Introducción a la Programación en Gobstones

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

Eduardo Bonelli
ebonelli@unq.edu.ar

2 de mayo de 2010

1. Introducción

El objetivo de este documento servir como introducción amena y gradual a Gobstones, y a algunas nociones básicas de programación. Gobstones es un lenguaje pensado para personas que no tienen conocimientos previos de lenguajes de programación. Se basa en elementos concretos simples: un tablero, bolitas de colores y un dispositivo que posee un cabezal con el cual puede recorrer el tablero, y al que se le pueden dar instrucciones sencillas mediante *comandos*. Además posee construcciones que permiten la combinación de comandos en otros más complejos, permitiendo la resolución de problemas no triviales. No es el propósito de Gobstones concentrarse en ningún aspecto de entrada/salida (o sea, la comunicación entre un programa y el mundo).

La versión actual es la 2.10, que posee varias diferencias con versiones previas, y fue diseñada para ser más simple de utilizar y comprender. Existe una herramienta muy sencilla que implementa Gobstones, y quizás en algún momento haya una herramienta más robusta. Se recomienda conseguir e instalar esta herramienta antes de empezar la lectura.

Es importante tener claro la forma en que fue pensado este documento, de manera de entender cómo es la mejor manera de leerlo. La idea es que cada concepto se presenta gradualmente, construyendo sobre los anteriores, y se van reforzando las ideas con ejemplos y ejercicios. Por esa razón, es mejor leerlo en orden por capítulos, sentado al lado de la computadora con la herramienta instalada, y realizando cada ejercicio, respondiendo cada pregunta, antes de continuar con la siguiente sección. Observar que esta forma de proceder no es necesariamente la misma con una guía de ejercicios. En ese caso, si un ejercicio no sale, debe continuarse con el siguiente. En ambos casos debe resolverse el problema consultando lo antes posible.

Si al leer tiene comentarios, sugerencias de mejoras, observaciones, etc. envíelos a alguna de las direcciones de mail de los autores.

La estructura del tutorial es la siguiente. En la sección 2 se describen los elementos básicos que se manejan con un programa Gobstones. En la sección 3 se comienzan a presentar elementos básicos de programación, y su manifestación en Gobstones. En el futuro habrá más secciones con los restantes elementos. Por ahora, la sección 5 establece el conocido "*Continuará...*" de todas las entregas en episodios. El documento finaliza con una sección de conclusiones (sección 9). En los apéndices se pueden encontrar la descripción detallada de la sintaxis de Gobstones (apéndice A) y la manera de conseguir e instalar la herramienta (apéndice B).

Índice

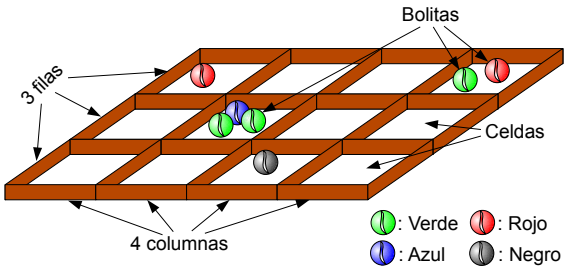


Figura 1: Un tablero de 3 filas y 4 columnas, en 3 dimensiones

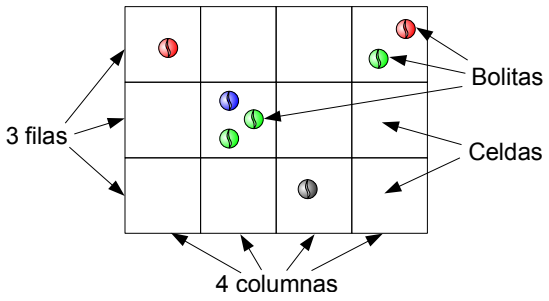


Figura 2: El mismo tablero en 2 dimensiones

2. Elementos básicos de Gobstones

2.1. Tablero y bolitas

El elemento más notable de Gobstones es el *tablero*. El tablero es una cuadrícula de *celdas* dispuestas en filas y columnas. Cada celda es un contenedor en el que puede haber *bolitas* de colores. En la versión actual de Gobstones existen bolitas de cuatro colores: `Verde`, `Rojo`, `Azul` y `Negro`. En las figuras 1 y 2 se observan dos representaciones de un tablero típico, con 3 filas y 4 columnas, y algunas bolitas.

Definición 1 *El* tablero *es una cuadrícula de* celdas. *Las celdas pueden contener* bolitas *de* colores.

Hay dos cuestiones del tablero que son importantes.

La primera es el hecho de que es finito. Siendo finito, habrá ciertas cosas que no se podrán realizar con él. Este tema se tratará en las siguientes secciones, cuando se presenten las acciones sobre el tablero. Es destacable que no hace falta conocer el tamaño del tablero, aunque el mismo podría determinarse de varias maneras, que se verán posteriormente.

La segunda cuestión es que las celdas poseen capacidad ilimitada. O sea, no hay límite en el número de bolitas que pueden contener. Esto puede parecer poco realista, pero está pensado para evitar complicaciones en el control de la capacidad.

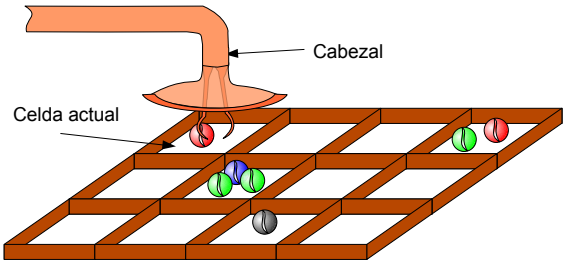


Figura 3: El tablero de la figura 1 con la representación del cabezal

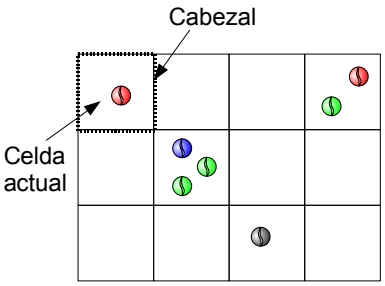


Figura 4: El mismo tablero en dos dimensiones

2.2. El cabezal

Tanto el tablero como las bolitas son elementos inanimados. El factor de movimiento en Gobstones viene dado por una máquina que puede operar sobre el tablero y las bolitas. Esta máquina puede realizar diversas acciones (siempre sobre una única celda por vez). Las acciones incluyen desplazarse a una celda vecina, poner y sacar bolitas en la celda sobre la que se encuentra, y consultar si hay bolitas en una celda, y cuántas hay.

La máquina, que será denominada *cabezal* para remarcar el hecho de que opera sobre una celda por vez, puede recibir instrucciones relativas a la celda sobre la que se encuentra el mismo, y a la que se denomina *celda actual*. En las figuras 3 y 4 se observa el tablero presentado antes con una representación del cabezal y la celda actual.

Definición 2 *El* cabezal *es una máquina que opera sobre una celda por vez, la* celda actual.

2.3. Comandos simples

La manera de darle instrucciones al cabezal es a través de un programa. Un *programa* Gobstones se puede entender como la descripción de las acciones que el cabezal debe intentar realizar cuando recibe la indicación de *ejecutar* el programa. Un programa está formado por *comandos*, que son descripciones de acciones individuales. Los comandos por sí mismos no son programas completos; al recibir la orden de ejecutar un programa, lo único que el cabezal hará es invocar a un procedimiento llamado `Main`. Esta idea se describirá en la sección 2.4.1.

Definición 3 *Un* programa *es una descripción de las acciones que el cabezal intentará realizar al ejecutar el mismo. Un programa está formado por comandos.*

Definición 4 *Un* comando *es la descripción de una acción individual.*

Por ejemplo, el comando `Poner(Verde)` describe la acción de agregar una bolita verde en la celda actual. Se asume que el cabezal tiene a su disposición una cantidad ilimitada de bolitas de cada color y, como ya se mencionó, que las celdas tienen capacidad infinita, con lo cual la acción de poner siempre puede realizarse.

La forma general del comando `Poner` está dada por la siguiente definición.

Definición 5 *El comando* `Poner(`$<$*color*$>$`)` *indica al cabezal que coloque una bolita del color* $<$*color*$>$ *en la celda actual.*

Los valores posibles para $<$*color*$>$ son los ya indicados: `Verde`, `Rojo`, `Azul` y `Negro`. Observar que el nombre $<$*color*$>$ se encuentra en diferente formato para indicar que no debe ir la palabra `color`, sino *un color particular*. De aquí en más, cuando aparezca el nombre $<$*color*$>$ de esta manera, querrá decir que debe elegirse la descripción de un color para colocar en ese lugar. O sea, son comandos válidos

- `Poner(Verde)`,
- `Poner(Rojo)`,
- `Poner(Azul)` y
- `Poner(Negro)`.

Cuando se agreguen otras formas de describir colores, tales formas también podrán ser usadas en donde se espera un $<$*color*$>$.

Definición 6 *Los valores posibles para un* $<$*color*$>$ *son* `Verde`*,* `Rojo`*,* `Azul` *o* `Negro`*.*

¿Qué debe hacerse para describir la indicación al cabezal que debe poner más de una bolita en la celda actual? Dicha descripción debe contener 2 comandos seguidos. La manera de colocar seguidos comandos en Gobstones es ponerlos en líneas separadas, por ejemplo

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

es un fragmento de programa que describe que al ejectuarlo, el cabezal debe colocar 2 bolitas verdes en la celda actual. Opcionalmente, pueden colocarse un punto y coma después de un comando. Esto es útil cuando se quiere colocar más de un comando en la misma línea. Así, los programas

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

y

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

también son válidos, y equivalentes al primero.

Definición 7 *La* secuenciación *de comandos se realiza colocando los mismos en líneas separadas, opcionalmente terminados con el caracter de punto y coma (* `;`*).*

Otra noción importante relacionada con los comandos es la delimitación de un grupo de comandos en un comando compuesto. Esto se lleva a cabo mediante lo que se denomina un *bloque*, que se compone de una serie de comandos delimitados por llaves. Entonces, un bloque de código válido sería

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

cuyo efecto al ser ejecutado sería colocar dos bolitas de cada color en la celda actual. Los bloques tendrán importancia en la definición de comandos compuestos.

Definición 8 *Un* bloque *es comando formado por un grupo de comandos delimitados por llaves (caracteres { y }).*

Un bloque también es un comando, con lo cual pueden secuenciarse con otros comandos. Entonces, es válido escribir

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

que es un grupo de comandos que tendrá el mismo efecto que el bloque dado previamente.

2.4. Procedimientos simples

Es importante entender que si bien se puede pensar un programa Gobstones de a un comando por vez (llamado *pensamiento operacional* pues se concentra en las operaciones individualmente), esta forma tiene como consecuencia que los problemas complejos sean difíciles de tratar. Es por ello que se recomienda pensar los programas en base a diferentes *tareas* que el cabezal debe realizar, y agruparlas de manera conveniente. El mecanismo utilizado para definir tareas en Gobstones es el de *procedimiento* que es un bloque al que se le asigna un nombre.

Para nombrar elementos se utilizan una forma especial de nombres, llamados *identificadores*. Un *identificador* es un grupo de letras sin interrupciones (o sea, no contiene espacios ni signos de puntuación).

Definición 9 *Un* identificador *es un nombre conformado exclusivamente por letras (mayúsculas o minúsculas), que se utiliza para denotar (o* identificar *o* referise a*) algún elemento.*

Teniendo la capacidad de denotar elementos a través de los identificadores, estamos en condiciones de definir procedimientos.

Definición 10 *Un* procedimiento simple *es un bloque con un nombre dado por un identificador. Su forma es*

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

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

Los paréntesis son necesarios; su utilidad se comprenderá en la sección 3.4, donde se definirán formas más complejas de procedimientos, para tener la capacidad de definir tareas más complicadas. Por el momento se verán las formas más simples de los mismos.

2.4.1. El procedimiento `Main`

Para que el cabezal sepa cuáles son todas las acciones que debe realizar antes de detenerse es necesaria alguna forma de indicarle cuáles son los comandos que debe intentar ejecutar. Tal mecanismo se lleva a cabo mediante un procedimiento particular, cuyo nombre es `Main`. Es el procedimiento más importante de un programa Gobstones que siempre debe estar presente y ser el último de los procedimientos declarados. Por ejemplo, el programa

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

es un programa completo, compuesto de la secuenciación de dos comandos `Poner`. Al indicarle al cabezal que ejecute el programa, se ejecutarán todos los comandos contenidos en el procedimiento `Main`.

Definición 11 *El* procedimiento principal *de un programa Gobstones es un procedimiento de nombre* `Main`*, que siempre debe estar, y debe aparecer como último procedimiento. El procedimiento* `Main` *determina completamente el comportamiento del cabezal al ejecutar el programa.*

Ejercicio 1 *Escribir un programa Gobstones que deposite una bolita verde, dos rojas, tres azules y cuatro negras en la celda actual, y ejecutarlo. ¿Qué puede concluirse acerca del estado del tablero al iniciar la ejecución del programa? ¿Y al finalizar la misma?*

> **Observación:** para poder ejecutar un programa Gobstones debe tipearlo en un archivo de texto (sin formato), e indicarle adecuadamente a la herramienta que ejecute el programa de ese archivo. Dado que pueden existir diferentes herramientas, y diferentes formas dentro de la misma herramienta para relizar esta tarea, este tutorial no dará detalles de operación de ninguna de ellas como parte integral del texto. En el apéndice B se pueden encontrar detalles de cómo obtener y utilizar una de tales herramientas.

Al escribir un programa es importante ser precisos respecto de la forma del mismo. Por ejemplo, no es lo mismo escribir `Main` que `main` o `Mani`. La herramienta dará un mensaje de error cuando esto suceda. La interpretación de este mensaje de error puede resultar más o menos compleja; comprenderlas tiene mucho de experiencia con una herramienta en particular, pues los mensajes varían drásticamente de una a otra. La sugerencia general al enfrentar errores de sintaxis es observar en qué punto del programa se reporta el error, y revisar por la precisión de lo que se escribió.

Ejercicio 2 *Escribir el siguiente programa Gobstones y ejecutarlo.*

```
procedur main() { poner(verde) }
```

Observar el mensaje de error que se produce, y corregir los errores hasta que el programa resulte ejecutable.

En el ejercicio 1 se pregunta acerca del estado inicial del tablero. El *estado del tablero* es la ubicación del cabezal y si hay o no bolitas en él, y em qué celdas. Es importante observar que todo lo que interesa de un programa Gobstones es el *efecto* que produce sobre el estado del tablero, o sea, los cambios que se realizarán sobre el mismo. Es por ello importante tener idea del estado inicial del tablero. Sin embargo, al escribir un programa Gobstones, no debería hacerse ninguna suposición acerca de este estado inicial. Cada herramienta es libre de usar como estado inicial uno cualquiera, e incluso podría ser que el estado inicial variase de una ejecución a otra.

Definición 12 *El* efecto *de un programa Gobstones es el conjunto de cambios que produce sobre el tablero inicial. No debe suponerse ningún estado inicial particular si se desea que el programa sea independiente de la herramienta.*

2.4.2. Procedimientos definidos por el programador

Otros procedimientos simples son aquellos que puede escribir el programador. La forma es la misma que la del procedimiento `Main`, salvo que el programador puede elegir cualquier nombre (que sea sólo formado por letras y que comience con mayúsculas), y que debe colocar sus definiciones antes que las de `Main`. Por ejemplo, un procedimiento definido (o declarado) por el programador podría ser

```
procedure PonerDosVerdes()
 {
   Poner(Verde); Poner(Verde)
 }
```

La forma general es la que se definió al principio de la sección.

La diferencia entre un procedimiento definido por el programador y el procedimiento `Main` es que mientras que este último es invocado de manera implícita al ejecutar el programa, un procedimiento definido por el programador debe ser invocado de manera explícita. El nombre de un procedimiento declarado seguido de paréntesis se puede utilizar como un comando. Entonces

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

es un programa Gobstones que coloca cuatro bolitas verdes en la celda actual.

Definición 13 *Un* procedimiento declarado por usuario *puede ser utilizado como* comando*. Se escribe el nombre seguido por paréntesis, de la siguiente forma*

<*procName*>`()`

El efecto de dicho comando es el mismo que el del bloque definido en la declaración del procedimiento.

3. Conceptos básicos de programación

3.1. Más comandos simples

Así como existe una forma de describir la acción del cabezal de poner una bolita en la celda actual, existe la acción opuesta, de sacarla. El comando `Sacar(Verde)` saca una bolita verde de la celda actual, si hay alguna.

La forma general del comando `Sacar` es similar a la de `Poner`, y está dada por la siguiente definición.

Definición 14 *El comando* `Sacar(<color>)` *indica al cabezal que quite una bolita del color <color> de la celda actual, si hay alguna.*

Debe observarse que el cabezal sólo puede realizar una acción de `Sacar` si se cumplen ciertos requisitos. Dado que el cabezal no es más que una máquina, y bastante limitada, no tiene ninguna manera de saber qué hacer cuando la descripción para la acción a realizar no incluya alguna situación. Cuando el cabezal recibe una orden que no puede ejecutar, toma una acción drástica: **se autodestruye**, y con él al tablero y todas las bolitas. Esto puede parecer exagerado para una persona, que siempre tiene otros cursos de acción para considerar, pero no lo es para una máquina, al menos no para una tan simple. Cuando un comando describe una acción que puede provocar la autodestrucción del cabezal en caso de que ciertos requisitos no se cumplan, se dice que es tal comando describe una *operación parcial*, pues sólo describe parcialmente la acción a realizar. Si en cambio la descripción conlleva una acción que siempre puede realizarse, se dice que es una *operación total*. Al requisito que debe cumplirse antes de intentar ejecutar una operación parcial para estar seguro que la misma no falla se la denomina *precondición* de la operación. De esta manera, `Poner` es una operación total, y `Sacar` es una operación parcial y su precondición es que haya una bolita del color indicado en la celda actual.

Definición 15 *Una* operación total *es la descripción de una acción que, al no tener requisitos para que el cabezal pueda llevarla a cabo, siempre puede ser realizada.*

Definición 16 *Una* operación parcial *es la descripción de una acción que precisa que ciertos requisitos se cumplan para que el cabezal pueda llevarla a cabo, y que en caso que los mismos no se cumplan, provoca la autodestrucción del cabezal y todos los elementos.*

Definición 17 *La* precondición *de una operación parcial expresa los requisitos que deben cumplirse para que la ejecución de la acción indicada por la operación pueda ser llevada a cabo sin provocar la autodestrucción del cabezal.*

De aquí en más, al presentar nuevos comandos, se establecerá si los mismos describen operaciones totales o parciales, y en caso de que sean parciales, cuáles son sus precondiciones.

Ejercicio 3 *Escribir un programa Gobstones que saque una bolita de la celda actual sin que haya ninguna, y comprobar la forma en que se manifiesta la autodestrucción del cabezal, el tablero y los otros elementos.*

Ejercicio 4 *Escribir un programa Gobstones que saque una bolita de la celda actual, pero que no produzca la autodestrucción del cabezal. ¿Cómo se puede asegurar que la precondición del comando* `Sacar` *se cumpla?*

Otro comando que describe una operación parcial es el que permite al cabezal moverse por el tablero. El comando `Mover(Norte)` describe la acción de que el cabezal se debe desplazar una celda hacia el norte, si hay alguna celda en esa dirección. El formato general está dado por esta definición

Definición 18 *El comando* `Mover(<dir>)` *indica al cabezal que debe moverse una celda en la dirección <dir>, si es posible.*

Las direcciones posibles (los *valores* que puede tomar *<dir>*) son `Norte`, `Sur`, `Este` y `Oeste`.

