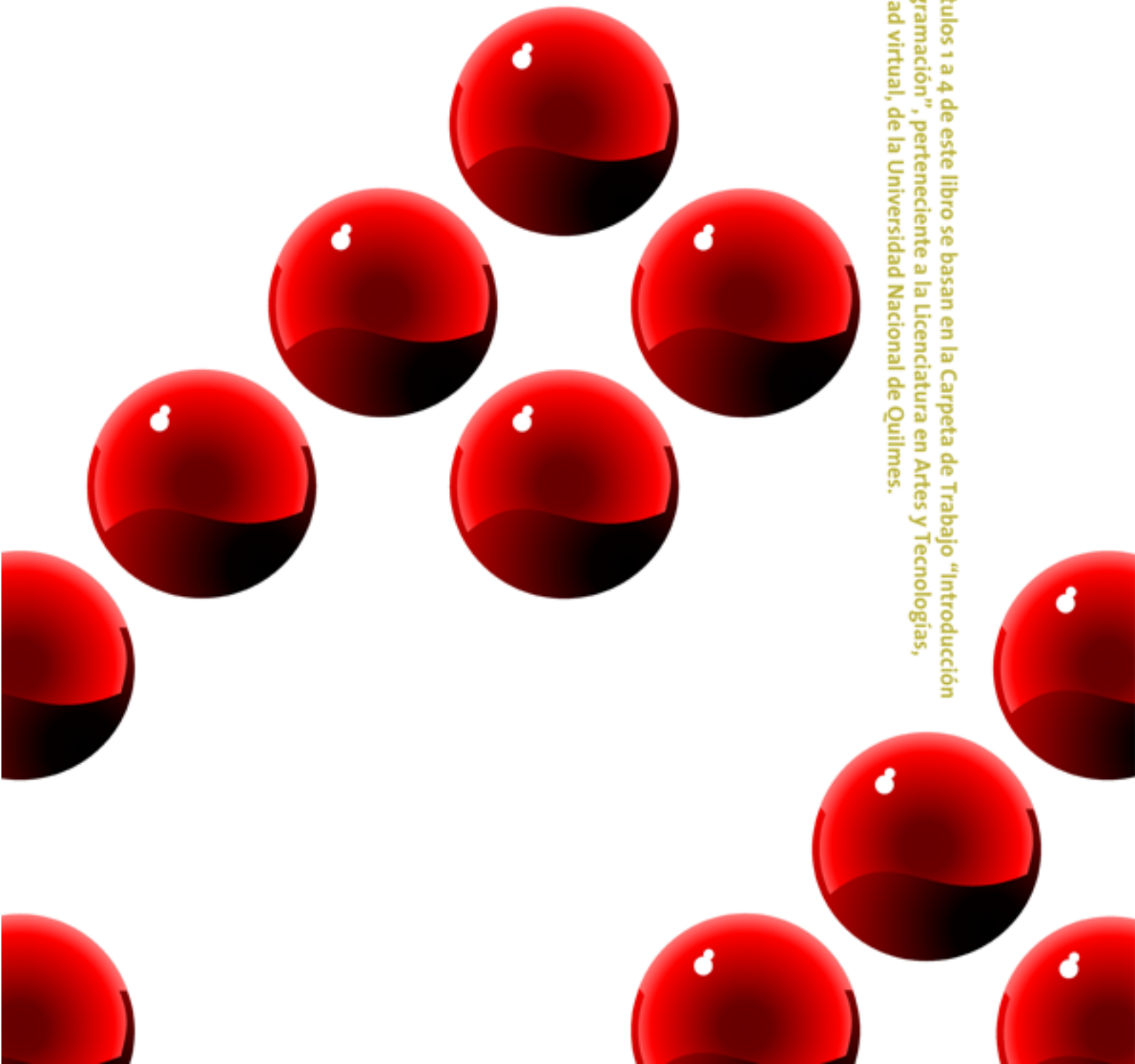
Las bases conceptuales de la Programación

Una nueva forma de aprender a programar

Pablo E. Martínez López

Los capítulos 1 a 4 de este libro se basan en la Carpeta de Trabajo "Introducción a la Programación", perteneciente a la Licenciatura en Artes y Tecnologías, modalidad virtual, de la Universidad Nacional de Quilmes.



Las bases conceptuales de la Programación

Una nueva forma de aprender a programar

Pablo E. "Fidel" Martínez López

Los capítulos 1 a 4 de este libro se basan en la Carpeta de Trabajo *"Introducción a la Programación"*, perteneciente a la Licenciatura en Artes y Tecnologías, modalidad virtual, de la Universidad Nacional de Quilmes.

Versión digital
Revisión del 10 de febrero de 2014

Martínez López, Pablo E.
Las bases conceptuales de la Programación: Una nueva forma de aprender a programar / - 1ra ed. - La Plata : el autor, 2013.
EBook.

ISBN 978-987-33-4081-9

1.Programación. 2.Didáctica. 3.Aprendizaje. I.Título
CDD 005.3.

Primera edición: diciembre 2013
ISBN: 978-987-33-4081-9

La obra está disponible para su descarga en:
http://www.gobstones.org/bibliografia/Libros/BasesConceptualesProg.pdf

Los capítulos 1 a 4 y el diseño de maqueta de esta obra se basan en la Carpeta de Trabajo “Introducción a la Programación para la carrera de Licenciatura en Artes y Tecnologías”, ISBN: 978-987-1856-39-8, publicado electrónicamente por la Universidad Virtual de Quilmes. Los créditos de dicha carpeta corresponden a las siguientes personas
Procesamiento didáctico: Bruno De Angelis, Ana Elbert
Diseño original de maqueta: Hernán Morfese, Marcelo Aceituno y Juan I. Siwak
Diagramación: Pablo E. Martínez López (con LaTeX)

Para Lautaro y Aylen, que soportaron el proceso y quizás vean sus frutos.
Y para mis viejos, que siempre lo soñaron y se habrían babeado de gusto.

Agradecimientos

Este libro no habría sido posible sin el trabajo de muchísima gente.

En primer lugar, Eduardo Bonelli, con quién venimos compartiendo muchísimas cosas desde hace mucho tiempo, y que fue el inspirador de GOBSTONES, además de su coautor. Con él discutimos los fundamentos conceptuales, seleccionamos los contenidos del curso que luego se plasmaron en GOBSTONES, fuimos mejorando juntos tanto el lenguaje como la herramienta. Y hoy día continuamos discutiendo sobre GOBSTONES.

En segundo lugar, Francisco Soulignac. Él se incorporó al equipo y rápidamente se apropió de GOBSTONES y contribuyó enormemente con su desarrollo. Francisco contribuyó especialmente con el desarrollo de XGOBSTONES, que si bien no está incluído en este libro, es parte de esta secuencia didáctica.

Luego quiero agradecer a Federico Sawady, quién aprendió con GOBSTONES y luego se incorporó al equipo de desarrollo. Con él discutimos sobre fundamentos, sobre posibles caminos, y él fue quién primero probó esta secuencia didáctica en la secundaria, contribuyendo a su mejoramiento. Además con Federico escribimos juntos "Introducción a la Programación para la Licenciatura en Artes y Tecnologías" en el cual se basa el presente libro.

El último de los agradecimientos personales es para Gabriel Baum. Gabriel fue mi mentor en la Universidad, y mi guía, y él fue quién puso la semilla que germinó en mi interés por la enseñanza de la programación. En las discusiones con él se cimentan muchas de las cosas buenas de este libro, y en su incansable búsqueda por mejorar la computación para el país encontré siempre una fuente de inspiración y de modelo.

Una vez Gabriel me dijo "Fidel, no sabemos enseñar a programar porque no sabemos enseñar a abstraer. Y no sabemos enseñar a abstraer porque los informáticos tenemos poca idea de qué es la abstracción realmente." Esa frase fue siempre una zanahoria que me motivó a continuar mi búsqueda.

También debo agradecer a todos los involucrados en la materia "Introducción a la Programación" de la Tecnicatura en Programación Informática de la UNQ. Ellos le dieron vida a GOBSTONES en el aula, y aportaron ejercicios, discusiones, puntos de vista, y muchas otras cosas, además de un ambiente ameno de trabajo.

Otro agradecimiento importante va para todos aquellos que hicieron lecturas del material preliminar y fueron sugiriendo mejoras, errores y omisiones, y que contribuyeron de esta manera a que el presente libro sea mucho mejor. Sepan todos ustedes que los voy a seguir necesitando, porque los errores no paran de aparecer, y siempre hay lugar para introducir mejoras.

En el plano institucional, no puedo dejar de agradecer a la Universidad Nacional de Quilmes, que proveyó todo lo necesario para que esto pudiera florecer. El ambiente de trabajo estimulante y ameno, la infraestructura adecuada, los recursos necesarios son todos méritos que deberían ser más comunes, y que son valiosísimos.

También en el plano institucional quiero agradecer a la Escuela Florentino Ameghino de Florencio Varela, que apostó por GOBSTONES y esta secuencia didáctica desde el principio.

Finalmente quiero agradecer a mi familia: a mis viejos que ya no están pero que pusieron los cimientos y aportaron siempre a mi formación, a mis hijos que son la luz de mi vida y que sufrieron el proceso de desarrollo, incluso como conejitos de indias, y a Julia, que también sufrió el proceso como conejito de indias, y que me banca y me acompaña en la aventura de la vida juntos más allá de los altibajos.

Íconos

Leer con Atención. Son afirmaciones, conceptos o definiciones destacadas y sustanciales que aportan claves para la comprensión del tema que se desarrolla.

Para Reflexionar. Propone un diálogo con el material a través de preguntas, planteamiento de problemas, confrontaciones del tema con la realidad, ejemplos o cuestionamientos que alienten a la reflexión.

Pastilla. Incorpora informaciones breves, complementarias o aclaratorias de algún término o frase del texto principal. El subrayado indica los términos a propósito de los cuales se incluye esa información asociada en el margen.

Cita. Se diferencia de la palabra del autor de la Carpeta a través de la inserción de comillas, para indicar claramente que se trata de otra voz que ingresa al texto.

Para Ampliar. Extiende la explicación a distintos casos o textos como podrían ser los periodísticos o de otras fuentes.

Actividades. Son ejercicios, investigaciones, encuestas, elaboración de cuadros, gráficos, resolución de guías de estudio, etcétera.

Recurso Web. Liks a sitios o páginas web que resulten una referencia dentro del campo disciplinario.

El autor

Pablo E. "Fidel" Martínez López

Pablo E. "Fidel" Martínez López se recibió de Doctor en Ciencias de la Computación en la UBA en noviembre 2005, y antes de eso de Magíster en Ciencias de la Computación en la Universidad de la República, Uruguay y de Licenciado en Informática en la UNLP y la ESLAI. Ejerce la docencia universitaria desde 1990, y desde 2007 es Profesor Asociado con Dedicación Exclusiva en la Universidad Nacional de Quilmes, habiendo trabajado como docente en UNLP, UBA, UNR, UNRC y UNLM. Posee la categoría 2 en el programa de incentivos a la investigación del gobierno nacional. Sus áreas de interés científico son los Lenguajes de Programación, especialmente Programación Funcional, Estructuras de Datos, la producción automática de programas, la Teoría de la Computación y los Lenguajes Formales y Autómatas, y recientemente la didáctica de la enseñanza de Programación. Ha participado en diversos proyectos de investigación desde 1993, habiendo dirigido incluso un proyecto de cooperación internacional en 2008, y varios proyectos de extensión y transferencia desde 2007. Es autor de numerosos artículos científicos nacionales e internacionales, de un capítulo de libro, y coautor del libro "Introducción a la Programación para la carrera de Licenciatura en Artes y Tecnologías" de la UNQ, que es precursor del presente libro. Ha formado recursos humanos en investigación en diversos niveles, y ha participado en tareas de gestión universitaria y transferencia de conocimientos, desempeñándose al momento de esta publicación como Director de Carrera de la Tecnicatura Universitaria en Programación Informática del Departamento de Ciencia y Tecnología de la Universidad Nacional de Quilmes (UNQ) y Director de la Unidad de Vinculación Tecnológica UTICs (Unidad de Tecnologías de la Información y la Comunicación) dependiente de la Dirección de Vinculación Tecnológica de la misma Universidad. En ambos cargos se desempeña desde el 2do semestre de 2007. En relación a la temática de este libro, dictó la materia *Introducción a la Programación* de la carrera *Tecnicatura en Programación Informática* de la Universidad Nacional de Quilmes entre 2008 y 2010, desarrollando junto al Dr. Eduardo Bonelli la propuesta didáctica aquí presentada y las herramientas asociadas. Posteriormente dirigió los grupos docentes que continuaron con el desarrollo de la propuesta y las herramientas y estuvo a cargo de varias capacitaciones a docentes con esta propuesta.

Índice general

Prefacio

La programación es una disciplina que en pocos años ha cobrado una relevancia fundamental en gran diversidad de ámbitos de la cultura y la sociedad humana. Hoy día es deseable que todas las personas tengan un mínimo de conocimientos relacionados con la programación, ya que la programación favorece el pensamiento algorítmico. Además, si consideramos a la programación desde un enfoque adecuado, abstracto, se adquiere una conciencia del uso de elementos de abstracción que resultan fundamentales para muchas otras actividades del conocimiento. La abstracción es una herramienta esencial del pensamiento humano, y la programación provee formas de explicitar el proceso de abstracción y de controlarlo de diversas maneras, orientando a una forma de conceptualizar los problemas que hacen mucho más simple el entender problemas y encontrarles solución, tal cual lo indicó Edsger Dijkstra en 1989 [Dijkstra and others, 1989].

Este libro busca ser una introducción amena para personas con poca o ninguna experiencia en temáticas vinculadas al desarrollo de software. Para ello ofrece una visión panorámica de los temas básicos, comenzando por la historia de la programación, y continuando con abstracciones básicas que permiten modelar programas. Este libro se basa principalmente de los primeros 4 capítulos del Cuaderno de Trabajo "Introducción a la Programación para la carrera de Licenciatura en Artes y Tecnologías" [Martínez López and Sawady O'Connor, 2013] del autor y Federico Sawady O'Connor, y presenta un enfoque nuevo para la enseñanza de la programación, guiado por la necesidad de focalizar el aprendizaje en el proceso de abstracción, y en los conceptos fundamentales, transversales a todos los paradigmas y lenguajes. La secuencia didáctica que guía este enfoque fue desarrollada por el autor y su colega Eduardo Bonelli durante el dictado de la materia Introducción a la Programación de la carrera Tecnicatura en Programación Informática de la UNQ, entre los años 2008 a 2010. Las bases conceptuales del enfoque se discuten en el artículo *"El nombre verdadero de la programación. Una concepción de la enseñanza de la programación para la sociedad de la información"* [Martínez López *et al.*, 2012]. Este enfoque ha sido utilizado con éxito desde 2010 en la mencionada carrera, y también se ha comenzado a utilizar en algunas escuelas secundarias. La secuencia didáctica específica se presenta en la próxima sección.

En el momento de la edición de este libro estamos escribiendo una versión más completa con título tentativo *"Introducción a la Programación. Una didáctica innovadora"*, y que completará muchísimo el material presente aquí, pero por razones de necesidad en la implementación de cursos masivos en escuelas secundarias se hace necesario contar con una versión incial que pueda salir a prensa antes.

Presentamos la programación de una manera amena y sencilla procurando brindar los conocimientos de los fundamentos básicos de la misma. Sin embargo, no por ello incurrimos en el defecto de sobresimplificar o infantilizar la programación a través de metáforas u otros recursos limitantes, sino que buscamos mantener una visión precisa, científica, aunque sin incurrir en detalles técnicos innecesarios. Articulamos la presentación alrededor del concepto de abstracción, idea vertebral a la actividad misma de programar. La misma noción de programación es una tarea abstracta, y que requiere de conceptualizaciones y representacio-

nes abstractas de la información y los procesos. Asimismo, los lenguajes de programación pueden ser vistos como herramientas de abstracción, y los elementos que en ellos aparecen se pueden comprender en función del tipo de abstracción que proveen. Los capítulos se centran en las formas de escribir programas, omitiendo adrede el tratamiento de temas más complejos (como las estructuras de datos, o la algorítmica), que, aunque fundamentales para la programación, pueden abordarse con posterioridad.

El material de este libro está pensado para que sirva no solo como referencia y guía del aprendizaje de la disciplina, sino también de material de consulta para las definiciones elementales y las ideas que son pilar de la fascinante disciplina de la programación.

En educación es clave el protagonismo y compromiso individual de cada estudiante, aunque muchas veces esto no sea adecuadamente puesto en foco. En el caso de la enseñanza de la programación, donde la práctica constante y sostenida es el único camino para aprender los conceptos fundamentales, este compromiso y protagonismo debe remarcarse aún más. Parafraseando un viejo dicho acerca de la matemática, podemos decir que “la programación no es un deporte para espectadores” (en inglés *computer science is not a spectator sport*), lo que viene a querer decir que para poder aprender es necesario involucrarse y ejercitar, buscar varias alternativas de solución al mismo problema, ampliar la información, y probar en forma práctica en las computadoras los programas realizados. En ese proceso este libro actúa como guía, y como consejero, pero el factor de éxito está en el estudiante mismo. No dejes de tener esto en cuenta eso al leer este libro para aprender, o al utilizar este libro para enseñar.

El libro se organiza en 6 capítulos. En el primero comenzamos presentando la idea de programa y lenguaje de programación, y haremos una revisión de la historia de los lenguajes de programación desde su surgimiento moderno, para comprender el rol que los lenguajes juegan en la construcción del mundo actual. En el segundo hacemos la primera aproximación a un primer lenguaje de programación, presentando sus elementos básicos (expresiones y comandos) y organizándolos para comenzar a resolver problemas sencillos. Elegimos el lenguaje GOBSTONES, desarrollado específicamente en la UNQ para la enseñanza de un curso inicial. En el tercero continuamos presentando elementos de lenguajes de programación en GOBSTONES, incorporando herramientas (funciones, procedimientos y parametrización) para manejar la complejidad y simplificar los programas resultantes. El cuarto completa la presentación del lenguaje GOBSTONES con una explicación de los elementos más complejos del mismo (estructuras de control para alternativa y repetición, y manejo de memoria elemental), y propone algunos ejercicios avanzados. El capítulo 5 completa la presentación mostrando la integración de los conceptos explicados en el libro mediante el desarrollo de una aplicación, para la que se explica su diseño y su codificación (el código fuente completo se presenta en el anexo B). Finalmente concluimos en el último con una discusión sobre cómo creemos los autores que se debe continuar profundizando el aprendizaje de la programación, a través del abordaje de temas fundamentales que no incluímos por tratarse de un enfoque inicial.

Secuencia didáctica y mapa conceptual

Como mencionamos antes, en este libro proponemos una secuencia didáctica novedosa, diferente a la tradicional forma de enseñar programación. Esta secuencia didáctica se diseñó teniendo en cuenta el trabajo con los estudiantes de primer año de la Tecnicatura en Programación Informática de la Universidad Nacional de Quilmes, y expresa una síntesis conceptual utilizable con personas que no tienen ninguna experiencia previa en programación, ni demasiada familiaridad con procesos de abstracción (típicamente presentados en cursos de matemáticas). La secuencia didáctica articula una serie de conceptos de diferentes categorías y niveles. En primer lugar brindamos una categorización de los con-

ceptos, para ofrecer un marco donde entender mejor la secuencia didáctica, y luego presentamos la secuencia didáctica propiamente dicha.

1) Elementos del lenguaje
   a) Universo de discurso (tablero, bolitas, cabezal)
   b) Programas
   c) Comandos
      - Comandos primitivos (`Poner`, `Mover`, `Sacar`, etc.)
      - Secuencia de comandos
      - Bloques
      - Invocación de procedimientos simples
      - Invocación de procedimientos con argumentos
      - Repetición simple
      - Repetición indexada
      - Alternativa condicional
      - Alternativa indexada
      - Repetición condicional
      - Asignación
   d) Expresiones
      - Expresiones literales
      - Uso de parámetros
      - Operaciones primitivas de expresiones (por tipo)
      - Uso de índices
      - Invocación de funciones simples
      - Invocación de funciones con argumentos
      - Uso de variables
   e) Definiciones
      - Procedimientos simples
      - Procedimientos con parámetros
      - Funciones simples
      - Funciones con parámetros
      - Funciones con procesamiento
   f) Mecanismos de nombrado
      - Parámetros
      - Índices
      - Variables
2) Elementos de abstracción
   a) Elementos de “contrato”
      - Propósito de una operación
      - Precondiciones de una operación
      - Requisitos a los parámetros
   b) División en subtareas
   c) Tipos de datos
   d) Esquemas de recorrido
3) Elementos de estilo
   a) Indentación
   b) Comentarios

c) Elección de nombres

4) Otros conceptos

a) Ejecución de programas
b) Mensajes de error
c) Biblioteca de operaciones

Los diferentes conceptos de esta categorización se van presentando en la secuencia didáctica como resultado de alguna necesidad surgida desde la práctica de la programación. O sea, para cada concepto se presenta un problema que resulta difícil o imposible de resolver con los elementos previos, y así se justifica la necesidad de dicho concepto. De allí que el orden de los conceptos en la secuencia didáctica sea extremadamente relevante, y que los ejercicios del libro deban realizarse en orden, con los elementos presentados hasta el punto donde el ejercicio aparece. La secuencia didáctica específica es la siguiente

Cap. 2

1. Universo de discurso (tablero, bolitas, cabezal)
2. Programas
3. Comandos
4. Comandos primitivos (`Poner`)
5. Expresiones
6. Expresiones literales
7. Secuencia de comandos
8. Bloques
9. Ejecución de programas
10. Mensajes de error
11. Más comandos primitivos (`Mover`, `Sacar`, etc.)
12. Procedimientos simples
13. Invocación de procedimientos simples
14. División en subtareas
15. Biblioteca de operaciones
16. Elementos de estilo
17. Indentación
18. Elección de nombres
19. Comentarios
20. Elementos de “contrato”
21. Propósito de una operación
22. Precondiciones de una operación

Cap. 3

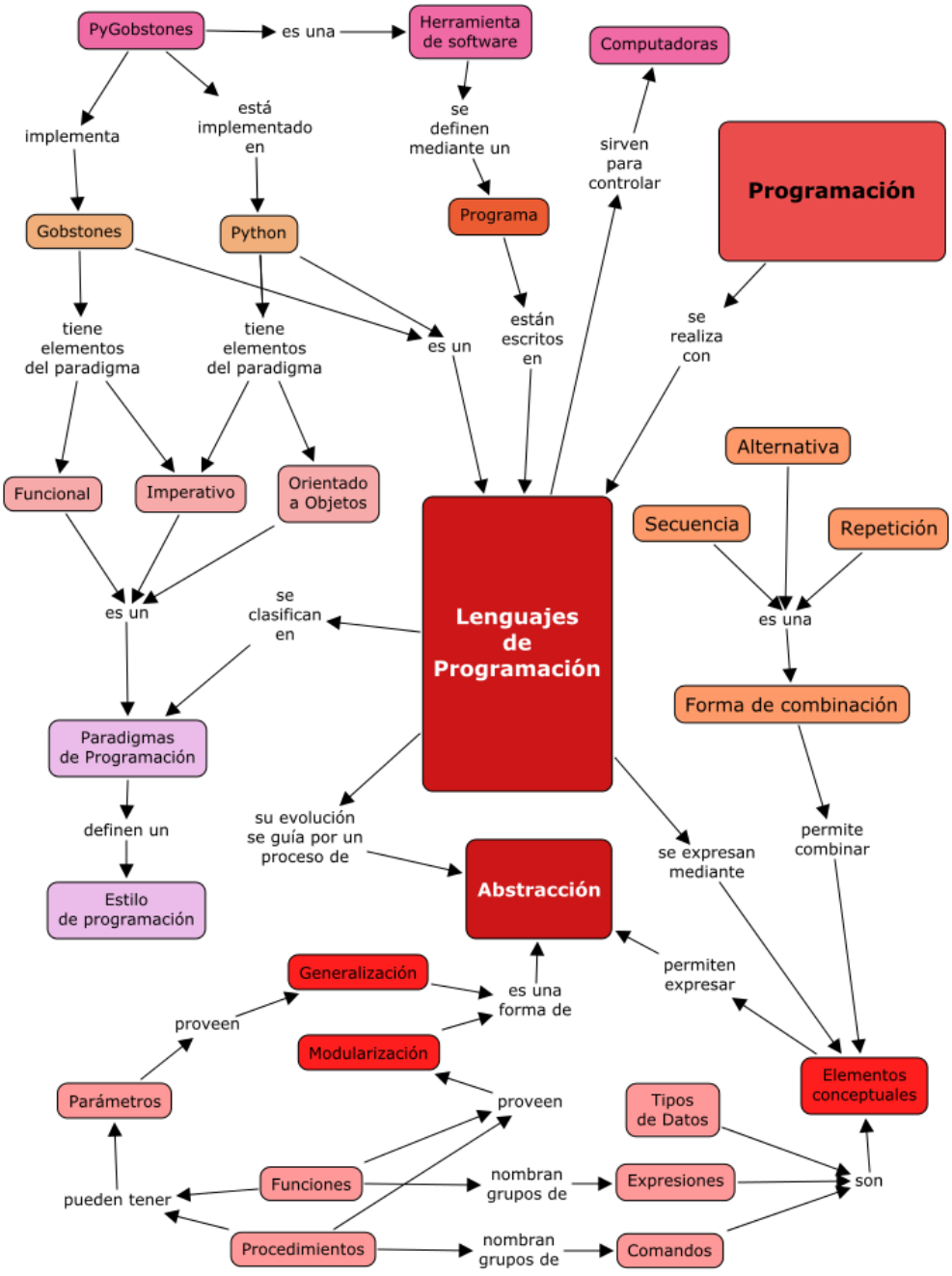
1. Parámetros
2. Procedimientos con parámetros
3. Invocación de procedimientos con argumentos
4. Uso de parámetros
5. Requisitos a los parámetros
6. Repetición simple

7. Repetición indexada
8. Índices
9. Uso de índices
10. Tipos de datos
11. Utilidad de los tipos
12. Operaciones primitivas de expresiones (por tipo)
13. Alternativa condicional
14. Funciones simples
15. Invocación de funciones simples
16. Funciones con parámetros
17. Invocación de funciones con argumentos
18. Funciones con procesamiento

Cap. 4

1. Alternativa indexada
2. Repetición condicional
3. Esquemas de recorrido
4. Variables
5. Asignación
6. Uso de variables

Otros conceptos dados en el libro no entran correctamente en esta clasificación. Por esa razón ofrecemos además un mapa conceptual de los conceptos principales (los principales de los que fueron categorizados, y los que no lo fueron), mostrando así la interrelación que existe entre todas las ideas trabajadas. El mapa conceptual de este libro se presenta en el gráfico G.1.



G.1. Mapa conceptual del libro

1

La disciplina de la programación

En este primer capítulo comenzaremos a conocer el mundo de la programación. Analizaremos introductoriamente la idea de lenguaje de programación y de programa, y veremos cómo surge la necesidad de contar con este tipo de lenguajes, a partir de analizar brevemente la historia del surgimiento de los mismos.

A partir de estas nociones iniciales acerca de qué trata la programación y del concepto de lenguaje de programación nos prepararemos para aprender las nociones fundamentales que todo programador maneja.

1.1. ¿Qué es la programación?

La programación es una disciplina que requiere simultánemente del uso de cierto grado de creatividad, un conjunto de conocimientos técnicos asociados y la capacidad de operar constantemente con abstracciones (tanto simbólicas como enteramente mentales).

La creatividad necesaria para programar no se diferencia demasiado de aquella utilizada para producir textos. Sin embargo, lo que hace a la programación algo especial es que requiere emplear un conjunto de conocimientos técnicos asociados a la manipulación de las computadoras. Esto agrega un grado notable de rigurosidad a esta actividad, ya que no podemos programar sin tener en cuenta este aspecto. Por otra parte, al poseer una naturaleza ligada a la resolución de diferentes problemas del mundo real, se requiere de una capacidad de abstracción que permita operar sin que los conocimientos técnicos limiten al programador a resolver adecuadamente dichos problemas.

Para Reflexionar

Ejemplos de actividades que requieren:

- un *uso intensivo de la creatividad* son las relacionadas con el arte;
- *conocimientos técnicos profundos* son las relacionadas con la medicina, electrónica y química;
- *operar continuamente en abstracto* son las relacionadas con filosofía, lógica y matemática.

Todas las actividades mencionadas parecen disímiles. ¿Por qué la programación incluye y utiliza intensamente dichas capacidades?

A lo largo de la vida los seres humanos continuamente enfrentamos todo tipo de problemas. Para ello nos valemos de diversas herramientas, que combinadas de maneras innovadoras amplían el espectro de soluciones y vuelven factible el desarrollo. Los programadores se dedican principalmente a construir programas.

Leer con Atención

¿Qué es un programa? Un programa es una *descripción ejecutable de soluciones a problemas computacionales*, es decir, un texto descriptivo que al ser procesado por una computadora da solución a un problema propuesto por los humanos. De esta manera, la parte descriptiva de los programas es el texto que el programador le provee a la computadora.

Definición 1.1.1. *Un* programa *es una descripción ejecutable de soluciones a problemas computacionales.*

La descripciones dadas por los programas pueden estar escritas con diferentes símbolos y para diferentes propósitos. Cuando el código consiste mayormente de palabras y nociones que son más sencillas para que manejen los humanos, con el objetivo de que puedan entender y construir los programas, hablamos de un lenguaje de *alto nivel*, y al código resultante lo llamamos *código fuente* del programa. Cuando el código consiste mayormente de números y símbolos de difícil comprensión para los humanos, pero de rápida interpretación para su ejecución por una máquina, hablamos de un lenguaje de *bajo nivel*, y al código resultante lo llamamos código objeto o *código ejecutable*. Hablaremos entonces de un nivel *alto* de abstracción cuando nos estemos refiriendo a abstracciones más cercanas a las ideas del problema a ser solucionado, a la mente de los programadores; y de nivel *bajo* de abstracción cuando nos refiramos a abstracciones más cercanas a las ideas relacionadas a las formas de funcionamiento de las máquinas que ejecutan los programas.

Es importante observar que tanto el código fuente como el código ejecutable están conformados por símbolos, y en ese sentido es correcto llamar a ambos *programas*. Esto suele crear confusión, pues entonces la palabra programa se utiliza para dos propósitos diferentes: el código que escribe el programador, y que es el objeto de estudio de este libro, y el código que ejecuta la computadora, que es el resultado de varios procesos de traducción y compilación sobre el código fuente (y que es de mucha menos relevancia a la hora de aprender a programar).

Actividad 1

Es sabido que a diario interactuamos constantemente con programas. Razone qué conceptos posee actualmente sobre los programas y contrástelos con la definición que acaba de leer. Además piense:

1. ¿qué problemas solucionan los programas que utiliza a diario?
2. ¿qué diferencias existen con los problemas que no puede resolver por medio de una computadora?
3. ¿puede ubicar entre sus programas algo de código fuente? (Sugerencia: busque en el programa navegador de internet una opción de “visualizar código fuente”.)

La llamamos *dura* porque debe respetar de manera estricta ciertas reglas de formación. En esto se diferencia bastante de la sintaxis de los lenguajes, donde si bien hay reglas, existen numerosas excepciones. E incluso cuando un texto en lenguaje natural no respeta totalmente las reglas es común que igual puede ser comprendido por un lector humano.

Si bien cuando escribimos el código fuente de un programa utilizamos símbolos como los del lenguaje natural, este texto debe poder ejecutarse. Esa característica hace que los programas se diferencien de otros textos, ya que no cualquier texto es ejecutable por una computadora. Lo que hace a un texto ejecutable es su *sintaxis dura*, que no es más que un conjunto de reglas estrictas de un determinado *lenguaje de programación*, con las que se escribe el código fuente.

En síntesis, la tarea diaria de un programador es codificar, es decir, escribir programas, y dar solución a problemas de diversa índole. La tarea consiste en poder traducir las ideas que los programadores razonan a código ejecutable, que es el que finalmente resolverá el problema en cuestión.

> *Programmers have to balance two very different worlds: a world of structure and a world of imagination. They create abstract concepts using very structured programming languages (like* PHP *or* JAVA*). It's not an easy task.*

http://lifedev.net/2008/07/programmer-creativity-boost/

"Los programadores tienen que balancear dos mundos bien diferentes: un mundo de estructura y un mundo de imaginación. Crean conceptos abstractos usando lenguajes de programación muy estructurados (como PHP o JAVA). No es una tarea fácil."

Un elemento fundamental en la codificación de problemas en forma de programas es la *representación simbólica* de la información: la información concerniente a un problema es representada mediante un *modelo* que luego se expresa a través de *símbolos*, que en el caso de los lenguajes de programación son representados por números o letras. Toda la actividad de programación es un proceso de *codificación* y *decodificación* de la información representada. Esta codificación/decodificación puede darse numerosas veces antes de dar con la solución buscada; por ejemplo, el movimiento de los dedos en un teclado es codificado en señales eléctricas que luego son codificadas como números, los cuales son decodificados como caracteres que se muestran en una pantalla codificados como gráficos, que a su vez son codificados con impulsos eléctricos que se traducen a luces que el cerebro humano interpreta como los mencionados caracteres, que resulta en lo que una persona entiende como escribir en un teclado y ver los caracteres en la pantalla. Rara vez los usuarios de un sistema de cómputo son concientes de la cantidad de codificaciones y decodificaciones que se producen cuando utilizan computadoras. Para que los programadores puedan especificar estas transformaciones existen lenguajes que permiten escribir estos programas.

