho de que nada es fijo, pero en cada equipo/herramienta/programa hay cosas fijas. Es algo que un programador aprende a percibir: "¿cuáles cosas son fijas acá?", "¿cuáles puedo cambiar, y cómo?". Para eso hay que prestar atención a los detalles.