Definición 19 *Los valores posibles para una dirección <dir> son* `Norte`*,* `Sur`*,* `Este` *y* `Oeste`*.*

Como se mencionó, `Mover` es una operación parcial. Su parcialidad está dada por el hecho de que el tablero es finito, y por lo tanto hay celdas que no tienen celda vecina en algunas direcciones. Por ejemplo, la celda de la esquina inferior izquierda del tablero no tiene celdas ni al `Sur` ni al `Oeste`, y es la única celda con esta característica. Observar que dado que el tamaño del tablero no se conoce a priori, puede ser que si un programa que no controle adecuadamente sus movimientos provoque la autodestrucción del cabezal. Por ejemplo

```
procedure Main()
 {
   Mover(Sur);Mover(Oeste)
 }
```

es un programa que tiene como precondición comenzar su ejecución en cualquier celda que no se encuentre en la fila de más al sur, ni en la columna de más al oeste. En caso que la ejecución del programa arranque en una de dichas celdas, provocará la autodestrucción del cabezal.

Puesto que la posición inicial del cabezal es desconocida, es útil contar con el comando `IrAlOrigen`. Este comando, cuya forma es simplemente `IrAlOrigen()`, describe una operación total cuyo efecto es posicionar el cabezal en la esquina inferior izquierda del tablero. Posteriormente, en la sección 4.4 se verán las herramientas necesarias para que este comando pueda ser reemplazado por un procedimento equivalente definido por el programador.

Asimismo, y puesto que el contenido de las celdas es arbitrario, es útil contar con el comando `VaciarTablero`. Este comando, cuya forma es simplemente `VaciarTablero()`, describe una operación total cuyo efecto es eliminar la totalidad de las bolitas del tablero, sin alterar la posición del cabezal. Al igual que el comando `IrAlOrigen`, el comando `VaciarTablero` puede ser reemplazado por un procedimiento definido por el programador, aunque para ello se requiere casi la totalidad del poder del lenguaje.

3.2. Utilizando adecuadamente procedimientos

Combinando las operaciones de `Mover` y `Poner` se pueden simular "dibujos" utilizando las bolitas como colores. Por ejemplo, el programa

```
procedure Main()
 {
   Mover(Norte); Poner(Negro)
   Mover(Este);  Poner(Negro)
   Mover(Sur);   Poner(Negro)
   Mover(Oeste); Poner(Negro)
 }
```

dibuja un pequeño cuadrado negro de dos celdas de lado.

Ejercicio 5 *Ejecutar el programa Gobstones dado antes, y comprobar que dibuja el cuadrado. ¿Cuál es la precondición de este programa?*

Es conveniente, como ya se mencionó, que se utilicen procedimientos para nombrar adecuadamente las tareas, pues al escribir programas para solucionar tareas más complejas, se vuelve imposible determinar adecuadamente la corrección del programa si se razona de manera operacional. Esta forma de pensar un programa sólo sirve con los programas más simples. Pensar por tareas permite generalizar adecuadamente el programa. Por esta razón, el programa anterior podría haberse escrito como

```
procedure DibujarLineaHaciaElNorte()
 { Mover(Norte); Poner(Negro) }
procedure DibujarLineaHaciaElEste()
 { Mover(Este); Poner(Negro) }
procedure DibujarLineaHaciaElSur()
 { Mover(Sur); Poner(Negro) }
procedure DibujarLineaHaciaElOeste()
 { Mover(Oeste); Poner(Negro) }
procedure DibujarCuadradoNegro()
 {
   DibujarLineaHaciaElNorte()
   DibujarLineaHaciaElEste()
   DibujarLineaHaciaElSur()
   DibujarLineaHaciaElOeste()
 }
procedure Main()
 { DibujarCuadradoNegro() }
```

Observar que el programa quedó más complicado de esta forma que antes. Sin embargo, ahora estamos en condiciones de generalizar. Por ejemplo, si quisiéramos dibujar un cuadrado de tamaño más grande, sólo deberíamos cambiar los procedimientos que dibujan líneas de la siguiente manera

```
procedure DibujarLineaHaciaElNorte()
 {
   Mover(Norte); Poner(Negro)
   Mover(Norte); Poner(Negro)
 }
procedure DibujarLineaHaciaElEste()
 {
   Mover(Este); Poner(Negro)
   Mover(Este); Poner(Negro)
 }
procedure DibujarLineaHaciaElSur()
 {
   Mover(Sur); Poner(Negro)
   Mover(Sur); Poner(Negro)
 }
procedure DibujarLineaHaciaElOeste()
 {
   Mover(Oeste); Poner(Negro)
   Mover(Oeste); Poner(Negro)
 }
```

dejando los restantes sin alteraciones.

Ejercicio 6 *Escribir un programa Gobstones que dibuje un cuadrado de tres celdas de lado, utilizando la idea dada antes. ¿Cuál es la precondición del programa en cuestión?*

Esta forma de dividir un programa en tareas y definir un procedimiento para cada una de ellas hace que el foco de atención esté puesto en los procedimientos individuales. De esta forma, muchas veces se pedirá la definición de un procedimiento particular, y no de todo un programa. Por ejemplo, el ejercicio de dibujar un cuadrado se podría enunciar de la siguiente manera:

Ejercicio 7 (reformulación del ejercicio 6)
Escribir un procedimiento `DibujarCuadrado` *que dibuje un cuadrado de tres celdas de lado.*

Al enunciar un problema se asume que el requerimiento de escribir el programa `Main` es implícito, o sea, debe crearse un procedimiento `Main` que invoque al procedimiento solicitado en el enunciado, aunque tal cosa no esté pedida de manera explícita en él.

Además, muchas veces serán necesarios varios procedimientos diferentes combinados adecuadamente para llevar a cabo una tarea. En ese caso, se presentará y discutirá cada uno de ellos por separado, dejando su combinación implícita. Como ejemplo de esto, considerar el siguiente enunciado.

Ejercicio 8 *Escribir un procedimiento llamado* `CuadradoNegroARojo` *que suponiendo que hay un cuadrado Negro de lado tres a partir de la celda actual, lo transforme en un cuadrado Rojo. ¿Cuál es la precondición de este procedimiento?*

La solución para este problema sería dada de la siguiente manera

```
procedure TrasformarLineaHaciaElNorte()
 {
   Mover(Norte); Sacar(Negro); Poner(Rojo)
   Mover(Norte); Sacar(Negro); Poner(Rojo)
 }
procedure TrasformarLineaHaciaElEste()
 {
   Mover(Este); Sacar(Negro); Poner(Rojo)
   Mover(Este); Sacar(Negro); Poner(Rojo)
 }
procedure TrasformarLineaHaciaElSur()
 {
   Mover(Sur); Sacar(Negro); Poner(Rojo)
   Mover(Sur); Sacar(Negro); Poner(Rojo)
 }
procedure TrasformarLineaHaciaElOeste()
 {
   Mover(Oeste); Sacar(Negro); Poner(Rojo)
   Mover(Oeste); Sacar(Negro); Poner(Rojo)
 }
procedure CuadradoNegroARojo()
 {
   TrasformarLineaHaciaElNorte()
   TrasformarLineaHaciaElEste()
   TrasformarLineaHaciaElSur()
   TrasformarLineaHaciaElOeste()
 }
```

Observar que, puesto que la precondición de `CuadradoNegroARojo` es que exista un cuadrado negro de lado tres ya dibujado en el tablero, no tendría sentido utilizar como programa `Main`

```
procedure Main()
 { CuadradoNegroARojo() }
```

pues este programa siempre fallaría. Se requiere, en cambio, asegurar primero la precondición de `CuadradoNegroARojo`, utilizando algún otro procedimiento. Por ejemplo

```
procedure Main()
 {
   DibujarCuadradoNegro()
   CuadradoNegroARojo()
 }
```

Ejercicio 9 *Escribir un programa Gobstones que utilice* `CuadradoNegroARojo`*, y que, si el tablero es lo suficientemente grande, no falle.*

Observar que en el ejercicio 9 no se indica cómo debe conseguirse que al invocar el procedimiento `CuadradoNegroARojo` se cumpla su precondición. La precondición de un procedimiento solicitado debe satisfacerse de alguna manera para poder probarlo. O sea, el ejercicio 8 implicará que debe hacerse también el ejercicio 9, aunque esto no se indicará de ninguna manera a partir de ahora. Observar que también está implícita la necesidad de comprobar que el programa funciona, ejecutándolo.

3.3. Comentando el código

Puesto que un programa Gobstones es una descripción de la solución a un problema, cabe hacer la pregunta de quién leerá esta descripción. La respuesta básica es que el programa debe ser *"leído"* por el cabezal para realizar las acciones. Pero otro punto importante es que el programador debe leerlo también, para realizar cambios, mejoras, descubrir posibles errores, etc. Puesto que la intención del programador no siempre es evidente a partir de la lectura exclusiva de los comandos, Gobstones (al igual que prácticamente todos los lenguajes) ofrece una posibilidad interesante: la de *comentar* el código. Comentar el código significa agregarle descripciones, llamadas *comentarios* que no son relevantes para el cabezal, pero que sí lo son para un lector humano. Normalmente estos comentarios se escriben en castellano (o inglés, o algún otro lenguaje natural), o bien en algún lenguaje formal, como puede ser la lógica. Los comentarios no tienen ningún efecto sobre las acciones del cabezal.

Definición 20 *Un* comentario *es una descripción en castellano o algún lenguaje formal que se agrega al código para mejorar su comprensión por parte de un lector humano. Los comentarios no tienen efecto alguno sobre el accionar del cabezal.*

En Gobstones, los *comentarios* se dividen en dos grupos:

- los comentarios de línea, y
- los comentarios de párrafo.

Un comentario de línea comienza con el símbolo `--` o con el símbolo `//` y continúan hasta el final de la línea actual; todo lo que se escriba desde el símbolo hasta el final de línea es ignorado durante la ejecución. Un comentario de párrafo comienza con el símbolo {- y continúa hasta la primer aparición del símbolo -}, o comienza con el símbolo `/*` y continúa hasta la aparición del símbolo `*/`; todo el texto que aparece enmarcado entre estos pares de símbolos es ignorado durante la ejecución.

Los comentarios son útiles para una serie de propósitos diferentes. Por ejemplo:

- describir la intención de un procedimiento (o cualquier parte del código), es decir su comportamiento esperado;

- describir la intención de un parámetro, el conjunto de sus valores posibles, o lo que se intenta representar con él;
- eliminar temporalmente un comando para probar el programa sin él, pero sin borrarlo del mismo;
- brindar información diversa acerca del código, como ser el autor del mismo, la fecha de creación, etc.

Como ejemplo de uso de comentarios, se ofrece una versión comentada del procedimiento `DibujarLineaHaciaElNorte`.

```
{- Autor: Fidel
   Fecha: 08-08-08
-}
procedure DibujarLineaHaciaElNorte()
 {- PRECONDICIÓN:
     Debe haber al menos dos celdas al Norte de la actual.
     (O sea, el cabezal no puede encontrarse ni en la última
      ni en la anteúltima fila)
 -}
 {
   -- No se dibuja en la celda incial
   Mover(Norte); Poner(Negro) -- Dibuja en la celda
                              --        al Norte de la inicial
   Mover(Norte); Poner(Negro) -- Dibuja en la celda dos lugares
                              --        al Norte de la inicial

   -- Al terminar el cabezal se encuentra dos lugares
   -- al Norte de donde comenzó
 }
```

Observar que el código efectivo (aquel que no será ignorado durante la ejecución) es idéntico al ofrecido antes. Sin embargo, los comentarios ayudan a entender el propósito de cada parte.

Es importante destacar que, puesto que los comentarios son ignorados en la ejecución, debe verificarse al leer un programa que los comentarios sean válidos respecto de los comandos. Es una práctica mala, pero demasiado común, el dejar comentarios desactualizados al modificar código. Por ejemplo, supongamos que utilizamos este último procedimiento comentado como base para un procedimiento similar utilizado en la solución del ejercicio 6. Para ello, copiamos el programa en otro archivo, y realizamos algunas modificaciones, para obtener:

```
{- Autor: Fidel
   Fecha: 08-08-08
-}
procedure DibujarLineaHaciaElNorte()
 {- PRECONDICIÓN:
     Debe haber al menos dos celdas al Norte de la actual.
     (O sea, el cabezal no puede encontrarse ni en la última
      ni en la anteúltima fila)
 -}
 {
   -- No se dibuja en la celda incial
   Mover(Norte); Poner(Negro) -- Dibuja en la celda
```

```
                              --        al Norte de la inicial
    Mover(Norte); Poner(Negro) -- Dibuja en la celda dos lugares
                              --        al Norte de la inicial
    Mover(Norte); Poner(Negro) -- Dibuja en la celda dos lugares
                              --        al Norte de la inicial

    -- Al terminar el cabezal se encuentra dos lugares
    -- al Norte de donde comenzó
}
```

Observar que el código fue obtenido copiando y pegando, y luego alterando algunas partes. Eso lleva a que haya por lo menos tres comentarios desactualizados en este código. El primero es la precondición, puesto que al agregar comandos de movimiento, la precondición cambió, y sin embargo el comentario no fue alterado. El segundo es el comentario de línea que se asocia al tercer grupo de comandos: como fue obtenido por *cut&paste* y no fue modificado, indica lo mismo que el anterior, pero su efecto es diferente. El tercero es el comentario de cierre, sobre la posición final del cabezal. Si además este código hubiera sido modificado por otra persona diferente de `Fidel`, o en otra fecha, el comentario sobre el autor y la fecha también estarían desactualizados.

La práctica de comentar el código de manera adecuada es extremadamente importante cuando varias personas trabajan sobre un programa, ya sea simultáneamente, o a lo largo de algún período de tiempo. Es recomendable comentar todo código que se escriba de manera correcta. Por otra parte, los comentarios desactualizados son más dañinos que la ausencia de comentarios, puesto que pueden inducir a ideas incorrectas. Es por ello recomendable no desestimar la importancia de los comentarios y actualizarlos al modificar el código.

Como observación final, salvo que cada procedimiento sea definido por un programador diferente, la información de autor y fecha suele ponerse una única vez por archivo o conjunto de definiciones, y no como parte de cada procedimiento.

3.4. Parámetros

Al resolver el problema de dibujar un cuadrado, fue necesario escribir cuatro procedimientos, uno para cada una de las líneas. Sin embargo, lo único que variaba de uno a otro (además del nombre), era la dirección en la que el cabezal debía desplazarse. Resulta mucho más conveniente contar con alguna forma de representar esos cuatro procedimientos en uno solo, e indicarle de alguna manera qué dirección queremos que tenga en cuenta en cada uso. La forma de conseguir esto se conoce con el nombre de *parámetros*. Un parámetro es un nombre que denota un valor que puede ser diferente cada vez que se usa un procedimiento, y que debe ser suministrado en la invocación como un valor concreto. Este valor concreto usado en la invocación recibe el nombre de *argumento*.

La forma de definir un parámetro es utilizar un identificador entre paréntesis luego del nombre del procedimiento. Ese identificador puede luego ser utilizado dentro del bloque que define el procedimiento, como si de un valor fijo se tratase.

Definición 21 *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*

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

donde <params> *es una lista de identificadores separados por comas. Los identificadores usados como parámetros deben comenzar con minúscula.*

Definición 22 *Un* argumento *es el valor específico usado en la invocación de un procedimiento con parámetros. Al invocar un procedimiento, se utiliza la siguiente forma*

<procName>(<args>)

donde <args> es una lista de valores específicos (los argumentos) para los parámetros.

En el caso de los procedimientos para dibujar líneas y el cuadrado, es más conveniente escribir un solo procedimiento para dibujar las líneas que reciba la dirección como parámetro, y que sea el procedimiento de dibujar el cuadrado el que se encargue de utilizar *el mismo* procedimiento, pero con diferentes datos. Esto daría como resultado el siguiente código

```
procedure DibujarLineaHacia(dir)
 {- PROPÓSITO:
    * dibuja una línea de longitud 1 de color Negro
    PRECONDICIÓN:
    * el parámetro dir es una dirección
    * existe al menos una celda en la dirección dada
      por el parámetro dir
 -}
 {
   Mover(dir); Poner(Negro)
 }

procedure CuadradoNegro()
 {- PROPÓSITO:
    * dibuja un cuadrado Negro de lado 2
    PRECONDICIÓN:
    * el cabezal no se encuentra ni en la última fila
      ni en la última columna
 -}
 {
   DibujarLineaHacia(Norte)
   DibujarLineaHacia(Este)
   DibujarLineaHacia(Sur)
   DibujarLineaHacia(Oeste)
 }
```

que es mucho más conciso, ya que evita la repetición del código de dibujar líneas sólo para tener diferentes direcciones. Observar que `dir` es el parámetro en la definición de `DibujarLineaHacia` y se lo utiliza en el comando `Mover(dir)`. Al invocar el procedimiento `DibujarLineaHacia` se utiliza un valor específico de dirección com argumento en cada caso.

El mecanismo utilizado para esta definición, llamado *abstracción*, consiste en concentrarse en las similaridades entre diferentes elementos, reconociéndolos como casos particulares de una misma idea. El uso de parámetros permite identificar en qué partes son diferentes los elementos, y el proceso de invocación de un procedimiento con argumentos, o sea valores específicos para los parámetros, permite obtener los elementos particulares a partir de la definición general. Este caso particular de abstracción se conoce con el nombre de *parametrización*, justamente por el uso de parámetros para representar las variaciones de una misma idea.

¿Qué otras nociones son generalizables en el caso de dibujar un cuadrado? La más sencilla de todas es el color del cuadrado. El programa definido dibuja un cuadrado

negro. Si quisiéramos un cuadrado rojo, o uno azul, tendríamos que definir otro procedimiento que utilizase un dibujar línea que tuviera la instrucción `Poner(Rojo)` o `Poner(Azul)` en lugar de la actual `Poner(`Negro`)`. Sin embargo, lo único que es diferente es el color elegido, y por lo tanto, la similiaridad de la instrucción permite colocar un parámetro, y tener un procedimiento de dibujar un cuadrado de lado dos al que haya que decirle el color del cuadrado mediante un argumento en la invocación.