1.2. ¿Qué son los lenguajes de programación?

Cuando programamos, no podemos utilizar el lenguaje natural con que nos comunicamos cotidianamente los seres humanos. Por el contrario, los programadores emplean un lenguaje que la computadora puede interpretar para realizar tareas. En otras palabras, un *lenguaje de programación*.

Una idea fundamental en la programación, que no es nativa de esta disciplina, pero que es donde se aplica con mayor rigor, es la *composicionalidad*, o sea, la capacidad de entender algo como una composición de partes, y tratar cada parte por separado; y si alguna de las partes es demasiado grande, ¡repetimos el procedimiento! Por ejemplo, una ciudad se compone de barrios, los cuales están compuestos de manzanas, las cuales están compuestas de casas, que a su vez están compuestas de ladrillos, y podríamos continuar hasta llegar a las partículas más elementales (¿los quarks?). En las disciplinas tradicionales, cada parte es estudiada por una especialidad diferente. En nuestro ejemplo, la línea de trabajo para diseñar una ciudad posee urbanistas, arquitectos, etcétera, hasta físcos subatómicos... Cada disciplina, entonces, aborda un nivel y a lo sumo se relaciona con los dos adyacentes en estas *jerarquías conceptuales*. En la programación, en cambio, es la misma disciplina la que aborda el tratamiento de todos los niveles de la jerarquía, desde los bits hasta los sistemas hechos con cientos de miles de componentes complejos. Esta particularidad de la programación fue señalada por el científico de la programación Edsger Wybe Dijkstra, que es considerado por muchos como uno de los fundadores de la computación moderna por sus trabajos fundacionales en programación estructurada y algoritmos y por sus opiniones sobre cómo debe hacerse y enseñarse la programación. En su trabajo *"Sobre la crueldad de enseñar realmente ciencias de la computación"*, él dice que esta noción es una de las *novedades radicales* de la computación, lo que la hace algo sin precedentes en la historia de la ciencia y la humanidad.

Se pronuncia *"ets-jer wíbe dáijkstra"*. Fue un científico holandés que vivió entre 1930 y 2002. Pionero en el desarrollo de las ciencias de la computación y ganador de numerosos premios por sus notables contribuciones a la programación.

Para Ampliar

Realice una búsqueda en internet acerca de la figura de Edsger Dijkstra, y considere algunas de sus contribuciones y las ideas que manejaba. Comience por leer el artículo *"On the cruelty of really teaching computing science"* [Dijkstra and others, 1989] y profundizar sobre la idea de novedad radical.

http://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1036.html

Para Ampliar

Entre las historias que se cuentan sobre Edsger Dijkstra hay una característica de su personalidad que se reflejó en todos sus trabajos. Intente descubrir cuál es.

Lo más notable de esta novedad radical reside en el hecho de que la base de la jerarquía de programación es la única que tiene sustento en el mundo físico. Está constituída por los circuitos que conforman las computadoras. Todas las demás son abstractas, imaginarias. Sobre esa capa física se construyen abstracciones mentales que permiten imaginarse que el movimiento de energía en los circuitos representa números y cuentas entre ellos, y que al encender diversas luces se conforman imágenes que pueden, a su vez, representar casi cualquier cosa: "ventanas" y "botones" en el caso de las interfases gráficas, "dragones" y "armas" en el caso de los juegos, "palabras" y "dibujos" en el caso de las herramientas de oficina como los procesadores de palabras. Con esta abstracción de la circulación de energía pueden hacerse cuentas de maneras sumamente veloces lo cual permite resolver problemas de mayor complejidad, simular inteligencia al realizar deducciones mecánicamente, controlar mecanismos diversos, preparar sistemas que reaccionen a cambios en el entorno, y un sinnúmero de etcéteras; en síntesis, todas las maravillas modernas a las que estamos tan acostumbrados y que hace 100 años atrás eran apenas una fantasía casi utópica.

Como podemos ver a partir de esto, la programación trata fundamentalmente de cuestiones abstractas, imaginarias, que no tienen correlato inmediato en el mundo real. Los programadores, entonces, tenemos que aprender a imaginarnos cosas inexistentes, dotarlas de estructura y volverlas reales a través de los programas. Y las herramientas que nos permiten llevar esto a cabo son los *lenguajes de programación*, el conjunto de reglas que nos permiten escribir programas de cierta manera.

Leer con Atención

Un lenguaje de programación es una serie de reglas que establecen qué descripciones serán aceptadas y ejecutadas y cuáles no tienen sentido para el mecanismo de ejecución provisto por la computadora. Además, estas reglas están diseñadas de manera composicional, para que sea sencillo construir programas de mayor envergadura.

Definición 1.2.1. *Un* lenguaje de programación *es un conjunto de reglas que permiten escribir programas para su ejecución por cierto mecanismo.*

Existen muchos y variados lenguajes de programación, de características sumamente diferentes y que emplean diferentes enfoques con los que podemos programar. Cada lenguaje comprende un conjunto de ideas que guían, como su

propósito final, la forma en que codificamos la descripción que otorgaremos a la máquina. En todo momento el lenguaje de programación permite especificar de manera precisa el trabajo que el programador espera que la computadora realice.

Esta forma viene dada por las reglas que definen cómo se combinan los elementos que el lenguaje de programación provee al programador.

Para Reflexionar

¿Conoce o escuchó el nombre de algún lenguaje de programación? ¿Sabe el lenguaje de programación con el que se desarrolló algún programa que use a diario?

A su vez, algunos lenguajes de programación están pensados para volcar mejor las ideas abstractas que el programador intenta emplear. Estos se conocen como lenguajes de *alto nivel*, ya que intentan, con cierto grado de eficacia, soslayar aquellas tareas que la computadora requiere realizar para que el programa cumpla con sus objetos. Pero existen también lenguajes de *bajo nivel*, que nos hacen definir en mayor o menor medida cada paso que el programa seguirá, y por ende, están ligados a la naturaleza operacional de la máquina. Esta denominación (de "alto" y "bajo") surge de la idea de imaginar que la computadora es la base fundacional sobre la que los programas ejecutan, y que las ideas abstractas se van construyendo sobre ellas. Justamente, denominamos a un lenguaje de más "alto nivel" de abstracción cuánto más lejos de la base se encuentra, o sea, cuanto más abstracto, menos relacionados con el proceso real de ejecución.

Definición 1.2.2. *Un* lenguaje de programación de alto nivel *es uno que expresa mejor las ideas en las que debe pensar el programador, y en alguna forma está más alejado (y menos dependiente) de la máquina específica que ejecutará cada programa escrito en dicho lenguaje.*

Definición 1.2.3. *Un* lenguaje de programación de bajo nivel *es uno que expresa mejor las ideas propias de los mecanismos de ejecución, por lo que es más dependiente de la máquina específica que ejecutará cada programa escrito en dicho lenguaje.*

En algunos cursos de programación inicial suele utilizarse una forma de lenguaje denominada comúnmente *pseudocódigo*, que se confunde con la de lenguaje de programación. En este pseudocódigo se utiliza una combinación entre reglas precisas de lenguajes de programación y formas más libres propias de los lenguajes naturales. Sin embargo, a nuestro entender, no es correcto considerar que el pseudocódigo es un lenguaje de programación, pues no existe un mecanismo exacto de ejecución, quedando librados muchos detalles a la interpretación del lector. En este libro sostenemos que la mejor manera de aprender a programar es enfrentarse con las nociones de código desde el comienzo, en todo caso simplificando, como hacemos aquí, las cuestiones más complejas, pero siempre buscando que los programas que se escriben puedan ser ejecutados correctamente, por lo que desaconsejamos de manera fuerte el uso de cualquier tipo de pseudocódigo.

Existen algunos proyectos que proveen un modelo de ejecución para cierta forma de "pseudocódigo". Sin embargo, al proveer un modelo de ejecución preciso y completo, el pseudocódigo pasa a transformarse en código, por lo que no es correcto seguirlo llamando de aquella manera.

El desarrollo de los lenguajes de programación es guiado fundamentalmente por la búsqueda del nivel de abstracción adecuado para poder expresar con facilidad cierto tipo de soluciones a los problemas a resolver. Podría decirse que los lenguajes de programación surgieron y se desarrollaron a partir de la necesidad de reconciliar dos mundos: el *mundo operacional* o físico, relativo a la ejecución de bajo nivel de las computadoras; y el *mundo denotacional* o abstracto, definido por las ideas que manejan los programadores. Las primeras computadoras con programas solo aceptaban programas escritos en lenguajes de bajo nivel. Eso hacía la tarea de programar extremadamente difícil, propensa a errores, y demasiado costosa. De ahí que los lenguajes fueron evolucionando para aumentar su nivel de abstracción, alejándose cada vez más del modelo específico de ejecución. La búsqueda de la abstracción impacta de manera fuertísima en la forma

en que pensamos programas. Por eso conviene revisar un poco la historia de la programación.

1.3. Breve historia de la programación

Alan Mathison Turing fue un matemático, lógico, criptoanalista e informático inglés que vivió entre 1912 y 1954. Famoso por desarrollar las ideas de algoritmo y de computación, determinó los límites de la computabilidad, o sea, de la capacidad de resolver problemas mediante algoritmos. Es considerado el padre de las ciencias de la computación.

En esta sección vamos a contar una breve (y no particularmente precisa) historia de la programación, con el objetivo de tener un panorama de cómo se clasifican los lenguajes modernos, y cómo esto influye en la forma en que se piensan programas. Nos limitaremos a hablar de la parte "moderna" de la historia, desde el surgimiento de las computadoras con válvulas y circuitos hasta nuestros días, omitiendo la "prehistoria" de las máquinas de calcular, teorías fundacionales, etcétera.

1.3.1. Surgimiento de las computadoras

Las computadoras como las conocemos actualmente surgieron durante la primera mitad del siglo XX. En 1936, Alan Turing desarrolló la idea de máquina de cómputo universal a través de sus *máquinas de Turing*, que fueron los primeros dispositivos teóricos en contar con la noción de un "programa", o sea un dato que se le suministraba a la máquina y que afectaba su comportamiento. Turing intentaba resolver el problema de decodificar los mensajes del ejército alemán durante la guerra, y se enfrentaba con el problema de que su equipo no era lo suficientemente veloz para resolver el problema a tiempo, a pesar de que las bases de cálculo eran sencillas. De ahí que pensó en utilizar máquinas para acelerar el proceso, y desarrolló la teoría necesaria para tales máquinas.

Se pronuncia "yon fon nóiman". Matemático húngaro-americano, que vivió entre 1903 y 1957. Realizó numerosas contribuciones en matemáticas, pero su nombre se conoce por su aporte a la arquitectura de las computadoras.

A partir de los trabajos de Turing, John von Neumann desarrolló una arquitectura de máquinas que usaban la misma memoria para almacenar los datos y los "programas". La *arquitectura von Neumann*, o alguna de sus variantes, es aún hoy utilizada por, prácticamente, todas las formas de computadoras existentes.

Estos dos descubrimientos permitieron, durante la Segunda Guerra Mundial el desarrollo práctico de las primeras computadoras: los trabajos del alemán Zuse, los trabajos ingleses en la línea Colossus y los trabajos americanos en la línea ENIAC. Pero recién en la década de 1950 se comenzaron a producir computadoras a nivel comercial. Estaban basadas en tubos de vacío llamados "válvulas" y eran enormes y costosas, y su poder de cómputo era menor que el de un teléfono digital moderno. Sin embargo significaron una revolución en términos de cálculo para la época. Fueron la llamada "primer generación" de computadoras. La "segunda generación" aparece a fines de esa década, cuando las válvulas son reemplazadas por transistores, y permiten iniciar la carrera para disminuir el tamaño de las computadoras. Las computadoras de la segunda generación son más pequeñas y consumen menos electricidad que las anteriores.

Durante la década de 1960 se inventó el circuito integrado, que fue la clave para el desarrollo del microprocesador, que dio lugar a la "tercera generación". Los circuitos integrados son pastillas de silicio en las que se colocan miles de componentes electrónicos en una integración en miniatura.

Para Ampliar

Para saber más sobre el desarrollo del *hardware*, especialmente las primeras computadoras, consultar:

http://en.wikipedia.org/wiki/Computer_history y
http://es.wikipedia.org/wiki/Generaciones_de_computadoras

Todas estas computadoras iniciales se programaban en lenguaje binario, o sea,

a través de lo que comúnmente se denomina “ceros y unos”, y que en realidad en esa época eran agujeros (o su ausencia) en tarjetas perforadas. Con este lenguaje binario se codificaban instrucciones para indicar las acciones que las máquinas intentaban llevar adelante. Los primeros lenguajes de programación reemplazaban entonces estas cadenas de 0s y 1s por “palabras” que fueran más fáciles de recordar por los humanos que construían los programas (*instrucciones mnemónicas*), como MOVE, LDA, ADD, etc., las cuales constituyeron la primer abstracción sobre la capa física. La relación entre las instrucciones mnemónicas y el código binario era directa: por cada mnemónico existía una instrucción en binario y viceversa. Así se construyen los primeros traductores que traducen instrucciones mnemónicas a binario; recibieron el nombre de *ensambladores*, y se convirtieron en los primeros lenguajes de programación de computadoras de la historia.

Cada computadora venía con su propio lenguaje ensamblador (*assembly languages* en inglés), y cada programador debía aprenderlo junto con las características específicas de la máquina en cuestión. Pero programar en *assembler* requería tremendos esfuerzos intelectuales y era muy fácil cometer errores, puesto que había que atender todos y cada uno de los aspectos de las máquinas que se programaban, los cuales no eran pocos ni simples, puesto que atendían a cada uno de los componentes específicos que cada máquina poseía con sus características de funcionamiento particulares. Además, al querer realizar un programa para otra computadora debía aprenderse todo el lenguaje de nuevo desde el comienzo, puesto que cada computadora tenía un conjunto diferente de componentes, y el lenguaje ensamblador de cada máquina reflejaba en su estructura estos componentes. El problema para los programadores con el lenguaje *assembler* es que el mismo se acerca más a la forma de operar de las computadoras y no a la forma de pensar de los humanos; o sea, en los términos que manejamos en este capítulo, es un lenguaje de **muy bajo nivel**. En ese momento era necesario este tipo de manejo de bajo nivel, puesto que los recursos de los que disponía una computadora eran extremadamente escasos y debían, por lo tanto, ser administrados con precisión y sumo cuidado.

Se suele decir que los 0s y 1s, constituyen la *base física* de la jerarquía conceptual de los programas. Sin embargo, este lenguaje es en realidad una abstracción que expresa la presencia o ausencia de algo a través de 2 números y de combinaciones de los mismos, aunque el hecho de que esta también sea una abstracción prácticamente no se tiene en cuenta al considerar las abstracciones dadas por los lenguajes.

Si intentamos ser precisos, no se trata de un lenguaje, sino de un tipo o familia de lenguajes. Pero es común referise a esta familia como si se tratase de un lenguaje único.

John Warner Backus fue un científico americano de la computación que vivió entre 1924 y 2007. Ganó diversos premios por sus contribuciones, entre las que se encuentran la creación del primer lenguaje de programación de alto nivel, y de un mecanismo de descripción de lenguajes de programación conocido hoy como forma Backus-Naur (Backus-Naur Form, o BNF).

1.3.2. Los primeros lenguajes de alto nivel

La situación con los lenguajes ensambladores era bastante preocupante para los programadores de aquella época pues les restaba productividad, les restringía la creatividad y el espectro de soluciones imaginables y los mantenía todo el tiempo pensando más en la máquina que en la programación en sí. Por esa razón, en 1955, John Backus y sus colegas de IBM desarrollaron, en el campus de esta compañía en California en Estados Unidos, el lenguaje FORTRAN, por FOR*mula* TRAN*slator* (o más detalladamente, *the IBM Mathematical* FOR*mula* TRAN*slating System*), que es típicamente considerado por toda la comunidad informática como el primer lenguaje de programación independiente de la máquina. FORTRAN introdujo muchas ventajas sobre los lenguajes ensambladores, e hizo más claras las operaciones básicas.

La idea rectora en esos días era hacer la programación más cercana al lenguaje natural humano; o dicho de otro modo, aumentar el nivel de abstracción. Por esa razón, estos lenguajes que empezaron a surgir fueron denominados “lenguajes de alto nivel”. Hoy en día se los suele entender como lenguajes de más bajo nivel, puesto que existen otros lenguajes (como los lenguajes funcionales o los orientados a objetos) que son de mayor nivel de abstracción que los aquí mencionados. En esta época el motor que llevaba adelante las investigaciones y el desarrollo de software eran las aplicaciones de defensa y las de administración a gran escala.

Técnicamente el primer lenguaje de programación de este estilo fue PLANKALKÜL (pronunciado “plánkalkil”), desarrollado por Konrad Zuse en 1943 en Alemania. Sin embargo, debido a la Guerra, este lenguaje no fue hecho público hasta 1972.

Al poco tiempo de la aparición de FORTRAN surgieron otros dos lenguajes: LISP (por LIS*t* P*rocessor*, procesador de listas), antecesor de los modernos lenguajes funcionales y COBOL (por CO*mmon* B*usiness* O*riented* L*anguage*,

lenguaje orientado a los negocios comunes) que fuera adoptado en bancos, compañías y dependencias oficiales.

Leer con Atención

El lenguaje LISP se basó en una teoría matemática que podría considerarse hoy el primer lenguaje de programación de alto nivel, y que fue desarrollada en la década de 1930 por el lógico-matemático Alonzo Church; esa teoría se conoce con el nombre de *lambda cálculo* (utiliza letras griegas como λ, α, β para identificar diferentes conceptos de programación). En paralelo, el lógico-matemático Haskell B. Curry desarrolló una teoría denominada *cálculo combinatorio*, de poder similar al lambda cálculo.

Uno de los primeros usos en computación del lambda cálculo fue el de permitir explicar en términos abstractos el funcionamiento de lenguajes imperativos (en particular, de ALGOL).

Es importante mencionar este lenguaje por dos razones: primero, porque hoy en día, lenguajes basados en las ideas del lambda cálculo (los lenguajes funcionales) se consideran una de las variantes de más alto nivel de los lenguajes de programación, y están haciendo gran diferencia en el campo de la programación paralela y concurrente; y en segundo lugar, porque ¡fueron desarrollados casi una década entera antes que existiese la idea de computadora como máquina física!

Luego de FORTRAN, LISP y COBOL, en menos de una década florecieron otra docena o más de lenguajes diferentes. Cada uno de estos lenguajes se enfocaba en cierta forma de resolver problemas, siendo la mayoría orientados a establecer una *secuencia de instrucciones* con foco en diferentes aspectos de cómo establecer la comunicación entre las diferentes partes que componían el programa.

Entre los más notorios que dieron origen a ideas modernas están CPL (antecesor de C), SIMULA (precursor de la programación orientada a objetos), y BASIC (originalmente diseñado para la enseñanza).

Un lenguaje que resultó un hito por la consolidación de muchas ideas al respecto de cómo debían definirse los lenguajes de programación fue ALGOL (por ALGO*rithmic* L*anguage*, lenguaje de algoritmos). Este lenguaje fue de gran influencia en muchísimos lenguajes modernos, puesto que sus ideas siguen usándose en el diseño actual de lenguajes de programación (por ejemplo, la forma de describir sus reglas de sintaxis, la Backus-Naur Form o BNF, se sigue usando en la especificación de la sintaxis de todos los lenguajes de programación modernos). También fue el primer lenguaje para el que se estudió cómo asignar significado (semántica) de manera independiente a la ejecución.

1.3.3. Los paradigmas de programación

Durante la década de 1960, la proliferación de lenguajes de programación siguió creciendo, y de a poco fueron diferenciándose grupos o "familias" de lenguajes, en torno a la predominancia de ciertas características. El foco del desarrollo de software se fue desplazando hacia la educación, para poder formar a los futuros programadores. La administración a gran escala siguió teniendo fuerte presencia, pero las aplicaciones orientadas a la defensa fueron disminuyendo.

Estos grupos o familias de lenguajes dieron origen a lo que se denomina *paradigmas de programación*, que no es otra cosa que un conjunto de ideas y conceptos al respecto del estilo con el que se expresan las soluciones a problemas a través de un lenguaje de programación. Cada paradigma privilegia ciertas ideas por sobre otras, y ciertas formas de combinación por sobre otras, dando lugar a estilos muy diferentes (aunque muchas veces complementarios) en la forma de programar.

Definición 1.3.1. *Un* paradigma de programación *es un conjunto de ideas y conceptos vinculados a la forma en que se relacionan las nociones necesarias para solucionar problemas con el uso de un lenguaje de programación.*

Para 1970 ya se pueden identificar cuatro grandes paradigmas, que están vigentes hoy día y que son claramente reconocidos por todos los miembros de la comunidad informática. Mucho después se intentó identificar paradigmas adicionales a estos cuatro originales, pero no hay consenso sobre si alguno de ellos llega o no a poder ser considerado un paradigma y merece ese nombre; por esa razón nos limitaremos a hablar de los cuatro paradigmas principales.

Los cuatro paradigmas de programación que surgieron a fines de los sesenta y principios de los 1970, y que resultaron de fundamental influencia en la forma de hacer programación, son:

- el paradigma *imperativo*,
- el paradigma *orientado a objetos*,
- el paradigma *funcional*, y
- el paradigma *lógico*.

Los dos primeros están más orientados a la forma de manejar estado y podrían ser denominados *procedurales*, mientras que los dos últimos están más orientados a expresar conceptos o nociones independientes del estado, y podrían ser denominados *declarativos*. El paradigma que se desarrolló con mayor ímpetu al principio fue el imperativo, debido a su cercanía con los lenguajes de bajo nivel. Los otros tardaron más tiempo en adoptar un estado de madurez, y no fue hasta mediados de la década de 1980 que tanto el paradigma funcional como el lógico y el orientado a objetos empezaron a ser foco de atención masiva.

No es usual asociar el término procedural con los lenguajes orientados a objetos, pero es una forma de indicar que los mismos se concentran más en el estado que en la descripción de información.

Dentro del paradigma imperativo se clasifican lenguajes más vinculados con la secuencia de instrucciones y cercanos al *assembler*. Algunos nombres notables que surgieron en esa época dentro del paradigma imperativo, y aún conocidos hoy en día, son: BASIC, desarrollado en 1965 por John Kemey y Thomas Kurtz con la intención de que se conviertiese en un lenguaje de enseñanza; PASCAL, creado, también con fines didácticos, por Niklaus Wirth en 1970 a partir del ALGOL; y C, diseñado por Dennis Ritchie y Ken Thompson en los laboratorios Bell (*Bell Labs*) entre 1969 y 1973, con el propósito proveer una traducción eficiente a assembler y permitir la administración eficaz de los recursos de cómputo de las máquinas con arquitectura von Neumann a través de abstracciones cercanas al bajo nivel, que brinda una forma cómoda e independiente de la computadora de administrar sus recursos. La fama del C se debe a que, por esta característica de su diseño, fue utilizado en la programación del sistema operativo UNIX y fue ampliamente portado a numerosos sistemas. Es uno de los lenguajes más difundidos y conocidos de todos los tiempos, y su estudio implica un conocimiento profundo de la forma en que se ejecutan las computadoras.

Actualmente el nombre “UNIX” es una marca registrada que se licencia para usar cuando un sistema operativo cualquiera satisface sus definiciones. La denominación “UNIX-*like*” (“de tipo UNIX”) se utiliza para referirse a una gran familia de sistemas operativos que se asemejan al UNIX original. La familia UNIX-*like* tiene diversas subcategorías, entre las que se encuentra el actualmente popular GNU/LINUX.

Para Ampliar

Para saber más sobre sistemas operativos, su función, su historia y su clasificación, consultar la Wikipedia:

http://en.wikipedia.org/wiki/Operating_system

Dentro del paradigma funcional se clasifican lenguajes orientados a la descripción de datos, de su forma, las relaciones entre ellos, y sus transformaciones. Si bien inicialmente no fueron tan populares, la investigación llevó a este paradigma a la madurez y desde el mismo se realizaron grandes aportes a todos los lenguajes modernos. Algunos lenguajes que surgieron en esa época dentro de este paradigma son: ML, desarrollado por Robin Milner y otros a principios de los setenta en la Universidad de Edimburgo en el Reino Unido con el propósito de servir para desarrollar tácticas de prueba en herramientas de demostración

automática de teoremas, utilizando un sistema de tipos estático que es una de sus grandes innovaciones; MIRANDA, desarrollado por David Turner en 1985 como sucesor de sus primeros lenguajes de programación, SASL y KRC, incorporando conceptos aprendidos del lenguaje ML; y SCHEME, derivado como dialecto de LISP por Guy L. Steele and Gerald J. Sussman en el laboratorio de inteligencia artifical del MIT, siguiendo principios de minimalidad en la cantidad de conceptos distintos a proveer, pero conservando un gran poder expresivo. A mediados de la década de 1980 se conformó un comité mundial de científicos de la programación funcional que definió un lenguaje estándar que expresase la suma de conocimientos sobre lenguajes funcionales, basándose fundamentalmente en MIRANDA. Este nuevo lenguaje, HASKELL (en honor a Haskell B. Curry), se publicó por primera vez en 1990, y es actualmente el lenguaje de alto nivel con mayor pureza conceptual, expresando el estado del arte en el desarrollo de lenguajes de programación funcional. Su impacto en la comprensión de conceptos de alto nivel no puede ser ignorada por un programador actual. El enfoque de este libro incorpora muchas ideas aprendidas a partir de HASKELL.

HASKELL es un lenguaje funcional puro avanzado estándar, con semántica no-estricta y tipado estático fuerte. Permite el desarrollo de software correcto, conciso y robusto. Cuenta con funciones como valores, mecanismos para la integración con otros lenguajes, concurrencia y paralelismo intrínsecos, diversas herramientas de producción de software, sofisticadas bibliotecas de funciones y una comunidad activa a lo ancho de todo el mundo. Más información en su página web:

http://www.haskell.org

Dentro del paradigma orientado a objetos se encuentran lenguajes que agrupan el código alrededor de la metáfora de “objeto”, y que intentan representar mediante datos encapsulados las entidades del mundo real. Al ya mencionado lenguaje SIMULA, pionero de los lenguajes orientados a objetos, se le agregó el lenguaje SMALLTALK, creado en el *Learning Research Group* (LRG) de Xerox por Alan Kay y otros, también en los setenta, pensado con fines educacionales basándose en la teoría constructivista del aprendizaje. Fue la base del desarrollo posterior en tecnología de objetos, que hoy es uno de los pilares de la construcción moderna de *software*.

Finalmente, dentro del paradigma lógico se encuentran distintos lenguajes orientados a la descripción de las relaciones lógicas entre aserciones, que habilitaban la posibilidad de realizar inferencias y ciertas formas de razonamiento automático, lo cual fue la inspiración para desarrollar el área conocida como *inteligencia artificial*. El lenguaje más conocido de este paradigma, que también surgió a finales de la década de 1970 en Marseille, Francia, en el grupo de Alain Colmerauer, es PROLOG, un lenguaje basado en la afirmación de hechos y reglas de inferencia, que se utiliza mediante “consultas” en la forma de afirmaciones que deben ser validadas a partir de los hechos.

Usualmente es definida como “el estudio y diseño de agentes inteligentes”, donde un agente inteligente es un sistema que decide de sus acciones con base en su entorno de manera de maximizar sus chances de éxito. Existen numerosos subcampos dentro del campo de la inteligencia artificial y los mismos no siempre están vinculados, siendo los más exitosos los vinculados a la optimización y búsqueda de información compleja.

Cada uno de los lenguajes mencionados dejó un sinnúmero de descendientes y prácticamente todos los lenguajes modernos se vinculan, de una forma u otra, con alguno de éstos.

Otro gran avance de esta época fue el desarrollo de lo que dio en llamarse *programación estructurada*, que sucedió dentro del paradigma imperativo, y consistió fundamentalmente en aumentar la abstracción de los lenguajes, eliminando primitivas de control “desestructurado”, o sea, que permitían moverse libremente por el código, sin tener en cuenta su estructura lógica. En este avance es cuando se establecen las formas fundamentales de combinación de componentes que veremos en este libro.

1.3.4. Consolidación y expansión del software

Durante la década de 1980 todos estos lenguajes y paradigmas comienzan un proceso de consolidación y de combinación, que dan origen a nuevos lenguajes híbridos entre paradigmas, refinan y estandarizan los lenguajes existentes, y sintetizan todo lo aprendido en nuevos lenguajes. Además, diversos grupos de investigación y desarrollo se comprometen con la implementación de compiladores y herramientas para ayudar en el proceso de construcción de software, dando lugar a numerosos avances en el campo de la implementación de lenguajes.

En este período surgen además las computadoras personales, actualmente conocidas como PC, que invaden los hogares y empujan el desarrollo de técnicas de programación a mucha mayor escala. Por ejemplo, la provisión de interfaces

gráficas útiles y eficientes, de sistemas de oficina personales, el desarrollo de juegos, la administración de pequeñas empresas (que antes no podían permitirse la administración computarizada por sus costos altísimos), etcétera. Esto dio origen a lo que actualmente se conoce como la *industria del software*, a través del desarrollo de empresas dedicadas exclusivamente a la producción de software.

En esta época surgen lenguajes como C++, que es un derivado del lenguaje C al que se le agregan nociones de programación con objetos; PERL, un lenguaje de scripting pensado para hacer reportes en sistemas UNIX; TCL (pronunciado "téce-ele", o también "tíkl"), otro lenguaje de scripting para tareas de prototipación de interfaces gráficas, el cual se suele combinar con el *toolkit* de interfaces TK, combinación conocida como TCL/TK; y ERLANG, un lenguaje funcional y sistema de ejecución desarrollado por la empresa Ericsson para construir aplicaciones de tiempo real tolerantes a fallos.

Son lenguajes basados en *scripts*, que a su vez son pequeños programas escritos con la intención de automatizar tareas de mantenimiento en ambientes de software, y que originalmente estaban ideados para ser corridos a mano por el administrador del sistema.

Hacia el final de este período la comunidad de programación funcional desarrolla el lenguaje HASKELL, que sintetiza los conocimientos de todos los lenguajes del área, y establece un estándar para el paradigma funcional, sentando las bases para la investigación de nuevas características. Este lenguaje, que es (junto con SMALLTALK en el paradigma de objetos) uno de los pocos lenguajes completamente puros con capacidad de desarrollo a nivel industrial, tiene la particularidad de haber sido desarrollado por un comité representativo de toda la comunidad funcional, encabezado por John Hughes y Simon Peyton-Jones. Es, en la opinión de muchos investigadores de lenguajes, uno de los lenguajes más sólidos conceptualmente, y nos atreveríamos a decir, uno de los lenguajes de más alto nivel entre los existentes actualmente y de los más adecuados para enseñar conceptos avanzados de programación. Además ha sido el soporte de numerosas investigaciones que desarollaron la teoría necesaria para enriquecer a muchos otros lenguajes.

1.3.5. La era de internet

La década de 1990 marcó la difusión de Internet, que constituyó una plataforma completamente nueva para el desarrollo de los sistemas de computadoras. Esta nueva forma de comunicar computadoras dio lugar a numerosas evoluciones, de las cuales los lenguajes de programación y el desarrollo de software no estuvieron ausentes.

Una evolución importante que se dio en este período fue la gran difusión de la idea de *software libre*, que aunque no tiene que ver con la parte técnica de los lenguajes en sí, impactó en la forma de hacer software y sistemas. Durante la expansión de las PC de la década de 1980 las empresas de software vendían *el derecho de usar* los ejecutables de los programas (práctica que muchas empresas aún mantienen), pero no de conocer el código fuente de los mismos. De esta manera, cualquier modificación que debiera ser hecha en el código debía esperar a que la empresa fabricante dispusiese recursos para ello. Esta es una forma de disponer del *derecho de autor* para limitar el uso del código producido. En esa época surgió también la idea de utilizar el derecho de autor de otra manera, para *permitir la inspección y modificación* del código fuente, siempre y cuando se respetase la autoría original. Esta idea evolucionó en los noventa en muchas formas de licenciamiento diferentes.

En inglés *free software*, pero usando *free* en el sentido de libertad (como en *free will*, libre albedrío), y no en el sentido de gratuidad (como en *free beer*, cerveza gratis). Si bien el software libre puede ser gratuito, la libertad tiene que ver más bien con la posibilidad de conocer y modificar el código fuente de los programas.

El ejecutable es un programa en una forma que puede ser ejecutado por una computadora específica, pero que no es normalmente entendible por las personas

Para Ampliar

Investigue sobre el concepto de software libre, sobre cómo el mismo fue producto de las formas de licenciamiento, sobre las variantes existentes de licencias y sus características, y sobre las profundas consecuencias sociales de esta idea.