Suponiendo que aún no se generalizó el dibujo de líneas en un único procedimiento, el código para generalizar el cuadrado de diferentes colores quedaría como sigue

```
procedure DibujarLineaHaciaNorteDeColor(color)
 {- PROPÓSITO:
    * dibuja una línea del color indicado de longitud
      1 hacia el Norte
    PRECONDICIÓN:
    * el parámetro color es un color
    * existe al menos una celda en al Norte
 -}
 {
   Mover(Norte); Poner(color)
 }
procedure DibujarLineaHaciaEsteDeColor(color)
 {- PROPÓSITO:
    * dibuja una línea del color indicado de longitud
      1 hacia el Este
    PRECONDICIÓN:
    * el parámetro color es un color
    * existe al menos una celda en al Este
 -}
 {
   Mover(Este); Poner(color)
 }
procedure DibujarLineaHaciaSurDeColor(color)
 {- PROPÓSITO:
    * dibuja una línea del color indicado de longitud
      1 hacia el Sur
    PRECONDICIÓN:
    * el parámetro color es un color
    * existe al menos una celda en al Sur
 -}
 {
   Mover(Sur); Poner(color)
 }
procedure DibujarLineaHaciaOesteDeColor(color)
 {- PROPÓSITO:
    * dibuja una línea del color indicado de longitud
      1 hacia el Oeste
    PRECONDICIÓN:
    * el parámetro color es un color
    * existe al menos una celda en al Oeste
 -}
 {
   Mover(Oeste); Poner(color)
 }
```

```
procedure CuadradoDeColor(c)
 {- PROPÓSITO:
    * dibuja un cuadrado de color c de lado 2
    PRECONDICIÓN:
    * el cabezal no se encuentra ni en la última fila
      ni en la última columna
 -}
 {
   DibujarLineaHaciaNorteDeColor(c)
   DibujarLineaHaciaEsteDeColor(c)
   DibujarLineaHaciaSurDeColor(c)
   DibujarLineaHaciaOesteDeColor(c)
 }
```

Es claro que es mucho mejor si se parametrizan *ambas* nociones, el color y la dirección.

Ejercicio 10 *Definir un procedimiento* `DibujarCuadrado` *que, dado un color, dibuje un cuadrado de lado dos de dicho color, utilizando un único procedimiento para dibujar líneas.*

Otra noción generalizable es el tamaño del cuadrado. Sin embargo, para poder generalizarla son necesarios comandos más complejos, por lo cual esta idea se tratará luego de presentar dichos comandos.

Un detalle importante a observar en el uso de parámetros es la correspondencia entre el número de parámetros declarados en un procedimiento, y el número de expresiones usadas para invocarlo. Por ejemplo, el procedimiento `DibujarLineaHacia` tiene un único parámetro `dir` y cuando se invoca dicho procedimiento, se debe suministrar un único valor, como en `DibujarLineaHacia(Norte)`. La misma correspondencia debe existir si se usan varios parámetros en la definición del procedimiento: al invocarlo deben usarse la misma cantidad de argumentos. Por ejemplo, si tomamos la solución al ejercicio 10

```
procedure DibujarLineaDeHacia(color,dir)
 {- PROPÓSITO:
    * dibuja una línea de longitud 1 del color indicado
      y en la dirección indicada
    PRECONDICIÓN:
    * el parámetro dir es una dirección
    * el parámetro color es un color
    * existe al menos una celda en la dirección dada por dir
 -}
 {
   Mover(dir); Poner(color)
 }
procedure CuadradoDeColor(color)
 {- PROPÓSITO:
    * Dibuja un cuadrado de lado 2 del color indicado
    PRECONDICIÓN:
    * el parámetro color es un color
    * el cabezal no se encuentra ni en la última fila
      ni en la última columna
 -}
 {
   DibujarLineaDeHacia(color,Norte)
```

```
    DibujarLineaDeHacia(color,Este)
    DibujarLineaDeHacia(color,Sur)
    DibujarLineaDeHacia(color,Oeste)
  }
```

vemos que `DibujarLineaDeHacia(color,Norte)` es una invocación válida del procedimiento `DibujarLineaDeHacia`, pues corresponden el número de los argumentos con los parámetros declarados. En cambio, `DibujarLineaDeHacia(Norte)` es una invocación inválida, ya que no concuerda la cantidad de argumentos entre el uso y la declaración.

3.5. Alternativas condicionales

Hasta el momento, además de la noción de comando, se ha utilizado además otra noción de manera implícita. Dicha noción es la de *valor*. La idea de valor se discute con más detalles en la sección 4.1. En Gobstones, ejemplos de valores son los colores y las direcciones. Observar que lo único importante de un color es ser uno particular, y distinto de los demás. Lo mismo sucede con las direcciones. Ninguno de ellos expresa más que su propia existencia (a diferencia, por ejemplo, del comando `Poner(Rojo)`, que expresa una acción posible del cabezal).

Dos valores que usamos implícitamente desde chicos son los que expresan la idea de verdad. Cualquiera distingue algo *verdadero* de algo *falso*. Entonces, así como podemos distinguir una bolita de color rojo e indicarlo mediante el valor `Rojo`, o una de color azul e indicarlo mediante el color `Azul`, podremos expresar la capacidad de distinguir entre verdadero y falso a través de dos valores: `True` y `False`. El valor `True` describe la idea de verdadero, mientras que el valor `False` describe la idea de falso. A estos dos valores se los conoce como *valores de verdad*, *valores booleanos* o simplemente *booleanos*[1].

Así como el cabezal posee un dispositivo para poner y sacar bolitas, y para moverse, posee también sensores que le permiten saber si hay o no bolitas de cierto color en la celda actual. Para describir el hecho de que haya o no bolitas en la celda actual, se usará la expresión `hayBolitas`, que debe recibir un color como argumento. Entonces `hayBolitas(Rojo)` es una expresión que será verdadera (o sea, describirá al valor `True`) cuando haya al menos una bolita roja en la celda actual, y falsa (o sea, describirá al valor `False`) cuando no haya ninguna. La forma general de `hayBolitas` está dada por la siguiente definición:

Definición 23 *La expresión* `hayBolitas(`*<color>*`)` *describe el hecho de que exista una bolita de color <color> en la celda actual.*

Esta expresión valdrá `True` si en la celda actual hay alguna bolita de color *<color>*, o `False` si no hay ninguna.

Los booleanos pueden utilizarse para distinguir entre diferentes alternativas al momento de describir comandos para el cabezal. Para ello es necesario un comando que, basado en un booleano, decida entre otros dos comandos. Por ejemplo, supongamos que queremos poner dos bolitas verdes en caso que en la celda actual haya una roja, o una negra si no la hay. Entonces deberíamos utilizar un nuevo comando, llamado `if-then-else`:

```
if (hayBolitas(Rojo))
  { Poner(Verde); Poner(Verde) }
else
  { Poner(Negro) }
```

[1] George Boole fue un matemático y filósofo inglés del siglo XIX que fundó las bases de la aritmética de computadoras, y que es considerado uno de los padres de la computación moderna. La estructura estudiada por este matemático, y que se compone fundamentalmente de los valores de verdad se conoce como *Álgebra de Boole*.

En inglés, `if-then-else` significa *si-entonces-sino*, y es la idea de estructura alternativa en base a una condición. Por ejemplo, en este caso, el comando se leería "*si hay una bolita roja en la celda actual, entonces poner dos bolitas verdes, y si no, poner una negra*". La forma general del comando `if-then-else` está dada por la siguiente definición:

Definición 24 *El comando de* alternativa condicional *(o simplemente* condicional*)* `if-then-else` *tiene la siguiente forma:*

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

siendo <*bool*> *una expresión que describe un valor de verdad (booleano), y donde los bloques* <*unBloque*> *y* <*otroBloque*> *son bloques de código cualesquiera (que normalmente son diferentes, aunque podrían ser iguales). Al bloque* <*unBloque*> *se lo denomina* rama verdadera *(o rama del* `then`*) y al bloque* <*otroBloque*> *se lo denomina* rama falsa *(o rama del* `else`*).*

El comando `if-then-else` describe la decisión de realizar la acción descrita por el bloque <*unBloque*>, en caso que la condición sea `True`, o de realizar la acción descrita por <*otroBloque*> en caso que la condición sea `False`. Observar que los paréntesis son necesarios siempre alrededor de la condición.

Ejercicio 11 *Escribir dos programas Gobstones que utilicen el condicional dado en el ejemplo. Combinarlo con el número mínimo de comandos de manera tal de obtener, en un caso un tablero con al menos dos bolitas verdes, y en el otro un tablero con al menos una bolita negra.*

Una variante útil del comando `if-then-else` es cuando no existe una acción alternativa para la rama del `else`. En ese caso, la alternativa se transforma realmente en un comando condicional (o sea, un comando que sólo se ejecuta si se cumple cierta condición). Esto es extremadamente útil para evitar que una operación parcial provoque la autodestrucción del cabezal. Para ello, basta con utilizar como condición del `if-then-else` una expresión <*condicion*> que valga `True` cuando la precondición de la operación dada en la rama del `then` sea válida. Por ejemplo, si se quiere sacar una bolita, puede preguntarse primero si la misma existe, de la siguiente manera:

```
if (hayBolitas(Rojo)) { Sacar(Rojo) }
```

Notar que la precondición de `Sacar(Rojo)` es que haya bolitas rojas, y la condición del `if-then-else` es una expresión que vale `True` siempre que haya bolitas rojas; o sea, la condición del `if-then-else` garantiza la precondición de la rama del `then`. De esta manera, la operación dada por la alternativa condicional es total, pues el comando `Sacar` sólo se invocará cuando haya una bolita roja en la celda actual; en caso de que no haya ninguna bolita roja, este comando no tendrá ningún efecto.

Ejercicio 12 *Escribir un programa que, suponiendo que en la celda actual hay hasta tres bolitas rojas (pero no se sabe cuántas exactamente), las elimine de la celda. El programa debe funcionar siempre, o sea debe describir una operación total.*

Los comandos condicionales pueden *anidarse* (o sea, usar un condicional en las ramas de otro), para tener diversas alternativas. Por ejemplo, suponer que se pide que si la celda actual y la vecina `Este` tienen bolitas verdes, se agregue una bolita roja en la celda actual, y si la celda actual y la vecina `Norte` no tienen bolitas verdes, se agregue una bolita azul en la celda actual, en otros casos, no agregar nada. El código para resolver este problema podría ser

```
if (hayBolitas(Verde))
  {
   Mover(Este)
   if (hayBolitas(Verde))
     { Mover(Oeste); Poner(Rojo) }
  }
else
  {
    Mover(Norte)
    if (not hayBolitas(Verde))
      { Mover(Sur); Poner(Azul) }
  }
```

Observar que la precondición de este programa es que la celda actual no puede encontrarse ni en la fila más al `Este`, ni en la fila más al `Norte`.

Es importante observar que si bien el ejemplo ilustra la posibilidad de anidar comandos condicionales, quedaría de manera más claro utilizando procedimientos, de la siguiente manera.

```
procedure ProcesarEste()
 {- PROPÓSITO:
    * verifica si hay bolitas de color Verde en la celda
      del Este y si es así, coloca una bolita de color Rojo
      en la celda actual
    PRECONDICIÓN:
    * hay una celda al Este
 -}
 {
  Mover(Este)
  if (hayBolitas(Verde))
    { Mover(Oeste); Poner(Rojo) }
 }
procedure ProcesarNorte()
 {- PROPÓSITO:
    * verifica si NO hay bolitas de color Verde en la celda
      del Norte y si es así, coloca una bolita de color Azul
      en la celda actual
    PRECONDICIÓN:
    * hay una celda al Norte
 -}
 {
  Mover(Norte)
  if (not hayBolitas(Verde))
    { Mover(Sur); Poner(Azul) }
 }
```

y luego construir el condicional como

```
if (hayBolitas(Verde))
  { ProcesarEste() }
else
  { ProcesarNorte() }
```

Ejercicio 13 *Escribir un programa que ponga una bolita roja en la primer celda al* `Este` *de la actual, si en la celda actual hay bolitas azules; si no las hay, pero hay verdes,*

la bolita roja debe ubicarse en la segunda celda al `Este`*; y si no hay ni verdes ni azules, pero hay negras, entonces la bolita roja debe ubicarse en la tercera celda al* `Este`*. Si hay sólo bolitas rojas, o no hay ninguna bolita, debe ubicarse una bolita roja en la celda actual. ¿Cuál es la precondición de su programa? Intentar determinar la precondición más precisa posible.*

Los condicionales anidados no son imprescindibles. En muchas ocasiones pueden ser reemplazados por condiciones más complejas, a través del uso de *conectivos lógicos*. Los conectivos lógicos son operaciones entre booleanos. Los más comunes, y que pueden ser representados directamente en Gobstones, son la negación, la conjunción y la disyunción.

Definición 25 *Un* conectivo lógico *es una operación entre booleanos. Los conectivos más comunes son la negación, la conjunción y la disyunción.*

La negación es un conectivo que toma un booleano y devuelve otro booleano. Si el booleano representa el valor de verdad de una proposición, como negar una proposición verdadera la hace falsa y viceversa, entonces la negación transforma `True` en `False`, y recíprocamente. La manera de escribir la negación de una expresión con un valor booleano es usar la operación prefija `not`. De esta manera, `not True` es igual a `False`, y `not False` es igual a `True`.

Definición 26 *La* negación *es un conectivo lógico unario. Su forma general es*

`not` $<$*expBooleana*$>$

donde $<$*expBooleana*$>$ *es cualquier expresión cuyo resultado sea* `True` *o* `False`*.*

En el ejemplo de condicionales anidados se utilizó una negación para decidir realizar una acción si *no* había bolitas de color verde en determinada celda. Por ejemplo, para agregar una bolita roja en la celda actual si no hay bolitas verdes en la misma, se puede utilizar el siguiente comando

```
if (not hayBolitas(Verde))
  { Poner(Rojo) }
```

Como `hayBolitas(Verde)` retorna `False` cuando no hay bolitas en la celda actual, la expresión `not hayBolitas(Verde)` retorna `True` en ese caso, y el comando `Poner(Rojo)` será ejectuado.

Un conectivo lógico importante es la conjunción, que toma dos valores de verdad y retorna un tercero. La conjunción de dos proposiciones, expresada mediante la operación infija `&&`, será verdadera cuando ambas lo sean, y falsa en otro caso. Es la manera de representar el '*y*' del lenguaje natural.

Definición 27 *La* conjunción *es un conectivo lógico binario. Su forma general es*

$<$*expBooleana*$_1>$ `&&` $<$*expBooleana*$_2>$

donde $<$*expBooleana*$_1>$ *y* $<$*expBooleana*$_2>$ *son expresiones cuyo resultado es* `True` *o* `False`*.*

La expresión `True && True` tendrá como resultado `True`, y las expresiones `True && False`, `False && True` y `False && False` tendrán como resultado `False`. Por ejemplo si se pide definir un comando que coloque una bolita roja si ya hay bolitas verdes y azules en la celda actual, se puede utilizar una conjunción de la siguiente manera:

```
if (hayBolitas(Verde) && hayBolitas(Azul))
  { Poner(Rojo) }
```

que establece exactamente la condición pedida (se lee *si hay bolitas verdes* ***y*** *hay bolitas azules, entonces poner una bolita roja*). Observar que en este caso hubiera sido equivalente utilizar el siguiente condicional anidado:

```
if (hayBolitas(Verde))
  { if (hayBolitas(Azul))
      { Poner(Rojo) }
  }
```

Sin embargo suele ser más claro utilizar conectivos lógicos en lugar de condicionales anidados en la mayoría de los casos. Es fundamentalmente una cuestión de gusto y claridad el decidir si utilizar un conectivo lógico o un condicional anidado.

El tercer conectivo lógico que se puede escribir en Gobstones de manera primitiva es la disyunción. La disyunción toma dos valores de verdad, y retorna otro. La disyunción es verdadera cuando al menos uno de los argumentos es verdadero, siendo falsa sólo si ambos son falsos. Para denotarla se utiliza la operación infija , `||` de la siguiente manera

Definición 28 *La* disyunción *es un conectivo lógico binario. Su forma general es*

$<expBooleana_1>$ `||` $<expBooleana_2>$

donde $<expBooleana_1>$ *y* $<expBooleana_2>$ *son expresiones cuyo resultado es* `True` *o* `False`*.*

Para ejemplificar el uso de la disyunción, se pide agregar una bolita de cada roja solamente cuando haya alguna bolita en la celda actual (de cualquier color). Esto se puede conseguir mediante el siguiente comando:

```
if (hayBolitas(Verde) || hayBolitas(Rojo)
 || hayBolitas(Negro) || hayBolitas(Azul))
  { Poner(Rojo) }
```

Observar que la condición será verdadera cuando al menos una de las cuatro proposiciones sea verdadera, y entonces se expresa la condición pedida (que haya alguna bolita, de cualquier color).

Ejercicio 14 *Escribir un programa que agregue bolitas de los cuatro colores en la celda actual sólo si no hay bolitas de ningún color. ¿Puede hacerlo con un único condicional? ¿Puede expresar la condición de más de una forma?*

Al combinar diferentes conectivos lógicos, la negación se resuelve primero, luego la conjunción, y finalmente la disyunción. Esto se denomina *precedencia* de los operadores lógicos.

Definición 29 *La* precedencia *de un operador es la prioridad que tiene el mismo en una expresión combinada para ser resuelto.*

La negación tiene más precedencia que la conjunción, y la conjunción más precedencia que la disyunción. En caso de precisar alterar este orden, deben utilizarse paréntesis. Entonces, la expresión

```
not hayBolitas(Verde) && not hayBolitas(Azul) || hayBolitas(Rojo)
```

significa lo mismo que

```
((not hayBolitas(Verde)) && (not hayBolitas(Azul))) || hayBolitas(Rojo)
```

Para obtener otros resultados, deben utilizarse paréntesis de manera obligatoria. Como ser

```
not (hayBolitas(Verde) && not (hayBolitas(Azul) || hayBolitas(Rojo)))
```

¿Cuál será el valor de verdad de las condiciones dadas arriba? Para determinarlo, escriba una tabla que consigne las diferentes posibilidades de una celda para contener bolitas. ¿Qué diferencia se observa entre ambas condiciones (con diferentes precedencias)?

3.6. Repetición indexada

Un concepto fundamental para poder resolver problemas complejos es el de la repetición. Existen varias formas de repetición. La primera que se presentará es la *repetición indexada*. En esta forma de repetición, un comando se repite un cierto número de veces, siendo ese número controlado por un índice. El comando que se utiliza para definir una repetición indexada es `repeatWith`. Por ejemplo, para poner 10 bolitas rojas en la celda actual, se puede escribir el siguiente comando:

```
repeatWith i in 1..10
  { Poner(Rojo) }
```

En este caso el rango es `1 2 3 4 5 6 7 8 9 10`, y el comando `Poner(Rojo)` se repetirá una vez por cada elemento del rango. El identificador `i` nombrará cada uno de los valores del rango en cada repetición; o sea, en la primera repetición `i` nombrará al `1`, en la segunda repetición `i` nombrará al `2`, etc.

Definición 30 *La* repetición indexada *permite repetir una acción un cierto número de veces, siendo esta cantidad indicada por un índice que varía dentro de un cierto rango de valores. El comando* `repeatWith` *que permite la repetición indexada tiene la siguiente forma*

`repeatWith` <*indexName*> `in` <*inicioRango*>`..`<*finRango*>
<*bloque*>

siendo <*indexName*> *un identificador con minúscula,* <*inicioRango*> *y* <*finRango*> *valores del mismo tipo, y* <*bloque*> *un bloque cualquiera.*

El efecto de un comando de repetición indexada es que la acción descrita por el bloque <*bloque*> se repetirá tantas veces como valores haya en el rango determinado por <*inicioRango*> y <*finRango*>. Por ejemplo, la acción de poner 10 bolitas rojas podría describirse también con el comando

```
repeatWith i in 3..12
  { Poner(Rojo) }
```

siendo <*inicioRango*> el valor `3` y <*finRango*> el valor `12`. En este caso el rango será `3 4 5 6 7 8 9 10 11 12`, y por lo tanto `i` nombrará a los valores de `3` a `12` en cada repetición.