La posibilidad de inspeccionar y modificar el código, sumado a la amplísima red de comunicación que proveyó Internet, permitió desarrollar comunidades alrede-

dor de ciertos sistemas de software, dando lugar a desarrollos como el sistema operativo GNU/LINUX, el navegador Mozilla, la suite de oficina OpenOffice y muchísimas otras.

Esta ampliación de la cantidad de gente en condiciones de programar, junto con nuevas formas de utilizar las computadoras a través de internet, favoreció la aparición de nuevos lenguajes. En este período el mayor motor de avance fue la abstracción de las operaciones en la web y el desarrollo de aplicaciones que aprovechasen Internet. En esta época surgen lenguajes como PYTHON, un lenguaje que combina los paradigmas orientado a objetos, imperativo y en menor medida, funcional; RUBY, un lenguaje dinámico y con capacidad de modificar su propio código basado en PERL y SMALLTALK; y PHP, un lenguaje de scripting originalmente pensado para producir páginas web dinámicas. Sin embargo, los lenguajes que alcanzan realmente mayor difusión son JAVA, un lenguaje desarrollado por la empresa Sun Microsystems que deriva su sintaxis de los lenguajes C y C++ incorporando nociones de objetos al estilo SMALLTALK, y con la principal característica de correr sobre una *máquina virtual*; y JAVASCRIPT (que no debe ser confundido con JAVA), un lenguaje de scripting basado en prototipos combinando características de los paradigmas imperativo, orientado a objetos y funcional, y cuya utilización es básicamente en la construcción de páginas web, al ser combinado con el lenguaje HTML de especificación de formato en dichas páginas.

Una máquina virtual es un programa que simula ser una computadora que ejecuta programas. Su principal objetivo es despreocupar al programador de muchas de las características específicas que posee la computadora real en donde finalmente correrán sus programas.

1.3.6. Tendencias actuales

El siglo XXI continuó viendo el desarrollo de la programación, e incorporó a la comunicación en Internet (que continúa en expansión) el motor producido por el desarrollo de juegos, la telefonía móvil, el surgimiento de la idea de la *nube digital*, la animación cinematográfica, y actualmente la Televisión Digital. Siguieron apareciendo lenguajes de la mano de grandes empresas. Se pueden mencionar C# y F# de la plataforma .NET de Microsoft, y los lenguajes DART y GO de la empresa Google. También surge SCALA, que se diseñó para misma la máquina virtual de JAVA, e incorpora nociones del paradigma funcional a dicho lenguaje.

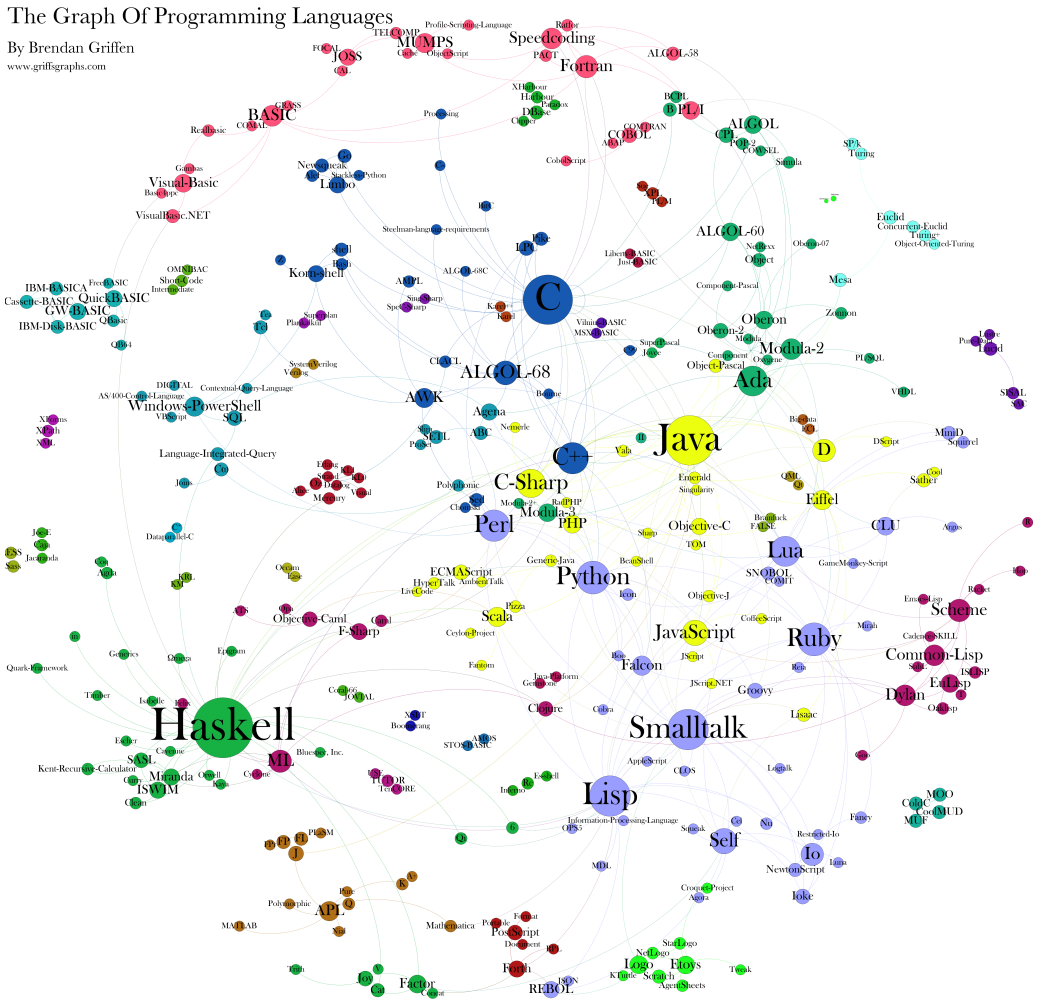
Este concepto tiene que ver con empresas que alojan en sus servidores desde los datos de los usuarios hasta las aplicaciones con los que los mismos son modificados, y permiten que cualquier persona mantenga en un único lugar su información.

También se desarrollaron numerosísimas líneas de trabajo, como ser la estandarización (dando lugar a desarrollos como *Unicode*, que es un estándar para la codificación de textos o el lenguaje extendido de descripción de documentos, XML), la integración de sistemas (diferentes lenguajes incorporan la capacidad de ser combinados en el mismo programa, diferentes herramientas como bases de datos pueden ser accedidas desde casi cualquier lenguaje, etc.), las aplicaciones en programación concurrente, paralela y masivamente paralela (que permiten el mejor aprovechamiento de los recursos computacionales), desarrollos en seguridad de la información herramientas como firewalls, antispams, antivirus y muchas otras, pero también la incorporación de nociones de seguridad en los lenguajes), investigaciones en código móvil, computación de alto desempeño, programación orientada a aspectos y muchísimos etcéteras.

Para hacerse una idea de la cantidad de lenguajes que existen, sus influencias relativas entre ellos y su clasificación según paradigmas puede observarse el gráfico G.1.1, donde se muestran los lenguajes en un grafo de tipo “galaxia”, donde se agrupan por paradigma y grupos dentro de los paradigmas (indicados por color), con las conexiones a los lenguajes que influenciaron y su tamaño relativo a su influencia. El original de este gráfico puede encontrarse en:

Recurso Web

http://griffsgraphs.com/2012/07/01/programming-languages-influences/



G.1.1. “Galaxia” de los lenguajes de programación, según paradigma e influencia

Cerramos esta sección con la certeza de que la próxima década nos deparará nuevas sorpresas en el desarrollo de sistemas de computación.

1.4. Lenguajes para dominios específicos

Todos los lenguajes descritos en las secciones anteriores tenían la particularidad de ser *lenguajes de propósitos generales* (GPLs, por su sigla en inglés, *General Purpose Languages*) y como vimos, la intención fue siempre abstraer de una forma u otra el modelo de computación subyacente.

Ahora bien, para poder cumplir con este propósito (abstracción con generalidad), estos lenguajes ofrecen una amplia gama de herramientas y formas de combinación. Esta amplitud resulta ser demasiada en ocasiones, cuando los propósitos son más particulares (como por ejemplo acceso a una base de datos, composición de textos, descripción de páginas web, etc.), sobre todo teniendo en cuenta que los usuarios que busquen utilizarlos serán especialistas de dichos dominios pero no necesariamente en conceptos generales de programación. Por esta razón surgieron otro tipo de lenguajes, orientados a abstraer las nociones e ideas de un dominio particular de trabajo. Estos lenguajes se conocen como *lenguajes específicos de dominio* (DSLs, por sus siglas en inglés, *Domain Specific Languages*). Son normalmente lenguajes altamente declarativos, y en muchos

Un lenguaje declarativo es un lenguaje orientado a la descripción de soluciones minimizando los componentes de ejecución que se explicitan. Se asocian típicamente a los paradigmas funcional y lógico y, debido a su capacidad de expresar cómputos paralelos de manera implícita, han cobrado mucha relevancia recientemente.

casos extremadamente pequeños, adecuados solamente para una tarea muy específica, que tanto pueden ser considerados lenguajes de programación como lenguajes de especificación.

Cada dominio particular tiene asociado uno o más lenguajes diseñados teniendo en cuenta las ideas de dicho dominio, y sus particularidades, abstrayendo lo necesario y no más que eso. Existe un estilo de programación que sostiene que para cada problema debe crearse un DSL particular para resolverlo, siendo esto parte del proceso propuesto para la solución de problemas; este estilo se suele denominar *programación orientada a lenguajes*. Existen numerosas investigaciones sobre la manera más eficiente y efectiva de definir nuevos lenguajes rápidamente y se han desarrollado numerosas herramientas para ello.

En esta sección mencionaremos dos o tres de los dominios más popularizados, y relevantes para el público general, pues una cobertura exhaustiva resulta prácticamente imposible e innecesaria para los objetivos aquí propuestos.

Puede resultar interesante investigar más en el tema, eligiendo un dominio de su interés y comprobando si existen DSLs para dicho dominio.

El primero de los dominios que consideraremos es el de consulta y administración de bases de datos. El modelo tradicional utiliza *bases de datos relacionales*, que son agrupaciones de datos en vínculos (o relaciones) dadas por el problema a modelar. Para describir las consultas a este tipo de bases de datos, IBM en 1970 desarrolló un DSL denominado SQL (por su denominación en inglés, *Structured Query Language*). Es uno de los DSLs más conocidos hoy día. Este lenguaje evolucionó con el tiempo, y dio lugar a diversos lenguajes de propósito general como FOX, CLIPPER, RBASE y los lenguajes asociados a los motores de bases de datos *Oracle* e *INFORMIX*. Todos los lenguajes de esta familia se denominan grupalemente como 4GL (Lenguajes de Cuarta Generación).

Otro dominio muy conocido es el de diseño y especificación de contenidos para la Web. En este dominio el lenguaje más conocido es HTML, un lenguaje de marcado (*markup language*) que permite especificar los elementos de una página web de manera tal que diferentes herramientas puedan elegir cómo mostrarlos. Como otros lenguajes de marcado, HTML especifica información a través de *etiquetas*, más conocidas por su denominación en inglés, *tags*. Los tags son utilizados para describir estructura, información visual, comportamiento u otras nociones, y son interpretados por las diferentes herramientas que analizan un programa en uno de estos lenguajes para decidir diferentes cursos de acción. Por ejemplo, en el caso de HTML, los diferentes navegadores pueden elegir mostrar los elementos de diferentes maneras dependiendo de factores tales como el tipo de dispositivo (pantalla de PC, teléfono, etc.), las opciones elegidas (ver el código fuente o mostrar el resultado) y otros. Por eso se dice que HTML es un *lenguaje de marcado presentacional*. Otro DSL asociado al dominio de presentación web es el CSS, por el inglés *Cascading Style Sheets* (hojas de estilo en cascada), utilizado para especificar el estilo de presentación de una familia de documentos de manera genérica.

Son lenguajes textuales que permiten anotar un documento con etiquetas o *tags* que sirven para especificar estructura, agregar significados asociados a diferentes elementos, asociar información de autoría, revisiones, etcétera.

Esta palabra también se utiliza en castellano como anglicismo aún no aceptado por la Real Academia Española.

Otro DSL que se clasifica dentro de los lenguajes de marcado es el XML, por *eXtensible Markup Language* (lenguaje de marcado extensible), que se utiliza para definir un conjunto de reglas de codificación de documentos en un formato que es simultáneamente legible por las personas y por las computadoras. Aquí los tags se utilizan para expresar la estructura del documento, sus partes y dependencias, y en ese sentido se puede considerar como un *lenguaje de marcado descriptivo*. Los documentos descritos con XML están correctamente formados por definición, siendo imposible la especificación de documentos con partes faltantes o incompletos. El lenguaje es extremadamente flexible, pudiendo utilizarse para describir *metadatos*, o sea, documentos que describen, de manera completamente formal y susceptible de ser analizada por las computadoras, la forma de los datos o documentos a definir con el mismo lenguaje XML. Una de las formas más conocidas de este tipo de descripciones es llamada DTD, por *Document Type Definition* (definición de tipo de documento), un conjunto de tags XML específicamente diseñado para describir formatos de documentos. Del XML evolucionaron otros lenguajes tales como RSS, ATOM, SOAP, and XHTML (una versión de HTML basada en las definiciones de XML).

Un tercer dominio, aunque por el momento solo difundido en la comunidad científica (más específicamente entre matemáticos, físicos e informáticos), es el de composición tipográfica (en inglés, *typesetting*). En este dominio existe un lenguaje de marcado extremadamente poderoso y flexible llamado LaTeX, utilizado para la preparación de documentos de altísima calidad de presentación con mínimo esfuerzo. LaTeX es la parte frontal del languaje TeX, un lenguaje de marcado para typesetting pero de más bajo nivel. La forma de trabajo propuesta por este lenguaje difiere de manera radical de la filosofía WYSIWYG (*What You See Is What You Get*, "lo que ves es lo que obtenés") promovida por las herramientas como *MS-Office* y *OpenOffice*, permitiendo concentrarse más en la estructura del documento que en su presentación o composición tipográfica, aportando eficiencia y flexibilidad con mínimo esfuerzo. Si bien la curva de aprendizaje de este lenguaje es elevada al inicio, una vez que se alcanza un cierto grado de familiaridad la productividad se multiplica exponencialmente, permitiendo escribir libros con la misma facilidad con la que se escriben documentos cortos. En particular este libro fue diseñado de manera completa utilizando LaTeX. Existen extensiones de LaTeX para escribir presentaciones (similares a las de *MS-Powerpoint*, pero con la filosofía LaTeX), páginas web (permitiendo traducir a HTML u otros DSLs para ese propósito) y documentos con contenido específico de numerosos dominios (como matemáticas, química, informática y muchos más).

Pronunciado *"látex"* en castellano, pero también *"léitek"* o *"léitej"* en el idioma inglés.

Es una forma de definir el esfuerzo requerido para aprender cierta noción en función del tiempo requerido y los resultados obtenidos. Una curva elevada indica que hace falta mucho esfuerzo en poco tiempo para obtener buenos resultados.

Otro dominio de interés especialmente para programadores es el desarrollo de lenguajes de programación, donde se definen herramientas conocidas *analizadores sintácticos*, útiles a los diseñadores de lenguajes de programación. Para este propósito existen lenguajes basados en las gramáticas BNF y utilizados en herramientas como YACC, BISON, y una multitud de derivados.

Para terminar enumeramos algunas de los lenguajes y herramientas más conocidas en diversos dominios: POSTSCRIPT (para descripción de páginas para impresión y publicación electrónica), MATHEMATICA (para manipulación simbólica en áreas científicas y de ingeniería), VHDL (para especificación de *hardware*), LOGO (para enseñanza de programación en niveles iniciales), CSOUND (para especificación de sonidos), lenguajes para procesamiento de imágenes, lenguajes de reescritura de grafos, de trazado de grafos, lenguajes para descripción de diagramas, y muchos, muchos otros.

Actividad 2

¿Puede determinar si existe algún DSL para su área de dominio? Si existe, investigue sobre las características de dicho lenguaje.

Para Ampliar

Puede complementar lo visto sobre la historia de la programación con un resumen visual en el video que se puede encontrar en:

http://programandoenc.over-blog.es/article-28741792.html

2

Primeros elementos de programación

Para empezar a entender la programación primero debemos abordar la forma de expresar ideas en forma de código: los lenguajes de programación. Y para entender esos lenguajes debemos entender primero los distintos elementos que podemos encontrarnos cuando construimos o leemos o miramos un programa. En este capítulo vamos a empezar a conocer los bloques elementales que conforman cualquier lenguaje de programación. Y también vamos a escribir nuestros primeros programas en un lenguaje de programación.

2.1. Introducción a elementos básicos

La programación de computadoras involucra cierta forma de "expresar código", actividad a la que denominamos programar. La forma de llevar adelante esta tarea es utilizar lenguajes de programación, que permiten escribir código en forma de texto.

Los lenguajes de programación están compuestos por distintos elementos, que se relacionan en el código de distintas maneras. Estos elementos y sus relaciones son similares a las que podemos encontrar en textos escritos en códigos linguísticos ya conocidos, como los lenguajes naturales, con la única diferencia que poseen cierta precisión adicional en su definición. Por ejemplo, en una oración en castellano podemos encontrar sustantivos, adjetivos, verbos, etc., combinados con ciertas reglas. En los lenguajes de programación vamos a encontrar expresiones, comandos, etc., también combinados con ciertas reglas. Una forma de empezar a entender un lenguaje de programación, es primero entender cuáles son las formas básicas de estos elementos. Es importante volver a remarcar que cada elemento en programación está definido de una manera rigurosa en la que no se admite ningún tipo de ambigüedad, como sí puede suceder en el idioma castellano.

Estas formas básicas quizás aparezcan combinadas o distorsionadas en diferentes lenguajes de programación, pero entendiéndolas se pueden también entender las formas complejas. Esto es igual que en los lenguajes naturales.

Al programar, el programador puede escribir directamente el texto, como es el caso del lenguaje que utilizaremos en este libro o los lenguajes de propósito general más conocidos y de nivel profesional, o puede utilizar algún tipo de interfaz visual, como es el caso de las herramientas SCRATCH o ALICE, ambas basadas en la idea de trasladar y enganchar bloques gráficos entre sí. Al usar únicamente la forma en texto podemos escribir combinaciones de palabras que no signifiquen nada en el lenguaje o podemos escribirlas en lugares donde no tenga sentido que aparezcan (somos libres de escribir lo que queramos, como queramos, donde queramos, y no todo lo que escribamos será un programa que pueda ejecutarse, dado que podría no seguir la estructura que define la sintaxis). En cambio, en los programas confeccionados con herramientas visuales como SCRATCH o ALICE, no podemos formar código incorrecto dado que los bloques están en una paleta

y son arrastrados al editor; además la forma en que son enganchados impide que el resultado no sea una secuencia o expresión bien formada). Por esta razón estas herramientas se suelen utilizar en los niveles iniciales (como primaria o primeros años del secundario), ya que facilitan la confección de código. Pero para poder formar programadores de nivel profesional es necesario enfrentar la tarea de escribir el texto a mano y ser capaces de distinguir las combinaciones válidas de las inválidas; es incluso deseable que al programar los programas se escriban y razonen en primer lugar en papel, y recién luego de tenerlos listos pasarlos en la computadora.

Empecemos entonces hablando de los elementos de los lenguajes de programación.

2.1.1. Valores y expresiones

El primero de los elementos comunes a la mayoría de los lenguajes de programación son los *valores*. Los valores pueden representar, por ejemplo, números, colores, nombres de personas, números de teléfono. Como puede observarse, estos elementos representan *datos*, que los programas manipulan y transforman.

Definición 2.1.1. *Un* valor *es una entidad o dato con sus características propias.*

Ejemplos de valores son el número dos, el color rojo, el nombre de mi tío Arturo, etcétera.

Sin embargo, para hablar o escribir valores sin ambigüedad, se utilizan otros elementos llamados *expresiones*, que son la forma en que se describen *valores*. Las expresiones están formadas por símbolos y cadenas de símbolos, y requieren una interpretación para entender qué valor describen.

Puede entablarse una analogía entre las expresiones y los valores, y las palabras del español y las ideas o conceptos a los que hacen referencia. Además, las palabras también se forman a través de símbolos, que específicamente son letras del abecedario más otros signos lingüísticos. Y así como las ideas son necesariamente expresadas a través de palabras, los valores en los programas deben ser descritos a través de expresiones.

Leer con Atención

En síntesis, las expresiones son la forma en que los valores se manifiestan en el código fuente de los programas, y los valores son aquellos conceptos o ideas que dichas expresiones referencian.

Definición 2.1.2. *Una* expresión *es un símbolo o cadena de símbolos que se utiliza para describir, denotar o nombrar un valor específico.*

La idea de denotar puede ser nueva o poco conocida. Por eso creemos conveniente definirla.

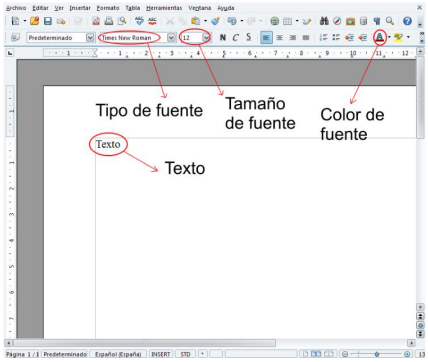
Definición 2.1.3. Denotar *algo es referirse de manera específica a eso, distinguirlo de otros de manera única. Según el diccionario es "significar".*

Por eso usamos denotar como sinónimo de describir, aunque estrictamente no lo sea: cuando describimos algo, estamos refiriéndonos a ello, significándolo. En este sentido es que se dice que las expresiones denotan valores.

Para Reflexionar

Por ejemplo, existen infinidad de formas posibles de describir al número cinco: números arábigos (5), números romanos (V), la cara de un dado (⚄), con fósforos en el truco (🜆), operaciones compuestas con números ($3+2$), y muchas otras. ¿Observa la diferencia entre las distintas expresiones, y el valor cinco? ¿Se da cuenta que con cualquiera de ellas, una persona que conoce el método de intepretación de un símbolo, puede identificar al valor referido?

Ejemplos de expresiones tal como aparecerían en un programa (forma textual),



G.2.1. Ejemplos de valores en el programa Open Office

junto con una descripción en castellano del respectivo valor que denotan (puesto que no podemos presentar los valores sin hablar de ellos en algún lenguaje...), son las que se presentan en la siguiente tabla:

Expresión	Valor
`2`	*número dos*
`5`	*número cinco*
`4+1`	*número cinco*
`Rojo`	*color rojo*
`"Arturo"`	*nombre de persona*

Observar que las expresiones pueden tener formas básicas, *atómicas* (como `5`), o pueden estar formadas por partes (como `4+1`), y sin embargo se refieren al mismo valor. En ese caso, se suele decir que las expresiones son *equivalentes*, y por ende pueden ser intercambiadas a elección.

Para Reflexionar

¿Cuántas formas equivalentes de describir al valor `5` usando números y sumas puede imaginarse? Reflexione sobre la diferencia entre una expresión cualquiera y el valor que describe.

Esta idea de valor de los lenguajes de programación se puede ver también en las interfases de los programas que implementan herramientas gráficas, como editores de texto, etcétera. En dichas herramientas informáticas aparecen valores, aunque las expresiones que se usan para describir estos valores adoptan diferentes formas (íconos, texto, etcétera). En el gráfico G.2.1 podemos visualizar distintos valores, y la forma en que aparecen graficados en un programa (el editor de textos de Open Office). En el ejemplo, los valores son el tipo de la fuente, el tamaño de la fuente, el color de la fuente, y un texto con la palabra "`Texto`". Todos estos valores aparecen de una forma u otra en el código que los programadores de dichas herramientas escriben.

Es un programa libre, de funcionalidad similar a otras herramientas no libres, como Word. Para conocer esta herramienta en mayor profundidad, ver http://www.openoffice.org/es/

Actividad 1

Trate de identificar otros valores que aparecen mostrados en el Open Office. Elija otro programa de uso tradicional (por ejemplo, una planilla de cálculo, un editor de gráficos, etc.), y trate de identificar valores que aparezcan graficados en los mismos.

Las expresiones que podemos utilizar cuando codificamos programas, su forma exacta y la forma en que podemos manipularlas dependerán del lenguaje de programación que estemos utilizando. Las reglas que rigen estas formas (y otras) se

conocen como *sintaxis* del lenguaje de programación. La sintaxis normalmente consiste en reglas rígidas que hay que aprender para cada lenguaje. Al aprender un lenguaje de programación deben aprenderse todas las reglas de sintaxis de cada uno de los elementos, en combinación con qué cosas describe cada uno de ellos (su significado o *semántica*). Volveremos sobre este punto más adelante.

Definición 2.1.4. *La* sintaxis *de un lenguaje de programación es el conjunto de reglas rígidas que establecen con qué símbolos se denotan las diferentes formas de combinación válidas en dicho lenguaje.*

Definición 2.1.5. *La* semántica *de un lenguaje de programación es el significado de cada una de las construcciones sintácticas del lenguaje. Puede definirse matemáticamente, o bien darse a través de un mecanismo de ejecución (en cuyo caso la semántica precisa será implícita).*

Por otra parte, dependiendo del tipo de programa con el que trabajamos, los valores serán en mayor o menor medida el centro de atención. Por ejemplo, de tratarse de un sistema de facturación, los datos registrados por el programa serán de suma importancia, y su correcta conservación representa el objetivo del mismo. Por el contrario, un programa de edición de videos se centrará principalmente en transformar información para producir un producto final, siendo los valores individuales menos relevantes.

2.1.2. Acciones y comandos

Sin embargo, no toda idea es un valor. En muchas situaciones simplemente describir datos no alcanza para que los programas resuelvan problemas. Por esta razón, los programas también describen *acciones*, que son la manera de producir *efectos* o *consecuencias operacionales* sobre diversos elementos, generalmente externos a los programas. Por ejemplo, existen programas que manipulan máquinas en fábricas, impresoras, fotocopiadoras, etc. o que reproducen imágenes y sonidos sobre dispositivos audiovisuales, y también los programas tradicionales como editores de texto o planillas de cálculo, que modifican archivos, imprimen documentos, etcétera. Todos estos programas están produciendo efectos sobre el mundo, a través de modificar los elementos sobre los que operan.

Definición 2.1.6. *Una* acción *es la manera de producir un efecto sobre un elemento u objeto del mundo, ya sea interno o externo al programa en sí.*

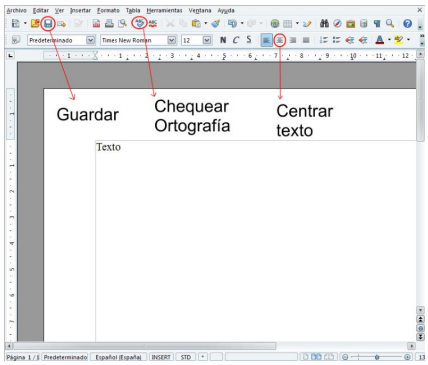
Así como los valores se manifiestan en los lenguajes de programación a través de las expresiones, las acciones son descritas mediante *comandos*. Lo que distingue a los comandos de las expresiones es la idea a la que hacen referencia. Mientras las expresiones referencian cosas (abstractas o concretas), los comandos referencian acciones (formas de producir efectos). Pero tanto expresiones como comandos están formados por cadenas de símbolos. Ejemplos de comandos, con su respectiva acción, tal como podrían aparecer en algunos programas, son:

Existen lenguajes en donde se puede distinguir visualmente cuándo una cadena de símbolos se trata de una expresion o un comando, haciendo que uno empiece con minúsculas y otro con mayúsculas por ejemplo; pero esto no es un requisito esencial y en muchos lenguajes estas construcciones pueden manifestarse bajo la misma forma, e incluso combinadas.

Comando	Acción
`MostrarEnPantalla`	*muestra un dato por pantalla*
`EliminarCarpeta`	*elimina la carpeta seleccionada de nuestro disco duro*
`ApagarComputadora`	*apaga la computadora que estemos utilizando*
`ImprimirArchivo`	*ordena a la impresora que imprima un archivo*
`EnviarMail`	*envía por correo electrónico el texto que redactamos*

Definición 2.1.7. *Un* comando *es una cadena de símbolos que describe una acción específica.*

De la misma forma que los valores, las acciones también se encuentran graficadas en los programas tradicionales. En el gráfico G.2.2 podemos visualizar distintas acciones encontradas en el programa Open Office.



G.2.2. Ejemplos de acciones en el programa Open Office

La mayoría de los lenguajes de programación trabajan con comandos. Al igual que con las expresiones, la forma exacta en que se construyen comandos corresponde a la *sintaxis* del lenguaje, y se rige por las mismas reglas rígidas, que como se mencionó, son una de las cosas a aprender al estudiar un lenguaje de programación. En el caso de los comandos, su significado (su *semántica*) es normalmente entendida a través de la forma en que la máquina que ejecuta el programa lleva a cabo las acciones correspondientes. Esto se conoce como *comportamiento operacional* del comando, y es común no distinguir entre el comando como descripción de la acción y la acción en sí. Sin embargo, en algunos casos es fundamental hacer esta distinción, y por ello es importante conocerla.

Leer con Atención

Un comando no es más que un texto que describe una acción, y como tal, no "hace" nada; solo describe lo que se pretende que una máquina haga. Sin embargo en el lenguaje cotidiano solemos decir que, por ejemplo, el comando `GrabarArchivo` **graba** el archivo, y no que **describe la acción de grabar** el archivo.

Para Reflexionar

Reflexione sobre la naturaleza de los comandos como cadenas de símbolos, y sobre el hecho de que la misma acción puede llegar a describirse de diferentes maneras. ¿Por qué cree que abusamos del lenguaje diciendo que un comando **hace** la acción, cuando en realidad solo la **describe**? Medite cuántas veces en el lenguaje cotidiano no somos exactos con lo que decimos, y sin embargo nos entendemos, y también cuántas veces ese abuso del lenguaje nos conduce a equívocos.

2.1.3. Operaciones sobre expresiones y comandos

De asignaturas de matemática es sabido que podemos realizar operaciones sobre distintas expresiones. Por ejemplo, los números pueden ser sumados, restados y multiplicados; las palabras pueden ser concatenadas (puestas una a continuación de las otras); y los colores pueden ser combinados para formar otros colores. A todas estas operaciones que utilizan expresiones para resultar en otras expresiones las denominaremos *operaciones sobre expresiones*. El resultado de una operación sobre expresiones es siempre una nueva expresión, y por lo tanto puede volver a combinarse. Entonces, podemos escribir $(2 + 3) * 4$ combinando la expresión $2 + 3$ con la expresión 4 a través de la operación de multiplicación.

Expresión	Tipo
`Rojo`	*Color*
`"Casa"`	*Palabra*
`2+4`	*Número*
`Norte`	*Dirección*

G.2.3. Ejemplos de expresiones con sus tipos

Nuevamente, las reglas de sintaxis de un lenguaje establecen qué operaciones son posibles sobre cada expresión, permitiendo de esa forma conocer el total de las expresiones del lenguaje.

Los comandos también pueden ser combinados mediante operaciones. En el caso de los comandos, las operaciones se suelen conocer con el nombre de *estructuras de control*. Por ejemplo, dos comandos pueden ponerse uno a continuación del otro para indicar que luego de terminar la acción descrita por el primero de ellos, debe llevarse a cabo la segunda acción; esta estructura de control se conoce como *secuenciación*. La sintaxis de un lenguaje también establece las posibles estructuras de control y su significado.

Cada lenguaje tiene sus propias estructuras de control. En el lenguaje que aprenderemos en breve veremos las formas más básicas de las mismas, que combinadas son extremadamente poderosas. Sin embargo el repertorio de estructuras de control es amplísimo y debe prestarse atención en cada lenguaje.

2.1.4. Tipos de expresiones

Los valores son parte de distintos conjuntos. Por ejemplo, el *número dos* forma parte de los números naturales, las palabras forman parte del conjunto de todas las palabras y los colores forman parte del conjunto total de colores. Denominaremos *tipo de una expresión* a un conjunto de expresiones determinado, y lo denotaremos mediante algún nombre especial. Diremos, por ejemplo, que la expresión `2` posee tipo *Número*, y el color `Rojo` posee tipo *Color*.

Actividad 2

Dé seis ejemplos más de expresiones utilizando como guía los tipos dados como ejemplo en el gráfico G.2.3

Para Reflexionar

¿Cuál puede ser la utilidad de los tipos en un lenguaje de programación? Piense sobre qué podría significar `Rojo+Norte`, o `"Casa"*"Auto"`.

Cuando veamos el conjunto de operaciones sobre expresiones, en el capítulo 3, subsección 3.2.1, veremos la utilidad que le daremos a los tipos en este libro. Sin embargo, los tipos son útiles en varias formas, que exceden completamente el alcance de este curso. Por otra parte, existen lenguajes con tipos y sin tipos, y dentro de los que utilizan tipos, las construcciones e ideas varían de lenguaje en lenguaje. Pero incluso en su forma básica, los tipos representan una ayuda importante a la hora de pensar un programa.

2.2. Elementos básicos de Gobstones

GOBSTONES fue diseñado e implementado por docentes de la UNQ con el único propósito de servir como lenguaje inicial para aprender a programar. El nombre fue tomado de un juego de bolitas mágicas que aparece en los libros de la saga de Harry Potter, por J.K.Rowling.

Para ejemplificar todas las nociones explicadas hasta el momento, y comenzar a aprender un lenguaje de programación, utilizaremos un lenguaje llamado GOBSTONES, que nos permitirá entender estos conceptos básicos de programación en profundidad, antes de pasar a conceptos de mayor complejidad.

GOBSTONES es un lenguaje conciso de sintaxis razonablemente simple, orientado a personas que no tienen conocimientos previos en programación. El len-

guaje maneja distintos componentes propios, ideados con el fin de aprender a resolver problemas en programación, pero al mismo tiempo intentando volver atractivo el aprendizaje, para lograr captar la atención y capacidad de asombro del estudiante.

Cada concepto relacionado a GOBSTONES será definido de una manera más estricta que la que habitualmente encontraríamos en otras disciplinas.

Para Reflexionar

¿A qué cree usted que se debe la necesidad de ser estrictos, particularidad de la programación? Intente relacionarla con el capítulo 1, en el que explicamos la naturaleza de la programación. Piense específicamente en la necesidad de que los programas ejecuten de manera automática muchas de las tareas que realizan, sin intervención del usuario en estos casos.

El lenguaje GOBSTONES queda definido por el conjunto de reglas de sintaxis y su significado asociado; sin embargo, para poder efectivamente ejecutar programas GOBSTONES, hace falta contar con alguna herramienta informática que lo implemente. Actualmente existen dos herramientas, una básica construída utilizando el lenguaje de programación Haskell, denominada sencillamente GOBSTONES, por haber sido la primera, y otra más elaborada construída utilizando el lenguaje de programación Python, denominada PYGOBSTONES. En este libro utilizaremos PYGOBSTONES; en el anexo A se explica cómo obtenerla y cómo utilizarla.

Para Ampliar

Para conocer más de Haskell, ver

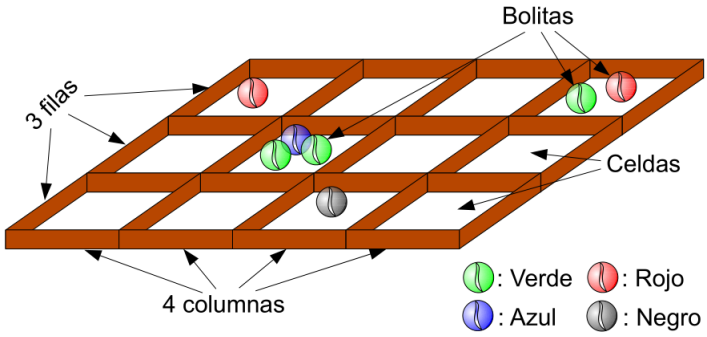
www.haskell.org

Para conocer más de Python, ver

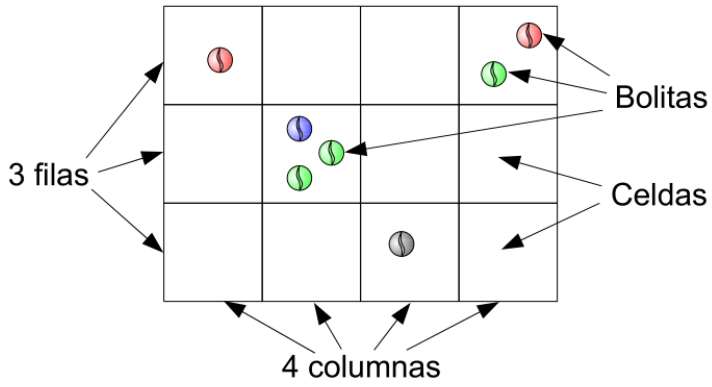
www.python.org

Es importante explicar la diferencia entre un lenguaje y una herramienta o ambiente que implementa y permite el uso de ese lenguaje. El lenguaje es *el conjunto de reglas de combinación* que permiten la construcción de programas válidos (en dicho lenguaje). En cambio, una herramienta o ambiente es un programa específico, casi siempre únicamente ofrecido en formato ejecutable, que asiste al programador en la construcción de programas en uno o más lenguajes dados mediante herramientas como editores de texto con funcionalidades adicionales (resaltado de sintaxis, indicación de posibles errores, etc.) o mecanismos automáticos para ejecutar los programas escritos con la herramienta. Entonces, dado un lenguaje puede haber diversas herramientas que lo implementen, y de manera análoga, una herramienta podría permitir la construcción de programas en diversos lenguajes. GOBSTONES es un *lenguaje* con el que vamos a construir programas elementales, y por lo tanto tiene sintaxis y semántica. En cambio PYGOBSTONES es una *herramienta* que nos ayudará a construir y ejecutar programas escritos en GOBSTONES, para lo cual posee diversas ventanas y botones que brindan diferentes funcionalidades. Lo que puede confundir es que las herramientas, al ser programas, están programadas usando algún lenguaje de programación. En este caso, PYGOBSTONES fue programada en el lenguaje PYTHON. En la mayoría de los casos no es importante con qué lenguaje fue programada una herramienta (e incluso el mismo podría cambiar entre dos versiones de la misma herramienta), pero a veces, como en este caso, influye en las cosas

Recordemos que la sintaxis está conformada por las reglas que permiten combinar elementos del lenguaje para construir programas válidos, y la semántica está conformada por las reglas que dan significado a cada construcción sintácticamente válida.



G.2.4. Un tablero de 3 filas y 4 columnas, en 3 dimensiones



G.2.5. El mismo tablero en 2 dimensiones

que deberemos instalar en nuestra computadora para poder ejecutar PyGobstones, y por eso es importante mencionarlo. Sin embargo, volvemos a remarcar que Gobstones y Python son lenguajes de programación que no tienen nada que ver entre sí, y que PyGobstones es una herramienta implementada usando Python y que implementa el lenguaje Gobstones.

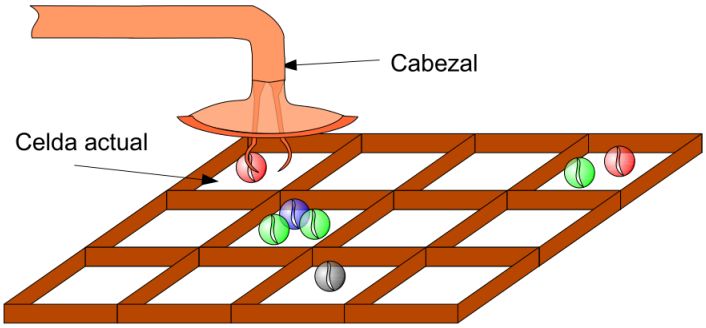
Antes de comanezar con el estudio de la sintaxis de Gobstones es necesario comenzar a comprender este lenguaje presentando algunos de los elementos que existen en su universo de discurso, que es el conjunto de ideas que los programas Gobstones describen a través de la sintaxis del lenguaje.

2.2.1. Tablero y bolitas

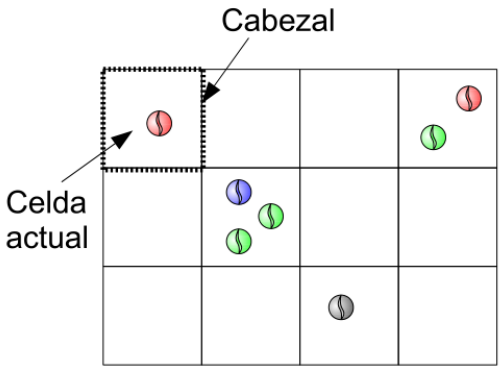
El componente más notable de Gobstones es el *tablero*. El tablero es una cuadrícula de *celdas* dispuestas en filas y columnas. Cada celda es un contenedor en el que puede haber *bolitas* de colores. En la versión actual de Gobstones existen bolitas de cuatro colores: `Azul`, `Negro`, `Rojo` y `Verde`. Las bolitas son simples canicas como las que se usan para el juego de niños del mismo nombre, siendo su color la única característica importante que nos interesa observar (o sea, ignoraremos el tamaño, si se trata de una plumita o una lechera, y lo mismo con otras posibles características).

`Azul`, `Negro`, `Rojo` y `Verde` son expresiones que denotan a los colores de las bolitas

En los gráficos G.2.4 y G.2.5 se observan dos representaciones de un tablero típico, con 3 filas y 4 columnas, y algunas bolitas.



G.2.6. El tablero del gráfico G.2.4 con la representación del cabezal



G.2.7. El mismo tablero en dos dimensiones

Definición 2.2.1. *El* tablero *es una cuadrícula de* celdas*. Las celdas pueden contener* bolitas *de* colores*.*

El tablero es finito, y en un principio no hay una forma directa de saber su dimensión exacta. Por otra parte, las celdas tienen una capacidad ilimitada, pudiendo contenter cualquier número de bolitas.

2.2.2. El cabezal

Tanto el tablero como las bolitas son componentes 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 los gráficos G.2.6 y G.2.7 se observa el tablero presentado antes con una representación del cabezal y la celda actual.

Definición 2.2.2. *El* cabezal *es una máquina que opera sobre una celda por vez, la* celda actual*.*

Para Reflexionar

¿Cuál de los conceptos que hemos visto hasta ahora representará en el código de programas GOBSTONES a las acciones que realiza el cabezal? ¿Y cuál permitirá describir los colores de las bolitas?

Otra característica importante del cabezal, además de que puede operar sobre una celda por vez, es que dispone de una cantidad ilimitada de bolitas de cada color. Esto, al igual que las características del tablero y las celdas, determinará algunas de las formas de hacer cosas en GOBSTONES.

Ahora bien, el cabezal *puede* realizar acciones, pero al no tener voluntad propia, ni ningún tipo de inteligencia, debe indicársele qué es lo que esperamos que haga. Sin embargo, el cabezal no es una máquina que podamos manejar (como un auto o un montacargas), sino que lo único que podemos hacer es darle una descripción de las acciones que queremos que haga, y decirle *"dale, ejecutalas"*. La forma de comunicarle al cabezal la descripción de las acciones que queremos que realice es un *programa*. Así, todo el objetivo de un programa GOBSTONES es servir para indicar al cabezal que lleve a cabo determinada serie de modificaciones al tablero, con algún propósito determinado. Para ello, precisamos los elementos del lenguaje.

2.2.3. Programas y 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, o sea, interpretar las acciones descritas allí y llevarlas a cabo sobre el tablero.

Un programa está formado por comandos, que, como ya mencionamos, son descripciones de acciones individuales; más adelante veremos que también puede llevar un conjunto de definiciones. O sea, los comandos por sí mismos no son programas completos, sino que integran unidades más grandes, denominadas *programas* y *procedimientos*. Todo programa contendrá, al menos, una serie de comandos que serán los que el cabezal intentará llevar a cabo al ejecutar dicho programa.

Los procedimientos serán presentados en la sección 2.2.8, y explicados en profundidad en el capítulo 3.

Definición 2.2.3. *Un* programa GOBSTONES *es una descripción de las acciones que el cabezal intentará realizar al ejecutar el mismo. Esta descripción se brinda a través de* comandos*.*

¡Estamos empezando a aprender la sintaxis específica de un lenguaje de programación!

Entonces, para poder construir programas, primero debemos conocer algunos comandos. El primer comando de GOBSTONES que vamos a presentar es el comando `Poner`. Un caso particular de este comando es el comando `Poner(Verde)`, que describe la acción de agregar una bolita verde en la celda actual. Esta acción siempre podrá ser llevada a cabo, pues, como ya se mencionó, se asume que el cabezal tiene a su disposición una cantidad ilimitada de bolitas de cada color y, que las celdas tienen capacidad infinita, con lo cual la acción de poner siempre puede realizarse.

Para definir un programa debemos colocar el comando recién visto de forma tal que el cabezal, al recibir la orden de ejecutar el programa esté en condiciones de decodificarlo y ejecutarlo. Para ello, se utiliza una "palabra especial", `program`, que indica al cabezal que a continuación, encerrados entre llaves (`{` y `}`), estarán los comandos que describen las acciones que deberá intentar llevar a cabo al recibir la orden de ejecutar el programa. Esta "palabra especial" recibe un nombre específico en el tratamiento de lenguajes de programación: *palabra reservada*. Así, el programa GOBSTONES más sencillo que podemos escribir será

```
program
 { Poner(Verde) }
```

Leer con Atención

El primer programa en la mayoría de los cursos tradicionales de programación es algo del tipo "¡Hola, mundo!", donde se imprime algo por pantalla. Sin embargo, en GOBSTONES no existe ningún mecanismo de entrada/salida explícito; la única interacción que se tiene con la máquina es la provisión de un tablero inicial. Esto es absolutamente premeditado, puesto que las nociones de entrada/salida son extremadamente complejas, al estar asociadas de manera indisoluble al estado del "mundo concreto" (o sea, lo que constituye nuestro ambiente cotidiano, pero que no es necesariamente el ambiente de computación de la máquina). Es mucho mejor diferir toda noción de entrada/salida hasta el momento en que se maneje con fluidez la noción de estado y de efectos sobre el estado. En particular, en la secuencia didáctica que proponemos en este libro hemos preferido no incluir ese tema. Creemos que intentar enseñar conceptos básicos mezclados con conceptos avanzados es uno de las mayores dificultades asociadas a la enseñanza de la programación.

Al indicarle al cabezal que ejecute el programa, se ejecutarán todos los comandos encerrados entre las llaves que aparecen después de la palabra reservada `program`.

Las *palabras reservadas* o *palabras clave* son nombres especiales de los lenguajes de programación que sirven para identificar diversas ideas en el programa. Cada lenguaje define su propio conjunto de palabras reservadas, que no pueden usarse, dentro de ese lenguaje, para ningún otro propósito que aquel para el que fueron pensadas.

En inglés, *reserved words* o *keywords*.

Definición 2.2.4. *Las* palabras reservadas *son nombres especiales de los lenguajes de programación que sirven para identificar diversas ideas en el programa.*

Definición 2.2.5. *Un* programa GOBSTONES *consiste de la palabra reservada* `program` *seguida de un bloque de comandos (encerrados entre llaves* `{` *y* `}`*). Este bloque de comandos determina completamente el comportamiento del cabezal al ejecutar el programa.*

En el caso del programa del ejemplo, describe la sencilla acción de que el cabezal debe intentar poner una bolita de color verde al recibir la instrucción de ejecutar el programa. En la sección 2.2.6 hablaremos más de cómo hacer esto, de lo que significa y de cómo se puede visualizar el resultado del programa. Pero antes de estar en condiciones de comenzar a realizar actividades de programación, vamos a profundizar un poco más sobre comandos básicos.

2.2.4. El comando `Poner`

Ya vimos una forma particular del comando `Poner`: `Poner(Verde)`. Sin embargo, no es la única. La forma general del comando `Poner` está dada por la siguiente definición.

Observar que el nombre *<color>* se encuentra en diferente formato (y color) 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.

Definición 2.2.6. *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`. 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 2.2.7. *Los valores posibles para un <color> son* `Verde`, `Rojo`, `Azul` *o* `Negro`.

2.2.5. Secuencias y bloques de comandos

¿Qué debe hacerse para describir la indicación al cabezal que debe poner más de una bolita en la celda actual? Dicha descripción debe contener dos comandos seguidos. La manera de colocar seguidos comandos en GOBSTONES es ponerlos uno a continuación del otro; por ejemplo

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

es un fragmento de programa que describe que al ejectuarlo, el cabezal debe colocar dos bolitas verdes en la celda actual.

Opcionalmente, pueden colocarse un punto y coma después de un comando, y en ese caso no es necesario colocar los comandos en líneas separadas. Así, los programas

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

y

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

también son válidos, y equivalentes al primero.

Esta idea de poner comandos uno a continuación del otro es una *estructura de control* elemental, y se conoce con el nombre de *secuenciación*. La secuenciación de dos comandos es un comando compuesto que describe la acción consistente en realizar las acciones más simples descritas por los comandos secuenciados, en el orden dado.

Definición 2.2.8. *La* secuenciación *de comandos se obtiene colocando los mismos uno a continuación del otro, normalmente en líneas separadas, y opcionalmente terminados con el caracter de punto y coma (*`;`*).*

Entonces, si queremos hacer un programa que ponga dos bolitas verdes, debemos escribir

```
program
 {
   Poner(Verde)
   Poner(Verde)
 }
```

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 secuencia 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)
}
```

que describe una acción, cuyo efecto al ejecutarse sería colocar dos bolitas de cada color en la celda actual.

Definición 2.2.9. *Un* bloque *es comando formado por un grupo de comandos delimitados por llaves (caracteres* `{` *y* `}`*).*

Los bloques tendrán importancia en la definición de comandos compuestos a través de ciertas estructuras de control y, como vimos, en la definición de un programa.

En este punto somos capaces de construir comandos cuyo único efecto es poner muchas bolitas de muchos colores y con ellos programas sencillos. Luego de ver a continuación cómo ejecutar programas, iremos enriqueciendo el repetorio de formas de construir descripciones de las acciones que queremos que el cabezal realice en nuestros programas.

2.2.6. Ejecución de progamas

Sabiendo cómo definir programas completos, estamos en condiciones de comenzar a realizar actividades de programación.

Leer con Atención

Las actividades de programación incluyen dos partes: por un lado, debe escribirse el programa en papel, para lo cual primero debe ordenar sus ideas, nombrar los diferentes elementos y estructurar la solución; por otro lado, debe cargarse el programa en alguna herramienta y ejecutarlo, para ver los resultados de manera fehaciente.

Al escribir el programa en papel, lo más importante es validar que las ideas son las correctas, y que están seleccionadas y ordenadas adecuadamente para resolver la tarea encomendada. Sin embargo, el código en papel puede considerarse como un borrador, y puede por lo tanto contener pequeños errores, detalles que pueden dejarse para un mejoramiento posterior porque no hacen a la estructura de la solución. Para poder afirmar que tenemos un programa, debemos poder ejecutarlo en alguna máquina, y verificar que realmente hace aquello para lo que lo construimos. Esta tarea de verificación puede ser muy compleja en el caso de programas complicados, pero en principio nos resultará sencillo.

Leer con Atención

Una herramienta muy importante para asistir al programador en la verificación de los progamas son los *sistemas de tipos* modernos. Un *sistema de tipos* es un conjunto de reglas sobre las partes de un programa, que pueden ser verificadas automáticamente *sin ejecutar el programa*, ayudando a verificar de esta formen la validez de ciertas construcciones. Si el sistema de tipos está bien definido (lo cual es sorprendentemente complejo), muchísimos de los errores comunes de los programas son detectados *antes* de realizar ninguna ejecución. Este hecho no siempre es bien comprendido, y se tiende a considerar a los sistemas de tipos como una restricción innecesaria, en lugar de como la gran herramienta que constituyen.

En GOBSTONES puede existir una herramienta de chequeo basada en un sistema de tipos. De hecho, las implementaciones actuales de las herramientas que implementan GOBSTONES (como PYGOBSTONES), poseen esta opción. Volveremos sobre este tema en el capítulo 3.

Repasemos la idea de un programa, antes de ver cómo ejecutarlo. Un *programa* es un bloque especial, identificado con la palabra reservada `program`. Pode-

mos entonces refinar la definición general que vimos en definición 2.2.5, usando la forma que vimos para el comando `Poner`.

Definición 2.2.10. *Un* programa *tiene la forma*

```
program
  <bloque>
```

siendo <*bloque*> *un bloque cualquiera.*

Para poder ejecutar un programa debemos utilizar alguna herramienta que nos permita interpretarlo y realice las acciones correspondientes.

Leer con Atención

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, no daremos detalles de operación de ninguna de ellas como parte integral del texto. En el anexo A se pueden encontrar detalles de cómo obtener y utilizar una de tales herramientas.

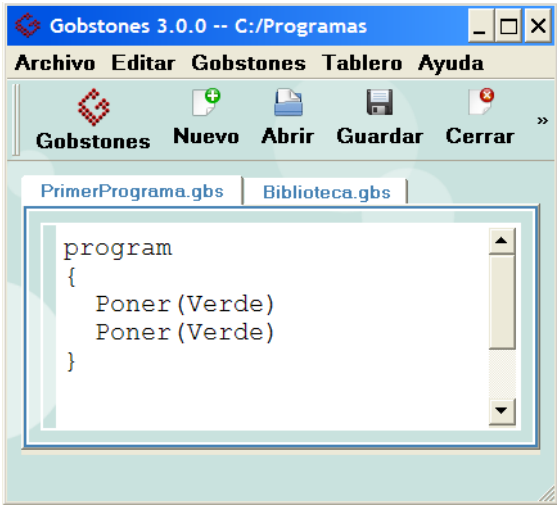
Al volcar el programa escrito en papel en la computadora, sin embargo, la precisión es fundamental, porque si no escribimos la forma exacta, la máquina fallará en reconocer el programa. Estos errores cometidos al escribir un programa en la computadora se conocen como *errores de sintaxis*, y tienen que ver con la rigidez que mencionamos en secciones anteriores. Por ejemplo, no es lo mismo escribir `program` que `porgram` o `programa`. El primero será reconocido adecuadamente por la máquina, pero, por tratarse de procesos automáticos, es mucho más complejo que la máquina reconozca que nuestra intención al escribir `porgram` fue realmente la de escribir `program` (aunque en los casos más simples esto sea factible, en casos más complicados es prácticamente infinita la cantidad de posibles interpretaciones).

Este tipo de errores es similar a las faltas de ortografía y de gramática cometidas al escribir en castellano, con la diferencia de que la máquina no analiza nuestra intenciones como haría un lector humano.

Leer con Atención

Cada herramienta dará un mensaje de error diferente cuando encuentre código que no sigue las reglas precisas de sintaxis. La interpretación de este mensaje de error puede resultar más o menos compleja. La comprensión de los errores que aparecen requiere experiencia con una herramienta en particular, pues los mensajes varían drásticamente de una a otra, e incluso de una versión a otra de la misma herramienta. **Uno de los puntos más frustrantes al aprender a programar es justamente la interpretación de mensajes de error.** Esto sucede porque los que programan los mensajes de error presuponen que el programador sabe de antemano lo que el mensaje indica, aunque en realidad el estudiante puede aún no haber aprendido los conceptos para comprenderlo.

La sugerencia general al enfrentar errores de sintaxis en una herramienta es observar en qué punto del programa se informa el error, y revisar por la precisión de lo que se escribió, recordando todas las reglas vistas. Es una tarea extremadamente tediosa al principio, pero debe serse paciente y persistente, pues con un poco de práctica, es relativamente simple descubrir los errores más obvios de un vistazo. La razón por la que pusimos tanto énfasis al comienzo en la precisión, y la razón por la cual volvemos a remarcarlo acá, es justamente que la mayor causa de fracasos en los intentos de aprender a programar vienen de la diferencia en dificultad entre pensar y escribir un programa para nosotros mismos pensando el problema que deseamos resolver (en papel, por ejemplo) y construirlo correctamente para que deba ser ejecutado por computadora.



G.2.8. Texto correcto del programa GOBSTONES del ejercicio 2.2.1

Actividad de Programación 3

Realizar el ejercicio 2.2.1. Por ser el primer ejercicio, la parte de escribirlo en papel ya fue realizada por nosotros. Por lo tanto, debe pasarlo en la computadora y ejecutarlo, para lo cual deberá instalar antes la herramienta, como se describen en el anexo A.

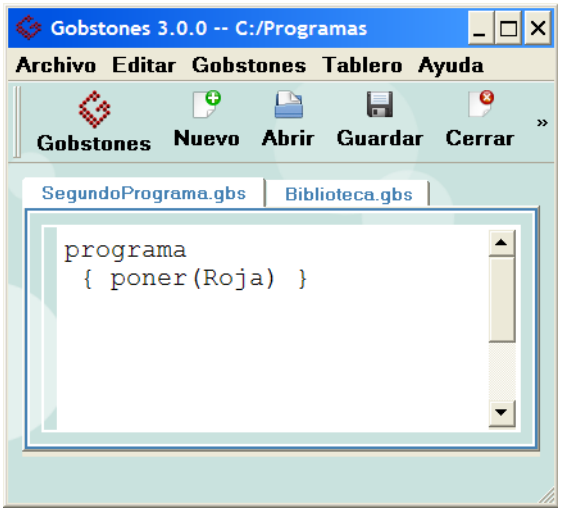
Ejercicio 2.2.1. *Cargar en la herramienta, y ejecutar, el programa* GOBSTONES *dado antes, que deposita dos bolitas verdes en la celda actual. ¿Qué puede concluirse acerca del estado del tablero al iniciar la ejecución del programa? ¿Y al finalizar la misma?*

Recuerde escribir **exactamente** el código como fuera mostrado. Su editor de textos debería verse como en el gráfico G.2.8. Si todo salió de la manera esperada, habrá obtenido un tablero final que en la celda actual tiene dos bolitas verdes más que el tablero inicial. Según la herramienta que esté utilizando, la forma de visualizar este hecho puede variar. Pero en todos los casos sabemos que hay un tablero inicial con dos bolitas verdes menos que el tablero final.

El *estado del tablero* es el conjunto de características que lo conforman: su tamaño, la ubicación del cabezal y si hay o no bolitas, y en 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 2.2.11. *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.*

Antes de pasar a realizar ejercicios donde usted escriba sus propios programas para solucionar tareas sencillas, veremos un ejercicio más, con el fin de trabajar sobre la noción de error.



G.2.9. Texto “correcto” del programa GOBSTONES del ejercicio 2.2.2, al comenzar el ejercicio

Actividad de Programación 4

Realice el ejercicio 2.2.2. Recuerde ser paciente y persistente con los mensajes de error, incluso aunque al principio la tarea pueda resultar algo frustrante.

Ejercicio 2.2.2. *Escribir el siguiente programa* GOBSTONES *y ejecutarlo.*

```
programa
 { poner(Roja) }
```

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

Recuerde que al escribirlo en la herramienta, el programa debe aparecer exactamente como lo escribimos. El editor de texto debería verse como en el gráfico G.2.9.

El informe de los errores en la herramienta PYGOBSTONES es razonablemente adecuado para un novato. Sin embargo, no hay que confiar en que la herramienta detecte siempre nuestras intenciones. Los errores aparecen siempre en los programas, ya sea por descuidos, suposiciones incorrectas o conocimientos insuficientes, y la única manera de encontrarlos y eliminarlos es aprender a interpretar mensajes de error. En todos los ejercicios posteriores no nos será posible mostrar la forma exacta en la que debe verse el editor, pues habrá cientos de opciones correctas – ¡y cientos de miles de opciones incorrectas! Por eso es tan importante que aprenda a manejar sus errores.

Ahora sí, podemos realizar el primer ejercicio completo de programación.

Actividad de Programación 5

Realizar el ejercicio 2.2.3. Recuerde que para poder realizarlo, debe primero escribirlo en papel, y luego pasarlo en la computadora y ejecutarlo.

Ejercicio 2.2.3. *Escribir un programa* GOBSTONES *que deposite una bolita verde, dos rojas, tres azules y cuatro negras en la celda actual, y ejecutarlo.*

Cuando ya se adquiere cierta práctica, la etapa de escribir en papel suele omitirse para los programas más simples. Sin embargo, incluso los programadores más experimentados tomamos una hoja de papel para empezar a organizar nuestras ideas antes de usar la computadora.

Para Reflexionar

Revise todo lo leído hasta el momento, e intente identificar cuáles son las actividades importantes al pensar sobre el papel, y cuáles al escribir el código en la computadora para ejecutarlo. Relacione esto con la definición de *programa* dada, y observe qué cosas tienen que ver con la parte de los programas en cuanto *descripciones* y qué cosas con la parte en cuanto descripciones *ejecutables*. La tarea del programador comienza por pensar lo que quiere describir, y luego darle forma ejecutable...

2.2.7. Más comandos simples

¿Qué otras órdenes elementales se le pueden dar al cabezal? 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 2.2.12. *El comando* `Sacar(`*<color>*`)` *indica al cabezal que quite una bolita del color <color> de la celda actual, si hay alguna.*

No olvidar que esto siginifica *un color* en particular, y no la palabra `color`...

Debe observarse que el cabezal solo 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. Si bien las personas ante tal situación podemos tomar decisiones, las computadoras no pueden. 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 tal comando describe una *operación parcial*, pues solo 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 2.2.13. *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 2.2.14. *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 de que los mismos no se cumplan, provoca la autodestrucción del cabezal y todos los elementos.*

Definición 2.2.15. *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.*

Recordar que los comandos **describen** acciones y no que **hacen** acciones. Tener en cuenta este abuso del lenguaje al leer el resto del libro.

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. En el caso de operaciones parciales, normalmente se indicará solamente la acción descrita por el comando en el caso en que la precondición sea verdadera. Por ejemplo, diremos que `Sacar` saca una bolita de la celda actual; si la bolita no existiese, la precondición sería falsa y por lo tanto tal situación no se indica. En algunos casos, podremos hacer la salvedad aclarando la precondición en la acción: `Sacar` saca una bolita de la celda actual, *si existe*.

Actividad de Programación 6

Realizar los ejercicios 2.2.4 y 2.2.5. Observe que si en el estado inicial del tablero ya había bolitas, el programa del ejercicio 2.2.4 no fallará. Pruebe con un tablero que no tenga bolitas en la celda actual.

Ejercicio 2.2.4. *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 2.2.5. *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? Piense solo en los elementos presentados hasta el momento. . .*

Toda vez que se utiliza una operación parcial, hay que tomar una de dos medidas: o bien se busca garantizar la precondición de la misma de alguna manera (como en el ejercicio 2.2.5) o bien se trasladan los requerimientos de dicha operación a la operación que se está definiendo. Veremos esto con cuidado más adelante.

Para Reflexionar

Los errores en los programas de computadora son algo que sufrimos cotidianamente. Basta pensar en las veces que escuchamos “se cayó el sistema”. Es responsabilidad de los programadores programar de manera que la cantidad de errores sea mínima. Las precondiciones son uno de los mecanismos más útiles para lograr esto. Reflexionar sobre la importancia de aprender a programar de manera adecuada, y sobre lo simple que resulta programar sin tener estas cosas en cuenta. Relacionarlo con el estado de situación de los programas que conoce.

Otro comando que describe una operación parcial es el que permite al cabezal moverse por el tablero. Esta operación es básica para poder operar en más de una celda. 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 2.2.16. *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 2.2.17. *Los valores posibles para una dirección 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

Comando	Precondición	Acción descrita
`Poner(`<*color*>`)`	Ninguna (operación total)	Pone una bolita del color indicado en la celda actual
`Sacar(`<*color*>`)`	Que exista una bolita del color indicado en la celda actual	Saca una bolita del color indicado de la celda actual
`Mover(`<*dir*>`)`	Que exista una celda en la dirección indicada	Mueve el cabezal una celda en la dirección indicada respecto de la celda actual
`IrAlBorde(`<*dir*>`)`	Ninguna (operación total)	Mueve el cabezal al borde indicado del tablero, en la misma fila o columna
`VaciarTablero()`	Ninguna (operación total)	Elimina todas las bolitas del tablero

T.2.1. Sumario de los comandos básicos conocidos hasta el momento

con esta característica. Observar que dado que el tamaño del tablero no se conoce *a priori*, puede ser que un programa que no controle adecuadamente sus movimientos provoque la autodestrucción del cabezal. Por ejemplo, el programa