El rango siempre debe ser creciente, o sea, el valor incial debe ser menor o igual que el valor final. Si no sucede esto, el rango es vacío, y el comando de repetición no realiza ninguna acción. Por ejemplo, el siguiente comando no tiene ningún efecto:

```
repeatWith i in 10..1
  { Poner(Rojo) }
```

puesto que el rango `10..1` es vacío.

El verdadero poder de la repetición indexada viene de la combinación de la misma con parámetros. Por ejemplo, se puede definir un procedimiento que coloque `n` bolitas de color Rojo en la celda actual, a través del siguiente procedimiento:

```
procedure PonerNRojo(n)
  {- PROPÓSITO:
     * coloca n bolitas rojas en la celda actual,
       si n>=0, y ninguna si n<0
     PRECONDICIÓN:
     * ninguna (es una operación total)
  -}
  {
    repeatWith i in 1..n
      { Poner(Rojo) }
  }
```

Entonces, el comando `PonerNRojo(3)` pondrá 3 bolitas en la celda actual, y el comando `PonerNRojo(10)` pondrá 10 bolitas.

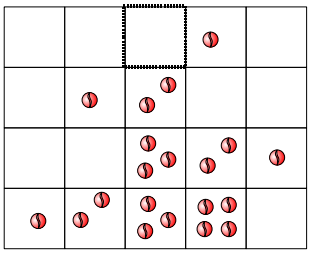
Ejercicio 15 *Escribir un procedimiento* `PonerN` *que deposite* `n` *bolitas de color* `color` *en la celda actual, suponiendo que* `n` *es un número positivo. Utilizarlo para colocar 17 bolitas azules y 42 bolitas negras en la celda actual. ¿Cuál es la precondición del procedimiento* `PonerN`*? ¿Y del programa?*

El índice de una repetición indexada es un identificador usado para denotar cada elemento del rango, y por lo tanto, puede utilizarse como un valor en el bloque que se repite. Por ejemplo, si se desea colocar 1 bolita roja en la celda actual, 2 en la celda lindante al Norte, 3 en la siguiente, y así hasta 5 bolitas rojas en la celda 4 lugares al Norte de la actual, se puede escribir el siguiente comando

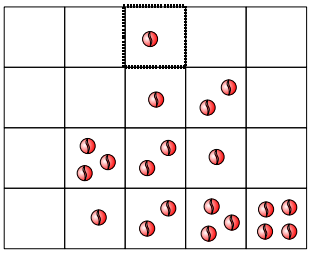
```
repeatWith cant in 1..5
  { PonerNRojo(cant); Mover(Norte) }
```

Observar que el índice, `cant` en este caso, denota un valor diferente en cada repetición del cuerpo. O sea, si el rango es `1 2 3 4 5`, el valor de `cant` será `1` en la primera iteración, `2` en la segunda, `3` en la tercera, `4` en la cuarta, y `5` en la última.

Ejercicio 16 *Escribir un procedimiento* `Progresion` *que dados un número* `n` *y una dirección* `dir`*, coloque 1 bolita roja en la celda actual, 2 bolitas rojas en la celda lindante en la dirección* `dir`*, 3 en la siguiente, etc. repitiendo esto* `n` *veces. Utilizarlo para obtener la siguiente configuración de bolitas en el tablero final:*



¿Qué tendría que cambiar en el procedimiento `Progresion` *para obtener, en cambio, la siguiente configuración?*



Los rangos de una repetición indexada no se limitan a números. También pueden ser colores, direcciones o booleanos. El orden de las direcciones es en el sentido de las agujas del reloj, comenzando por el Norte, el orden de los colores es alfabético, y el de los booleanos es primero falso y luego verdadero. El orden en cada caso es importante para saber qué rango corresponde a cada par de valores.

Por ejemplo, el procedimiento para dibujar un cuadrado de 2 celdas de lado puede escribirse de la siguiente manera:

```
procedure CuadradoNegro()
  {- PROPÓSITO:
     * dibuja un cuadrado Negro de lado 2
     PRECONDICIÓN:
     * el cabezal no se encuentra ni en la última fila
       ni en la última columna
  -}
  {
    repeatWith dir in Norte..Oeste
      { DibujarLineaHacia(dir) }
  }
```

Observar que la repetición utiliza las direcciones como índice, y que, por el orden de las mismas, utiliza las 4 direcciones.

Con la repetición indexada puede realizarse un procedimiento para dibujar cuadrados de cualquier longitud.

Ejercicio 17 *Escribir un procedimiento* `DibujarCuadrado` *que dibuje cuadrados de color* `color` *y de* `n` *celdas de lado. Confirmar que el lado del cuadrado tenga* exactamente `n` *celdas, y no más o menos.*

4. Mecanismos más complejos

4.1. Expresiones, valores y tipos

Como se vio al inicio de la sección 3.5, además de los comandos, que son descripciones de órdenes al cabezal, se han usado ciertos *valores*: los colores (`Azul`, `Negro`, `Verde` y `Rojo`), las direcciones (`Norte`, `Este`, `Sur`, `Oeste`), los booleanos (`False` y `True`) y los números (determinados por secuencias de dígitos). Los *valores* son elementos diferenciados por sus propiedades, y pueden ser usados como argumentos de un procedimiento o como elementos en el rango de una repetición indexada, por ejemplo.

Definición 31 *Un* valor *es un elemento que se diferencia de otro por sus propiedades.*

La forma de las expresiones puede ser, o bien directamente un valor (como ser los colores o las direcciones), utilizar identificadores que denoten valores (como ser, los

parámetros o los índices – por ejemplo `hayBolitas(color)`, utilizar *operaciones primitivas* (como ser `hayBolitas`), o utilizar *expresiones* compuestas (como ser las conformadas con conectivos lógicos o con operaciones aritméticas – por ejemplo, `(hayBolitas(Rojo) && hayBolitas(Verde))`, o `(2+(4*3))`).

Definición 32 *Una* expresión *es la descripción de un valor, o sea, denota un valor. Está formada a partir de valores, identificadores que denoten valores, y operaciones que combinen otras expresiones.*

La denotación es una manera de referise de manera específica a algo. Entonces, al decir que un identificador *denota* un valor, se está diciendo que el mismo se refiere a ese valor, lo distingue de otros.

Definición 33 Denotar *algo es referirse de manera específica a eso, distinguirlo de otros de manera única. En este sentido se dice que las expresiones denotan valores.*

Una tercera noción fundamental asociada a la noción de valores y expresiones es la noción de tipo. En su forma más simple, un *tipo* puede entenderse como la descripción de un conjunto de valores específicos, con propiedades comunes. En Gobstones existen cuatro tipos elementales: los colores, las direcciones, los booleanos y los números.

La noción de tipo es utilizada, entre otras cosas, para identificar usos erróneos de un valor, o sea, cuáles combinaciones de expresiones están permitidas al armar una expresión compuesta, o qué comandos pueden armarse con seguridad para describir acciones. Por ejemplo, la operación `Poner` espera que su argumento sea un color. Si `Poner` recibe un valor de otro tipo, provocará la autodestrucción del cabezal. La precondición de `Poner` es que su argumento sea un valor de tipo color.

Definición 34 *Un* tipo *es la descripción de un conjunto de valores con ciertas propiedades comunes. Se utilizan, entre otras cosas, para distinguir entre usos permitidos y erróneos de una expresión.*

Ejercicio 18 *Escribir un programa que incluya el comando* `Poner(17)` *y comprobar el error obtenido.*

Al presentar la operación `Poner` se había establecido que era una operación total, y ahora resulta que la misma es en realidad parcial. ¿Cómo debe entenderse este hecho? Lo que sucede es que los errores producidos por utilizar valores de tipo distinto al esperado (llamados *errores de tipo*) pueden ser controlados antes de la ejecución del programa. De este control se suele decir que "impone el uso de tipos", pues en el caso que un programa contenga combinaciones erróneas de tipos en las expresiones, es rechazado sin ejecutarlo.

La versión actual de Gobstones no impone el uso de tipos. Esto quiere decir que, por ejemplo, puede usarse un argumento de tipo numérico para invocar un comando que espera un tipo color. Sin embargo dicho programa fallará al ser ejecutado. Otros lenguajes más avanzados imponen restricciones en la forma de utilizar tipos. Si bien la versión actual de Gobstones no controla los errores de tipo de esta manera, se seguirá la convención de no escribir en la precondición los requerimientos de tipo, sino dejarlos en la definición de la operación. Por ejemplo, se dirá que `Poner` *espera un argumento de tipo color* para indicar que fallará con la autodestrucción del cabezal si recibe un valor de otro tipo. Consecuentemente, sólo se indicará la precondición en los casos en los que el argumento sea un color, y por eso `Poner` aparece como una operación total a pesar de no serlo realmente.

La idea de tipos también se aplica a las expresiones. Por ejemplo, la suma sólo es posible realizarla entre números. ¿Qué sucede si se intenta armar una suma utilizando

un color? Consideremos, por ejemplo, la expresión `Verde+1`. La misma no describe a ningún valor. Si se intenta determinar qué valor representa esta expresión, también se obtendrá la autodestrucción del cabezal.

Ejercicio 19 *Escribir un programa que incluya la expresión* `Verde+1` *y comprobar el error obtenido.*

La suma es también una forma de operación parcial, cuya precondición es que sus dos sumandos sean descripciones de números, y sólo en ese caso describe a un número. Esto se describe diciendo que la suma *espera dos argumentos de tipo numérico para producir un número*. Al igual que con el comando `Poner`, el tipo de los argumentos no se incluirá entre las precondiciones de la suma. De esta manera, diremos que la suma es una *operación total* ***sobre*** *números*.

La idea de tipos se aplica también a los parámetros de un procedimiento. Por ejemplo, en el procedimiento `DibujarLineaHacia`, el parámetro `dir` es utilizado como argumento del comando `Mover`, y por lo tanto se espera que sea una dirección. Si se invocase al procedimiento con un valor diferente de una dirección, dicho programa sería erróneo. Para indicar esto, se dirá que `dir` es de *tipo* dirección, y que `Mover` sólo espera valores de tipo dirección. De esta manera, la invocación `DibujarLineaHacia(Rojo)` es fácilmente identificable como un error, pues `Rojo` no es de tipo dirección, sino de tipo color. En el caso de más de un parámetro en el mismo procedimiento, cada uno tendrá un tipo específico, y al invocar dicho procedimiento, deberán suministrarse argumentos de los tipos correctos, en el orden establecido. Por ejemplo, en el procedimiento `DibujarLineaDeHacia`, el primer parámetro, `color`, es de tipo color, y el segundo parámetro, `dir`, es de tipo dirección. Entonces, `DibujarLineaDeHacia(Verde, Norte)` es una invocación correcta del procedimiento, mientras que las inovacaciones `DibujarLineaDeHacia(Norte, Verde)` y `DibujarLineaHacia(Verde)` no lo son. La correspondencia entre el tipo de un parámetro y el de un argumento es básica, y sólo se la consignará en la precondición del procedimiento en cuestión estableciendo el tipo que se espera que tenga cada parámetro.

4.2. Operaciones predefinidas

Cada tipo viene acompañado por una serie de operaciones predefinidas sobre sus elementos. Es importante conocer cuáles son las operaciones predefinidas de cada tipo.

El tipo de los booleanos trae las operaciones asociadas con los conectivos lógicos, y las operaciones de mínimo y máximo booleano:

- la negación, `not`
- la conjunción, `&&`
- la disyunción, `||`
- el booleano mínimo en el orden, `minBool`
- el booleano máximo en el orden, `maxBool`

La negación se usa de manera prefija (o sea, la operación se escribe antes del operando), y las conjunción y la disyunción, infijas (o sea, la operación se escribe entre medio de los operandos). Es decir, la forma de la operación `not` es `not` $<condicion>$, y la forma de la operación `&&` es $<cond_1>$ `&&` $<cond_2>$. El valor de `minBool()` es `False`, y el de `maxBool()` es `True`.

El tipo de los números trae las operaciones aritméticas usuales:

- la suma, `+`

- la resta, `-`
- la multiplicación, `*`
- la división entera, `div`
- el resto de la división entera, `mod`
- la exponenciación, `^`

Todas estas operaciones se usan infijas. En el caso de la división entera y el resto, el comportamiento esperado es que si `n div m` = `q` y `n mod m` = `r`, entonces `m` = `q*n + r`, y `r` es positivo y menor estricto que `n`.

Las únicas operaciones especiales de los colores son el cálculo de los valores mínimo y máximo.

- el color mínimo en el orden, `minColor`
- el color máximo en el orden, `maxColor`

El valor de `minColor()` es `Azul`, y el de `maxColor()` es `Verde`.

El tipo de las direcciones tiene las operaciones de mínima y máxima dirección:

- la dirección mínima en el orden, `minDir`
- la dirección máxima en el orden, `maxDir`

El valor de `minDir()` es `Norte`, y el de `maxDir()` es `Oeste`.

Ejercicio 20 *Escriba un procedimiento* `LateralesVerdes`*, que tome una dirección* `dir` *como parámetro, y coloque una bolita verde en la celda lindante a la celda inicial en la dirección dada y otra bolita verde en la celda lindante a la inicial en la dirección opuesta. Al terminar, el cabezal debe quedar en la misma celda en la que comenzó. ¿Qué precondición tiene el procedimiento?*

Hay operaciones que trabajan sobre todos los tipos básicos. Las más comunes de éstas son las operaciones relacionales, que permiten realizar comparaciones entre dos elementos del mismo tipo, y cuyo resultado es un booleano (o sea, son operaciones que provocan la autodestrucción del cabezal si reciben argumentos de distinto tipo). Además hay operaciones que permiten moverse dentro del orden establecido de cada tipo. Estas operaciones son:

- las comparaciones por igualdad
  - iguales, `==`
  - diferentes, `/=`
- las comparaciones de orden
  - menor, `<`
  - menor o igual, `<=`
  - mayor, `>`
  - mayor o igual, `>=`
- la operaciones de movimiento en el orden
  - cálculo del siguiente, `siguiente`
  - cálculo del previo, `previo`

- operaciones especiales sólo para algunos tipos
    - cálculo del opuesto, `opuesto` ó `-` (unario)

Todas, salvo `siguiente`, `previo` y `opuesto`, se usan de manera infija. Las operaciones de orden siguen el orden de cada tipo, establecido al presentar la repetición indexada (sección 3.6). La operación `siguiente(`<*expresion*>`)` devuelve el elemento siguiente en el orden de los elementos del tipo del valor de <*expresion*>, volviendo al mínimo en caso de que el elemento sea el máximo. Por ejemplo, `siguiente(2)` es `3`, `siguiente(Norte)` es `Este`, y `siguiente(Oeste)` es `Norte`. La operación `previo(`<*expresion*>`)` es la operación inversa, devolviendo el valor anterior en el orden, y volviendo al máximo en caso que se trate del mínimo. Entonces, `previo(3)` es `2`, `previo(Oeste)` es `Sur`, y `previo(Norte)` es `Oeste`. La operación `opuesto` funciona sobre direcciones o números. Se utiliza como `opuesto(`<*expresion*>`)` o `-`<*expresion*>. En el caso de las direcciones transforma `Norte` en `Sur`, `Este` en `Oeste`, y viceversa. Por ejemplo, el valor de `opuesto(Sur)` es `Norte` y el valor de `-Este` es `Oeste`. En el caso de los números, si la expresión vale n, calcula el valor de $-n$.

Hay expresiones que tienen que ver con la operatoria del cabezal, y sirven para determinar ciertos valores asociados a la celda actual. Las mismas son:

- `hayBolitas(`<*color*>`)` que dado un color, representa a un booleano que indica si en la celda actual hay o no bolitas de ese color;
- `puedeMover(`<*direccion*>`)` que dada una dirección, representa a un booleano que indica si el cabezal se puede mover en la dirección indicada sin provocar su autodestrucción;
- `nroBolitas(`<*color*>`)` que dado un color, representa al número de bolitas de ese color que hay en la celda actual.

Esto completa el repertorio de expresiones de Gobstones, con las cuales se pueden realizar una serie importante de programas.

Ejercicio 21 *Utilizar el procedimiento* `PonerN` *del ejercicio 15 para igualar el número de bolitas verdes de la celda actual con el número de bolitas rojas, suponiendo que hay más bolitas rojas que verdes. ¿Cuál es la precondición del programa?*

Ejercicio 22 *Escribir un procedimiento* `MoverN` *que tome un número* `n` *y una dirección* `dir` *como parámetros, y mueva el cabezal* `n` *celdas en la dirección indicada, o provoque la autodestrucción del cabezal si no hay suficientes celdas. Asumir como precondición que el número es positivo. Utilizarlo para mover el cabezal al norte tantas veces como bolitas verdes y rojas hay en la celda actual. ¿Cuál es la precondición del programa?*

Ejercicio 23 *Escribir un procedimiento* `Arcoiris`*, que dado un número* `n` *y una dirección* `dir` *como parámetros, dibuje un arcoiris en la dirección* `dir`*. Un arcoiris es una serie de celdas que alternan los colores de las bolitas que poseen (por ejemplo, una bolita azul en la celda inicial, una bolita negra en la lindante en dirección* `dir`*, una bolita roja en la celda siguiente en esa dirección, una verde en la siguiente, etc.)*

Ejercicio 24 *Escribir un procedimiento* `PonerMultiple` *que reciba un color* `c` *y un número cualquiera* `n` *como parámetros, y ponga* `n` *bolitas de color* `c`*, si* `n` *es positivo, o saque* `-n` *bolitas de color* `c`*, si* `n` *es negativo. Con la misma idea, escribir procedimientos* `SacarMultiple` *y* `MoverMultiple`*.*

4.3. Variables y asignación

La idea de dar nombres a expresiones, que ya fue utilizada en los parámetros de un procedimiento y en el índice de una repetición indexada, puede utilizarse de manera independiente a estos dos casos. Para ello, se define la noción de *variable*, que no es más que un identificador que se utiliza para denotar algún valor en cierto momento de la ejecución de un programa. Para establecer la correspondencia entre una variable y el valor que la misma denota se utiliza un comando particular, conocido como *asignación*.