```
program
 { Mover(Sur); Mover(Oeste) }
```

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 `IrAlBorde`. Este comando, cuya forma es `IrAlBorde(`<*dir*>`)`, describe una operación total cuyo efecto es posicionar en el borde del tablero indicado, en la misma fila o columna en que se encuentra originalmente. Posteriormente, en el capítulo 4 se verán las herramientas necesarias para que este comando pueda ser reemplazado por un procedimiento equivalente definido por el programador.

Definición 2.2.18. *El comando* `IrAlBorde(`<*dir*>`)` *indica al cabezal que se posicione en el borde del tablero indicado por la dirección* <*dir*>*, en la misma fila o columna en que se encontraba.*

No olvidar que esto siginifica *una dirección* en particular, y no la palabra `dir`...

Asimismo, y puesto que el contenido de las celdas es arbitrario al iniciar, 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 `IrAlBorde`, el comando `VaciarTablero` puede ser reemplazado por un procedimiento definido por el programador, aunque para ello se requiere conocer casi la totalidad de elementos que posee el lenguaje.

Definición 2.2.19. *El comando* `VaciarTablero()` *indica al cabezal que quite todas las bolitas del tablero, dejando un tablero final vacío.*

En este punto, es bueno hacer un pequeño repaso de los comandos vistos hasta el momento, y sus características. Esto se puede encontrar en la tabla T.2.1. Estos comandos pueden ser combinados mediante *secuenciación*, agrupados en *bloques*, utilizados en un programa y, como veremos en breve, usados en definiciones de procedimientos que determinarán nuevos comandos, definidos por el programador.

Es el momento de comenzar a escribir algunos programas más interesantes.

2.2.8. Procedimientos simples

Combinando las operaciones de `Mover` y `Poner` se pueden simular "dibujos" utilizando las bolitas como colores. Por ejemplo, ¿cómo podemos decir la tarea que lleva a cabo el siguiente programa?

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

En lugar de decir que pone 8 bolitas negras en diferentes lugares, es más fácil decir que dibuja un pequeño cuadrado negro de tres celdas de lado.

Recordar que en realidad sería correcto decir que el programa describe la tarea de dibujar un cuadrado negro.

Actividad de Programación 7

Realizar el ejercicio 2.2.6. Probarlo varias veces, con el cabezal en distintas posiciones, y comprobar que este programa describe una operación parcial.

Ejercicio 2.2.6. *Ejecutar el programa Gobstones dado antes, y comprobar que dibuja el cuadrado. ¿Cuál es la precondición de este programa?*

Leer con Atención

Si bien GOBSTONES no es un lenguaje para expresar dibujos, y el tablero no es un papel o una pantalla, podemos interpretar el problema de "dibujar un cuadrado" en términos del tablero y bolitas. Esta interpretación es una de las maneras fundamentales de representar cosas en los lenguajes de programación: se utilizan ciertos elementos, pero se entienden como si se tratase de otros elementos.

Para los que conocen la película "Matrix", en programación solemos decir que debemos *"tratar de entender la verdad... la cuchara no existe..."* y ese es el primer paso para poder doblarla.

Entonces, a través de algunos dibujos bien coordinados podemos ver *ventanas*, *botones* y otros elementos abstractos en una pantalla de computadora, aunque solo se trate de puntitos de luz que los programadores podemos controlar. Al enunciar ejercicios haremos uso intuitivo de esta forma de interpretación. En la sección 2.3 profundizaremos un poco la idea de representar información en términos de los elementos presentes en el universo de discurso de un lenguaje.

Ahora supongamos que queremos dibujar 2 cuadrados, uno a la izquierda (al `Oeste`) del tablero y otro a la derecha (al `Este`), ambos lo más abajo posible (al `Sur`). Con los elementos que tenemos hasta ahora es posible hacerlo, resultando en el siguiente código:

```
program
 {
   VaciarTablero()

   IrAlBorde(Sur); IrAlBorde(Oeste)
   Mover(Norte); Poner(Negro); Mover(Norte); Poner(Negro)
   Mover(Este);  Poner(Negro); Mover(Este);  Poner(Negro)
   Mover(Sur);   Poner(Negro); Mover(Sur);   Poner(Negro)
   Mover(Oeste); Poner(Negro); Mover(Oeste); Poner(Negro)

   IrAlBorde(Este); Mover(Oeste); Mover(Oeste)
   Mover(Norte); Poner(Negro); Mover(Norte); Poner(Negro)
```

G.2.10. Resultado de ejecutar el programa dado como ejemplo

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

El resultado de ejecutar este programa dará un tablero final como el de la gráfico G.2.10 (el tamaño del tablero puede variar en función de la herramienta, y en caso de ser un tablero demasiado chico, puede que la ejecución falle, produciendo la autodestrucción del cabezal). Ahora, analicemos el código. Primero podemos observar que se indica vaciar el tablero; luego que se indica posicionar el cabezal en la celda más al sur y el oeste (el *origen* del tablero), y luego que está el código que habíamos usado antes para dibujar el cuadrado negro de lado 3; finalmente se vuelve a posicionar el cabezal cerca de la esquina sureste, y se vuelve a repetir el código de dibujar el cuadrado. Sin embargo, ese análisis requiere de una cuidadosa lectura, puesto que el programa tiene ¡38 comandos primitivos! Y solo estamos dibujando un par de cuadrados...

Sería mucho más fácil si el lenguaje GOBSTONES ya tuviese un comando primitivo `DibujarCuadradoNegroDeLado3()`. Solo tendríamos que escribir:

```
program
 {
   VaciarTablero()

   IrAlBorde(Sur); IrAlBorde(Oeste)
   DibujarCuadradoNegroDeLado3()

   IrAlBorde(Este); Mover(Oeste); Mover(Oeste)
   DibujarCuadradoNegroDeLado3()
 }
```

¡Sin embargo, no es posible pedir que el lenguaje traiga predefinidos **todos** los comandos que se nos puedan ocurrir! Lo que sí podemos pedirle al lenguaje es un mecanismo por el cual podamos definir nuestros propios comandos. La idea es escribir algo como

```
‘‘definir comando’’ DibujarCuadradoNegroDeLado3() ‘‘como’’
 {
   Mover(Norte); Poner(Negro); Mover(Norte); Poner(Negro)
   Mover(Este);  Poner(Negro); Mover(Este);  Poner(Negro)
   Mover(Sur);   Poner(Negro); Mover(Sur);   Poner(Negro)
   Mover(Oeste); Poner(Negro); Mover(Oeste); Poner(Negro)
 }
```

para que el lenguaje use esas 16 instrucciones de `Mover` y `Poner` cada vez que alguien use el comando `DibujarCuadradoNegroDeLado3()`, y entonces el problema estaría resuelto.

La herramienta que se utiliza para este tipo de definición es un *procedimiento*. Básicamente, un procedimiento simple es un *comando definido por el programador*, o sea, un comando nuevo, resultante de la combinación de otros comandos más elementales. La manera de definir un comando es, entonces, declarar la existencia de un nuevo *procedimiento*, para lo cual hay que indicar cuál es el nombre que se usará para este nuevo procedimiento, y cuál es el efecto que describirá (el significado del nuevo comando). Este significado se define a través de alguna combinación de comandos más elementales, ya existentes (primitivos, o declarados previamente). La declaración de un procedimiento se indica con la palabra reservada `procedure`, y entonces para declarar el comando `DibujarCuadradoNegroDeLado3()` debe escribirse:

```
procedure DibujarCuadradoNegroDeLado3()
 {
   Mover(Norte); Poner(Negro); Mover(Norte); Poner(Negro)
   Mover(Este);  Poner(Negro); Mover(Este);  Poner(Negro)
   Mover(Sur);   Poner(Negro); Mover(Sur);   Poner(Negro)
   Mover(Oeste); Poner(Negro); Mover(Oeste); Poner(Negro)
 }
```

Las llaves que delimitan el bloque que se coloca luego del nombre del nuevo procedimiento determinan cuáles son los comandos elementales que serán ejecutados cuando el cabezal trate de ejecutar `DibujarCuadradoNegroDeLado3()`. Dicho de otra manera, el comando `DibujarCuadradoNegroDeLado3()` es simplemente otra forma de nombrar a la combinación dada por esos 16 `Mover` y `Poner`.

Identificadores

Para nombrar elementos del lenguaje se utiliza 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 2.2.20. *Un* identificador *es un nombre conformado exclusivamente por letras (mayúsculas o minúsculas) o números, que se utiliza para denotar (o* identificar *o* referise a*) algún elemento.*

La idea de identificador es mayor que la idea de palabra a la que estamos acostumbrados en castellano. En la siguiente lista de ejemplos de identificadores, pueden observarse letras, palabras y combinaciones diversas, todas las cuales constituyen identificadores, pues son cadenas de letras sin separación.

1. `x`
2. `sumar`
3. `Dibujar`
4. `unidentificadormuylargoymuyconfuso`
5. `UnIdentificadorMasEntendible`

6. `UnIdentificadorTanLargoQueNoValeLaPenaNiPensarEnEscribirlo`
7. `DibujarCuadradoNegroDeLado3`

Los ejemplos 1, 2 y 4 comienzan con minúscula. Los ejemplos 3, 5, 6 y 7 comienzan con mayúsculas. Por otra parte, los ejemplos 5, 6 y 7 están escritos en una forma mixta conocida como *camelCase*, que es la que vamos a utilizar en este libro. Existen otras formas de escribir identificadores, y cada programador tiene su propio estilo. Vamos a hablar de cuestiones de estilo en la sección 2.3.

De Wikipedia: "CamelCase es un estilo de escritura que se aplica a frases o palabras compuestas. El nombre se debe a que las mayúsculas a lo largo de una palabra en CamelCase se asemejan a las jorobas de un camello. El nombre CamelCase se podría traducir como Mayúsculas/Minúsculas en Camello."

es.wikipedia.org/wiki/CamelCase

Actividad 8

Escribir al menos otros diez identificadores, que sirvan para describir valores complejos y acciones cotidianas. Por ejemplo, `sumarDosYTres`, `FreirUnHuevo`, etcétera.

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

Definición de procedimientos simples

La forma general de una declaración de procedimiento simple está dada por la siguiente definición.

Definición 2.2.21. *Un* procedimiento simple *es un nuevo comando definido por el programador mediante el mecanismo de darle un nombre a un bloque mediante un identificador. La definición de un procedimiento simple tiene la forma*

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

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

¡Recordar que la notación quiere decir que esto debe reemplazarse por un identificador que nombre al procedimiento, y no por la palabra `procName`!

Los paréntesis son necesarios; su utilidad se comprenderá en el capítulo 3, donde se definirán formas más complejas de procedimientos. Por el momento solo se verán las formas más simples de los mismos.

Leer con Atención

Un procedimiento simple es un bloque al que se le pone nombre. Ese nombre podrá ser utilizado como un comando.

Un ejemplo de procedimiento definido (o declarado) por el programador sería

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

Es importante observar que un procedimiento definido por el programador por sí solo NO constituye un programa. La diferencia entre un procedimiento definido por el programador y un programa es que el el programa determina completamente la tarea que el cabezal debe llevar a cabo, mientras que el procedimiento define solo una parte. El programa se usa indicando externamente al cabezal que lo ejecute, pero un procedimiento definido por el programador debe ser invocado de manera explícita como comando por el código del programa para que el mismo sea tenido en cuenta por la máquina. Entonces

```
program
 {
   PonerDosVerdes()
   PonerDosVerdes()
```

```
  }

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

es un programa GOBSTONES que coloca cuatro bolitas verdes en la celda actual. Observemos que la definición del procedimiento `PonerDosVerdes` debe darse a continuación del programa para que el comando `PonerDosVerdes()` tenga sentido durante la ejecución.

Definición 2.2.22. *Un* procedimiento *declarado por usuario puede ser utilizado como* comando*. Se escribe el nombre seguido por paréntesis:*

<*procName*>`()`

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

Si se utiliza un comando que no es primitivo ni fue definido por el programador, el programa fallará.

Observar cómo la tarea de poner dos bolitas verdes ya no queda expresada únicamente por una secuenciación, sino que a través del uso de procedimientos hemos podido darle un *nombre significativo* a esa tarea. Cada vez que precisemos poner dos bolitas verdes podemos recurrir al nuevo comando `PonerDosVerdes()`.

Para que sea realmente un nuevo comando, su definición debe encontrarse entre las definiciones de procedimientos del programa.

Un defecto muy común en la enseñanza de la programación (imperativa) es ignorar completamente a los procedimientos hasta que el estudiante conoce muchas otras formas básicas de comandos. Y esto lleva a escribir programas que son extremadamente difíciles de leer, por estar pensados en demasiado bajo nivel. En el pasado, cuando las computadoras eran lentas y tenían poca memoria, se hacía imprescindible ahorrar los recursos (tiempo y memoria) al máximo, por lo que no resultaba conveniente utilizar muchos procedimientos. Pero hoy en día, cuando los recursos se han ido multiplicando exponencialmente hasta no ser un problema, y donde además la tecnología de procesamiento de programas (construcción de compiladores, análisis estático de código, etcétera) ha evolucionado muchísimo, resulta mucho mejor utilizarlos desde el comienzo para favorecer el pensamiento denotacional, de más alto nivel. Por ello es extremadamente importante aprender a utilizar los procedimientos de maneras que resulten adecuadas a este propósito.

2.2.9. Uso adecuado de procedimientos

Un *algoritmo* es una secuencia bien definida de pasos para alcanzar una tarea. Muchos programadores piensan sus programas siguiendo fielmente esta línea. Si bien la ejecución de los programas se aproxima exactamente hacia esta definición, sugerimos entender a los programas como descripciones precisas construidas por humanos, pese a que sean ejecutadas por máquinas. De esto surgen muchas construcciones abstractas que enriquecen el trabajo de los programadores, aunque carezcan de significado para las máquinas.

Si privilegiamos la secuenciación como forma de combinar comandos, sin utilizar adecuadamente los procedimientos y otras herramientas de abstracción (que sirven para simplificar código, para escribir menos o para que su significado sea mejor entendido por humanos), los programas tendrían que ser pensados de a un comando por vez y focalizados en los efectos que los mismos producen sobre el estado, y así se complicaría el trabajo en equipo que caracteriza a la programación moderna. Esta forma de pensar programas se conoce como *pensamiento operacional*, pues ubica como la herramienta principal de programación a las secuencias de operaciones, y obliga al programador a pensar en sus programas solo en los efectos de la ejecución individual de cada operación, relegando, y muchas veces mal interpretando, el uso de muchas otras formas de abstracción de programas. El enfoque tradicional de enseñanza de la programación (aquel que propone el pensamiento algorítmico por sobre otras abstracciones) privilegia el pensamiento operacional, postergando el aprendizaje de procedimientos para más adelante. El problema con aprender a pensar operacionalmente es que tiene como consecuencia que los problemas complejos sean difíciles de tratar y la programación en equipos sea prácticamente imposible. Es por ello que se

recomienda aprender a pensar los programas como diferentes *tareas* que deben realizarse, agrupándolas de manera conveniente. Para poder pensar con tareas, el lenguaje debe proveernos algún mecanismo; en GOBSTONES el mecanismo utilizado para definir tareas es el de *procedimiento*. Por eso, aprender a utilizar adecuadamente procedimientos es algo que debe hacerse, en nuestra opinión, desde el mismísimo comienzo del aprendizaje de la programación. Fomentar el pensamiento por tareas, en detrimento del pensamiento operacional, es crucial para que el aprendizaje conseguido sea valioso no solo en el corto plazo, sino durante toda la vida del programador, independientemente de la envergadura de la tarea que se le encomiende.

Leer con Atención

Podríamos decir que las tareas son situaciones complejas a resolver, y los procedimientos, la forma de describir tareas (de la misma forma que las acciones son la manera de producir efectos, y los comandos, la forma de describir acciones).

Para Reflexionar

Esta habilidad es fundamental para poder encarar tareas complejas. Uno de los errores más tradicionales en los cursos básicos de programación consiste en mantener el pensamiento operacional demasiado tiempo, pues resulta más sencillo pensar los primeros ejemplos de manera operacional. Sin embargo, esto complica y dificulta luego el aprendizaje posterior. A partir del ejemplo visto, reflexione sobre el altísimo impacto que tiene la capacidad de agrupar detalles e ideas en componentes que se puedan combinar para aumentar la productividad y reducir la complejidad de las descripciones.

Es conveniente, entonces, 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 y se vuelve casi imposible dividir diferentes tareas entre varios programadores de un mismo equipo. Por eso la forma operacional de pensar un programa solo sirve con los programas más simples. Pensar por tareas permite generalizar adecuadamente el programa.

Volvamos al ejemplo del cuadrado. Vimos que era mejor escribirlo utilizando un procedimiento llamado `DibujarCuadradoNegroDeLado3` en lugar de utilizar 38 comandos (resultado del pensamiento operacional). El resultado final fue (vale la pena reiterar el código) el siguiente programa

```
program
 {
   VaciarTablero()

   IrAlBorde(Sur); IrAlBorde(Oeste)
   DibujarCuadradoNegroDeLado3()

   IrAlBorde(Este); Mover(Oeste); Mover(Oeste)
   DibujarCuadradoNegroDeLado3()
 }

procedure DibujarCuadradoNegroDeLado3()
 {
   Mover(Norte); Poner(Negro); Mover(Norte); Poner(Negro)
   Mover(Este);  Poner(Negro); Mover(Este);  Poner(Negro)
```

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

Como podemos ver, el procedimiento `DibujarCuadradoNegroDeLado3()` sigue siendo el resultado del pensamiento operacional. Si tuviéramos que modificarlo para cambiar alguna característica, sería complicado, ya que habría que pensar en cada operación de manera individual. Por ejemplo, si quisiéramos transformarlo en un procedimiento que dibujase un cuadrado de lado 4, ¿qué operaciones habría que agregarle?

Actividad de Programación 9

Realice el ejercicio 2.2.7 y pruébelo dibujando 2 cuadrados como antes. Para ello deberá modificar el programa principal anterior para que el 2do cuadrado se comience a dibujar un lugar más al oeste (pues si no no habrá lugar, y el programa fallará siempre).

Ejercicio 2.2.7. *Modificar el programa anterior cambiando el código del procedimiento* `DibujarCuadradoNegroDeLado3` *escrito usando pensamiento operacional para que dibuje 2 cuadrados de tamaño 4. Cambiar el nombre del procedimiento a* `DibujarCuadradoNegroDeLado4`.

¿Cuántas modificaciones fueron necesarias para realizar el cambio de procedimiento? ¿Cuánto trabajo fue darse cuenta dónde modificar?

Para Reflexionar

A partir de la experiencia de modificar este código, reflexione nuevamente sobre cómo se complicaría el trabajo de programar si solo utilizásemos pensamiento operacional y todos nuestros programas se armasen solamente en base a una larga secuencia de instrucciones, sin distinción de diferentes tareas.

Claramente, podríamos tratar de aplicar la técnica de división en subtareas al dibujo del cuadrado, y expresar su código mediante algunas tareas bien elegidas, para las que se deben definir procedimientos adecuadamente.

Puesto que se trata de dibujar un cuadrado, una forma natural de pensar este problema es mediante dibujar 4 líneas del mismo tamaño. Esto dará lugar a cuatro procedimientos que podrían denominarse

Prestar atención a los identificadores elegidos para nombrar los procedimientos. Discutiremos acerca de esto en la sección 2.3.

- `DibujarLineaNegra2HaciaElNorte`,
- `DibujarLineaNegra2HaciaElEste`,
- `DibujarLineaNegra2HaciaElSur` y
- `DibujarLineaNegra2HaciaElOeste`.

Entonces, el procedimiento de dibujar un cuadrado de lado 3 podría expresarse como

```
procedure DibujarCuadradoNegroDeLado3()
 {
   DibujarLineaNegra2HaciaElNorte()
   DibujarLineaNegra2HaciaElEste()
   DibujarLineaNegra2HaciaElSur()
   DibujarLineaNegra2HaciaElOeste()
 }
```

Cada uno de los procedimientos de dibujar línea se compondrá de comandos `Mover` y `Poner` que dibujen la línea en la dirección indicada. Por ejemplo, el procedimiento `DibujarLineaNegra2HaciaElNorte` quedaría

```
procedure DibujarLineaNegra2HaciaElNorte()
 {
   Mover(Norte); Poner(Negro)
   Mover(Norte); Poner(Negro)
 }
```

Actividad de Programación 10

Realizar el ejercicio 2.2.8 y ejecutar el programa resultante. Observar además que este programa es equivalente al del ejercicio 2.2.6, pero es más fácil de entender.

Ejercicio 2.2.8. *Completar las definiciones de los procedimientos para dibujar líneas y componer todo el código en un único programa que describa la misma tarea que el programa ejemplo que usamos para producir el tablero del gráfico G.2.10, pero de manera tal que el código resultante esté adecuadamente dividido en subtareas (incluído el procedimiento que dibuja el cuadrado).*

Actividad de Programación 11

Realizar el ejercicio 2.2.9 y ejecutar el programa resultante. Observe detenidamente este programa respecto del definido en el ejercicio anterior y piense cuántas modificaciones precisó.

Ejercicio 2.2.9. *Modificar el ejercicio 2.2.8, realizado utilizando división en subtareas, para que realice dos cuadrados de lado 4.*

Para Reflexionar

Reflexione sobre la utilidad de los procedimientos, y cómo éstos son absolutamente necesarios una vez que se encaran tareas mínimamente no triviales. Reflexione sobre los cambios necesarios en los procedimientos para realizar una tarea diferente y en cómo es sencillo determinar en qué lugares realizar los cambios si los procedimientos fueron definidos y nombrados adecuadamente.

Observar que el programa con procedimientos es más extenso en este caso que el que sigue el pensamiento operacional. Sin embargo, ahora estamos en condiciones de generalizarlo de varias maneras. Por ejemplo, cambiando solamente los procedimientos que dibujan líneas y copiando los restantes procedimientos sin alteraciones, podríamos hacer diferentes procedimientos para dibujar otros cuadrados de tamaño más grande.

Observar que se trata de dibujar solo los lados del cuadrado. O sea, no se trata de un cuadrado sólido.

Esta forma de dividir un programa en tareas y definir un procedimiento para cada una de ellas hace que el foco de atención en programación sea puesto en los procedimientos individuales. De esta forma, en lugar de pedir un programa que haga cierta tarea, se pedirá la definición de un procedimiento particular que realice la tarea. Por ejemplo, el ejercicio de dibujar un cuadrado se va a enunciar de la siguiente manera:

Ejercicio 2.2.10. *Escribir un procedimiento* `DibujarCuadradoNegroDeLado3` *que dibuje un cuadrado negro de tres celdas de lado, utilizando una adecuada división en subtareas.*

Leer con Atención

Al enunciar un problema se asume que el requerimiento de escribir el programa es implícito, o sea, debe crearse un programa que invoque al procedimiento solicitado en el enunciado, aunque tal cosa no esté pedida de manera explícita en ese enunciado. Lo mismo sucede con los procedimientos auxiliares, necesarios para realizar una adecuada división en subtareas de la tarea encomendada: no se enuncian, pero deben realizarse. Es tarea del programador determinar cuáles subtareas resolver con qué procedimientos auxiliares.

Además, en ocasiones serán necesarios varios procedimientos diferentes combinados adecuadamente para llevar a cabo alguna tarea. En ese caso, se presentará y discutirá cada uno de ellos por separado (en diferentes ejercicios), dejando su combinación implícita o como parte de algún ejercicio final. O sea, siempre se asumirá que los ejercicios previos han sido realizados, y pueden por lo tanto reutilizarse.

Como ejemplo de cómo los enunciados mencionarán solamente el procedimiento que se pide, pero no otros que pueden resultar necesarios, consideremos el siguiente enunciado.

Ejercicio 2.2.11. *Escribir un procedimiento llamado* `CuadradoNegroARojo3` *que suponiendo que hay un cuadrado Negro de lado tres a partir de la celda actual, lo transforme en un cuadrado Rojo.*

La solución para este problema sería dada de la siguiente manera

```
procedure TransformarCelda()
 { Sacar(Negro); Poner(Rojo) }

procedure TrasformarLinea2HaciaElNorte()
 {
   Mover(Norte); TransformarCelda()
   Mover(Norte); TransformarCelda()
 }
procedure TrasformarLinea2HaciaElEste()
 {
   Mover(Este); TransformarCelda()
   Mover(Este); TransformarCelda()
 }
procedure TrasformarLinea2HaciaElSur()
 {
   Mover(Sur); TransformarCelda()
   Mover(Sur); TransformarCelda()
 }
procedure TrasformarLinea2HaciaElOeste()
 {
   Mover(Oeste); TransformarCelda()
   Mover(Oeste); TransformarCelda()
 }
procedure CuadradoNegroARojo3()
 {
   TrasformarLinea2HaciaElNorte()
   TrasformarLinea2HaciaElEste()
   TrasformarLinea2HaciaElSur()
   TrasformarLinea2HaciaElOeste()
 }
```

El primer detalle a observar es que si bien el procedimiento solicitado es solamente `CuadradoNegroARojo3`, la solución involucra la definición de cinco pro-

cedimientos diferentes. Esto no es parte del requerimiento original explícito, pero una buena práctica de programación se consigue eligiendo convenientemente las tareas a describir, y nombrándolas adecuadamente (volveremos sobre el punto de elección de nombres en la sección 2.3).

Otro detalle a observar tiene que ver con cómo probar este procedimiento. Puesto que la precondición de `CuadradoNegroARojo3` es que exista un cuadrado negro de lado tres ya dibujado en el tablero, no tendría sentido utilizar como programa

```
program
 {
   VaciarTablero()
   IrAlBorde(Sur);IrAlBorde(Oeste)
   CuadradoNegroARojo3()
 }
```

pues este programa siempre fallaría. Se requiere, en cambio, asegurar primero la precondición de `CuadradoNegroARojo3`, utilizando alguna otra forma. Una de ellas, por ejemplo, es utilizar algún otro procedimiento para preparar el tablero antes de llamar al procedimiento a probar, como en

```
program
 {
   VaciarTablero()
   IrAlBorde(Sur);IrAlBorde(Oeste)
   DibujarCuadradoNegroDeLado3()
   CuadradoNegroARojo3()
 }
```

Otra de las formas, si la herramienta lo permite, es tener un tablero predefinido donde el cuadrado negro ya ha sido dibujado a partir del origen, y entonces usar como programa simplemente:

Prestar atención a que suponemos que al ejecutar este programa NO estamos usando cualquier tablero, sino uno que ya contiene un cuadrado negro del tamaño correcto dibujado en el lugar correcto.

```
program
 {
   IrAlBorde(Sur);IrAlBorde(Oeste)
   CuadradoNegroARojo3()
 }
```

Pero como esto es dependiente de cada herramienta, no nos extenderemos en estas alternativas. Será tarea del estudiante buscar la forma de probar los procedimientos que se solicitan en las actividades de programación. En el caso que buscamos ejemplificar, la solución elegida será la de utilizar el procedimiento `DibujarCuadradoNegroDeLado3` como en el programa del medio dado antes.

Actividad de Programación 12

Realice el ejercicio 2.2.12. Observe que para que el mismo funcione en cualquier herramienta su programa debe contener al menos 10 definiciones de procedimientos además del programa principal (y no solo los 5 correspondientes al ejercicio 2.2.11).

Ejercicio 2.2.12. *Escribir un programa que utilice* `CuadradoNegroARojo3`*, y que, si el tablero es lo suficientemente grande, no falle, y culmine con un cuadrado rojo en el tablero.*

Observar que en el ejercicio 2.2.12 no se indica cómo debe conseguirse que al invocar el procedimiento `CuadradoNegroARojo3` se cumpla su precondición. La precondición de un procedimiento solicitado debe satisfacerse de alguna manera

para poder probarlo. O sea, en general el ejercicio 2.2.11 implicará que debe hacerse también el ejercicio 2.2.12, 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.

Procedimientos de uso común

Un detalle para observar es que en todos los programas del último ejemplo (el de pasar un cuadrado negro a rojo) utilizamos reiteradas veces la combinación de comandos

```
IrAlBorde(Sur);IrAlBorde(Oeste)
```

Esta combinación nos permite ir a la esquina inferior izquierda del tablero, lo que en un diagrama de ejes cartesianos llamaríamos *el origen* de coordenadas. Para no tener que repetir esta combinación podemos utilizar un procedimiento, según el mecanismo de división en subtareas que aprendimos en esta sección.

Actividad de Programación 13

Realice el ejercicio 2.2.13 y modifique los ejemplos anteriores para que utilicen este procedimiento en lugar de la combinación de dos comandos para ir al borde.

Ejercicio 2.2.13. *Definir un procedimiento* `IrAlOrigen` *que lleve el cabezal a la celda más al sur y el oeste del tablero (el* origen *del tablero).*

Este procedimiento es extremadamente probable que sea utilizado en la mayoría de los programas que escribamos. Eso significará que, en ausencia de alguna herramienta que nos permita no tener que volver a definir procedimientos cada vez, en cada programa que hagamos vamos a tener que volver a definirlo copiando el código de manera idéntica. ¡Pero toda la idea de definir procedimientos era no tener que repetir una y otra vez las mismas combinaciones! Para eso GOBSTONES provee otra herramienta simple: una *biblioteca*. Una *biblioteca* de operaciones es un conjunto de definiciones que se utilizan una y otra vez en diferentes programas. La mayoría de los lenguajes provee mecanismos complejos para la definición e inclusión de diferentes bibliotecas de operaciones en nuestros programas. Como no es la intención de GOBSTONES presentar herramientas complejas, el concepto de biblioteca que presenta es extremadamente sencillo: si existe un archivo de nombre `Biblioteca` en el mismo lugar que el archivo que contiene el programa (usualmente la misma carpeta o directorio en el sistema de archivos), GOBSTONES indica que ese archivo también debe ser tenido en cuenta a la hora de buscar definiciones, sin tener que indicarlo de ninguna manera explícita.

El término en inglés es *library*, que usualmente se traduce (incorrectamente) como *librería*; sin embargo, la traducción correcta es *biblioteca*, y por eso es la que usaremos.

Definición 2.2.23. *Una* biblioteca *de operaciones es un conjunto de definiciones que se pueden utilizar en diferentes programas. En* GOBSTONES *se consigue a través de un archivo de nombre* `Biblioteca` *que se coloca en el mismo lugar que el archivo que contiene el programa (la misma carpeta o directorio).*

En esta biblioteca conviene poner principalmente definiciones que sean tan generales que puedan utilizarse en casi cualquier programa. El primer procedimiento que cumple con esta característica es, justamente, `IrAlOrigen`. A medida que vayamos avanzando con los ejercicios, surgirán otros procedimientos que podrán ser incorporados en la biblioteca; en cada caso esto será indicado de manera explícita. Además, si al realizar procedimientos surgiera alguno de uso común que no haya sido indicado aquí para ir a la biblioteca, ¡no hay que dudar en colocarlo también! La experimentación es la base del aprendizaje.

Leer con Atención

La `Biblioteca` es una herramienta conceptual de GOBSTONES para facilitar la reutilización de ciertas definiciones. Sin embargo, cada herramienta que implementa el lenguaje puede proveer diferentes facilidades para utilizarla, que van más allá de la cuestión conceptual.

Actividad de Programación 14

Rehaga el ejercicio 2.2.12 pero colocando el procedimiento `IrAlOrigen` en la `Biblioteca`. Pruébelo en la herramienta que utilice.

Antes de proseguir con los conceptos estructurales, vamos a hablar de algunas cuestiones de estilo que afectan la manera en que las personas leemos los programas.

2.3. Acerca de las cuestiones de estilo

Hasta ahora vimos que al aprender un lenguaje de programación, existen numerosas reglas rígidas que debemos aprender para poder escribir programas. Sin embargo, también aparecieron ciertas cuestiones de estilo, que si bien no remarcamos demasiado, son parte importante de la forma en que escribimos programas.

Puesto que un programa GOBSTONES es una descripción de la solución a un problema, cabe hacer la pregunta: ¿quién leerá esta descripción? La respuesta tiene dos partes. La más fundamental es que el programa debe ser *"leído"* por el cabezal para realizar las acciones. Pero son los programadores quiénes también deben leerlo, para realizar cambios, mejoras, descubrir posibles errores, etcétera.

Las reglas rígidas, como ya vimos, tienen que ver con los programas en cuanto a piezas *ejecutables*, o sea, con las máquinas que deben leer y analizar los programas, que son limitadas y esperan las cosas de cierta manera fija. Las cuestiones de estilo, en cambio, tienen que ver con los programas en cuanto *descripciones*, con las personas que deben leer y comprender el código. Un buen estilo resulta en código con mayor *legibilidad*, esto es, hace al programa más sencillo de leer y comprender por las personas que lo lean.

La *legibilidad* es una propiedad del código que muestra el grado de simplicidad con el que una persona que lee el código puede entenderlo.

Como con toda cuestión de estilo, no hay reglas fijas, universales, que establezcan qué es correcto y qué es incorrecto, o qué es cómodo o incómodo hacer. Cada programador posee un gusto diferente con respecto a las cuestiones de estilo. Sin embargo, hay algunos extremos que son claramente entendibles o claramente inentendibles, y por lo tanto sencillos de identificar. Entonces, la tarea del programador a medida que adquiere experiencia, es identificar estas cuestiones de estilo, evitar prácticas que conduzcan a códigos ilegibles, y en el camino, incorporar su propio estilo.

Esto es similar a lo que ocurre cuando uno aprende a bailar un baile específico, como tango o salsa, o cuando aprende a dibujar o pintar

Por supuesto, adquirir un estilo propio no es tarea sencilla, y el consejo al comenzar es copiar el estilo de otros. Copiando de la suficiente cantidad de programadores, pueden identificarse muchas de las prácticas consideradas correctas o incorrectas en estilo. En este libro hemos utilizado un estilo específico, y esperamos que los estudiantes lo imiten, al menos mientras adquieren el propio. Para que esto no sea algo totalmente impensado y automático, en este apartado vamos a presentar algunas de las cuestiones de estilo más importantes, y los principios por los que nos guiamos en este libro.

2.3.1. Indentación de código

La primera cuestión de estilo está relacionada con la forma en la que el código es visualizado. Como hemos visto, el código no se muestra como algo completamente lineal, sino más bien bidimensional: usamos ciertos elementos debajo de otros para indicar que los primeros están subordinados a los segundos. Por ejemplo, al bloque nombrado por la definición de un procedimiento simple lo hemos puesto hasta ahora siempre debajo de la línea que indica que se está definiendo el mismo.

Desde el punto de vista de la herramienta que ejecuta los programas, es lo mismo un *carácter de espaciado* que quinientos, ya que lo que importa es poder distinguir que hay más de un elemento uno a continuación del otro, y no a qué distancia están. Los caracteres de espaciado están dados por los espacios en blanco, los fines de línea y los tabuladores. Entonces, son equivalentes el código

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

y el código

```
procedure PintarCelda() { Poner(Rojo); Poner(Rojo) }
```

y también el código

```
procedure PintarCelda()
  { Poner(Rojo); Poner(Rojo) }
```

e incluso también el código

```
                                        procedure


             PintarCelda()


                                 {
Poner(Rojo)


                                 Poner(Rojo)

 }
```

Lo repetimos: estas cuatro variaciones del código son equivalentes. Recordemos que esto quiere decir que tendrán exactamente el mismo efecto. Entonces, ¿por qué molestarse con cuál de las 4 elegir? Claramente, la última de las formas es casi inentendible para un lector humano, a pesar de que la computadora las considerará exactamente igual. La diferencia entre las 3 primeras, en cambio, es más sutil.

Actividad de Programación 15

Pruebe las variaciones anteriores en la herramienta, y verifique que las formas son equivalentes. Encuentre otras variaciones posibles que resulten en código equivalente (aunque sea ilegible o innecesariamente complicado).

A la forma exacta en que el código es mostrado en dos dimensiones, en función de variar el número de caracteres de espacio que se utilizan, se lo denomina *indentar* el código. El término *indentar* es un neologismo basado en el término inglés *indent*, que significa *sangrar*, como en el castellano *sangría*. Entonces, indentar un código es sangrarlo, o sea, comenzar ciertas líneas más adentro que otras, como se hace en el primer párrafo de los textos en castellano; en otros términos, es agregarle sangrías al texto del código. Dado que el término castellano *sangrar* es sinónimo también de, y más comúnmente reconocido como, *perder sangre*, muchos programadores de habla castellana (pero no los españoles) preferimos usar el término *indentar*.

Indent también significa *dejar marcas en una superficie*, además de sangrar.

Los españoles son extremadamente recelosos de los neologismos o anglicismos, y traducen todo de manera literal. Esto resulta confuso, pues hablan de *sangrado de código*, lo cual a mí al menos me hace pensar en código al que se le saca sangre...

Para Ampliar

Buscar en la web discusiones sobre indentación (o sangrado) de código. El primer sitio que sale en Google es la Wikipedia:

http://es.wikipedia.org/wiki/Indentacion

que dice

> Indentación es un anglicismo (de la palabra inglesa indentation) de uso común en informática que significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores para separarlo del texto adyacente, lo que en el ámbito de la imprenta se ha denominado siempre como sangrado o sangría. En los lenguajes de programación de computadoras la indentación es un tipo de notación secundaria utilizado para mejorar la legibilidad del código fuente por parte de los programadores, teniendo en cuenta que los compiladores o intérpretes raramente consideran los espacios en blanco entre las sentencias de un programa. Sin embargo, en ciertos lenguajes de programación como Haskell, Occaml y Python, la indentación se utiliza para delimitar la estructura del programa permitiendo establecer bloques de código. Son frecuentes discusiones entre programadores sobre cómo o dónde usar la indentación, si es mejor usar espacios en blanco o tabuladores, ya que cada programador tiene su propio estilo.

Definición 2.3.1. Indentar *código es seleccionar la forma en que el mismo se muestra en dos dimensiones, en función de variar el número de caracteres de espacio que se utilizan.*

Leer con Atención

Entonces, realizamos nuevamente la pregunta: ¿por qué puede resultar mejor indentar el código de cierta manera? La indentación de código realza la comprensión de la *estructura* del texto, o sea, qué elementos están subordinados a qué elementos, y qué información es válida en qué contexto. Básicamente, permite una comprensión mucho más rápida de esa estructura, y por lo tanto del programa. Por ello, una adecuada indentación es fundamental.

Como con todas las cuestiones de estilo, la percepción sobre lo que es adecuado y lo que no varía con las personas, e incluso con el contexto. Algunas formas son claramente incorrectas (como la tercer variante que vimos antes), pero otras son solo cuestión de costumbre. Por ejemplo, en este libro hemos elegido colocar el

bloque nombrado por un procedimiento en una línea independiente al encabezado de su declaración, y en el caso que el bloque ocupe más de una línea, con las llaves en sus propias líneas, y uno o dos caracteres más adentro que la palabra `procedure`; los comandos del bloque van entre medio de ambas llaves, otros dos caracteres más adentro. El resultado, que ya hemos visto numerosas veces, es el siguiente

```
procedure␣DibujarLinea()
␣{
␣␣␣Poner(Negro);␣Mover(Norte)
␣␣␣Poner(Negro);␣Mover(Norte)
␣}
```

Sin embargo, otros estilos son posibles. El mismo código, indentado por un programador de la comunidad JAVA, habría sido

```
procedure␣DibujarLinea()␣{
␣␣Poner(Negro);␣Mover(Norte)
␣␣Poner(Negro);␣Mover(Norte)
}
```

JAVA es uno de los principales lenguajes de programación de hoy día. Para conocer más sobre el lenguaje JAVA, ver el sitio

www.java.com/es

Observe cómo la llave de apertura se coloca en la misma línea que el encabezado, y la correspondiente de cierre a la misma altura que la palabra `procedure`.

Otra variación de estilo, que usaremos en ocasiones, consiste en que, cuando el código del procedimiento es breve, se muestra en una única línea, como en

```
procedure DibujarLineaCorta() { Poner(Negro); Mover(Norte) }
```

o a veces

```
procedure DibujarLineaCorta()
 { Poner(Negro); Mover(Norte) }
```

Al principio intente imitar el estilo de indentación utilizado en este libro, pero a medida que avance en la escritura de sus propios programas, experimente con otras variaciones para ver cuál le resulta más cómoda. Recuerde que la forma en que otros programadores leen el código también se verá afectada por sus cambios. ¡Y también que una de las cuestiones que se evalúan al corregir código de los estudiantes es la comprensión sobre la estructura del mismo! Eso es lo que debe reflejar en la indentación.

Leer con Atención

Si bien en la mayoría de los lenguajes la indentación es una cuestión de estilo, tal cual mencionamos aquí, en algunos lenguajes modernos como HASKELL, PYTHON, etc. la indentación puede utilizarse para indicarle a la máquina que ejecuta cuál es la estructura del programa. Consecuentemente, en esos lenguajes la indentación sigue reglas fijas, rígidas, no sujetas a las consideraciones estilísticas.

2.3.2. Elección de nombres adecuados de identificadores

La segunda cuestión fundamental de estilo tiene que ver con la elección de nombres para los identificadores. A este respecto, hay dos aspectos básicos:

1. qué nombres elegir para describir adecuadamente la solución;
2. qué forma de escribir nombres complejos se utiliza.

El aspecto de cómo escribir nombres complejos tiene varias opciones más o menos estandarizadas en la comunidad de programadores. Por ejemplo, en este libro hemos elegido usar *CamelCase*, que como vimos, es un estilo de escritura para identificadores compuestos de varias palabras, y donde cada palabra se escribe con mayúsculas.

Otro estilo muy difundido es utilizar guiones bajos (*underscores* en inglés) para separar las palabras, en lugar de poner en mayúsculas cada una de las letras. Comparemos, por ejemplo,

- `UnIdentificadorEnCamelCase` con
- `un_identificador_NO_en_camel_case`.

Cada comunidad de programación tiene su propia preferencia por uno u otro estilo, o por algunas variaciones, e incluso en algunos casos por estilos completamente distintos. Lo que es claro es que se trata de una cuestión puramente estilística, y que, por lo tanto, genera discusiones fundamentalistas, cuasi-religiosas (o para decirlo de otra manera más elegante, semi-científicas).

En algunas comunidades de programación estas cuestiones de estilo se utilizan para alimentar herramientas adicionales de tratamiento de código, y en ese caso se constituyen en un "sublenguaje" que se expresa en reglas rígidas de estilo necesarias para que la ejecución de las herramientas adicionales sea correcta.

Para Ampliar

Para ampliar sobre el tema de las discusiones semi-científicas al respecto de cuál estilo es mejor, ver, por ejemplo

Recurso Web

http://whathecode.wordpress.com/2011/02/10/camelcase-vs-underscores-scientific-showdown/

¡Pero no olvide que en este libro favorecemos el CamelCase!

Ahora bien, el otro aspecto, referente a los nombres de identificadores, es crucial, y es por ello la que más debates fundamentalistas genera en los programadores. En este aspecto, se trata de **qué** nombres elegir para describir una tarea y así tener procedimientos bien nombrados.

Debemos tener en cuenta que el nombre exacto de un procedimiento no es importante para la herramienta, pues al ejecutar, se limita a buscar la definición de un procedimiento con ese nombre para saber qué debe hacer. Entonces, para la herramienta, y puesto que los tres generan el mismo efecto, son equivalentes el programa

```
program { PonerGotaAcuarelaRoja() }
procedure PonerGotaAcuarelaRoja() { Poner(Rojo) }
```

con el programa

```
program { ColocarAlienigena() }
procedure ColocarAlienigena() { Poner(Rojo) }
```

y con el programa

```
program { A() }
procedure A() { Poner(Rojo) }
```

Sin embargo, para un lector humano, el primero da cuenta de que el programador está intentando representar elementos en un dominio de pintura, el segundo, que el programador está pensando en algún juego de ciencia-ficción, por ejemplo, y el tercero puede tratarse de cualquier cosa, ya que la intención original del programador se ha perdido.

A partir de este ejemplo, podemos ver que el nombre `A` para un procedimiento rara vez es adecuado. De la misma forma, un nombre excesivamente largo, como `PonerAlienigenaInvasorCuandoSeProduceLaInvasionALaTierra`, resulta

inadecuado pues poco contribuye a la legibilidad del código, e incluso lo complica un poco.

Elegir nombres adecuadamente es, entonces, una cuestión de estilo de fundamental importancia. Y como todo estilo, es complejo poder dominarlo. Requiere práctica, mucha práctica. Pero se consigue. En GOBSTONES elegimos la convención de usar, para los comandos, identificadores que comienzan con verbos en infinitivo, seguidos de alguna descripción adicional, como `Poner`, `DibujarLinea`, etcétera. Otros lenguajes siguen otras convenciones.

Como dijimos antes, ningún nombre que se elija está *realmente mal*, pues no es importante para la ejecución del programa, sino que es más adecuado o más inadecuado para la lectura; y el grado de adecuación depende mucho del gusto del que lo lee. Sin embargo, al evaluar un programa, existen elecciones claramente incorrectas pues no solo no contribuyen a la legibilidad, sino que confunden. Por ejemplo, observe el siguiente código

```
program { Poner(Verde); PonerBolitaRoja() }
procedure PonerBolitaRoja() { Sacar(Verde) }
```

¿Qué hace este programa? ¿Es correcto? ¿Se da cuenta que la elección del nombre `PonerBolitaRoja` es terriblemente inadecuada, porque lleva a pensar en otra idea? ¡Pero sin embargo el código funciona! Entonces diremos que este código está *"mal"*, porque su elección de nombres es engañosa, aunque el código funcione.

Leer con Atención

Lo que sucede es que no es suficiente que el código funcione para ser considerado un buen código. Debe ser posible que las personas –otros programadores– lo lean y entiendan el propósito del mismo. La correcta elección de nombres contribuye enormemente a esto. Por correcta, queremos decir que refleje una adecuación entre el problema que se está resolviendo y la forma de resolverlo, que dé cuenta de las intenciones del autor.

¿Y qué hacemos cuando el nombre no es suficiente para mostrar nuestros propósitos? ¿Cómo damos cuenta, por ejemplo, de los requerimientos de un procedimiento para funcionar? En las próximas secciones se aborda precisamente este tema.

2.3.3. Comentarios en el código

Puesto que la intención del programador no siempre es evidente a partir de la lectura exclusiva de los comandos, incluso a pesar de la adecuada indentación del código y de la buena elección de identificadores, 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 y son considerados igual que los caracteres de espacio desde el punto de vista de la herramienta.

Definición 2.3.2. *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.*

Leer con Atención

Al igual que en el caso de los nombres de identificadores y de la indentación, en los comentarios también existen comunidades donde se utilizan los comentarios para alimentar ciertas herramientas externas de tratamiento de código, por lo que ciertos comentarios deben ser escritos utilizando una sintaxis rígida, adecuada para el tratamiento automático. En este caso, los comentarios incluyen un sublenguaje específico que si bien no es parte del programa original, es importante para la aplicación de las herramientas asociadas. Es importante, entonces, al aprender un lenguaje nuevo, establecer cuáles son las reglas rígidas y cuáles las nociones de estilo. En el caso de GOBSTONES, todos los comentarios son meramente estilísticos.

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

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

Un comentario de línea comienza con un símbolo o grupo de símbolos, y continúa hasta el final de la línea actual; todo lo que se escriba desde el símbolo de comentario de línea hasta el final de línea es ignorado durante la ejecución. Un comentario de párrafo comienza con un símbolo o grupo de símbolos y continúa hasta la primera aparición de otro símbolo o grupo de símbolos coincidente con el primero; todo el texto que aparece enmarcado entre estos pares de símbolos es ignorado durante la ejecución. En GOBSTONES coexisten 3 estilos de comentarios. En el estilo C-like, el símbolo `//` se utiliza para indicar comentarios de línea, y los símbolos `/*` y `*/` para encerrar comentarios de párrafo. En el estilo HASKELL-like, el símbolo `--` indica comentarios de línea, y los símbolos `{-` y `-}` encierran comentarios de párrafo. Finalmente, en el estilo PYTHON-like, el símbolo # indica comentarios de línea y el símbolo `'''` indica el inicio y fin de los comentarios de párrafo.

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, etcétera.

Todavía no hemos hablado de parámetros. Es un tema que se tratará en la Unidad siguiente.

Como ejemplo de uso de comentarios, se ofrece una versión comentada del procedimiento `DibujarLineaNegra2HaciaElNorte`.

```
{-
   Autor: Fidel
   Fecha: 08-08-08
-}
procedure DibujarLineaNegra2HaciaElNorte()
 {-
    PROPÓSITO:
     Dibujar una línea negra de longitud 2
     hacia el norte de la celda actual.
```

Esta forma de comentar pretende mostrar la mayoría de las formas usuales de comentarios en un solo programa, lo que resulta en un estilo recargado que no es el usualmente utilizado. En general, los comentarios se limitan al propósito y precondición, y alguna que otra aclaración ocasional en el código (¡y no como en este ejemplo, que cada línea tiene comentarios!)

```
    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. Eso sí: es tan malo abusar de los comentarios (en una línea similar al ejemplo anterior, que sobreabunda en ellos) como evitarlos completamente. Una buena práctica de la programación adopta un estilo donde los comentarios son parte importante del código, pero no lo recargan innecesariamente.

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 2.2.10. Para ello, copiamos el programa en otro archivo, y realizamos algunas modificaciones, para obtener:

```
{-
   Autor: Fidel
   Fecha: 08-08-08
-}
procedure DibujarLineaNegra3HaciaElNorte()
 {-
    PROPÓSITO:
     Dibujar una línea negra de longitud 2
     hacia el norte de la celda actual.
    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
```

```
{-
   Autor: Fidel
   Fecha: 08-08-08
-}
procedure DibujarLineaNegra3HaciaElNorte()
 {-
    PROPÓSITO:
    Dibujar una línea negra de longitud 2
    hacia el norte de la celda actual.
    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
  }
```

¡Comentarios desactualizados!

G.2.11. Código con comentarios desactualizados

```
}
```

Observar que el código fue obtenido copiando y pegando (suele decirse, con *cut&paste*), y luego alterando algunas partes

Para Reflexionar

¿Puede encontrar las partes que se agregaron?

Estas alteraciones llevan a que haya por lo menos tres comentarios desactualizados en este código. El primero es el del propósito y la precondición, puesto que al agregar comandos de movimiento, tanto el propósito como la precondición cambiaron, 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. Puede verse en el gráfico G.2.11 las indicaciones de qué comentarios están desactualizados. Puesto que los comentarios están en lenguaje natural, y el propósito de un programador no es analizable automáticamente, este tipo de situaciones no se pueden detectar con ninguna herramienta. Es absoluta responsabilidad del programador que sus comentarios estén actualizados, sean pertinentes y útiles.

La forma en que la mayoría lo leemos es *"cátan péist"*.

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 adecuada. 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 observaciones finales, 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. Además, rara vez suele comentarse un código tan profusamente como el ejemplo de `DibujarLineaHaciaElNorte` como hemos hecho aquí.

Durante este último ejemplo hablamos de propósito y precondición de un procedimiento, pero sin haber explicado exactamente qué eran esas cosas. En la sección siguiente completaremos esa falta.

2.3.4. Contrato de un procedimiento

Si bien la herramienta provista por los procedimientos puede utilizarse para darle nombre a cualquier grupo de comandos, lo más adecuado desde el punto de vista de la comprensión del código es que los comandos se agrupen en función de la resolución de alguna tarea específica. Por otra parte, y dado que algunos de los comandos que utilizamos son parciales, es posible que al armar un procedimiento el mismo falle porque no se cumplen las condiciones necesarias para que esos comandos parciales se ejecuten. Para cubrir ambos aspectos, presentaremos la idea de *contrato* de un procedimiento. El *contrato* de un procedimiento está formado principalmente por dos partes:

- el *propósito* del procedimiento, y
- las *precondiciones* del procedimiento.

El propósito de un procedimiento establece cuál es la tarea que el procedimiento pretende resolver, o sea, la funcionalidad que esperamos que el procedimiento tenga. Las precondiciones de un procedimiento, al igual que la precondición de un comando primitivo, establecen los requisitos que deben cumplirse para que el procedimiento pueda llevar a cabo su tarea con éxito. Entre ambos forman las reglas que permitirán saber cómo debe utilizarse el procedimiento en cuestión. O sea, si un programador quiere lograr cierta funcionalidad en su programa, se fija entre los procedimientos que ya tiene definidos si alguno sirve para ese objetivo (mirando los propósitos de los mismos), y elige el que corresponda. Luego se fija cuál es las precondiciones de dicho procedimiento, para poder llamarlo con la seguridad de que no fallará.

Definición 2.3.3. *El* contrato *de un procedimiento está formado por su* propósito *y sus* precondiciones*. El* propósito *establece qué tarea resuelve el procedimiento; las* precondiciones *establecen los requisitos que deben satisfacerse para poder completar el propósito con éxito.*

Tanto el propósito como las precondiciones pueden expresarse de diferentes maneras. Lo usual es que los mismos se expresen coloquialmente en lenguaje castellano, aunque también existen otras opciones, más avanzadas; por ejemplo, una muy común es utilizar alguna forma de lenguaje lógico para expresarlos, de manera de poder proveer mecanismos formales de tratamiento. Sin embargo, en este libro, usaremos solo la forma coloquial, pues se trata solo de una introducción y no pretende un conocimiento exhaustivo o profundo de las variantes.

Para dejar constancia del propósito y precondiciones de un procedimiento se suelen utilizar comentarios, tal cual se mostró en la sección anterior.

Veamos un ejemplo del contrato de un procedimiento. Para ello, utilizaremos el procedimiento `DibujarCuadradoNegroDeLado3` visto en el ejercicio 2.2.10. El código completo para el mismo, comentado con el contrato, podría ser

```
procedure DibujarCuadradoNegroDeLado3()
 {-
    PROPÓSITO:
```

```
      * dibujar un cuadrado negro de 3 celdas de lado
        con el vértice inferior izquierdo ubicado en
        la celda actual, dejando el cabezal en la misma
        posición
    PRECONDICIONES:
      * existe suficiente espacio hacia el norte y el este
        para poder dibujar el cuadrado (2 filas al Norte y
        2 columnas al Oeste)
-}
{
  DibujarLineaNegra2AlNorte()
  DibujarLineaNegra2AlEste()
  DibujarLineaNegra2AlSur()
  DibujarLineaNegra2AlOeste()
}
```

Podemos observar que tanto el propósito como las precondiciones están escritas en castellano, lo cual, siendo que figuran en un comentario, es perfectamente válido.

Es importante tener en cuenta que las precondiciones deben establecer condiciones que den los requerimientos del procedimiento para poder cumplir con su propósito sin fallar. Esto puede lograrse de manera muy precisa, o con menos precisión. En general vamos a preferir precondiciones lo más precisas posible. Para ejemplificar, consideremos la siguiente precondición:

```
  PRECONDICIONES:
    * el tablero debe tener 3 filas y 3 columnas y la celda actual
      debe ser el origen
```

Claramente sirve también para el procedimiento `DibujarCuadradoNegroDeLado3`, pues en ese tablero podrá dibujar sin problemas; pero en general este tipo de precondiciones son consideradas demasiado restrictivas, pues no dice que hay muchísimos otros tableros donde el procedimiento *también* funcionará. Entonces, si usásemos esta última precondición en lugar de la que vimos, un programador al verla pensaría que solo puede limitar el uso del procedimiento a tableros de ese tamaño, y no la usaría para otros casos donde podría serle útil.

Leer con Atención

En general es extremadamente importante escribir bien el propósito y la precondición de manera que indiquen claramente el objetivo y los requerimientos de la manera *más precisa posible*, ya que es la forma en que un programador se entera de las características importantes del procedimiento. O sea, si el contrato está bien expresado, el programador que precisa usarlo normalmente no lee el código del mismo, ya sea por no estar disponible o por no ser relevante para su solución.

Establecer el propósito tiene además una segunda ventaja: cuando definimos un procedimiento debemos pensar *para qué* lo estamos definiendo, o sea, *qué tarea* esperamos que el procedimiento resuelva. De esa forma evitamos definir procedimientos que no tienen sentido desde el punto de vista de nuestra solución, y podemos determinar si ciertas operaciones deben ser incluídas (o no) como parte del mismo. Volveremos a este punto cuando veamos ejemplos más complejos.

Por todas estas razones, en el resto del curso insistiremos con establecer siempre el propósito y las precondiciones de un procedimiento. La forma en que se redactan ambos es también una cuestión de estilo. Al igual que las restantes cuestiones de estilo, se puede dominar perfectamente con suficiente práctica.

2.4. Ejercitación

En este apartado se enuncian una serie de ejercicios de práctica adicionales a los ya dados durante el capítulo. Para su correcta resolución son necesarios todos los elementos aprendidos en las secciones anteriores. A través de su resolución iremos repasando algunos de los conceptos principales, e incorporando preguntas que prepararán el material de capítulos posteriores.

Es importante que al resolverlos se utilicen **solamente** los conceptos ya vistos. Las personas que ya conocen otros lenguajes tendrán la tentación de utilizar herramientas más avanzadas para pensar la solución; sin embargo, los ejercicios fueron pensados para ser resueltos con las herramientas provistas. En algunos casos puede resultar más complejo de la cuenta, ¡pero esa complejidad es buscada con fines didácticos!

Actividad de Programación 16

Realice los ejercicios enunciados en esta sección. Recuerde separar adecuadamente su código en procedimientos, elegir convenientemente los nombres de los mismos, y comentar su código de manera que el mismo sea fácilmente entendible.

El primer ejercicio es simplemente una guía para recordar cómo definir un procedimiento, para experimentar cómo probarlo en un programa, y para preparar ideas que luego se utilizarán en ejercicios posteriores.

Ejercicio 2.4.1.

Escribir un procedimiento `PonerUnaDeCadaColor`*, que coloque una bolita de cada color en la celda actual. ¿Cuál es su precondición?*

Leer con Atención

Recuerde que para probar su procedimiento debe escribir un programa principal que lo utilice.

El siguiente ejercicio permite volver a repasar la idea de reutilización de procedimientos ya definidos. Toda vez que una tarea de programación aparece una y otra vez, hay un candidato para introducir un procedimiento que resuelva esa tarea. En muchos casos en este libro, los procedimientos a utilizar en una solución se presentan antes en otros ejercicios previos, orientando de esta manera la solución deseada.

Ejercicio 2.4.2.

Escribir el procedimiento `Poner5DeCadaColor` *que coloque cinco bolitas de cada color en la celda actual. Reutilizar el procedimiento definido en el ejercicio 2.4.1 (*`PonerUnaDeCada`*) para definir esta tarea.*

Para Reflexionar

¿Cuánto más difícil habría sido el ejercicio 2.4.2 si no hubiera contado con el ejercicio 2.4.1? Piense en la necesidad de identificar estas subtareas, y definir sus propios procedimientos para expresarlas, especialmente cuando no se hayan definido en ejercicios previos.

Y en el caso de este libro, también la perspicacia, puesto que ya hemos anunciado la intención de presentar, en ejercicios previos a uno más difícil, algunos procedimientosque pueden ser útiles para solucionar el ejercicio complejo...

Si bien en el ejercicio 2.4.2 se pidió expresamente la reutilización del ejercicio 2.4.1, esto no es lo usal. Normalmente la determinación de qué otros procedimientos pueden servir para expresar la solución a un problema (o sea, en qué subtareas es interesante o relevante dividir ese problema) se deja librado a la imaginación y la habilidad del programador.

Leer con Atención

Recuerde que una vez que definió la forma de realizar una tarea, siempre puede reutilizar dicha definición en ejercicios posteriores. Al probar su programa no olvide incluir todas las definiciones necesarias en el mismo.

En los próximos dos ejercicios se repasa la idea de precondición, y de nombrar adecuadamente, y se trabaja con las ideas de definición de nomenclatura y de minimización de efectos de un procedimiento. La definición de nomenclatura es necesaria para poder hablar de un problema en los términos del dominio del mismo, y no en términos de bolitas, y es algo usual en los ejercicios más complejos. La minimización de efectos de un procedimiento tiene que ver con el hecho de que el mismo **haga lo pedido y nada más que lo pedido.**

Ejercicio 2.4.3.

La celda lindante en dirección *<dir> es la primera celda en la dirección dada, a partir de la celda actual. Por ejemplo la* celda lindante al Norte *es la primera celda al Norte de la celda actual.*

Escribir el procedimiento `PonerAzulAlNorteYMoverse` *que ponga una bolita de color azul en la celda lindante al Norte y se quede en esa posición. ¿Este procedimiento siempre funciona? ¿Qué requerimientos debe cumplir? ¿Cuál sería su precondición?*

Ejercicio 2.4.4.

Escribir el procedimiento `PonerAzulAlNorte` *que ponga una bolita de color azul en la celda lindante al Norte pero deje el cabezal en la posición original. ¿Cuál es su precondición?*

Para Reflexionar

¿Por qué puede ser importante devolver el cabezal a su posición original? Reflexione sobre este hecho antes de continuar. Reflexione también sobre las diferencias en el nombre de los procedimientos de los ejercicios 2.4.3 y 2.4.4. ¿Por qué no llamamos al segundo `PonerAzulAlNorteYQuedarseEnElLugar`? Recuerde la minimización de efectos.

Ejercicio 2.4.5.

Escribir el procedimiento `PonerAzulEnLos4Vientos` *que ponga una bolita de color azul en la celda lindante al Norte, Sur, Este y Oeste y deje el cabezal en la posición original. ¿Cuál es la precondición del procedimiento?*

Para Reflexionar

¿Cómo habría sido la solución del ejercicio 2.4.5 si en lugar de usar la solución del ejercicio 2.4.4, hubiera utilizado la del ejercicio 2.4.3? Reflexione acerca de cómo algunas decisiones sobre qué tareas agrupar en cada procedimiento pueden afectar la complejidad de la solución final.

En el siguiente ejercicio trabajamos con la idea de dominio de un problema, repasando la idea de definición de nomenclatura. En este caso, si estamos construyendo un programa de pintura, tiene sentido hablar de témperas de colores y

no de bolitas, y representar y nombrar las ideas de manera adecuada al problema y no al lenguaje de programación. También se repasa la idea de que usted identifique sus propias subtareas, definiendo procedimientos para ellas.

Leer con Atención

Al realizar el ejercicio, recuerde definir procedimientos auxiliares (como `PonerUnaDeCadaColor` en el caso de `Poner5DeCadaColor`), con los cuales definir sus procedimientos le resulte sencillo. Para ello, intente identificar subtareas y nombrarlas adecuadamente.

Ejercicio 2.4.6. *Supongamos que representamos a 1 gota de témpera con 3 bolitas de un color determinado. Escribir el procedimiento* `Mezclar2GotasDeAmarillo` *que agregue en la celda actual 1 gota de témpera azul y 1 gota de témpera verde.*

Para Reflexionar

¿Qué tan adecuado es el nombre `Mezclar2GotasDeAmarillo`? ¿Cómo denominó a sus procedimientos auxiliares? Nombres adecuados habrían sido `Poner1GotaDeAzul` y `Poner1GotaDeVerde`. ¿No sería mejor nombrar de otra forma al procedimiento `Mezclar2GotasDeAmarillo`? **Pista:** ¿cuál es el verbo que utiliza en cada caso?

El siguiente ejercicio insiste sobre la idea de reuso de procedimientos, y trabaja sobre la adecuación de nombrar a los procedimientos de determinada manera.

Ejercicio 2.4.7.

Escribir el procedimiento `Poner3GotasDeNaranja` *que agregue en la celda actual 2 gotas de témpera amarilla y 1 gota de témpera roja. Para resolver este ejercicio, debe cambiarse el nombre del procedimiento del ejercicio 2.4.6 convenientemente para lograr uniformidad y adecuación en los nombres.*

El uso uniforme de nombres está relacionado con la comprensión del problema a resolver, y su adecuada expresión en términos de los elementos del lenguaje de programación. Representar bien las ideas básicas del problema en términos de procedimientos hace que luego la solución resulte sencilla de entender y modificar. Por ejemplo, si uno quisiera cambiar la representación, solo tendría que cambiar los procedimientos más elementales.

Ejercicio 2.4.8.

Modificar el procedimiento `Poner3GotasDeNaranja` *para que la representación de 1 gota de témpera de color sea realizada con 5 bolitas en lugar de 3. ¿Qué procedimientos fue necesario modificar? ¿Debió alterarse el código del procedimiento* `Poner3GotasDeNaranja`*, o bastó con modificar los procedimientos elementales?*

Para Reflexionar

Cuando las subtareas fueron identificadas adecuadamente y expresadas correctamente en términos de procedimientos, modificaciones en la representación no deberían afectar la solución en su nivel más cercano al problema. Si en lugar de 3 o 5 bolitas para representar una gota, hubiéramos elegido 1, ¿habría seleccionado un procedimiento para representar la idea de poner una gota? Si su respuesta es no, reflexione sobre lo sencillo que es caer en el pensamiento operacional, que se concentra en las operaciones individuales en lugar de en las tareas a resolver, y en las terribles consecuencias de esto en la calidad del código producido.

Los próximos ejercicios repasan las siguientes ideas:

- la correcta identificación de subtareas reduce significativamente la complejidad del código resultante y facilita su modificación ante cambios en la representación;
- el adecuado uso de nombres de procedimientos mejora la legibilidad y facilita la comprensión de dicho código; y
- la minimización de los efectos de un procedimiento permite que su combinación con otros sea más sencilla.

Ejercicio 2.4.9.

Completar los procedimientos necesarios para que el procedimiento `LetraERoja` *pinte una letra E con acuarela roja, suponiendo que en cada celda se coloca una gota de acuarela, y que una gota de acuarela se representa con 1 bolita. La letra E se puede pintar en un grupo de 5 por 3 celdas, como se muestra en el gráfico G.2.12a y un posible código para el procedimiento* `LetraERoja` *sería*