Definición 35 *Una* variable *es un identificador que comienza con minúsculas y que se utiliza para denotar un valor en cierto momento de la ejecución de un programa.*

La asignación de un nombre a un valor se realiza escribiendo la variable, luego el signo `:=`, y luego la expresión que la variable nombra. Por ejemplo

```
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. La forma general de la asignación queda dada por la siguiente definición:

Definición 36 *La* asignación *es un comando mediante el cual se establece un nombre para cierto valor. Su forma es*

<variable> `:=` *<expresion>*

siendo <variable> un identificador con minúsculas, y <expresion> una expresión cualquiera.

El efecto de la acción descrita por un comando de asignación es la de "recordar" que el nombre de la variable se usará para denotar al valor descrito por la expresión. Para efectivizar la acción descrita por un comando de asignación, el cabezal primero calcula el valor de la expresión, y luego establece la correspondencia entre la variable y ese valor. Es importante destacar que el valor de la variable no cambiará si el valor de la expresión cambia a posteriori de la asignación, a menos que se realice una nueva asignación. Para observar este fenómeno, puede estudiarse el siguiente ejemplo:

```
procedure DosRojasMasQueVerdes()
  {- PROPÓSITO:
      * ilustra el uso de variables y el hecho de que la
        correspondencia no cambia aunque la expresión usada
        para definirla sí lo haga
     PRECONDICIÓN:
      * es una operación total
  -}
  {
    -- cantRojas representa el número de bolitas rojas en la
    --  celda actual al COMENZAR el procedimiento
    cantRojas := nroBolitas(Rojo)

    -- se agregan dos bolitas rojas a la celda actual
    PonerN(2, Rojo)

    -- se ponen tantas bolitas verdes como rojas había al COMIENZO
    PonerN(cantRojas, Verde)
        -- Observar en el tablero que el valor de cantRojas sigue
```

```
        --  siendo el de ANTES de agregar las 2 bolitas rojas
        --  (o sea, hay dos bolitas rojas más que verdes)
  }
```

Esto sucederá también en el caso en que el cabezal se mueva. Las variables son útiles para recordar ciertos valores al moverse por el tablero. Por ejemplo, en el siguiente ejercicio deberían usarse variables `cantRojas`, `cantVerdes`, `cantAzules` y `cantNegras`.

Ejercicio 25 *Escribir un procedimiento* `CopiarCeldaAlEste` *que, suponiendo que la celda lindante al Este no tiene bolitas, haga que la misma termine con el mismo número de bolitas de cada color que la celda actual. El procedimiento debe realizar un único movimiento al Este, y debe dejar el cabezal en el lugar de inicio.*

Las expresiones utilizadas en una asignación pueden ser de cualquier complejidad; esto es concordante con el hecho de que cualquier descripción de un valor puede ser utilizada de igual manera para denotarlo. Por ejemplo, si se quisiera poner en la celda lindante al Norte tantas bolitas negras como el total de bolitas de la celda actual, moviéndose sólo una vez a la celda lindante, se podría usar la siguiente asignación:

```
procedure NegrasAlNorte()
  {- PROPÓSITO:
      * pone tantas bolitas negras en la celda lindante al Norte
        como el total de bolitas de la celda actual,
        pero moviéndose sólo una vez a la celda lindante
     PRECONDICIÓN:
      * que haya una celda al Norte
     OBSERVACIÓN:
      * no es la mejor manera de abstraer este problema
  -}
  {
    -- nombra el número de bolitas de la celda actual
    aDepositar := nroBolitas(Rojo)  + nroBolitas(Azul)
                + nroBolitas(Verde) + nroBolitas(Negro)

    -- se mueve a la celda lindante y deposita la cantidad pedida
    Mover(Norte)
    PonerN(aDepositar, Negro)

    -- vuelve a la celda original
    Mover(Sur)
  }
```

Este ejercicio será revisado cuando se estudie la repetición condicional en la sección 4.4 (procedimiento `NegrasAlNorteConRepetir`), y nuevamente cuando se estudie el mecanismo de funciones en la sección 4.5 (procedimiento `NegrasAlNorteConFunciones`).

Las variables pueden nombrar valores de cualquier tipo. Por ejemplo, si se pide colocar en la celda lindante al Sur una bolita del color de las que haya más en la celda actual (en el caso que haya el mismo número de bolitas de cada color, debe colocarse una verde), moviéndose sólo una vez, podría usarse el procedimiento que se presenta a continuación (tener en cuenta la observación, que establece que esta forma de calcular el máximo es muy mala; sólo se usa este ejemplo para mostrar la capacidad de anidar alternativas condicionales).

```
procedure BolitaDeColorMaximo()
  {- PROPÓSITO:
```

```
     * colocar en la celda lindante al Sur, una bolita del color
        de las que haya más en la celda actual
    PRECONDICIÓN:
     * que haya una celda al Sur
    OBSERVACIÓN:
     * hay mejores formas de calcular el máximo, pero requieren
       características más avanzadas de Gobstones
  -}
  {
    if (nroBolitas(Azul) > nroBolitas(Negro) &&
        nroBolitas(Azul) > nroBolitas(Rojo)  &&
        nroBolitas(Azul) > nroBolitas(Verde))
      { colorAPoner := Azul }
    else          -- Las azules no son más que las demás
    {
      if (nroBolitas(Negro) > nroBolitas(Rojo)  &&
          nroBolitas(Negro) > nroBolitas(Verde))
        { colorAPoner := Negro }
      else        -- Ni las azules ni las negras son más
                  --    que las demás
      {
        if (nroBolitas(Rojo) > nroBolitas(Verde))
          { colorAPoner := Rojo }
        else      -- Las verdes son más o son todas iguales
          { colorAPoner := Verde }
      }
    }

    -- Observar que el valor de colorAPoner es un color!!
    Mover(Sur)
    Poner(colorAPoner)
    Mover(Norte)
  }
```

Observar que, tal cual lo consigna el comentario, hay mejores formas de calcular el máximo, pero requieren características más avanzadas de Gobstones, por lo que este ejemplo será revisado luego de ver las mismas en la sección 4.5.

¿Qué sucederá si se intenta usar una variable a la que no se le asignó ningún valor? Sencillamente, eso provoca la autodestrucción del cabezal. Un caso muy simple de cometer este error es realizar una asignación en alguna de las ramas de un condicional, y olvidarlo en la otra. Por ejemplo, si se pidiese agregar una bolita roja en la celda actual si no hubiese ninguna, y si hay rojas pero no verdes, agregar una bolita verde, el siguiente código no sería la solución buscada.

```
procedure DestruccionPorDescuido()
 {- PROPÓSITO:
     * ilustrar el uso de una variable que no fue asignada
    PRECONDICIÓN:
     * que no haya bolitas rojas en la celda actual,
       o si hay rojas, que no haya bolitas verdes
     * que haya una celda al Norte
 -}
 {
   if (not hayBolitas(Rojo))
```

```
    { colorAPoner := Rojo }
  else
    { if (not hayBolitas(Verde))
        { colorAPoner := Verde }
      -- NO HAY ELSE! Por lo tanto la variable colorAPoner no
      --  corresponde con ningún valor si las condiciones no
      --  valen!
    }

  Mover(Norte)
  Poner(colorAPoner)
  Mover(Sur)
}
```

¿Qué sucederá si en la celda actual hay bolitas rojas y verdes? Como establece la precondición, el procedimiento provocaría la autodestrucción del cabezal. El problema es que la acción de Poner en este caso es condicional, y por lo tanto no puede realizarse fuera de un `if-then-else`. Posibles soluciones a este problema incluyen repetir código en cada rama, o utilizar variables booleanas para saber si la variable tiene valor. La solución repitiendo el código en cada rama sería la siguiente:

```
procedure RojaOVerdeAlNorte_A()
 {- PROPÓSITO:
     * agregar una bolita roja o verde en la celda lindante al Norte
     * ilustrar la repetición de código en las ramas de un if
    PRECONDICIÓN:
     * que haya una celda al Norte
 -}
 {
   if (not hayBolitas(Rojo))
     {
       colorAPoner := Rojo
       Mover(Norte)               -- Código a repetir
       Poner(colorAPoner)
       Mover(Sur)
     }
   else
     { if (not hayBolitas(Verde))
         {
           colorAPoner := Verde
           Mover(Norte)           -- Código repetido
           Poner(colorAPoner)
           Mover(Sur)
         }
       -- NO HAY ELSE! Sin embargo, no importa porque
       --  la variable no se usa más allá de este punto
     }
 }
```

Por otra parte, la solución que utiliza una variable booleana para controlar que la variable `colorAPoner` tenga valor sería:

```
procedure RojaOVerdeAlNorte_B()
 {- PROPÓSITO:
```

```
    * agregar una bolita roja o verde en la celda lindante al Norte
    * ilustrar el uso de variables booleanas
   PRECONDICIÓN:
    * que haya una celda al Norte
-}
{
  if (not hayBolitas(Rojo))
    {
      colorAPoner := Rojo
      hayQuePonerBolita := True
    }
  else
    { if (not hayBolitas(Verde))
        {
          colorAPoner := Verde
          hayQuePonerBolita := True
        }
      else
        { hayQuePonerBolita := False }
        -- HAY ELSE! No se asigna colorAPoner porque no
        --  hay ningún valor razonable para ella. Sin
        --  embargo, no importa, pues no se usará a menos
        --  que hayQuePonerBolita sea True
    }

  if (hayQuePonerBolita)
    {
      Mover(Norte)
      Poner(colorAPoner)
      Mover(Sur)
    }
}
```

Observar en este último procedimiento que la variable `hayQuePonerBolita` siempre tiene valor, pues todas las ramas realizan una asignación para ella, y que la variable `colorAPoner` sólo tiene valor cuando la variable `hayQuePonerBolita` vale `True`.

Otra forma de provocar la autodestrucción por una variable que no fue definida es utilizar una variable cuando la misma no tiene sentido, por no corresponder con ningún valor. En Gobstones, la correspondencia entre una variable y un valor sirve solamente dentro del procedimiento donde se realiza la asignación; o sea, cada procedimiento tiene sus propios "recuerdos" que no comparte con ningún otro. La parte del código donde tal correspondencia tiene sentido para una variable determinada se denomina *alcance* de la variable. Así, en Gobstones, las variables tienen alcance dentro de un procedimiento exclusivamente.

Definición 37 *El* alcance *de una variable es la parte del código donde la correspondencia entre la misma y un valor tiene sentido. En Gobstones todas las variables tienen alcance sólo dentro del procedimiento que las asigna.*

El efecto de que el alcance de las variables sea entre procedimientos es que la única manera de pasar un valor entre procedimientos sea a través de los parámetros (existe otra forma más, asociada con las funciones, que se verán en la sección 4.5). O sea, no tiene sentido utilizar en un procedimiento una variable asignada en otro. Cada procedimiento posee su propia asignación de variables a valores. Por ejemplo, el siguiente código es

erróneo porque cada procedimiento tiene su propio espacio de variables, y por lo tanto no comparten las variables.

```
procedimiento Llamador_DefineVariable()
  {- PROPÓSITO:
      * ilustrar el uso incorrecto de variables
     PRECONDICIÓN:
      * falla siempre, por invocar un procedimiento que
        no puede tener éxito
  -}
  {
    cant := nroBolitas(Rojo)
    Mover(Norte)
    Llamado_UsaVariable()
  }
procedimiento Llamado_UsaVariable()
  {- PROPÓSITO:
      * ilustrar el uso incorrecto de variables
     PRECONDICIÓN:
      * falla siempre, pues cant no es una variable definida,
        ni un parámetro
  -}
  {
    PonerN(cant,Rojo)
        -- cant ES UNA VARIABLE SIN ASIGNAR!
        --  (La asignación de cant en el otro procedimiento no
        --   tiene validez en este)
  }
```

La forma correcta de realizar la comunicación entre procedimientos en este caso es utilizar parámetros, de la siguiente forma.

```
procedimiento Llamador_VariableComoArgumento()
  {- PROPÓSITO:
      * ilustrar el uso correcto de variables y parámetros
     PRECONDICIÓN:
      * debe haber celda al Norte
  -}
  {
    cant := nroBolitas(Rojo)
    Mover(Norte)
    Llamado_ConParametro(cant)
  }
procedimiento Llamado_ConParametro(cant)
  {- PROPÓSITO:
      * ilustrar el uso correcto de variables y parámetros
     PRECONDICIÓN:
      * siempre funciona
  -}
  {
    PonerN(cant,Rojo)
        -- cant ES UN PARÁMETRO!
        --  (Si bien se llama igual que la variable del otro
        --   procedimiento, podría llamarse diferente)
```

```
}
```

¿Cómo se relacionan las variables con los parámetros y los índices? La respuesta a esta pregunta es que no se mezclan. O sea, los parámetros y los índices, si bien son identificadores al igual que las variables, **no pueden ser asignados**. Entonces, no pueden existir, en un procedimiento dado, variables con el mismo nombre que un parámetro o un índice. Si esto sucediese, se producirá la autodestrucción del cabezal.

```
procedure VariosErrores(color)
  {- PROPÓSITO:
      * ilustrar la combinación incorrecta de parámetros y variables
        y de índices y variables
     PRECONDICIÓN:
      * este código siempre produce la autodestrucción
  -}
  {
    color := Negro   -- color no puede ser una variable, pues es
                     --  un parámetro
    cantidad := 1
    repeatWith cantidad in 1..5
             -- cantidad no puede ser un índice, pues es una variable
      {
        PonerN(cantidad, color)
            -- hace referencia a un índice o a una variable?
      }
  }
```

Ejercicio 26 *Escribir el código del ejemplo anterior, y realizar cambios en los nombres de las variables, parámetro o índices, hasta que ejecute correctamente. ¿Cuál es el efecto de este procedimiento? ¿Podría renombrar las variables de manera diferente para producir otro efecto distinto?*

Un punto de importancia con respecto a la asignación es que la misma variable puede asignarse varias veces a valores diferentes, dentro del mismo procedimiento. La correspondencia entre la variable y un valor producida por cada asignación dura hasta que la siguiente asignación es ejecutada. Esto hace que la utilización de variables sea muy dependiente del orden de ejecución. Para observar el efecto de esto, puede analizarse el siguiente ejemplo.

```
procedure VariasAsignaciones()
  {- PROPÓSITO:
      * ilustrar la posibilidad de reasignación de una variable
     PRECONDICIÓN:
      * debe haber una celda al Norte
     OBSERVACIÓN:
      * observar como se repite código. Esto se puede abstraer mejor
        como se vé en el ejercicio VariasAsignacionesConRepetir
  -}
  {
    cantAPoner := nroBolitas(Rojo)
    Mover(Norte)
    PonerN(cantAPoner, Rojo)
       -- Se refiere a la cantidad de bolitas ROJAS de
       --  la celda inicial
```

```
    Mover(Sur)

    cantAPoner := nroBolitas(Verde)
    Mover(Norte)
    PonerN(cantAPoner, Verde)
       -- Se refiere a la cantidad de bolitas VERDES de
       --  la celda inicial
    Mover(Sur)
  }
```

Observar que la variable `cantAPoner` se refiere a distintos valores en distintas partes del cuerpo del procedimiento, según cuál fue la última asignación ejecutada. Este fenómeno de reasignación es extremadamente útil para ser combinado con repetición. En este caso particular, el código anterior podría haberse escrito como:

```
procedure VariasAsignacionesConRepetir()
  {- PROPÓSITO:
      * ilustrar la combinación de reasignación de una variable
        con un comando de repetición
     PRECONDICIÓN:
      * debe haber una celda al Norte
  -}
  {
    repeatWith color in Rojo..Verde
      {
        cantAPoner := nroBolitas(color)
        Mover(Norte)
        PonerN(cantAPoner, color)
          -- Se refiere a la cantidad de bolitas de la celda
          -- inicial del color correspondiente a la iteración
        Mover(Sur)
      }
  }
```

¿Cuántas veces se reasigna la variable `cantAPoner` en este caso? Tantas veces como repeticiones del cuerpo del comando de iteración se realicen – en este caso, dos, una para el color `Rojo` y otra para el color `Verde`.

La última característica destacada de la asignación y la reasignación que se mencionará tiene que ver con el uso de una variable `v` en un comando que reasigna a esa misma variable `v`. Esto se conoce como *modificar* la variable `v`, y en el caso de que `v` sea un número al que se le suma un valor, se dice que se *incrementa* la variable. La modificación o incremento no debería producir ningún problema de interpretación si se lee un comando de asignación de la manera correcta. Por ejemplo, el siguiente comando

```
cant := cant + nroBolitas(Rojo)
```

debe leerse como "calcular el valor de la suma entre el valor designado por `cant` actualmente, sumarlo a la cantidad de bolitas rojas de la celda actual, y *luego* realizar una nueva asignación para `cant` con ese valor". Para simplificar esta lectura, se leerá "la variable `cant` se *incrementa* en la cantidad de bolitas rojas de la celda actual". Por ejemplo, en el siguiente código se agregan 2 bolitas verdes y 3 azules!

```
procedure ReasignacionYUso()
  {- PROPÓSITO:
      * ilustrar la modificación de una variable (o sea,
```

```
        el uso de una variable en una expresión que reasigna
        a la misma variable)
      * coloca dos bolitas verdes y tres bolitas azules en la
        celda actual (suponiendo que no haya rojas previamente)
    PRECONDICIÓN:
     * es una operación total
 -}
{
  Poner(Rojo)
  cant := 2
  PonerN(cant, Verde)
      -- en este punto cant vale 2

  cant := cant + nroBolitas(Rojo)
       -- en este punto cant todavía vale 2 pero pasará
       -- a valer 3 o más al terminar la asignación
       -- O sea, se incrementó cant!

  PonerN(cant, Azul)
      -- en este punto cant vale 3 (o más)
}
```

La modificación de una variable es extremadamente útil para llevar cuentas parciales al realizar una suma. Por ejemplo, si se desea el número total de bolitas de la celda actual, podría usarse el siguiente código:

```
totalBolitas := 0
repeatWith color in Azul..Verde
  {
    -- se incrementa totalBolitas para considerar un nuevo color
    totalBolitas := totalBolitas + nroBolitas(color)
  }
```

En cada repetición se incrementa `totalBolitas` con el número de bolitas de un nuevo color (o sea, se la reasigna a un valor que le agrega al previo el número de bolitas de cada color). Como empieza en `0` y recorre todos los colores, al terminar la repetición, el valor de `totalBolitas` corresponde al número de bolitas de la celda actual. Para probar este efecto, se puede usar este código como variante del procedimento `NegrasAlNorte`, presentado antes.