```
procedure LetraERoja()
 {- PROPOSITO: pinta una letra E roja en el tablero
    PRECONDICION: Hay 4 celdas al Norte y
                      2 al Este de la celda actual
    OBSERVACION: No modifica la posicion del cabezal
  -}
 {
   PintarLineaHorizontal3() -- La base de la E
   Mover(Norte)
   PintarCelda()            -- Nivel 2 de la E
   Mover(Norte)
   PintarLineaHorizontal3() -- La barra de la E
   Mover(Norte)
   PintarCelda()            -- Nivel 4 de la E
   Mover(Norte)
   PintarLineaHorizontal3() -- El tope de la E
   -- Retorna el cabezal
   Mover(Sur);Mover(Sur);Mover(Sur);Mover(Sur)
 }
```

Para Reflexionar

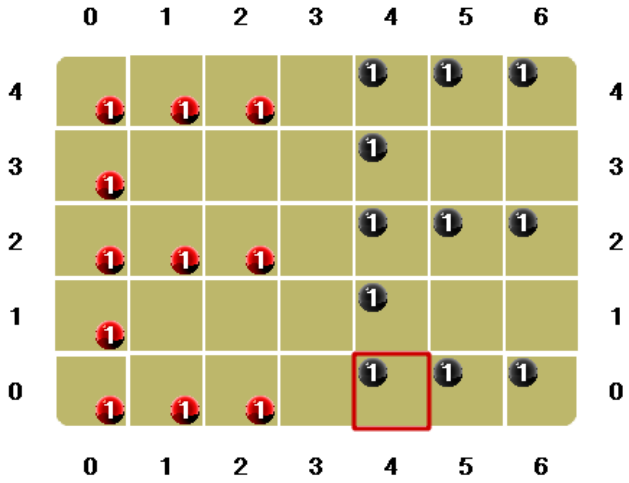
Observe con detenimiento los procedimientos que utilizamos en el procedimiento `LetraERoja`, y cómo los mismos fueron comentados. Al identificar correctamente las partes se requieren pocos procedimientos, y el código resultante queda estructurado con base en el problema, y no en las celdas en las que se dibuja. Reflexione sobre cómo habría sido el código si en lugar de definir y usar `PintarLineaHorizontal3`, hubiera usado solo `PintarCelda` y `Mover`. ¿Y si `PintarLineaHorizontal3` hiciese más de lo mínimo necesario?

Ejercicio 2.4.10.

Escribir un procedimiento `LetraE`*, que pinte una letra E con acuarela marrón, suponiendo que la acuarela marrón se consigue mezclando una gota de cada uno de los 4 colores. La letra E debería quedar como se muestra en el gráfico G.2.12b.* ***Ayuda:*** *Modificar adecuadamente el ejercicio anterior. ¿Puede hacerse con solo 2 cambios y a lo sumo 4 procedimientos nuevos? Uno de los cam-*

(a). Letra E roja (b). Letra E “marrón”

G.2.12. Representación de distintas letras E en el tablero de GOBSTONES



G.2.13. Representación de dos letras E en el tablero de GOBSTONES

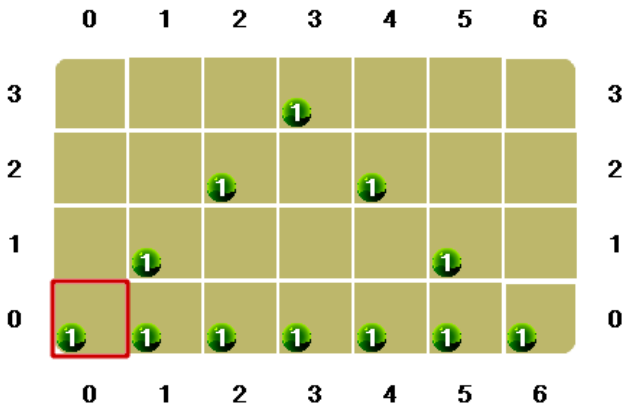
bios es modificar el nombre del procedimiento `LetraERoja` *por* `LetraE`. *¿Cuál debería ser el otro? ¿Qué procedimientos nuevos definir para la representación?*

El siguiente ejercicio motiva la necesidad de alguna herramienta del lenguaje que permita separar de alguna manera, por un lado el dibujo de una letra, y por el otro la elección sobre el color con la que se la dibuja. Esta herramienta se verá en unidades posteriores.

Ejercicio 2.4.11.

Escribir un procedimento `LetrasERojaYNegra` *que dibuje una letra E roja seguida de una letra E negra. El dibujo debería quedar como en el gráfico G.2.13.*

Sugerencia: *utilizar como base para la solución de este ejercicio, el código dado en el ejercicio 2.12a, duplicándolo y cambiando lo necesario en la versión duplicada para tener una letra E negra además de la roja.*



G.2.14. Dibujo de un triángulo de base 7

Para Reflexionar

Con los elementos presentados en esta Unidad, la única alternativa para resolver el ejercicio 2.4.11 es duplicar el código y cambiar el color con el que se dibuja. Eso obliga a duplicar todos los procedimientos auxiliares, y buscar nuevos nombres para cada uno. ¿Se da cuenta como es imprescindible, para poder hacer un programa razonable, contar con algún elemento que permita evitar esta duplicación?

Ejercicio 2.4.12.

Escribir un procedimiento `DibujarTrianguloBase7` *que dibuje un triángulo verde que tenga una base de 7 celdas. No utilizar más de 3 comandos en el cuerpo del procedimiento definido. El dibujo debe quedar como en el gráfico G.2.14.*
Ayuda: *para lograrlo hay que pensar en 3 subtareas que consigan dibujar las 3 partes de un triángulo. . . ¡No olvidar escribir las precondiciones de cada uno de los procedimientos así obtenidos!*

Para Reflexionar

En este caso, no era conveniente que las subtareas dejasen inalterada la posición del cabezal, puesto que al dibujar, estamos simulando un lápiz. Hemos intentado expresar esta sutileza utilizando el término “Dibujar” en lugar de “Poner”. Esto es conveniente expresarlo en las observaciones del procedimiento, como comentario.

Leer con Atención

Si calculó bien las precondiciones de los procedimientos auxiliares y del que dibuja el triángulo, podrá observar que no todos los requisitos de los auxiliares aparecen en el procedimiento general. Por ejemplo, dibujar la línea izquierda requiere espacio al `Este`, pero puesto que los dibujos anteriores se movieron hacia el `Oeste`, ese lugar existe seguro. Las precondiciones son una herramienta poderosa, pero en los casos complejos puede ser difícil calcularlas con exactitud. Por esa razón en este curso las usamos solo como guía y de manera relativamente informal.

El siguiente ejercicio motiva la necesidad de alguna herramienta del lenguaje que permita separar el dibujo de una figura del tamaño de la misma. Esta es una nueva manifestación de la repetición que vimos en relación a los colores de la letra E, salvo que en este caso se trata del tamaño en lugar del color.

Ejercicio 2.4.13.

Escribir un procedimiento `DibujarArbol`*, que dibuje un árbol de tipo conífera, como se muestra en el gráfico G.2.15.*
***Ayuda:** basarse en el procedimiento del ejercicio 2.4.12. Copiarlo 3 veces y modificar cada copia para obtener triángulos de distintos tamaños. Luego dibujar cada pieza, posicionándose adecuadamente para comenzar cada una.*

Para Reflexionar

Nuevamente, con los elementos presentados en esta Unidad, la única alternativa para resolver el ejercicio 2.4.13 es duplicar el código y cambiar el tamaño del dibujo. Todos los procedimientos auxiliares deben duplicarse, y buscar nombres adecuados para cada uno de ellos. ¿Se da cuenta cómo es imprescindible, para poder hacer un programa razonable, contar con algún elemento que permita evitar esta duplicación?

El siguiente ejercicio busca practicar los conceptos adquiridos.

Ejercicio 2.4.14.

Supongamos que cada celda del tablero representa una sección de un 1 metro de longitud de un cantero. Además, representaremos flores con combinaciones de bolitas. Las rosas se representan con 1 bolita de color rojo, las violetas se representan con 1 bolita de color azul, y los girasoles se representan con 1 bolita de color negro. Se desea armar un cantero de flores de 3 metros de longitud, donde cada metro del cantero posee 5 rosas, 8 violetas y 3 girasoles.

Escribir el procedimiento `ArmarCantero` *que cumpla con esta representación.*

Para Reflexionar

¿Usó procedimientos auxiliares? ¿Comprende la manera importantísima en la que los mismos permiten simplificar la comprensión del programa y su análisis de corrección? Si no lo hizo, observe la complejidad del código que resulta, y repita el ejercicio usando procedimientos auxiliares.

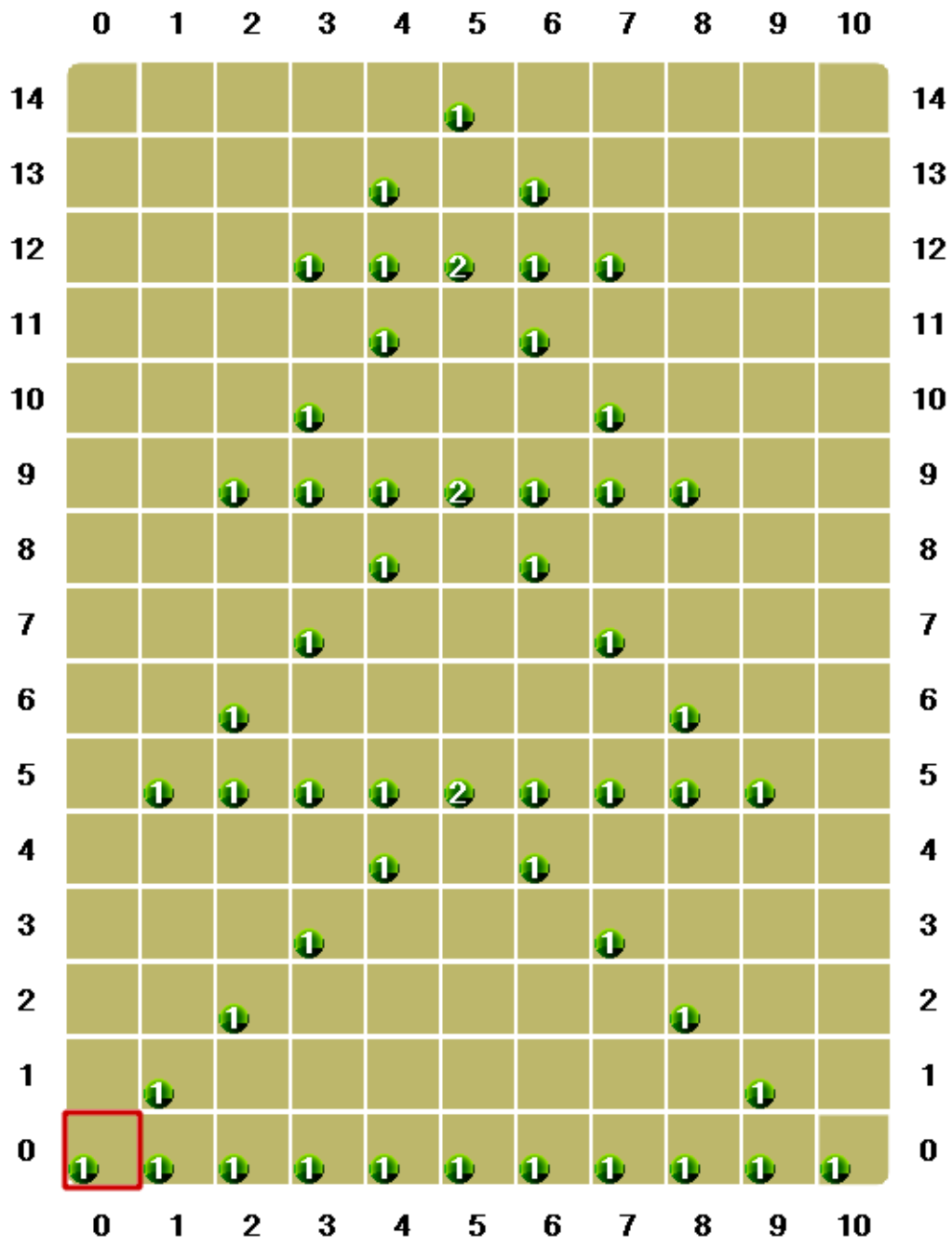
Ejercicio 2.4.15.

Modificar el programa del ejercicio 2.4.14 para que ahora las flores se representen utilizando el siguiente esquema. Las rosas se representan con 3 bolitas de color rojo y 1 de color negro, las violetas se representan con 2 bolitas de color azul y 1 de color rojo, y los girasoles se representan con 3 bolitas de color negro y 2 de color verde.

Para Reflexionar

¿Le resultó fácil la modificación? Si no, revise los conceptos de separación en procedimientos y de representación de información.

El último ejercicio busca combinar todos los elementos vistos hasta ahora. Por un lado, usted debe decidir acerca de la representación de los elementos. Por otro,



G.2.15. Dibujo de un árbol de tipo conífera

debería utilizar adecuadamente procedimientos para representar cada parte, no solo la representación de los elementos, sino también las acciones involucradas. Finalmente, debe establecer las precondiciones necesarias, y al probarlo, garantizar que las mismas se cumplen de alguna manera. ¡No olvide comentar el código!

Ejercicio 2.4.16.

La Base Marambio, en la Antártida, está siendo invadida por extraterrestres de color verde. La base es de color azul y está totalmente rodeada por ellos en las 8 direcciones del radar. La misión es escribir un programa GOBSTONES *que muestre cómo eliminar la amenaza alienígena. Para ello hay que representar de alguna manera la información del radar en el tablero de* GOBSTONES*, y luego escribir el procedimiento* `AcabarConAmenazaAlienigena` *que elimine a los invasores. ¿Cuál sería la precondición de este procedimiento?*

Ayuda: *¿De qué manera podría ser representado un alienígena en el radar? ¿Y la base? Imaginar una representación para cada uno de estos objetos, intentando ser fiel a la descripción dada sobre ellos, y elegir la representación mínima necesaria.*

La manera de eliminar a los invasores estará relacionada con la forma en la que son representados – si, por ejemplo, un elemento se representa utilizando 2 bolitas de color negro, la forma de eliminarlos deberá ser quitar esa cantidad de bolitas de ese color.

Para Reflexionar

¿Pudo probar el procedimiento anterior? ¿Qué mecanismo eligió para garantizar la precondición de `AcabarConAmenazaAlienigena` al invocarlo desde el programa principal? Tenga en cuenta que en ocasiones utilizar las ventajas que nos proveen las herramientas simplifica mucho el trabajo. ¿Utilizó la idea de precondición para armar el tablero de ejemplo?

3

Procedimientos, funciones y parametrización

En el capítulo anterior vimos los bloques básicos de un lenguaje de programación (expresiones, comandos y procedimientos simples, y las cuestiones de estilo); en el presente capítulo procederemos a ver cómo hacer bloques más generales, y estudiaremos el concepto de *parametrización*. En el proceso agregaremos algunas formas básicas de combinación de comandos y expresiones.

3.1. Procedimientos

Lo primero que aprendimos es cómo escribir procedimientos simples (definición 2.2.21), y trabajamos con ellos, aprendiendo cómo utilizarlos adecuadamente. Vimos que los procedimientos simples son una forma de nombrar comandos compuestos para identificarlos como tareas de nuestro problema y así definir un nuevo comando especializado para una tarea. Y también vimos que además de los comandos, los lenguajes poseían expresiones para describir valores, entidades para representar información.

En este apartado vamos a agregar la idea de parámetro a los procedimientos, y comenzaremos a estudiar su potencial.

3.1.1. Procedimientos con parámetros

Al resolver el ejercicio 2.4.11 donde se pedía dibujar una E roja y una E negra, vimos que se hacía necesario, con los elementos que contábamos en ese momento, duplicar el procedimiento y cambiar solo el dato del color. Volvamos a repasar el problema de la duplicación, pero utilizando el ejemplo inicial del cuadrado negro dado en la página 59 con el procedimiento `DibujarCuadradoNegroDeLado3()`, cuyo resultado se mostró en el gráfico G.2.10. Entonces, ahora supongamos que queremos que en lugar de dibujar dos cuadrados negros, uno de ellos sea rojo y el otro negro, como se muestra en el gráfico G.3.1. El código para ello, suponiendo solo las herramientas del capítulo anterior, sería algo como

```
program
 {
   VaciarTablero(); IrAlOrigen()
   DibujarCuadradoNegroDeLado3()
   IrAlBorde(Este); Mover(Oeste); Mover(Oeste)
   DibujarCuadradoRojoDeLado3()
 }

procedure DibujarCuadradoNegroDeLado3()
 /*
```

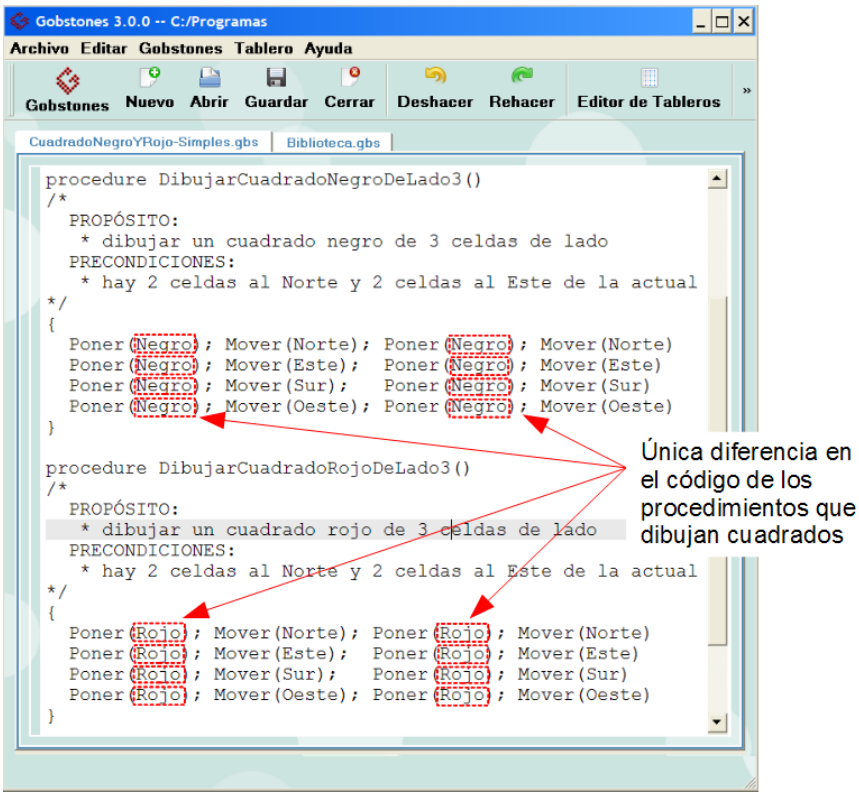
G.3.1. Resultado de ejecutar el programa que dibuja dos cuadrados

```
    PROPÓSITO:
     * dibujar un cuadrado negro de 3 celdas de lado
    PRECONDICIONES:
     * hay 2 celdas al Norte y 2 celdas al Este de la actual
 */
 {
   Poner(Negro); Mover(Norte); Poner(Negro); Mover(Norte)
   Poner(Negro); Mover(Este);  Poner(Negro); Mover(Este)
   Poner(Negro); Mover(Sur);   Poner(Negro); Mover(Sur)
   Poner(Negro); Mover(Oeste); Poner(Negro); Mover(Oeste)
 }

procedure DibujarCuadradoRojoDeLado3()
 /*
    PROPÓSITO:
     * dibujar un cuadrado rojo de 3 celdas de lado
    PRECONDICIONES:
     * hay 2 celdas al Norte y 2 celdas al Este de la actual
 */
 {
   Poner(Rojo); Mover(Norte); Poner(Rojo); Mover(Norte)
   Poner(Rojo); Mover(Este);  Poner(Rojo); Mover(Este)
   Poner(Rojo); Mover(Sur);   Poner(Rojo); Mover(Sur)
   Poner(Rojo); Mover(Oeste); Poner(Rojo); Mover(Oeste)
 }
```

Observar la repetición de código, puesto que tenemos dos procedimientos casi idénticos, que difieren únicamente en el color de cada uno de los `Poner` utilizados. En el gráfico G.3.2 podemos ver resaltadas esas diferencias.

Pero tener varios procedimientos únicamente porque un valor determinado es diferente es incómodo y poco eficiente. Si quisiéramos cuadrados de los 4 colores habría que tener 4 procedimientos. Y si hubiese que modificarlos para cambiar el tamaño del cuadrado, por ejemplo, habría que hacer las modificaciones en cada



G.3.2. Diferencias en los procedimientos simples que dibujan un cuadrado negro y otro rojo.

uno de ellos.

Actividad de Programación 1

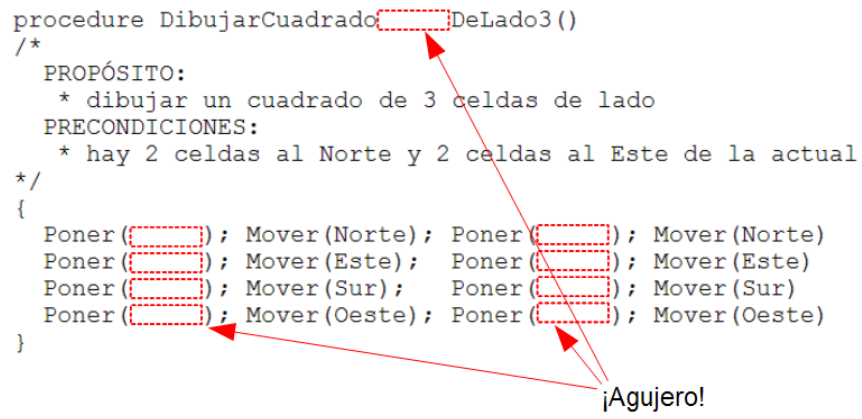
Escriba los dos procedimientos simples faltantes para poder dibujar cuadrados de los 4 colores. El objetivo de esta actividad es mostrar lo tedioso que resulta duplicar código solo por el cambio de un valor (en este caso, el color).

Sin embargo, los procedimientos, al diferir únicamente en el color, tienen todos algo en común. Si tomamos uno de ellos y recortamos el color, quedaría un procedimiento con un "agujero", como si fuera un rompecabezas al que le falta una pieza. El resultado de tal técnica se muestra en el gráfico G.3.3.

A este "procedimiento con agujero" se lo suele denominar *esquema* o *template*, y requiere que el agujero sea rellenado antes de poder funcionar.

¡Si ahora repetimos este procedimiento con los otros procedimientos, podemos observar que el esquema es exactamente el mismo! Esta observación nos permite pensar en un mecanismo donde definiésemos el esquema, y cada vez que precisásemos dibujar un cuadrado de cierto color, usaríamos el esquema, rellenando el "agujero" con el color que deseáramos. Según el valor que eligiésemos para rellenar el agujero, podríamos tener diferentes procedimientos. Este proceso se ilustra en el gráfico G.3.4.

Sin embargo, el agujero no puede expresarse gráficamente en un lenguaje de programación. Para poder expresarlo debemos utilizar texto. Entonces, la idea es ponerle un nombre al "agujero" y entender que cada vez que querramos mencionar al agujero, usaremos ese nombre. Ese mecanismo se conoce con el nombre de *parámetro*.



G.3.3. Procedimiento para dibujar un cuadrado, con un “agujero” para el color

Leer con Atención

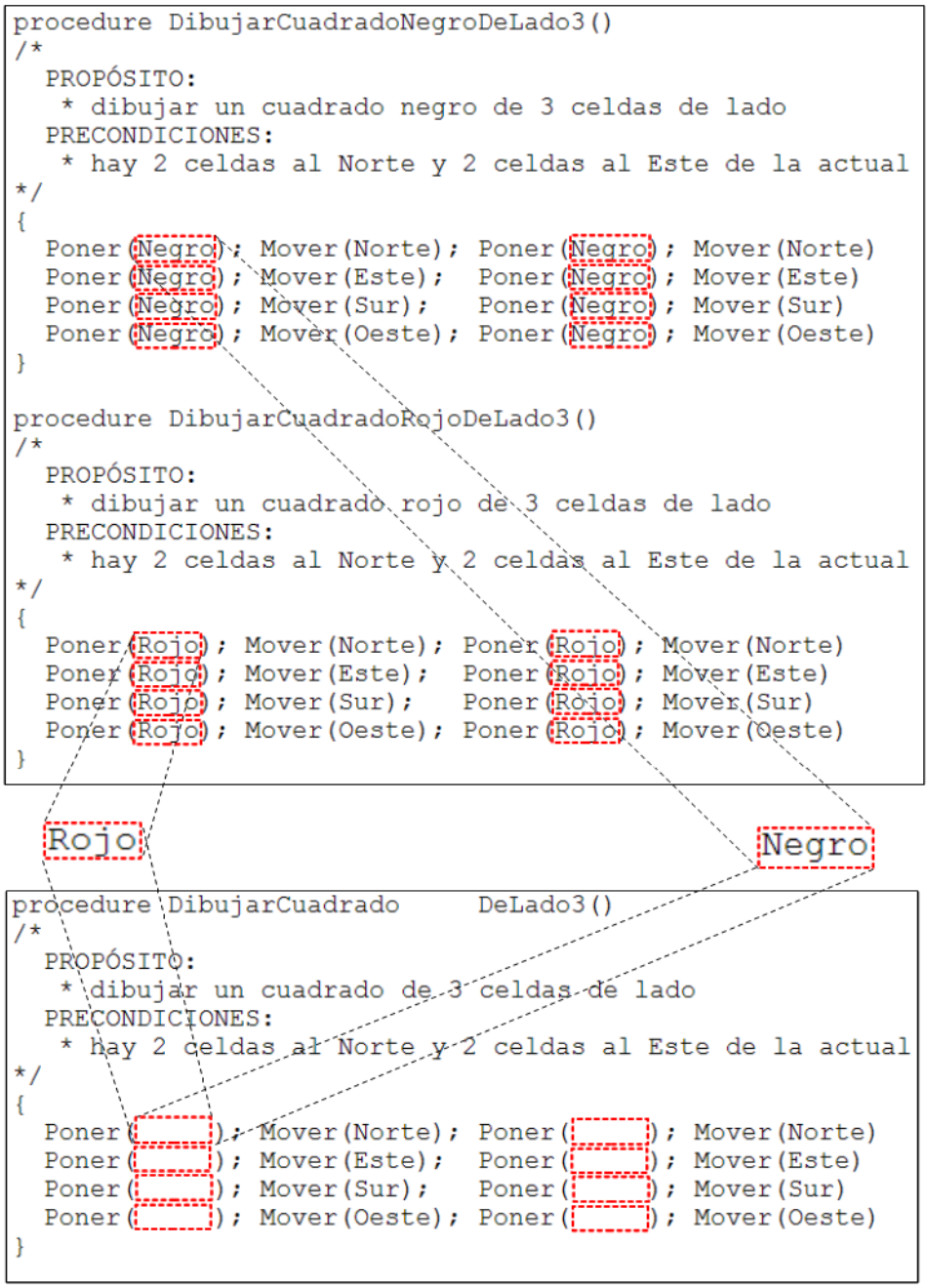
Un *parámetro* es un nombre que usamos para escribir un “agujero” dentro de un (esquema de) procedimiento. Este “agujero” debe ser rellenado adecuadamente con un valor antes de poder utilizar dicho procedimiento.

En nuestro caso, un nombre adecuado para el agujero (el parámetro) sería, por ejemplo, `colorDelCuadrado`. Así, en lugar de escribir un agujero gráficamente, usaremos ese nombre para indicar que ahí hay un agujero. Ese nombre aparecerá en el encabezado del procedimiento, como se ve en el siguiente código:

```
procedure DibujarCuadradoDeLado3DeColor(colorDelCuadrado)
 /*
    PROPÓSITO:
     * dibujar un cuadrado de 3 celdas de lado,
       de color colorDelCuadrado
    PRECONDICIONES:
     * hay 2 celdas al Norte y 2 celdas al Este de la actual
 */
 {
   Poner(colorDelCuadrado); Mover(Norte)
   Poner(colorDelCuadrado); Mover(Norte)
   Poner(colorDelCuadrado); Mover(Este)
   Poner(colorDelCuadrado); Mover(Este)
   Poner(colorDelCuadrado); Mover(Sur)
   Poner(colorDelCuadrado); Mover(Sur)
   Poner(colorDelCuadrado); Mover(Oeste)
   Poner(colorDelCuadrado); Mover(Oeste)
 }
```

Observar que en cada lugar donde aparecía el agujero en el gráfico G.3.3 ahora aparece el nombre `colorDelCuadrado`. Además, el nombre `colorDelCuadrado` también aparece mencionado en el encabezado del procedimiento; esto indica que este procedimiento tiene un agujero con ese nombre, y provee la forma mediante la cual luego el agujero puede ser rellenado con un valor.

El agujero que nombramos mediante un parámetro debe ser siempre rellenado con un valor. Así, un parámetro es un nombre que denota un valor (el que rellena el agujero), el cual cambia en cada llamado al procedimiento (pues el valor usado puede cambiar en cada llamado). La forma de indicar que el procedimiento tiene un parámetro es utilizar un identificador entre paréntesis luego del



G.3.4. Relación entre los procedimientos de dibujar cuadrados y un *esquema* de procedimiento (o “procedimiento con agujero”)

nombre del procedimiento. Ese identificador puede luego ser utilizado dentro del bloque que define el procedimiento como si de un valor fijo se tratase; al utilizarlo es como si hubiésemos llenado el agujero nombrado por el parámetro con el valor específico. En un procedimiento puede incluirse más de un agujero, o sea, puede haber más de un parámetro.

Definición 3.1.1. *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.*

Los nombres de los parámetros pueden ser usados en los mismos lugares que las expresiones que reemplazan. Por esta razón, la utilización de un parámetro es una forma de expresión que sirve para describir un valor que no se conoce.

Al igual que con la definición de un procedimiento simple, al definir un procedimiento con parámetros estamos agregando un nuevo comando al lenguaje. En este caso, el nuevo comando será `DibujarCuadradoDeLado3DeColor(`<*color*>`);` o sea, al invocarlo debemos suministrar un color específico para dibujar. Entonces, podremos escribir el programa que dibuja un cuadrado negro y uno rojo como

```
program
 {
   VaciarTablero(); IrAlOrigen()
   DibujarCuadradoDeLado3DeColor(Negro)
   IrAlBorde(Este); Mover(Oeste); Mover(Oeste)
   DibujarCuadradoDeLado3DeColor(Rojo)
 }
```

Observar como se usan los comandos `DibujarCuadradoDeLado3DeColor(Negro)` y `DibujarCuadradoDeLado3DeColor(Rojo)`. Tener en cuenta que en el archivo del programa también debe colocarse la definición del procedimiento `procedure DibujarCuadradoDeLado3DeColor(colorDelCuadrado)`.

Al utilizar un procedimiento como un nuevo comando debemos suministrar un valor para el parámetro, ya que el parámetro por sí solo no tiene sentido (pues representa a un “agujero”). Este valor concreto usado en la invocación recibe el nombre de *argumento*.

Definición 3.1.2. *Un* argumento *es el valor específico usado en la invocación de un procedimiento con parámetros.*

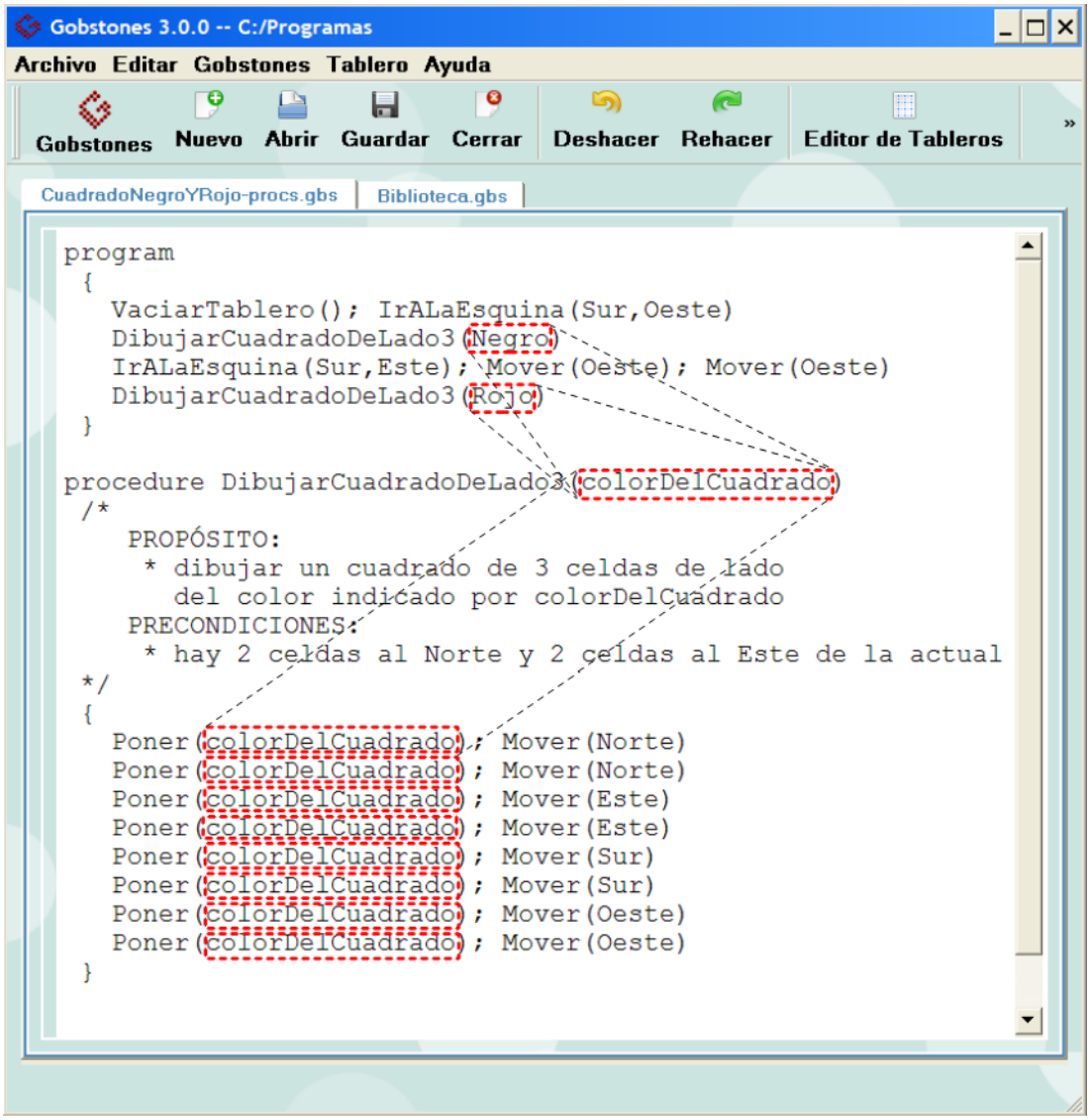
Definición 3.1.3. *Un procedimiento definido por el programador puede ser utilizado como comando, de la siguiente forma*

<*procName*>(<*args*>)

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

Llamaremos *invocación* de un procedimiento al uso del mismo como un comando nuevo. El argumento aparece entre paréntesis al invocar un procedimiento con parámetros como comando. Esta es la forma de indicar que el valor del argumento debe ser utilizado en el lugar donde se colocó el agujero nombrado por el parámetro. Este proceso se ilustra en el gráfico G.3.5.

Puede verse que es más conveniente escribir un solo procedimiento para dibujar el cuadrado que reciba el color como parámetro, ya que entonces en el



G.3.5. Programa que dibuja un cuadrado negro y uno rojo con un único procedimiento parametrizado para dibujar el cuadrado

programa principal podemos utilizar el mismo comando para dibujar cuadrados de dos colores diferentes, evitando la repetición del código de dibujar cuadrados solo para tener diferentes colores. Observar que `colorDelCuadrado` es el parámetro en la definición de `DibujarCuadradoDeLado3DeColor` y se lo utiliza en el comando `Poner(colorDelCuadrado);` el nombre utilizado no es importante, pero debe indicar de alguna manera (por cuestiones de estilo) la intención de para qué se utilizará el parámetro. Al invocar el procedimiento paramétrico `DibujarCuadradoDeLado3DeColor` se utiliza un valor específico de color como argumento en cada caso. ¡El código final es equivalente al código original, pero que define solo un procedimiento, en lugar de cuatro!

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 (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*, por el uso de parámetros para representar las variaciones de una misma idea. Al uso de parametrización para expresar varios procedimientos simples mediante uno parametrizado se lo denomina también *generalización*, pues el procediminento parametrizado es más general que los casos individuales.

¿Qué otras nociones son generalizables en el caso de dibujar un cuadrado? Otra noción generalizable en este caso 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, en la subsección siguiente.

Hay una noción más que resulta generalizable, y tiene que ver con los visto en la sección 2.2.9: es mejor dibujar el cuadrado con procedimientos que dibujen líneas. Para ver como aprovechar esta idea, volveremos momentáneamente al ejemplo de dibujar un cuadrado exclusivamente negro (sin parametrizar el color). En ese ejemplo utilizamos procedimientos

- `DibujarLineaNegra2HaciaElNorte`,
- `DibujarLineaNegra2HaciaElEste`,
- `DibujarLineaNegra2HaciaElSur` y
- `DibujarLineaNegra2HaciaElOeste`.

para dibujar líneas. Podemos observar en estos procedimientos el mismo patrón de repetición que vimos en el caso del cuadrado: ¡los procedimientos solo difieren en la dirección en la que deben moverse! Acá tenemos un candidato para utilizar parametrización.

Actividad de Programación 2

Realice el ejercicio 3.1.1 y utilice dicho procedimiento para dibujar dos cuadrados negros de lado 3.

Ejercicio 3.1.1. *Escribir un procedimiento* `DibujarCuadradoNegro3` *similar al realizado en el ejercicio 2.2.8, pero utilizando un único procedimiento parametrizado para dibujar líneas. Observar que en este caso lo que debe parametrizarse es la dirección.*

El resultado debería contener un programa principal, un procedimiento para dibujar el cuadrado, y un único procedimiento parametrizado para dibujar líneas. El procedimiento para dibujar el cuadrado debería realizar 4 invocaciones al procedimiento de dibujar líneas con diferentes argumentos (de dirección en este caso).

Para Reflexionar

¿Qué nombre utilizó para el procedimiento de dibujar líneas? ¿Y para el parámetro de dirección?
Si utilizó un nombre como `DibujarLineaNegra2HaciaEl` para el procedimiento y `direccionDeLinea` o `dirDeLinea` para el parámetro significa que viene entendiendo las nociones impartidas hasta el momento. Si en cambio utilizó un nombre poco significativo, debería volver a revisar la subsección 2.3.2 que explica la importancia de un buen estilo al elegir nombres y reflexionar sobre lo que allí explica.

En el caso de las líneas, además de parametrizar la dirección, podríamos también parametrizar el color. Ello es posible porque un procedimiento puede tener más de un parámetro. En este caso, el procedimiento para dibujar líneas quedaría así