```
procedure NegrasAlNorteConRepetir()
  {- PROPÓSITO:
      * ilustrar el uso de incremento de variables
      * pone tantas bolitas negras en la celda lindante al Norte
        como el total de bolitas de la celda actual,
        pero moviéndose sólo una vez a la celda lindante
    PRECONDICIÓN:
     * que haya una celda al Norte
    OBSERVACIÓN:
     * esta no es la mejor manera de abstrer este problema;
       sin embargo, mejora a la del procedimiento NegrasAlNorte
  -}
  {
    -- nombra el número de bolitas que se consideró hasta el momento
```

```
  totalBolitas := 0
  repeatWith color in minColor()..maxColor()
   {
     totalBolitas := totalBolitas + nroBolitas(color)
   }
  -- se consideraron todas las bolitas de la celda actual

  -- se mueve a la celda lindante y deposita la cantidad pedida
  Mover(Norte)
  PonerN(totalBolitas, Negro)

  -- vuelve a la celda original
  Mover(Sur)
}
```

Esta forma de abstraer el problema mejora a la que se utilizó en el procedimiento `NegrasAlNorte` presentado en la sección 4.3. Nuevamente debe destacarse que hay formas todavía más interesantes de dividir este problema, que requieren el uso de características que se tratan en la sección 4.5.

La técnica de ir incrementando una variable se denomina *contar*, y por lo tanto a la variable que se incrementa se la denomina *contador*. El uso de contadores es más útil al combinarse con movimiento, como se muestra en el siguiente código:

```
procedure PonerMuchasAzules()
  {- PROPÓSITO:
      * ilustrar un uso interesante de un contador
      * pone tantas bolitas azules en la celda actual,
        como bolitas verdes haya en las celdas lindantes
        en las cuatro direcciones
     PRECONDICIÓN:
      * que haya celdas lindantes en las cuatro direcciones
  -}
  {
    -- contará el número de bolitas vistas hasta el momento
    totalBolitas := 0   -- al principio no se vio ninguna
    repeatWith dir in minDir()..maxDir()
     {
       -- va a una celda lindante
       Mover(dir)

       -- incrementa el contador totalBolitas para considerar
       -- las bolitas verdes de la celda lindante visitada
       totalBolitas := totalBolitas + nroBolitas(Verde)

       -- retorna a la celda original
       Mover(opuesto(dir))
     }

    -- pone el número especificado de bolitas azules en la
    --  celda original
    PonerN(totalBolitas, Azul)
  }
```

Observar cómo el valor asociado a la variable `totalBolitas` va llevando la cuenta a medida que se visitan las distintas celdas lindantes.

Ejercicio 27 *¿Cómo podría hacerse este ejemplo sin incrementar variables? (Sugerencia: considerar cómo habría que modificar el código para que poner azules se realice en cada iteración.)*

4.4. Repetición condicional

En la sección 3.6 se presentó la repetición indexada, pero se mencionó que existen otras variantes de repetición. La más importante de todas estas variantes se denomina *repetición condicional*. Mientras que la repetición indexada repite un número fijo de veces, cantidad controlada por un rango de valores, la repetición condicional puede repetirse un número arbitrario de veces, dependiendo de que una condición se cumpla o no. El comando para describir una repetición condicional es llamado `while`, que en inglés significa "mientras", para marcar que el bloque a repetir será iterado *mientras* se cumpla cierta condición. La forma de este comando se establece en la siguiente definición:

Definición 38 *La* repetición condicional *es una forma de repetición que depende de una condición booleana para indicar cuando debe cesar de iterarse. La forma del comando que describe esta repetición es*

`while (`*<condicion>*`)`
<bloque>

donde <condicion> es una expresión booleana y <bloque> un bloque cualquiera al que se denomina cuerpo *de la repetición.*

El efecto del comando `while` es la repetición del comando descrito por el *<bloque>* mientras la *<condicion>* sea verdadera. Para obtener este efecto, la ejecución del comando comienza evaluando la *<condición>* en el estado actual. Si esta evaluación resulta verdadera (o sea, la expresión *<condicion>* evalúa a `True`), se realiza una iteración del comando descrito por el *<bloque>*, y a continuación se vuelva a evaluar la condición. Esto último se repite hasta tanto la condición resulte falsa, en cuyo caso termina la ejecución del comando. Observar que es necesario que el bloque del cuerpo altere el estado de tal manera de que la condición llegue a resultar falsa en algún momento.

El ejemplo más sencillo de repetición condicional es llevar el cabezal hasta un borde del tablero. El siguiente código lleva el cabezal hasta el borde superior:

```
procedure CabezalAlTopeDeLaColumna()
  {- PROPÓSITO:
      * ilustrar el uso de while
      * llevar el cabezal a la última fila de la columna actual
     PRECONDICIÓN:
      * es una operación total
  -}
  {
    while (puedeMover(Norte))
      { Mover(Norte) }
  }
```

Observar que el cuerpo del `while` modifica el estado, y en determinado momento esa modificación hace que la condición pase a ser falsa, dando por finalizada la ejecución del comando. Esta tarea no podría hacerse con repetición indexada, pues no se conoce el tamaño del tablero, ni hay forma de determinar la posición de la celda actual.

La repetición condicional es una de las formas más poderosas de repetición, y por ello, también más difícil de manejar correctamente. Por ejemplo, podría suceder que la condición jamás se hiciese verdadera. En este caso, el programa provocará que la máquina quede infinitamente intentando realizar una tarea, lo cual no puede ser comprobado de ninguna forma externa durante la ejecución de dicho programa. Como ejemplo, considerar el siguiente código:

```
procedure LaBuenaPipa()
  {- PROPÓSITO:
      * ilustrar un comando de repetición cuya ejecución no termina
     PRECONDICIÓN:
      * ninguna: el programa jamás termina, por lo que nunca produce
        ningún resultado
  -}
  {
    QueresQueTeCuenteElCuentoDeLaBuenaPipa()
    respuesta := True
    while (respuesta)
     {
       YoNoDije_True_Dije_QueresQueTeCuenteElCuentoDeLaBuenaPipa()
       respuesta := True
     }
  }
```

donde los dos procedimientos internos son cualesquiera que definan operaciones totales. Es claro que como la respuesta es siempre la misma, este cuento seguirá para siempre.

Esta capacidad de ejecutar infinitamente es equiparable a la autodestrucción del cabezal, en tanto y en cuanto no produce un tablero determinado. Sin embargo, su manifestación es diferente, pues se limita a mantener una apariencia de trabajo cuando en realidad no hay ningún trabajo que termine dando beneficios. A pesar de tener esta manifestación diferente, la precondición de un procedimiento debe incluir las condiciones para asegurar que todas las repeticiones condicionales terminan. Por ejemplo, considerar el siguiente código:

```
procedure NIndiecitos(n)
  {- PROPÓSITO:
      * ilustrar el uso de precondiciones para evitar no terminación
      * sacar n indiecitos (representados por bolitas rojas) de
        la celda actual
     PRECONDICIÓN:
      * n debe ser un número positivo
        (si no, el procedimiento no termina)
  -}
  {
    PonerN(n,Rojo)
    indiecitos := n             -- Al principio habia n indiecitos
    while (indiecitos /= 0)
      {
        Sacar(Rojo)             -- A uno le dio dolor de panza
        indiecitos := indiecitos - 1
                                -- y me quedaron menos indiecitos
      }
  }
```

Observar que la condición establece que el valor de la variable `indiecitos` debe ser distinto de cero. Si el número `n` es mayor o igual a cero, y como dicha variable se decrementa en `1` en cada iteración, habrá un momento en el que valdrá `0` y el ciclo terminará. Si en cambio `n` es negativo, al decrementarlo se alejará del `0` y por lo tanto nunca concluirá el ciclo, provocando la ejecución infinita.

La repetición condicional es útil cuando se trata de realizar una tarea repetitiva de la que no se sabe a priori cuántas veces debe repetirse (o sea, no se puede generar un rango sobre el cuál realizar la iteración). Un ejemplo de esto es procesar todas las celdas de una columna, o todas las columnas de una grilla, como se muestra a continuación:

```
procedure PintarColumna(color)
  {- PROPÓSITO:
      * pinta una columna entera con bolitas de color
     PRECONDICIÓN:
      * ninguna, es una operación total
  -}
  {
    -- va al tope para saber que las hace todas
    CabezalAlTopeDeLaColumna()

    while (puedeMover(Sur))
        -- la condición dice "no llegó a la base"
      {
        Poner(color)  -- pintar celda de color
        Mover(Sur)    -- pasar a la siguiente celda
      }

    -- pinta la última celda, pues no entró al while si
    --  la celda no tenía una al Sur
    Poner(color)
  }
```

Observar que no hay manera de realizar esta tarea con una repetición indexada pues no se conoce de antemano el número de celdas que hay que pintar. Otros puntos interesantes a observar son que se comienza llevando el cabezal al tope de la columna para asegurarse que todas las celdas resultarán pintadas, y que al finalizar el ciclo hace falta pintar la última celda a mano, pues el pintar celda de dentro del ciclo no se realiza sobre la última celda (ya que desde ella no se puede mover al Sur).

Este mismo esquema de repetición sobre celdas puede utilizarse sobre otros elementos. Por ejemplo, si se desea pintar toda el tablero de un color, puede descomponerse esta tarea en pintar cada una de las columnas del tablero. El código para esta tarea sería

```
procedure PintarTablero(color)
  {- PROPÓSITO:
      * pinta todo el tablero con bolitas de color
     PRECONDICIÓN:
      * ninguna, es una operación total
  -}
  {
    -- va a la primera columna para saber que las hace todas
    CabezalALaPrimeraColumna()

    while (puedeMover(Este))
```

```
      -- la condición dice "no llegó a la última columna"
    {
      PintarColumna(color)  -- pintar columna de color
      Mover(Este)           -- pasar a la siguiente columna
    }

  -- pinta la última columna, pues no entró al while si
  --  la columna era la última
  PintarColumna(color)
}
```

Al igual que en `PintarColumna`, no es posible utilizar una repetición indexada, pues no se conoce de antemano el número de columnas. Además, se observa que el esquema de resolución es el mismo: se comienza ubicando la primera columna, y a partir de ahí se van pintando cada una de ellas (utilizando el procedimiento `PintarColumna`), hasta la última, que debe ser pintada de manera separada (pues la última columna no satisface la condición del `while`, y por lo tanto no resulta pintada).

Debe definirse el procedimiento `CabezalALaPrimeraColumna` para que el código quede completo.

Ejercicio 28 *Escribir un procedimiento* `CabezalALaPrimeraColumna` *que lleve el cabezal a la primera columna del tablero.*

Este esquema de repetición donde se realiza una tarea para cada uno de una serie de elementos se puede reusar para diferentes elementos. Por ejemplo, si se plantea el problema de agregar una bolita roja a cada celda de una columna que contenga una bolita verde, el código podría ser el siguiente

```
procedure PonerRojasAVerdesEnColumna()
  {- PROPÓSITO:
      * Poner una bolita roja en cada celda de una columna
        que contenga una bolita verde
     PRECONDICIÓN:
      * ninguna, es una operación total
  -}
  {
    -- va al tope para saber que las hace todas
    CabezalAlTopeDeLaColumna()

    while (puedeMover(Sur))
        -- la condición dice "no llegó a la base"
      {
        if (hayBolitas(Verde))
          { Poner(Rojo) }       -- sólo se agrega en las celdas que
                                --  cumplen la condición
        Mover(Sur)              -- pasar a la siguiente celda
      }

    -- realiza la acción en la última celda, pues no entró
    --  al while si la celda no tenía una al Sur
    if (hayBolitas(Verde))
      { Poner(Rojo) }
  }
```

Otro caso donde es más simple utilizar repetición condicional es cuando la separación entre los valores de un rango no es uniforme. Por ejemplo, si se quiere colocar una progresión de bolitas verde en 8 celdas, tal que la cantidad de bolitas en cada celda sean sólo los múltiplos de 2 o de 3 (o sea, 2, 3, 4, 6, 8, 9, 10 y 12), podría usarse una repetición condicional de la siguiente manera

```
procedure ProgresionVerdeDosOTres()
  {- PROPÓSITO:
      * armar una progresión de 8 celdas con cantidad de
        bolitas que sea múltiplo de 2 o de 3
     PRECONDICIÓN:
      * que haya 8 celdas al Este
  -}
  {
    procesadas := 0      -- la cantidad de celdas ya tratadas
    cantAColocar := 2    -- el inicio de la progresión
    while (procesadas < 8)
      {
        if (cantAColocar mod 2 == 0    -- múltiplo de 2
           || cantAColocar mod 3 == 0) -- o múltiplo de 3
          {
            PonerN(cantAColocar, Verde)
            procesadas := procesadas + 1  -- se procesó una celda
            Mover(Este)                   -- por eso pasa a la sig.
          }
         cantAColocar := cantAColocar + 1 -- probar un nuevo nro.
      }
  }
```

4.5. Funciones

Así como un procedimiento es una forma de dar nombre a un grupo de comandos, existe un mecanismo para darle nombre a un grupo de expresiones. Tal mecanismo es conocido con el nombre de *funciones*. Sin embargo, en Gobstones, para ser capaces de obtener valores basados en el tablero, es necesario que las funciones puedan describir acciones que el cabezal deberá realizar. Por ello, el cuerpo de una función se compondrá de una serie de comandos, finalizados con las expresiones que la función representa.

Definición 39 *Una* función *simple es una forma de asignar un nombre a una expresión. La forma de definir una función en Gobstones es*

```
function <funcName>(<params>)
 {
    return(<expresion>)
 }
```

siendo <funName> un identificador que comienza con minúscula, <params>, una lista de identificadores que comienzan con minúsculas y <expresion>, una expresión de cualquier tipo básico.

La operación de `return` de la declaración de una función indica la expresión que la misma representa.

Las funciones se pueden invocar de manera similar a un procedimiento, con la diferencia de que, como representan a un valor, deben utilizarse como cualquier expresión (y no como un comando). Por ejemplo, un llamado de función puede ser usado como argumento, puede ser asignado, etc.

Definición 40 *Una función puede ser usada como una expresión. La forma de invocarla es escribir su nombre seguida por argumentos, de la siguiente manera*

<funcName>(<args>)

donde <args> es una lista de valores específicos (los argumentos) para los parámetros de la función.

Un ejemplo sencillo de función simple es la que calcula el número total de bolitas de la celda actual.

```
function nroBolitasTotal()
  {- PROPÓSITO:
      * calcula el total de bolitas de la celda actual
     PRECONDICIÓN:
      * ninguna, es una operación total
  -}
  {
    return ( nroBolitas(Azul) + nroBolitas(Negro)
           + nroBolitas(Rojo) + nroBolitas(Verde)
           )
  }
```

Al utilizar esta función, el valor representado por su invocación es el número total de bolitas de todos los colores en la celda actual. Por ejemplo, si se pidiese poner en la celda lindante al norte tantas bolitas negras como el total de bolitas de la celda actual, se podría usar el siguiente procedimiento

```
procedure NegrasAlNorteConFunciones()
  {- PROPÓSITO:
      * ilustrar el uso de funciones
      * pone tantas bolitas negras en la celda lindante al Norte
        como el total de bolitas de la celda actual,
        pero moviéndose sólo una vez a la celda lindante
     PRECONDICIÓN:
      * que haya una celda al Norte
     OBSERVACIÓN:
      * esta es una manera adecuada de abstraer este problema,
        a diferencia de la usada en los procedimientos
        NegrasAlNorte y NegrasAlNorteConRepetir
  -}
  {
    -- usa una función para calcular el número de bolitas de
    --  la celda actual y le coloca un nombre a tal número
    aDepositar := nroBolitasTotal()

    -- se mueve a la celda lindante y deposita la cantidad pedida
    Mover(Norte)
    PonerN(aDepositar, Negro)

    -- vuelve a la celda original
    Mover(Sur)
  }
```

Otro ejemplo de uso de funciones es el de calcular el color de las bolitas con más cantidad en la celda actual. Sin embargo, en este caso, se observa la necesidad de que

las funciones puedan ejecutar comandos, ya que tanto el condicional como la repetición sólo son obtenibles usando comandos, y alguno de los dos es necesario para calcular el máximo. Si hay más de un color con el mismo número de bolitas máximo, entonces debe devolverse cualquiera de ellos.

La definición general de las funciones es, entonces, la siguiente.

Definición 41 *Una* función *puede llevar un comando como parte de su definición. La forma general de una función en Gobstones es*

```
function <funcName>(<params>)
 {
    <comando>
    return(<expresion>)
 }
```

siendo <funName>, <params> y <expresion> como antes, y <comando> un comando (que puede no aparecer).

Con esta nueva forma, la definición con condicionales sería

```
function colorMaximoConCondicionales()
  {- PROPÓSITO:
      * calcula el color de las bolitas de las que
        hay más en la celda actual
     PRECONDICIÓN:
      * ninguna, es una operación total
     OBSERVACIÓN:
      * no es la mejor manera de implementar esta función. Ver la
        función colorMaximo para una manera más adecuada
  -}
  {
    if (nroBolitas(Azul) > nroBolitas(Negro) &&
        nroBolitas(Azul) > nroBolitas(Rojo)  &&
        nroBolitas(Azul) > nroBolitas(Verde))
      { colorAPoner := Azul }
    else          -- Las azules no son más que las demás
    {
      if (nroBolitas(Negro) > nroBolitas(Rojo)  &&
          nroBolitas(Negro) > nroBolitas(Verde))
        { colorAPoner := Negro }
      else        -- Ni las azules ni las negras son más
                  --    que las demás
      {
        if (nroBolitas(Rojo) > nroBolitas(Verde))
          { colorAPoner := Rojo }
        else      -- Las verdes son más o son todas iguales
          { colorAPoner := Verde }
      }
    }

    return (colorAPoner)
  }
```

En este caso, el cuerpo de la función posee un comando de alternativa condicional, con varias ramas, cada una de las cuales consiste de una asignación, y el valor final de retorno se toma de la variable en cuestión.

Otra manera de calcular un máximo es utilizar una repetición, de la siguiente manera:

```
function colorMaximo()
  {- PROPÓSITO:
      * calcula el color de las bolitas de las que
        hay más en la celda actual, con repeticion
     PRECONDICIÓN:
      * ninguna, es una operación total
     OBSERVACIÓN:
      * notar que el código es independiente de la cantidad de colores
        y del orden de los mismos
  -}
  {
    colorAPoner := minColor()
    nroMaximo := nroBolitas(minColor())
    repeatWith c in siguiente(minColor())..maxColor()
      {
        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)
          }
      }

    return (colorAPoner)
  }
```

En este caso, el cuerpo de la función consiste de asignaciones y la repetición indexada de un condicional. Esta forma de calcular el máximo con repetición es generalizable a cualquier cantidad de elementos, mientras que la versión con condicionales no. La función de calcular el color máximo se puede utilizar en el procedimiento de `BolitaDeColorMaximo`, de la siguiente manera

```
procedure BolitaDeColorMaximoConFunciones()
  {- PROPÓSITO:
      * ilustrar el uso de funciones
      * colocar en la celda lindante al Sur, una bolita del color
         de las que haya más en la celda actual
     PRECONDICIÓN:
      * que haya una celda al Sur
     OBSERVACIÓN:
      * notar que esta versión es mucho más clara y conceptualmente
        adecuada que la del ejercicio BolitaDeColorMaximo
  -}
  {
    colorAPoner := colorMaximo()
    Mover(Sur)
    Poner(colorAPoner)
    Mover(Norte)
  }
```

Una característica distintiva de las funciones en Gobstones es que no pueden alterar el estado de manera permanente. A pesar de que pueden indicar al cabezal que realice

acciones, estas acciones no serán tenidas en cuenta por el procedimiento que invoca la función. O sea, toda alteración del estado que se haga en una función es estricamente local a la ejecución de la misma. Como ejemplo de ello, veamos una función que calcula el número total de bolitas verdes de la columna actual.

```
function verdesEnColumna()
  {- PROPÓSITO:
      * retorna el número de bolitas verdes de la columna actual
     PRECONDICIÓN:
      * ninguna, es una operación total
     OBSERVACIONES:
      * altera el estado, pero sólo de manera local
  -}
  {
    CabezalAlTopeDeLaColumna() -- para considerar todas las celdas
    totalVerdes := nroBolitas(Verde)
                -- al principio sólo consideró las bolitas de
                --    la celda actual
    while (puedeMover(Sur))
      {
        Mover(Sur)                        -- pasa a la siguiente
        totalVerdes := totalVerdes +
                       nroBolitas(Verde) -- y cuenta sus bolitas
      }

    return (totalVerdes)
  }
```

Esta función es un ejemplo de que si bien las funciones pueden describir acciones del cabezal, las mismas sólo son simuladas, y no se efectúan en el código del procedimiento llamador. Como ejemplo, si se desea poner tantas bolitas rojas en la celda actual como el número de verdes de la columna actual, el código sería el siguiente

```
procedure RojasAcaComoVerdesEnColumna()
  {- PROPÓSITO:
      * ilustra la localidad del efecto de las funciones
      * coloca tantas bolitas rojas como bolitas verdes
         hay en la columna actual
     PRECONDICIÓN:
      * ninguna, es una operación total
     OBSERVACIONES:
      * si bien la función altera el estado, lo hace sólo
         de manera local por lo que el Poner sucede en la
         misma celda inicial
  -}
  {
    -- el cabezal está en el mismo lugar que empezó, a pesar
    --  de que la función indicó que se moviera
    PonerN(verdesEnColumna(), Rojo)
  }
```

Observar que desde el punto de vista de este procedimiento, la función `verdesEnColumna` sólo calcula el número de bolitas verdes en la columna actual. A pesar de que el cuerpo de la función indica que el cabezal debe moverse, tal movimiento no es recordado por

el procedimiento `RojasAcaComoVerdesEnColumna`. Esta pureza de las funciones es una capacidad única de Gobstones.

Las funciones, al igual que los procedimientos, pueden tomar parámetros, para representar algún valor que sea diferente en cada invocación de la misma. Por ejemplo, si se trata de determinar cuál es la celda lindante con el mayor número de bolitas de un color dado (y suponiendo que si hay dos con la misma cantidad, se retorna cualquiera), se podría resolver con la siguiente función

```
function direccionConMasBolitas(color)
  {- PROPÓSITO:
      * ilustra el uso de parámetros en una función
      * retorna la dirección de la celda lindante con más
        bolitas del color especificado
     PRECONDICIÓN:
      * debe haber celdas lindantes en las cuatro direcciones
  -}
  {
    maximoActual := 0  -- hasta ahora no se vio ninguna bolita
    repeatWith d in minDir() .. maxDir()
      {
        Mover(d)
        if (nroBolitas(color) >= maximoActual)
          -- Entra por acá por lo menos una vez, pues
          --  nroBolitas siempre es >= 0
          {
            -- Actualiza el máximo actual
            maximoActual := nroBolitas(color)
            -- y la dirección en que lo encontró
            direccionFinal := d
          }
        Mover(opuesta(d))
      }

    return (direccionFinal)
         -- La variable tiene valor, pues entró al menos una vez
  }
```

Ejercicio 29 *¿Qué modificaciones habría que hacerle a esta función para que trabaje aún sobre un borde, o sea, que sea una función total?*

Otra caracteterística interesante de las funciones en Gobstones es que pueden devolver más de un valor. Esto es extremadamente útil en varias circunstancias. Por ejemplo, supongamos que representamos las paradas del 159 con celdas con tres bolitas verdes. Queremos averiguar la distancia que hay desde la celda actual a la primera parada del 159 al este. ¿Qué debe devolver la función si no hay ninguna parada al este? Como se puede devolver más de un valor, se puede hacer lo siguiente

```
function distanciaALaParadaDel159()
  {- PROPÓSITO:
      * ilustra una función con varios valores de retorno
      * retorna si existe una parada del 159 al Este,
        y si existe, la distancia a la primera de ellas
     PRECONDICIÓN:
      * ninguna, es una función total
```

```
  -}
  {
     encontreParada := estoyEnLaParadadel159()
                                -- hasta ahora, solo vi aca
     cuantoCamine := 0          -- y no camine nada
     while (puedeMover(Este) && not encontreParada)
       {
         Mover(Este)
         cuantoCamine := cuantoCamine + 1
         encontreParada := estoyEnLaParadadel159()
       }

     return (encontreParada, cuantoCamine)
          -- si hay o no,     y  cuanto camine (si encontre,
          --                     es el valor que quiero, pero
          --                     si no, es cualquiera)
  }

function estoyEnLaParadadel159()
  { return (nroBolitas(Verde) == 3) }
```

Observar cómo la función posee dos valores de retorno. En este caso, el segundo de ellos sólo tendrá sentido cuando el primero sea verdadero. Esta es una forma muy adecuada de informar que cierto valor no fue encontrado. Para utilizar una función que retorna más de un valor, debe asignarse a varias variables. Por ejemplo, si suponemos que representamos un cartel de la campaña de donación de sangre mediante una bolita roja, y tenemos que poner el carte en la primera parada del 159 al Este de la celda actual, pero que no se mueva si no existe, se puede proceder de la siguiente forma

```
procedure AnunciarDonacionDeSangreEnLaParadaDel159()
  {- PROPÓSITO:
      * ilustra el uso de varios valores de retorno
      * si existe alguna parada del 159 al este se mueve ahí y
         pone un cartel de la campaña de donacion de sangre,
         y si no existe, no se mueve
     PRECONDICIÓN:
      * ninguna, es una operación total
  -}
  {
    (deboMover, nroCeldas) := distanciaALaParadaDel159()
    if (deboMover)
      {
        MoverN(nroCeldas, Este)
        PonerCartelDeDonacionDeSangre()
      }
    -- Observar que si deboMover es falso, nroCeldas no se usa!
  }

procedure PonerCartelDeDonacionDeSangre()
  { Poner(Rojo) }
```

Las funciones con múltiples valores de retorno sólo se pueden usar en asignaciones que contengan la cantidad correcta de variables.

5. CONTINUARÁ...

Main con return TO DO!
Case TO DO!
BOOM TO DO!
Skip TO DO!
includes (version 2.2) TO DO!

6. Versiones

Las diferentes versiones de Gobstones 2 fueron introduciendo pequeñas variaciones y corrigiendo bugs. Se reportan aquí las mismas, para consulta de quienes conocen o utilizan una versión anterior.

Gobstones 2.0

- Cambia la estructura sintáctica de manera drástica.
  - Se agregan los bloques.
  - Se cambian los nombres y la estructura de los comandos básicos.
- Se agregan procedimientos y funciones.
- Se definen scopes para las variables, donde todas las variables son locales.
- Se agregan las nociones de parámetros e índices.
- Los numerales pueden ser negativos.

Gobstones 2.01

- Las funciones no pueden alterar el estado del tablero (antes eran como los procedimientos).

Gobstones 2.02

- Se corrige el error de orden de los colores. El correcto es el orden alfabético.
- Se agregan operaciones de mínimo y máximo para colores, direcciones y booleanos.

Gobstones 2.03

- Se deshabilitan las funciones básicas `cantFilas` y `cantColumnas`, para evitar iteraciones por fila usando `repeatWith`.
- Las funciones que retornan un solo valor pueden usarse como expresiones (en versiones anteriores sólo podían asignarse).
- Las funciones pueden consistir sólo de una instrucción de retorno (antes tenían que tener un comando).

Gobstones 2.04

- Se agregan bloques vacíos.
- Se agrega el comando `BOOM`, que provoca la autodestrucción deliberada del cabezal.
- En la versión 2.04b se corrije un bug en el uso de puntos y comas, y uno en el comando `previo`.

Gobstones 2.05

- Se agrega el reporte del procedimiento donde se produce una autodestrucción, y la secuencia de llamadas que lo produce.
- El case ahora funciona para cualquiera de los tipos básicos.
- La función `opuesta` cambió a `opuesto(,)` y sirve para números además de para direcciones.

Gobstones 2.06

- El cabezal inicia en una posición aleatoria.
- La función opuesto se puede abreviar con la operación unaria de `-` (opuesto).

Gobstones 2.07

- Se agrega el comando `IrAlOrigen`.

Gobstones 2.10

- El tablero muestra coordenadas de los cuatro lados.
- Mejor control de error para overflow de bolitas.
- Existe una forma de grabar el tablero final en un archivo de extensión "`.gbt`".
- Existe una forma de leer un tablero (puede o no fijarse la celda inicial).
- El tablero es aleatorio a menos que se lea.
- Se agrega el comando `VaciarTablero`.

Gobstones 2.11

- La lectura del tablero acepta las bolitas en cualquier lugar dentro de la celda, y acepta que el número y la letra del color estén separadas.

Gobstones 2.12

- La lectura del tablero acepta que las letras para los colores estén con minúsculas.
- Se corrigieron algunos mensajes de error.
- Se llevaron todos los mensajes de error a castellano.
- Se proveyó un módulo `Babel` para que sea posible cambiar la herramienta al inglés.
- Se mejoró el *pretty printing*, usando la biblioteca estándar de Haskell98.

7. Wish List

En esta sección se irán consignando las características proyectadas para futuras versiones del lenguaje, pero que por razones de complejidad o falta de tiempo no han sido agregadas en la versión actual. Si tiene su deseo de alguna característica que le desearía ver en Gobstones, envíe un mail a la dirección de los autores, y será agregado aquí para tener tal característica en cuenta al revisar el lenguaje.

Generales:

- Que en los tableros grabados se puedan ver las bolitas de colores.

Para Gobstones 2.2:

- Que exista un mecanismo de *includes* que permita separar un programa en varios archivos.

Para Gobstones 3.0:

- Que exista un sistema de tipos que rechace como inválidas las expresiones mal formadas por no coincidir los tipos.

8. Agradecimientos

Agradecemos la atenta lectura de versiones preliminares de este tutorial por parte de Nicolás Passerini y Fernando Boucquez así como los comentarios que ellos sugirieron. Lilián García Rojas ha aportado las capturas de pantalla como así también ha integrado el grupo de estudiantes que cursaron la materia *Introducción a la Programación* durante el primer cuatrimestre de 2008. Las preguntas e inquietudes de este grupo han determinado, en gran parte, el contenido del presente tutorial. También queremos agradecer a Aylen quien a pesar de su juventud, señaló deficiencias interesantes en la presentación de algunos temas.

9. Conclusiones

Definir un lengujaje de programación desde cero, sobre todo de tal manera que sea lo más simple posible y que permita transmitir las ideas básicas de programación sin complicaciones innecesarias es una tarea de gran envergadura. Brindar, simultáneamente, una herramienta mínimamente utilizable y un tutorial que permita el acceso sencillo a las ideas que se busca impartir es un agregado que implica muchísimo esfuerzo y trabajo. Y si además esto hay que hacerlo sin descuidar ninguna de otras muchas obligaciones, pues... esto es lo que resulta.

Ambos autores estamos extremadamente satisfechos con los avances logrados. La herramienta conseguida y este tutorial distan de estar en su versión más acabada, y sin embargo, es mucho lo que se puede lograr con lo que hemos conseguido escribir e implementar. Esperamos que sea de gran provecho para quienes intentan aprender a programar, y que sea tan divertido usarlo como lo fue para nosotros definirlo.

Para cerrar, nos gustaría volver a repetir que cualquier comentario, observación, duda, idea, ejercicio adicional, crítica (siempre que sea constructiva... ;-)), etc. es bienvenido. ¡Suerte con Gobstones!

A. Sintaxis de Gobstones

Esta sección presenta una breve referencia de la estructura de un programa Gobstones, con la intención de servir de consulta rápida.

Para la presentación se utiliza notación estilo BNF, que es común en la manera de transmitir estructura de lenguajes de programación. En esta notación se presentan conjuntos de elementos a través de un nombre, lo cual se escribe como

$$< conjunto >$$

y se asocia ese nombre con las diferentes formas que puede tener a través de una cláusula de formato llamada *producción*, la cual se escribe como

$$< conjunto > \rightarrow definición$$

y donde la definición puede contener alternativas separadas por el símbolo |, o sea,

$$< conjunto > \rightarrow alternativa_1 \mid alternativa_2 \mid \ldots \mid alternativa_n$$

y los símbolos en `negrita` determinan elementos finales.

Por ejemplo, la siguiente definición

$< conj >$ → $< elem_1 >$ `->` $< elem_2 >$
$< elem >$ → `X` | `Y`

determina que los elementos del conjunto $< elem >$ son `X` e `Y`, y los del conjunto $< conj >$ son `X->X`, `X->Y`, `Y->X` e `Y->Y`. Observar que $< elem_1 >$ adopta todas las posibles formas de un elemento del conjunto $< elem >$, y lo mismo para $< elem_2 >$, y que poner dos elementos juntos simplemente los coloca uno a continuación del otro.

La notación permite además elementos opcionales, escrito como

$$[< elemOpcional >]$$

y eliminación de los elementos de un conjunto de otro conjunto dado, escrito como

$$< conjBase > / < conjAEliminar >$$

Por ejemplo, modificando la definición anterior a

$< conj >$ → $< elem_1 >$ [`->`$< elem_2 >$]
$< elem >$ → `X` | `Y`

los elementos de $< conj >$ serían los de antes, más `X` e `Y`, pues el símbolo `->` y el siguiente elemento se anotaron como opcionales. Si en cambio se definiese

$< ej >$ → $< conj > / < idem >$
$< conj >$ → $< elem_1 >$ `->` $< elem_2 >$
$< elem >$ → `X` | `Y`
$< idem >$ → `X -> X`

los elementos del conjunto $< ej >$ serían `X->Y`, `Y->X` e `Y->Y`. Observar que `X->X` está en el conjunto $< conj >$ pero no en el conjunto $< ej >$.

Con esta notación, podemos escribir la forma de un programa Gobstones, lo cual se lleva a cabo en las siguientes subsecciones.

A.1. Programas Gobstones

Un programa Gobstones es un elemento del conjunto <*gobstones*>. Cada programa está conformado por una lista de definiciones de procedimientos o funciones, la última de las cuales es la definición del procedimiento `Main`.

<*gobstones*> → <*defs*>

<*defs*> → <*maindef*> | <*def*><*defs*>
<*def*> → `procedure` <*procName*> <*params*> <*procBody*>
| `function` <*funcName*> <*params*> <*funBody*>
<*maindef*> → `procedure Main()` <*mainBody*>

<*params*> → <*varTuple*>

El cuerpo de un procedimiento es una lista de comandos encerrados entre llaves. El cuerpo de una función es similar, excepto que termina con el comando `return`. Finalmente, el cuerpo de `Main` tiene un `return` opcional, pero sólo de variables.

<*procBody*> → { <*cmds*> }
<*funcBody*> → { <*cmds*> `return` <*gexpTuple1*>[`;`] }
<*mainBody*> → { <*cmds*> [`return` <*varTuple*>[`;`]] }

Los comandos se definen en la siguiente sección (A.2), las tuplas de expresiones y variables, en la sección A.4, y los nombres de funciones y procedimientos también en la sección A.4.

A.2. Comandos

Los comandos pueden ser simples o compuestos, y pueden estar agrupados en bloques.

<*blockcmd*> → { <*cmds*> }
<*cmds*> → [<*necmds*>[`;`]]
<*necmds*> → <*cmd*> | <*cmd*>[`;`] <*necmds*>
<*cmd*> → <*simplecmd*> | <*compcmd*>

Los comandos simples son los comandos básicos del cabezal, la invocación de procedimientos y la asignación (de variables, y de resultados de llamados a función).

<*simplecmd*> → `Skip`
| `BOOM(`<*string*>`)`
| `Poner(`<*gexp*>`)`
| `Sacar(`<*gexp*>`)`
| `Mover(`<*gexp*>`)`
| `IrAlOrigen()`
| `VaciarTablero()`
| <*procCall*>
| <*varName*> `:=` <*gexp*>
| <*varTuple1*> `:=` <*funcCall*>
<*procCall*> → <*procName*> <*args*>

Las invocaciones a función y los argumentos para las invocaciones se describen en la sección A.3, y los nombres de variables en la sección A.4.

Los comandos compuestos son las alternativas (condicional e indexada), las repeticiones (condicional e indexada) y los bloques.

<*compcmd*>	→	`if (`<*gexp*>`)` <*blockcmd*> `else` <*blockcmd*>
	|	`if (`<*gexp*>`)` <*blockcmd*>
	|	`case (`<*gexp*>`) of` <*branches*>
	|	`while (`<*gexp*>`)` <*blockcmd*>
	|	`repeatWith` <*varName*> `in` <*range*> <*blockcmd*>
	|	<*blockcmd*>
<*range*>	→	<*gexp*>`..`<*gexp*>
<*branches*>	→	`_ ->` <*blockcmd*>
	|	<*lits*> `->` <*blockcmd*>[`;`] <*branches*>
<*lits*>	|	<*literal*> | <*literal*>`,` <*lits*>

El conjunto de literales <*literal*> se define en la sección A.3.

Observar que las condiciones de las alternativas y de la repetición condicional deben ir obligatoriamente entre paréntesis.

A.3. Expresiones

La expresiones se obtienen combinando ciertas formas básicas en distintos niveles. El nivel básico tiene las variables, las expresiones atómicas para indicarle al cabezal que cense del tablero, los literales, y las invocaciones de función y primitivas. Sobre ese nivel se construyen las expresiones aritméticas (sumas, productos, etc.) con la precedencia habitual. Sobre el nivel aritmético se construyen las expresiones relacionales (comparación entre números y otros literales) y sobre ellas, las expresiones booleanas (negación, conjunción y disyunción) también con la precedencia habitual.

$$
\begin{array}{lcll}
<gexp> & \to & <bexp> & \\
<bexp> & \to & <bterm> \mid <bterm>\texttt{||}<bexp> & \text{(infixr)} \\
<bterm> & \to & <bfact> \mid <bfact>\texttt{\&\&}<bterm> & \text{(infixr)} \\
<bfact> & \to & \texttt{not } <batom> \mid <batom> & \\
<batom> & \to & <nexp> & \\
 & \mid & <nexp><rop><nexp> & \\
<nexp> & \to & <nterm> \mid <nexp><nop><nterm> & \text{(infixl)} \\
<nterm> & \to & <nfactH> \mid <nterm>\texttt{*}<nfactH> & \text{(infixl)} \\
<nfactH> & \to & <nfactL> \mid <nfactL><mop><nfactL> & \\
<nfactL> & \to & <natom> \mid <nfactL>\texttt{\^{}}<natom> & \text{(infixl)} \\
<natom> & \to & <varName> & \\
 & \mid & <liter> & \\
 & \mid & \texttt{-}<natom> & \\
 & \mid & \texttt{nroBolitas(}<gexp>\texttt{)} & \\
 & \mid & \texttt{hayBolitas(}<gexp>\texttt{)} & \\
 & \mid & \texttt{puedeMover(}<gexp>\texttt{)} & \\
 & \mid & \texttt{minBool()} \mid \texttt{maxBool()} & \\
 & \mid & \texttt{minDir()} \mid \texttt{maxDir()} & \\
 & \mid & \texttt{minColor()} \mid \texttt{maxColor()} & \\
 & \mid & \texttt{siguiente(}<gexp>\texttt{)} \mid \texttt{previo(}<gexp>\texttt{)} & \\
 & \mid & \texttt{opuesto(}<gexp>\texttt{)} & \\
 & \mid & <funcCall> & \\
 & \mid & \texttt{(}<gexp>\texttt{)} & \\
 & & & \\
<rop> & \to & \texttt{==} \mid \texttt{/=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} & \\
<nop> & \to & \texttt{+} \mid \texttt{-} & \\
<mop> & \to & \texttt{div} \mid \texttt{mod} & \\
 & & & \\
<funcCall> & \to & <funcName> <args> & \\
<args> & \to & <gexpTuple> &
\end{array}
$$

Las tuplas de expresiones se definen en la sección A.4.

Los literales pueden ser numéricos, booleanos, de color o de dirección.

$$
\begin{array}{lcl}
<literal> & \to & <literN> \mid <literB> \mid <literC> \mid <literD> \\
<literN> & \to & <num> \\
<literB> & \to & \texttt{False} \mid \texttt{True} \\
<literC> & \to & \texttt{Verde} \mid \texttt{Rojo} \mid \texttt{Azul} \mid \texttt{Negro} \\
<literD> & \to & \texttt{Norte} \mid \texttt{Sur} \mid \texttt{Este} \mid \texttt{Oeste}
\end{array}
$$

La forma de los números se definen en la sección A.5

A.4. Definiciones auxiliares

En esta sección se definen diversos conjuntos utilizados como auxiliares en las definiciones previas. Los nombres de variables y de funciones son identificadores que comienzan con minúsculas. Los nombres de los procedimientos son identificadores que empiezan con mayúsculas.

$$
\begin{array}{lcl}
<varName> & \rightarrow & <lowerid> \\
<funcName> & \rightarrow & <lowerid> \\
<procName> & \rightarrow & <upperid>
\end{array}
$$

Las tuplas son listas de elementos encerrados entre paréntesis y separados por comas. Opcionalmente, una tupla puede estar vacía, o sea, no contener ningún elemento.

$$
\begin{array}{lcl}
<varTuple> & \rightarrow & \texttt{()} \mid <varTuple1> \\
<varTuple1> & \rightarrow & \texttt{(}<varNames>\texttt{)} \\
<varNames> & \rightarrow & <varName> \mid <varName>\texttt{,}<varNames>
\end{array}
$$

$$
\begin{array}{lcl}
<gexpTuple> & \rightarrow & \texttt{()} \mid <gexpTuple1> \\
<gexpTuple1> & \rightarrow & \texttt{(}<gexps>\texttt{)} \\
<gexps> & \rightarrow & <gexp> \mid <gexp>\texttt{,}<gexps>
\end{array}
$$

A.5. Definiciones lexicográficas

Las definiciones lexicográficas establecen la forma de las palabras que conforman el lenguaje. Ellas incluyen los números, los identificadores, las palabras reservadas, los operadores reservados y los comentarios.

Los números son simplemente secuencias de dígitos.

$$
\begin{array}{lcl}
<num> & \rightarrow & <digit> \mid <nonzerodigit> <digits> \mid \texttt{-}<num> \\
<digits> & \rightarrow & <digit> \mid <digit> <num> \\
<digit> & \rightarrow & \texttt{0} \mid <nonzerod> \\
<nonzerod> & \rightarrow & \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{array}
$$

Los identificadores son de dos tipos: los que comienzan con minúscula y los que comienzan con mayúscula. El símbolo de tilde (') puede ser parte de un identificador (excepto el primero). Las palabras reservadas no pueden ser identificadores.

$$
\begin{array}{lcl}
<lowerid> & \rightarrow & <lowname>/<reservedid> \\
<lowname> & \rightarrow & <lowchar> \mid <lowchar> <chars> \\
<lowchar> & \rightarrow & \texttt{a} \mid \ldots \mid \texttt{z}
\end{array}
$$

$$
\begin{array}{lcl}
<upperid> & \rightarrow & <uppname>/<reservedid> \\
<uppname> & \rightarrow & <uppchar> \mid <uppchar> <chars> \\
<uppchar> & \rightarrow & \texttt{A} \mid \ldots \mid \texttt{Z}
\end{array}
$$

$$
\begin{array}{lcl}
<chars> & \rightarrow & <char> \mid <char> <chars> \\
<char> & \rightarrow & <lowchar> \mid <uppchar> \mid <digit> \mid \texttt{'} \mid \texttt{\_}
\end{array}
$$

Los strings son secuencias de caracteres distintos de la comilla doble ("), encerrados entre comillas dobles.

$$
<string> \quad \rightarrow \quad \texttt{"} <anySymbols> /\texttt{"}\ \texttt{"}
$$

Algunos caracteres pueden usarse precedidos por una barra (\) llamada escape. Dentro de un string, la barra \\ y las comillas dobles \" siempre deben escaparse. La

categoría de caracteres escapados incluye representaciones portables para los caracteres "alert"(`\a`), "backspace"(`\b`), "form feed" (`\f`), "new line" (`\n`), "carriage return"(`\r`), "horizontal tab" (`\t`), and "vertical tab" (`\v`). Observar que las comillas escapadas no son consideradas comillas, por lo que pueden usarse dentro de un string.

Las palabras y los símbolos reservados son todos aquellos utilizados en algún comando predefinido o como separadores.

<*reservedid*>	→	`if` | `else` | `not` | `True` | `False`
	|	`case` | `of`
	|	`while` | `Skip`
	|	`repeatWith` | `in`
	|	`procedure` | `function` | `return`
	|	`Mover` | `Poner` | `Sacar`
	|	`BOOM`
	|	`IrAlOrigen` | `VaciarTablero`
	|	`div` | `mod`
	|	`hayBolitas` | `nroBolitas` | `puedeMover`
	|	`Norte` | `Sur` | `Este` | `Oeste`
	|	`minBool` | `maxBool`
	|	`minDir` | `maxDir`
	|	`minColor` | `maxColor`
	|	`siguiente` | `previo`
	|	`opuesto`
	|	`Verde` | `Rojo` | `Azul` | `Negro`
<*reservedop*>	→	`:=` | `..`
	|	`,` | `;` | `(` | `)` | `{` | `}`
	|	`||` | `&&` | `+` | `*` | `-` | `^`
	|	`==` | `/=` | `<` | `<=` | `>=` | `>`
	|	`--` | `{-` | `-}` | `//` | `/*` | `*/`

Finalmente, los comentarios son de línea o de párrafo. Los primeros empiezan con uno de los símbolos reservados `--` o `//` y terminan con el fin de línea, y los segundos empiezan con los símbolos reservados `{-` o `/*` y terminan con la primera aparición del símbolo `-}` o `*/` respectivamente.

<*comment*>	→	<*linecomm*> | <*parcomm*>
<*linecomm*>	→	`--` <*anySymbols*>/`\n` `\n`
	|	`//` <*anySymbols*>/`\n` `\n`
<*parcomm*>	→	`{-` <*anySymbols*>/`-}` `-}`
	|	`/*` <*anySymbols*>/`*/` `*/`

B. Instalando la herramienta

La herramienta actual de Gobstones 2.08 es un prototipo muy sencillo, pensado más como una forma de experimentar las ideas, antes que la de servir como ambiente de trabajo cotidiano. Está implementada en el lenguaje funcional Haskell [PJH+99], y prácticamente no ofrece soporte para poco más que la carga y ejecución de un programa Gobstones.

En su versión actual, el programa se compone de una serie de scripts, que pueden utilizarse desde un intérprete de Haskell. Futuras versiones podrían brindarse como una versión compilada de los mismos, en un archivo ejecutable. Por esta razón, antes de poder ejecutar Gobstones es importante instalar un intérprete de Haskell. Detallaremos aquí los pasos necesarios para correr Gobstones usando Hugs [JP99], pero otras alternativas son posibles (por ejemplo, usar ghci [GHC07]).

El primer paso es obtener los scripts que conforman el Gobstones 2.0. Los mismos se encuentran en un archivo zip que puede bajarse del sitio de la materia. La dirección del sitio es `http://sites.google.com/site/introprog2008c2/`, y allí debe entrarse en el link "`Repositorio de archivos`", donde se encuentra `Gobstones.2.0.zip` (ver figura 5).

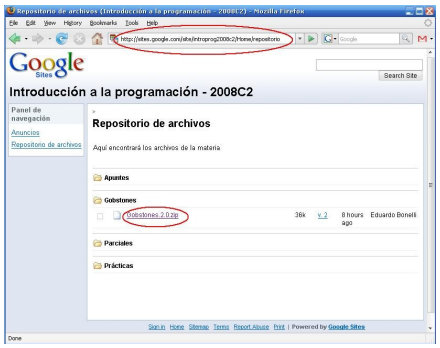


Figura 5: Bajando Gobstones 2.0

El archivo `Gobstones.2.0.zip` debe descomprimirse en un directorio. Allí encontrarán diversos scripts de Haskell que conforman la implementación de Gobstones.

Un programa Gobstones debe generarse un archivo de texto sin formato en el directorio `examples` que viene junto con los scripts de Gobstones. Si lo que se desea es crear un programa para resolver el ejercicio 1, debe crearse un archivo de nombre `ejerc1.gbs` en el directorio mencionado (ver figura 6) Con un editor de textos cualquiera se edita

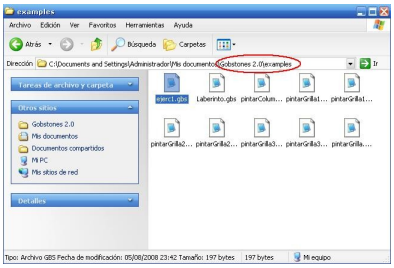


Figura 6: Creando un archivo para el programa `ejerc1.gbs`

el archivo en cuestión, escribiendo el programa, y grabándolo.

Los restantes pasos dependen del sistema operativo que se esté utilizando.

B.1. Usuarios de Windows

Para los usuarios de Windows, el siguiente paso es conseguir e instalar una versión de Hugs. Ello se hace en la URL `http://haskell.org/hugs`, en la sección "Downloading", como se puede observar en la figura 7. Puede elegirse cualquier versión de Hugs. Se

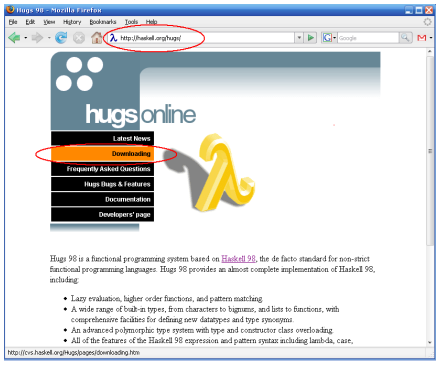


Figura 7: Página de Hugs, un sistema de programación funcional basado en Haskell

recomienda, por la relación tamaño/prestaciones que se instale la versión que contiene el número mínimo de bibliotecas adicionales. La misma se llama `MinHugs`, y actualmente va por la versión `Sep2006`. En la figura 8 se indica dónde se puede encontrar la misma. Una

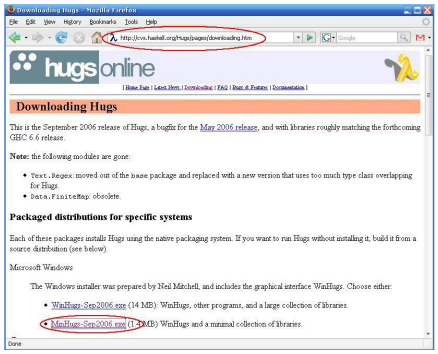


Figura 8: Bajando MinHugs para su instalación

vez obtenido el programa `MinHugs-Sep2006.exe`, debe ejecutárselo para proceder con la instalación. Alcanza con realizar la instalación con todas las opciones predeterminadas (ver figura 9).

Hecho esto ya se está en condiciones de probar el programa con la herramienta. Para cargar la herramienta debe abrirse el Hugs con el script `Main.hs` que se encuentra en el directorio de Gobstones. Ello se puede hacer utilizando el botón derecho sobre el archivo `Main.hs` y eligiendo la opción "`Abrir con`" (figura 10), de donde debe seleccionarse *WinHugs*. Alternativamente, en la mayoría de las configuraciones de Windows se puede hacer doble click directamente sobre el archivo `Main.hs`.

El último paso, una vez cargado el intérprete de Hugs, es invocar la ejecución del programa Gobstones a través de la evaluación de la expresión `ejecutar .ejerc1"`, como se observa en la figura 11.

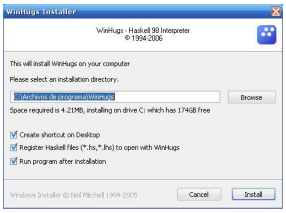


Figura 9: Instalando Hugs

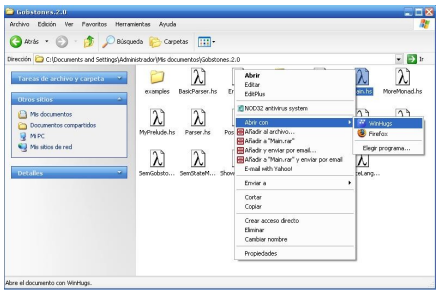


Figura 10: Abriendo el Hugs con `Main.hs` de Gobstones

Para más detalles sobre variantes para ejecutar programas Gobstones con esta herramienta, consultar la sección B.3.

B.2. Usuarios de Ubuntu

Las distribuciones de Linux normalmente ya traen el Hugs instalado. Si esto no es así, debe buscarse el paquete correspondiente uilizando el administrador de paquetes (se precisa permiso de administrador para esto) e instalarlo.

Una vez hecho esto, se debe abrir una terminal, ubicarse en el directorio de Gobstones como el directorio corriente, y tipear el comando `hugs Main`. Una vez abierto el Hugs con el `Main` de Gobstones, se procede a ejecutar el archivo de la misma manera que en Windows (o sea, se tipea `ejecutar "archivo"`, donde `archivo.gbs` es un archivo ubicado en el directorio `examples`).

B.3. Usando la herramienta

Para probar programas deben crearse archivos de extensión `.gbs` en el directorio `examples`, y evaluar la expresión `ejecutar "nombre-de-archivo"` dentro del Hugs. Observar que el nombre del archivo se coloca entre comillas y sin la extensión `.gbs`. Observar que, a partir de la versión 2.10, la función `ejecutar` inicia el programa sobre un tablero aleatorio, tanto en el contenido de las celdas como en la posición inicial del cabezal.

B.3.1. Variantes para la ejecución de un programa

A partir de la versión 2.10 pueden utilizarse variantes de la función `ejecutar`. Las mismas se detallan en la siguiente lista:

- La primera variante es

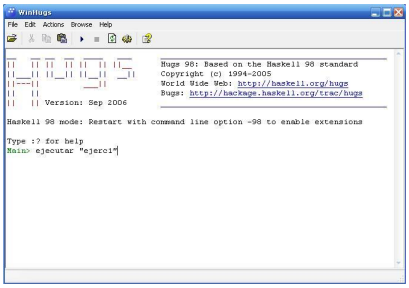


Figura 11: Ejecutando un programa Gobstones

`ejecutarA "nombre-programa" "nombre-tablero-salida"`.

Esta versión es similar a `ejecutar`, con el agregado que graba el tablero resultante en un archivo de extensión `.gbt` con el nombre dado. Debe tenerse en cuenta que si el archivo existe será sobreescrito. Se mantiene la aleatoriedad del tablero inicial.

- La segunda variante es

  `ejecutarEn "nombre-programa" "nombre-tablero-entrada"`.

  Esta versión es similar a `ejecutar`, pero en lugar de iniciar con un tablero aleatorio, lee el tablero contenido en el archivo de extensión `.gbt` con el nombre dado. Debe tenerse en cuenta que si el archivo no existe o no contiene una codificación válida de un tablero, se manifestarán diversos errores.

- La última variante es

  `ejecutarEnA "programa" "tablero-entrada" "tablero-salida"`.

  Esta versión combina las dos anteriores.

Utilizando estas variantes es posible lograr una flexibilidad mucho mayor que la que es posible en Gobstones 2.07 y anteriores.

B.3.2. Sobre los archivos de especificación de tableros

Los archivos de extensión `.gbt` codifican al tablero. Son archivos posicionales, de manera que una celda queda determinada por cada uno de los caracteres dentro del espacio delimitado de la siguiente manera:

```
+---------+
|         |
|         |
|         |
+---------+
```

La cantidad de celdas debe ser fija, lo cual determina el tamaño y la cantidad de líneas de un archivo `.gbt`. Alteraciones en estas cantidades arrojarán errores de lectura.

Para especificar cuántas bolitas de cada color hay dentro de una celda, deben usarse combinaciones de números y letras que determinen un color. Las letras posibles son cuatro: `N`, `A`, `V` y `R`, tanto mayúsculas como minúsculas (a partir de la versión 2.12); cualquier otra letra arrojará un error de lectura. Además, el número debe preceder a la letra, y no hace falta que haya blancos. Por ejemplo, una celda con 99 bolitas de cada color, se escribiría de la siguiente manera:

```
+---------+
| 99N 99A |
|         |
| 99V 99R |
+---------+
```

Además, cualquier secuencia que alterne números con letras identificadoras de color será válida. Entonces, otra forma de especificar la celda anterior sería:

```
+---------+
| 90N 19A |
|9N 99R 7V|
| 92V 80A |
+---------+
```

(observar que hay dos alternativas para Negro, dos para Azul y dos para Verde, y en cada caso, ambos números suman 99).

En un archivo `.gbt` también se puede especificar la posición incial del cabezal. Para especificarlo, debe rodearse la celda actual con caracteres `X`, de la siguiente manera

```
+---------+XXXXXXXXX+---------+
|         X         X         |
|         X         X         |
|         X         X         |
+---------+XXXXXXXXX+---------+
```

De no especificarse la celda actual, la posición de la misma será determinada aleatoriamente.

Finalmente, cada línea debe encerrarse entre comillas (`"`) y todas menos la última deben terminarse con coma (`,`). Además las líneas deben estar precedidas por un corchete izquierdo (`[`) y seguidas de un corchete derecho (`]`) para poder ser leídas correctamente. Además, la cantidad de líneas y el ancho de las mismas es prefijado. Los archivos `.gbt` generados por la herramienta conforman todas las reglas necesarias para ser leídos correctamente. Se puede ver un tablero completo en la figura 12, y también encontrar ejemplos de tableros predeterminados en el directorio `examples`.

```
[
"        0         1         2         3         4         5         6         7         8        ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  8 |         |         |         |         |         |         |         |         |         | 8  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  7 |         |         |         |         |         |         |         |         |         | 7  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  6 |         |         |         |         |         |         |         |         |         | 6  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  5 |         |         |         |         |         |         |         |         |         | 5  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  4 |         |         |         |         |         |         |         |         |         | 4  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  3 |         |         |         |         |         |         |         |         |         | 3  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  2 |         |         |         |         |         |         |         |         |         | 2  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  1 |         |         |         |         |         |         |         |         |         | 1  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"    |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |  1N  1A |    ",
"  0 |         |         |         |         |         |         |         |         |         | 0  ",
"    |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |  1V  1R |    ",
"    +---------+---------+---------+---------+---------+---------+---------+---------+---------+    ",
"        0         1         2         3         4         5         6         7         8        "
]
```

Figura 12: Ejemplo de un archivo `.gbt` con un tablero completo.

Referencias

[GHC07] GHC Team. The glorious Glasgow Haskell Compilation system user's guide. Available on-line: `http://www.haskell.org/ghc/`, 2007.

[JP99] Mark P. Jones and John Peterson. The Hugs 98 user manual. Available on-line: `http://www.haskell.org/hugs/`, 1999.

[PJH+99] S. Peyton Jones, J. Hughes, et al. Haskell 98: A non-strict, purely functional language. Available on-line: `http://www.haskell.org/onlinereport/`, February 1999.