```
procedure DibujarLinea2Hacia(colorDeLinea, dirDeLinea)
 /*
    PROPÓSITO:
     * dibujar una línea de 2 celdas de longitud en la dirección
       dirDeLinea y de color colorDeLinea
    PRECONDICIONES:
     * hay 2 celdas en la dirección indicada por dirDeLinea
 */
 {
   Mover(colorDeLinea); Poner(dirDeLinea)
   Mover(colorDeLinea); Poner(dirDeLinea)
 }
```

Podemos observar que este procedimiento tiene 2 parámetros: el primero es un color para el comando `Poner`, y el segundo es una dirección para el comando `Mover`. Al invocarlo deben pasársele, consecuentemente, 2 argumentos. Por ejemplo, para dibujar el cuadrado rojo haríamos

```
procedure DibujarCuadradoRojoDeLado3()
 /*
    PROPÓSITO:
     * dibujar un cuadrado de 3 celdas de lado,
       de color rojo
    PRECONDICIONES:
     * hay 2 celdas al Norte y 2 celdas al Este de la actual
    OBSERVACIONES:
     * utiliza un procedimiento parametrizado para
       dibujar líneas
 */

 {
   DibujarLinea2Hacia(Rojo, Norte)
   DibujarLinea2Hacia(Rojo, Este)
   DibujarLinea2Hacia(Rojo, Sur)
   DibujarLinea2Hacia(Rojo, Oeste)
 }
```

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 y orden de los argumentos usados para invocarlo. Por ejemplo, el procedimiento `DibujarLineaNegra2HaciaEl` del ejercicio 3.1.1 tiene un único parámetro `dirDeLinea` y cuando se invoca dicho procedimiento se debe suministrar un único valor; es válido invocar al procedimiento como

```
DibujarLineaNegra2HaciaEl(Norte)
```

así como también con el resto de las direcciones (¡pero no con colores, por ejemplo!). La misma correspondencia debe existir si se usan varios parámetros en la definición del procedimiento: al invocarlo deben usarse la misma cantidad de argumentos y en el orden correcto. Si consideramos el procedimiento `DibujarLinea2Hacia` vemos que `DibujarLinea2Hacia(Negro,Norte)` es una invocación válida del procedimiento, pues corresponden el número de los argumentos con los parámetros declarados y están en orden. En cambio una invocación inválida sería `DibujarLinea2Hacia(Norte)`, ya que no concuerda la cantidad de argumentos entre el uso y la declaración. Otro error posible sería hacer la invocación con los parámetros invertidos: `DibujarLinea2Hacia(Norte,Negro)`, pues en la definición el parámetro de nombre `colorDeLinea` se utiliza para invocar al comando `Poner`, y si rellenamos dicho parámetro con el argumento `Norte`, eso producirá un error (y lo mismo con `Negro`, `dirDeLinea` y `Mover`).

¿Y cómo podemos hacer para utilizar el procedimiento `DibujarLinea2Hacia` en un procedimiento `DibujarCuadrado3` que dibuje un cuadrado parametrizando su color? La respuesta es sencilla: debemos usar el parámetro `colorDeCuadrado` como argumento para el procedimiento de dibujar líneas. El resultado será

Puesto que un parámetro representa un valor, podemos utilizarlo como argumento al invocar otros procedimientos. La idea sería que el parámetro representa un agujero, el cual fue llenado con un valor al invocar el procedimiento principal, y al utilizarlo como argumento tomamos el valor que está en el agujero y lo colocamos en el agujero del procedimiento secundario.

```
procedure DibujarCuadrado3(colorDeCuadrado)
 /*
    PROPÓSITO:
     * dibujar un cuadrado de 3 celdas de lado,
       del color dado por colorDeCuadrado
    PRECONDICIONES:
     * hay 2 celdas al Norte y 2 celdas al Este de la actual
    OBSERVACIONES:
     * utiliza un procedimiento parametrizado para
       dibujar líneas
     * utiliza el parámetro de DibujarCuadrado3 como
       argumento de DibujarLinea2Hacia
 */

 {
   DibujarLinea2Hacia(colorDeCuadrado, Norte)
   DibujarLinea2Hacia(colorDeCuadrado, Este)
   DibujarLinea2Hacia(colorDeCuadrado, Sur)
   DibujarLinea2Hacia(colorDeCuadrado, Oeste)
 }
```

Esto sería como llenar el “agujero” de `DibujarLinea2Hacia` con el contenido del “agujero” `colorDeCuadrado`. Esto es válido, pues el parámetro representa a un valor específico dentro del procedimiento, y por lo tanto puede utilizarse ese valor como argumento. Agrupamos todas estas nociones en la siguiente actividad de programación.

Actividad de Programación 3

Realizar el ejercicio 3.1.2, y probarlo dibujando cuadrados de al menos 3 colores en 3 posiciones diferentes del tablero. Recordar que para probar un ejercicio debe escribir un programa que invoque adecuadamente los procedimientos necesarios.

Ejercicio 3.1.2. *Definir un procedimiento* `DibujarCuadrado3` *que, dado un color como parámetro, dibuje un cuadrado de lado dos de dicho color, utilizando un único procedimiento para dibujar líneas.*

Otro ejercicio interesante donde podríamos intentar utilizar parámetros es el de dibujar la letra E visto en el capítulo anterior.

Actividad de Programación 4

Realizar nuevamente los ejercicios 2.4.5 y 2.4.11 de la unidad anterior, intentando no duplicar la definición de procedimientos auxiliares, pero capturando las subtareas adecuadas con procedimentos. Utilizar la idea de parámetro vista en este apartado.

Esta idea de utilizar el parámetro de un procedimiento como argumento en la invocación de otro procedimiento abre una nueva pregunta: ¿dónde tiene sentido utilizar un parámetro? Dado que un parámetro permite definir un esquema de procedimiento, nombrando un valor que será diferente en cada llamado de dicho procedimiento, el único lugar donde tendría sentido el parámetro es dentro de ese mismo procedimiento.

Leer con Atención

Es por tanto razonable esperar que la validez del parámetro sea solo dentro del cuerpo del procedimiento que lo define. Esta noción de validez se conoce con el nombre de *alcance* (en inglés *scope*), y es extremadamente importante, pues es un error usual al comenzar a utilizar parámetros el intentar utilizar un parámetro fuera de su alcance, provocando errores extraños.

Definición 3.1.4. *El* alcance *de un parámetro es la región del programa donde el mismo tiene validez y puede ser utilizado.*

Diferentes herramientas (parámetros, variables, etc.) poseen alcance, y la forma de definirlo varía de herramienta en herramienta y de lenguaje en lenguaje. En GOBSTONES el alcance es el más simple de todos, siendo exclusivamente *local*, lo cual quiere decir que un parámetro solamente es conocido en el cuerpo del procedimiento que lo define. Podríamos entender esto pensando en un procedimiento como una familia, y en el nombre del parámetro como un apodo familiar: personas ajenas a la familia no conocerán este apodo, y diferentes familias pueden usar el mismo apodo para diferentes personas.

Antes de profundizar en estas nociones, vamos a incorporar algunos elementos para poder armar ejemplos más complejos.

3.1.2. Formas básicas de repetición de comandos

Al pensar en parametrizar el dibujo del cuadrado, surge inmediatamente la idea de parametrizar su tamaño. Esto se haría pasando un parámetro numérico que indicase el tamaño esperado del lado del cuadrado. Sin embargo, con las herramientas del lenguaje que hemos presentado hasta el momento no es posible construir ningún procedimiento que aproveche un parámetro numérico. Por ello, es necesario presentar una nueva forma de armar comandos compuestos: la *repetición*.

Repetición simple

Cuando es necesario repetir un comando un cierto número de veces, en lugar de copiar el mismo comando esa cantidad de veces, podemos describir que queremos el mismo comando repetido varias veces. La forma que se utiliza para definir una *repetición simple* en GOBSTONES es `repeat`. Por ejemplo, para poner 10 bolitas rojas en la celda actual se puede escribir el siguiente comando:

```
repeat(10)
  { Poner(Rojo) }
```

El comando `repeat` describe de manera concisa la repetición simple de otro comando una cierta cantidad de veces. La definición es la siguiente.

Definición 3.1.5. *Un comando de* repetición simple *permite que una acción se llevada a cabo un cierto número de veces, siendo la cantidad a repetir indicada por un número. El comando* `repeat` *que permite la repetición simple tiene la siguiente forma:*

`repeat (`*<numero>*`)`
<bloque>

siendo <numero> una expresión que describe una cantidad, y <bloque> un bloque cualquiera que se repetirá.

El efecto de un comando de repetición simple es que la acción descrita por el bloque *<bloque>* se repetirá tantas veces como fuera indicado por el número dado. Si el número fuera negativo o cero, el comando de repetición simple no realiza ninguna acción. Por ejemplo, el siguiente comando no tiene ningún efecto:

```
repeat(-10)
  { Poner(Rojo) }
```

puesto que no se puede repetir una cantidad negativa de veces.

Leer con Atención

El verdadero poder de la repetición viene de su combinación con parámetros.

Por ejemplo, se puede definir un procedimiento que coloque un cantidad distinta de bolitas de color Rojo en la celda actual cada vez que se lo usa, a través del siguiente procedimiento parametrizado:

```
procedure PonerNRojas(cantidad)
  {-
     PROPÓSITO:
     * coloca en la celda actual la cantidad de bolitas rojas
       indicada (y ninguna si cantidad<=0)
     PRECONDICIÓN: ninguna (es una operación total)
  -}
  {
    repeat(cantidad) { Poner(Rojo) }
  }
```

Entonces, el comando `PonerNRojas(3)` pondrá 3 bolitas en la celda actual, y el comando `PonerNRojas(10)` pondrá 10 bolitas. Como se puede observar, utilizando este comando no es necesario copiar una y otra vez un comando en secuencia para repetir varias veces, y además podemos controlar la cantidad con un parámetro.

Observar que ciertos datos son directamente dados como parámetro, indicando incluso el nombre que debe usarse para los mismos. Por ejemplo, el enunciado "*que deposite* `cant` *bolitas*" quiere decir que el procedimiento tendrá un parámetro de nombre `cant` que indique la cantidad en forma de número. Lo mismo con otros parámetros.

Actividad de Programación 5

Realizar los procedimientos solicitados en el ejercicio 3.1.3. Utilizar el primero de ellos para colocar 17 bolitas azules y 42 bolitas negras en la celda actual. ¿Cuál es la precondición del programa?

Ejercicio 3.1.3. *Escribir los procedimientos que se describen.*

1. `PonerN` *que deposite* `cant` *bolitas de color* `color` *en la celda actual, suponiendo que* `cant` *es un número positivo. ¿Cuál es su precondición?*
2. `MoverN` *que se mueva* `cant` *celdas en la dirección* `dir` *desde la celda actual, suponiendo que* `cant` *es un número positivo. ¿Cuál es su precondición?*
3. `SacarN` *que quite* `cant` *bolitas de color* `color` *de la celda actual, suponiendo que* `cant` *es un número positivo. ¿Cuál es su precondición?*

Leer con Atención

Los procedimientos `PonerN`, `MoverN` y `SacarN` son excelentes candidatos para ser colocados en la `Biblioteca`.

Actividad de Programación 6

Realice los ejercicios 3.1.4 y 3.1.5 y pruébelos adecuadamente. No olvide escribir el propósito y las precondiciones de los procedimientos involucrados.

Ejercicio 3.1.4. *Escribir un procedimiento* `DibujarLineaVerticalRoja` *que dada una cantidad* `cant`*, dibuje una línea vertical de bolitas rojas desde la celda actual hacia el norte, y retorne el cabezal a la posición inicial.*

Ayuda: *es necesario utilizar una repetición para dibujar y otra diferente para volver a la posición inicial.*

Ejercicio 3.1.5. *Escribir un procedimiento* `DibujarCuadradoSolido5` *que dada una cantidad* `cant`*, dibuje un cuadrado sólido rojo de* `cant` *bolitas de lado (o sea, no solo el perímetro, sino toda la superficie del cuadrado) hacia el norte y el este, y retorne el cabezal a la posición inicial.*

Ayuda: *utilizar el procedimiento del ejercicio anterior para dibujar cada una de las barras verticales del cuadrado mediante una repetición, y otra repetición para volver.*

Para Reflexionar

Al parametrizar direcciones o colores expresamos 4 procedimientos simples en uno parametrizado. Sin embargo, al parametrizar cantidades estamos expresando una cantidad infinita de procedimientos simples con uno solo. Reflexione sobre la necesidad de la repetición para conseguir esto, y sobre las limitaciones que impondría el lenguaje de programación si no contásemos con la misma. Cuando decimos que al aprender a programar debemos concentrarnos en ideas importantes nos estamos refiriendo exactamente a este tipo de imprescindibilidad.

Repetición indexada

La repetición, en combinación con los parámetros, es una forma poderosa de aumentar la expresividad del lenguaje, y ser capaces de escribir pocas líneas de código para lograr muchas variaciones. Sin embargo, hay variaciones que la forma simple de la repetición no permite. Por ejemplo, si en lugar de un rectángulo sólido como en el ejercicio 3.1.5, quisiéramos dibujar un triángulo sólido como el del gráfico G.3.6, dibujando cada una de las barras verticales, no sería posible hacerlo con repetición simple, ¡pues la longitud de cada barra vertical es diferente! Está claro que precisamos alguna forma más poderosa de la repetición,

G.3.6. Triángulo rectángulo a dibujar por barras verticales

que nos permita utilizar un valor diferente cada vez que se repite el comando indicado.

La *repetición indexada* es una forma de repetición que considera una secuencia de valores, y repite un cierto comando una vez por cada elemento de ella. Esta secuencia puede determinarse a partir de un rango de valores (especificado a través de dos valores, un *valor inicial* y un *valor final*) y comprende todos los valores que se encuentren entre el inicial y el final del rango, incluídos ellos. Por ejemplo, el rango `[1..10]` determina la secuencia que contiene los valores `1 2 3 4 5 6 7 8 9 10`; en cambio el rango `[12..17]` determina la secuencia que contiene los valores `12 13 14 15 16 17`.

Definición 3.1.6. *Un* rango *de valores es una secuencia de valores crecientes comprendidas entre un* valor inicial *y un* valor final*, incluyendo a estos dos. La forma de escribir un rango es*

[<*inicioRango*>..<*finRango*>],

donde <*inicioRango*> *establece el valor inicial y* <*finRango*> *establece el valor final.*

Mediante el uso de rangos y utilizando el comando `foreach`, podemos hacer ahora una repetición indexada. Por ejemplo, para dibujar triángulos sólidos como el visto antes usaríamos el siguiente comando

```
foreach cant in [1..5]
  {
    DibujarLineaVerticalRoja(cant)
    Mover(Este)
  }
```

Observar que esta forma de repetición indica un secuencia de valores mediante un rango, y además un nombre (`cant` en el ejemplo), que en cada repetición tomará el valor del elemento de la secuencia para el que se está repitiendo en ese momento. A este nombre se lo conoce con el nombre de *índice* y de ahí la forma de denominar a este comando. El índice nombrará cada uno de los valores de la

secuencia en cada repetición; o sea, en la primera repetición `cant` nombrará al `1`, en la segunda repetición `cant` nombrará al `2`, etc.; por eso el comando se llama `foreach`. Así, el código anterior sería equivalente a haber escrito

En inglés, *for each* significa *para cada uno*. El comando entonces se podría leer como "para cada uno de los cant que están en el rango [1..5], realizar el comando indicado".

```
DibujarLineaVerticalRoja(1)
Mover(Este)
DibujarLineaVerticalRoja(2)
Mover(Este)
DibujarLineaVerticalRoja(3)
Mover(Este)
DibujarLineaVerticalRoja(4)
Mover(Este)
DibujarLineaVerticalRoja(5)
Mover(Este)
```

puesto que la secuencia dada por el rango `[1..5]` está compuesta por los valores `1 2 3 4 5`.

Definición 3.1.7. *El comando de* repetición indexada *permite repetir una acción un cierto número de veces, siendo esta cantidad indicada por un índice que varía dentro de una cierta secuencia de valores. El comando* `foreach` *que permite la repetición indexada tiene la siguiente forma:*

`foreach` <*indexName*> `in` <*rango*>
<*bloque*>

siendo <*indexName*> *un identificador con minúscula que nombra al índice de la repetición,* <*rango*> *el rango de valores que describe a la secuencia sobre la que variará dicho índice, y* <*bloque*> *un bloque cualquiera que se repetirá.*

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 la secuencia, y de tal manera que en cada repetición el índice representará a un valor diferente. El nombre de un índice puede utilizarse en cualquier lugar donde se habría utilizado el valor que el mismo describe; por esta razón, el uso de un índice es una expresión que sirve para describir un valor que varía de una forma fija durante las repeticiones.

El comando de repetición simple puede obtenerse como una forma simplificada del comando de repetición indexada, simplemente haciendo que el bloque ignore el valor del índice. Así, en lugar de escribir

```
repeat(10) { Poner(Rojo) }
```

podría haberse escrito

```
forach i in [1..10] { Poner(Rojo) }
```

ya que el bloque no utiliza para nada el índice `i`. También podría haberse escrito

```
foreach i in [17..26] { Poner(Rojo) }
```

pues la secuencia dada por este otro rango, aunque tiene diferentes valores, tiene la misma cantidad de elementos que el anterior.

Para Reflexionar

¿Por qué entonces hará falta tener ambas formas de repetición? La cuestión es la simplicidad conceptual, y asociado a esto, la gradualidad didáctica. El concepto de repetición no tiene asociado necesariamente un índice, y existen muchas formas de poner 10 bolitas utilizando repetición indexada. Reflexionar sobre la importancia de la claridad y la simplicidad de los conceptos impartidos en un curso inicial.

(a). Tablero a producir como resultado.

(b). Tablero a producir como variante.

G.3.7. Tableros a producir en el ejercicio 3.1.6

Una característica particular de los rangos es que siempre determinan secuencias crecientes; o sea, el valor inicial debe ser menor o igual que el valor final para que la secuencia tenga elementos, y el incremento por el momento es siempre 1. Si el valor inicial es mayor que el final, la secuencia es vacía y el comando de repetición no realiza ninguna acción. Por ejemplo, el siguiente comando no tiene ningún efecto:

Veremos en el próximo capítulo como permitir variaciones como incrementos diferentes de 1, o rangos decrecientes.

```
foreach i in [10..1]
  { DibujarLineaVerticalRoja(i) }
```

puesto que el rango `10..1` describe a la secuencia vacía (no tiene elementos), ya que el valor inicial, `10`, es mayor que el valor final, `1`.

Actividad de Programación 7

Realizar el ejercicio 3.1.6. Utilizarlo para obtener en el tablero final la configuración de bolitas del gráfico G.3.7a.
Ayuda: observe que cada fila del dibujo puede ser generada usando una instancia particular del procedimiento `Progresion`, utilizando los argumentos adecuados.

Ejercicio 3.1.6. *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 y dejando el cabezal en la celda siguiente a la última ocupada.*

Actividad de Programación 8

¿Qué tendría que cambiarse en el procedimiento `Progresion` de la actividad anterior para obtener la configuración del gráfico G.3.7b sin cambiar el programa principal?

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 negro de 3 celdas de lado puede escribirse de la siguiente manera:

```
procedure DibujarCuadradoNegro3()
 {-
    PROPÓSITO:
    * dibuja un cuadrado Negro de lado 3
    PRECONDICIÓN:
    * hay 2 celdas al Norte y 2 al Este de la celda actual
    OBSERVACIÓN:
    * utiliza una repetición indexada sobre direcciones
 -}
 {
                      -- OBSERVAR que son direcciones!
   foreach dir in [Norte..Oeste]
     { DibujarLineaNegra2Hacia(dir) }
 }
```

Observar que la repetición utiliza las direcciones como índice, y que por el orden de las mismas, utiliza las 4 direcciones. El significado de esta repetición indexada es exactamente el mismo que si hubiéramos escrito

```
DibujarLineaNegra2Hacia(Norte)
DibujarLineaNegra2Hacia(Este)
DibujarLineaNegra2Hacia(Sur)
DibujarLineaNegra2Hacia(Oeste)
```

puesto que el rango `[Norte..Oeste]` es la secuencia `Norte Sur Este Oeste`.

Con la repetición indexada puede realizarse un procedimiento para dibujar cuadrados de cualquier longitud. Esta será la forma final que daremos a este procedimiento, generalizando todo lo que es posible generalizar (el tamaño, el color y la dirección en las líneas), y utilizando los recursos adecuados para no repetir código innecesariamente.

Actividad de Programación 9

Realizar el ejercicio 3.1.7 y utilizar el procedimiento definido para dibujar varios cuadrados de distintos tamaños y colores.

Ejercicio 3.1.7. *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 (tener en cuenta que las líneas tienen ancho 1, y el ancho de la línea que se superpone también cuenta para el total). Utilizar la menor cantidad de procedimientos posible.*

3.1.3. Ejercitación

En esta subsección revisamos algunos de los ejercicios de práctica del capítulo anterior, con el fin de incorporar las nociones de parámetros y repetición.

Actividad de Programación 10

Realice los ejercicios enunciados en esta subsección. Recuerde utilizar todas las buenas prácticas que venimos estudiando (adecuada separación en subtareas, elección de buenos nombres de procedimientos y parámetros, indentación de código, reutilización de código ya realizado, etcétera).

Recuerde que *generalizar* un procedimiento es cambiar algún dato fijo del mismo por un parámetro, de forma tal que con un código casi idéntico al anterior, ahora se expresan muchos más procedimientos.

Ejercicio 3.1.8. *Rehacer el ejercicio 2.4.1 usando repetición. Rehacer el ejercicio 2.4.2 usando repetición. Generalizarlos a un procedimiento* `PonerNDeCada` *que tome la cantidad como parámetro.*

¿Reutilizó el procedimiento `PonerUnaDeCadaColor` al escribir los demás? Si no lo hizo, ¡vuelva a revisar el ejercicio y hágalo!

Para Reflexionar

¿Puede observar cómo el uso de repetición y parámetros simplifica el código producido, al tiempo que lo hace más generalizable? Reflexione sobre la importancia de contar con adecuadas herramientas abstractas de programación, tales como los procedimientos, los parámetros y las estructuras de control como la repetición.

Ejercicio 3.1.9. *Rehacer el ejercicio 2.4.4, pero generalizando el color y la dirección. Usar el resultado para rehacer el ejercicio 2.4.5 generalizando el color, y usando una repetición sobre direcciones.*

Ejercicio 3.1.10. *Rehacer los procedimientos del ejercicio 2.4.9 generalizando el color con el cual se dibuja la letra E. Rehacer el ejercicio 2.4.10 para que reutilice el procedimiento anterior y una repetición sobre colores.*

Como habrá podido comprobar mediante estos ejercicios, la combinación de repetición y parámetros es extremadamente poderosa. Además, ambos juegan un rol fundamental al permitir la generalización de muchas de las nociones que se utilizan al escribir procedimientos. Sin embargo, éstas no son las únicas herramientas que brindan expresividad y poder a la programación. En la sección que sigue veremos otro conjunto de herramientas poderosas.

3.2. Expresiones y funciones

En esta sección veremos cómo definir en GOBSTONES expresiones complejas, resultantes de combinar otras expresiones. Es algo similiar a lo hecho con comandos a través de procedimientos definidos por el usuario, es decir, agruparemos y nombraremos expresiones, solo que con una nueva herramienta conocida como *funciones*.

3.2.1. Expresiones compuestas y tipos

Las expresiones compuestas son algo familiar cuando hablamos de números. Así, sabemos que `2+3` está formado por tres partes: dos descripciones de números y un símbolo que denota la operación de sumar ambos números. En cualquier lenguaje de programación habrá diversas formas de armar expresiones compuestas que vendrán dadas por reglas de formación.

Para poder introducir las reglas de formación de expresiones compuestas en GOBSTONES vamos a comenzar por profundizar en la idea de tipo presentada en la subsección 2.1.4. 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.

Se definirán en esta unidad.

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 operación `Poner` no se considera válida a menos que su argumento sea un valor de tipo color.

Definición 3.2.1. *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.*

Actividad de Programación 11

Realice el ejercicio 3.2.1. Si puede, pruébelo en diferentes herramientas que implementen GOBSTONES.

Ejercicio 3.2.1. *Escribir un programa que incluya el comando* `Poner(17)` *y comprobar el error obtenido.*

Al presentar la operación `Poner` se estableció que es una operación total, y ahora resulta que existen formas inválidas de la misma. ¿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 de que un programa contenga combinaciones erróneas de tipos en las expresiones es rechazado sin ejecutarlo. Entonces `Poner(17)` no debería ser ejecutado; puesto que esto puede ser detectado automáticamente por un chequeo basado en sistemas de tipos, se elige no incluir errores de tipo en la noción de error de ejecución, y consecuentemente, se elige no incluir este caso en la precondición.

Leer con Atención

En GOBSTONES se permite invocar definiciones que esperan expresiones de un determinado tipo con expresiones de cualquier otro tipo, incluso cuando su uso es incorrecto. No obstante, si esto sucede el programa fallará al ser ejecutado. Sin embargo, al igual que otros lenguajes más avanzados, GOBSTONES permite la verificación de ciertas restricciones para los tipos de datos, chequeando de alguna forma en qué casos su uso es correcto. Sin embargo, esta característica es opcional, y puede no estar disponible en todas las versiones de la herramienta.

Por ejemplo, se dirá globalmente que `Poner` *espera un argumento de tipo color* para indicar que su uso no es válido si recibe un valor de otro tipo.

Para Ampliar

En la herramienta PYGOBSTONES existe una opción en la que se verifica, sin correr el programa, que nuestro código no contiene errores de tipos, y de esa manera asegurar el programa no tendrá errores de este estilo. Puede resultar interesante probarla y explorar las ventajas de utilizarla.

La idea de tipos también se aplica a las expresiones. Por ejemplo, la suma solo 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 un error de tipo.

Ejercicio 3.2.2. *Escribir un programa que incluya la expresión* `Verde+1` *y comprobar el error obtenido.*

Leer con Atención

La suma es también una forma de operación que requiere que sus dos sumandos sean descripciones de números, y solo en ese caso describe a un número y se considera válida. Esto describe así que la suma *espera dos argumentos de tipo numérico para producir un número*. Al igual que como con `Poner`, la suma se considera una operación total. De esta manera, diremos que la suma es una *operación total* ***sobre*** *números*.

Para Reflexionar

¿Recuerda la expresión `Rojo+Norte` que mencionamos al hablar de tipos en el capítulo anterior? Puesto que la suma es una operación sobre números, esta expresión, de ser ejecutada, producirá un error. ¿Qué le parece la idea de contar con una herramienta que verifique que esta forma de errores no suceda en el programa?

La idea de tipos se aplica también a los parámetros de un procedimiento. Por ejemplo, en el procedimiento `DibujarLinea2Hacia`, 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` solamente espera valores de tipo dirección. De esta manera, la invocación `DibujarLinea2Hacia(Rojo)` es fácilmente identificable como un error, pues `Rojo` no es de tipo dirección, sino de tipo color.

Leer con Atención

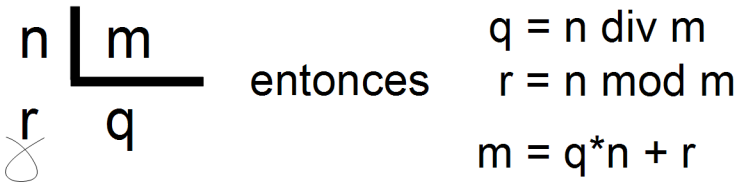
En el caso de más de un parámetro en el mismo procedimiento, cada uno tendrá un tipo específico, y al invocar dicho procedimiento, deberán suministrarse argumentos de los tipos correctos, en el orden establecido. Por ejemplo, en el procedimiento `DibujarLinea2Hacia`, el primer parámetro, `colorDeLinea`, es de tipo color, y el segundo parámetro, `dirDeLinea`, es de tipo dirección. Entonces, `DibujarLinea2Hacia(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 solo se la consignará en el contrato del procedimiento en cuestión estableciendo el tipo que se espera que tenga cada parámetro.

A través del uso de tipos se pueden visualizar rápidamente los usos correctos o erróneos de los diferentes elementos de un lenguaje de programación, y muchos lenguajes hacen uso de esta característica. En este libro usaremos la terminología de **tipos** para guiar la comprensión sobre las operaciones que iremos introduciendo.

3.2.2. Operaciones predefinidas para construir expresiones

Las expresiones compuestas se obtienen a través de un conjunto de expresiones predefinidas, combinándolas de maneras adecuadas. Las formas adecuadas quedan determinadas por el tipo de los argumentos y resultados, Así, para conocer el conjunto de operaciones predefinidas se puede seguir una clasificación de acuerdo con estos tipos. Para conocer un lenguaje de manera completa es



G.3.8. Descripción gráfica del resultado de las operaciones `div` y `mod` para números `n` y `m`.

importante conocer cuáles son las operaciones predefinidas de cada tipo. A continuación presentamos todas las operaciones predefinidas para cada uno de los cuatro tipos básicos de GOBSTONES, y damos ejemplos de su utilización.

Operaciones sobre números

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` (en otras palabras, `q` es el resultado entero de dividir `n` por `m`, y `r` es el resto de esa división, como se muestra en el gráfico G.3.8). Estas dos operaciones son útiles para dividir cuando no se tienen números decimales. Con ellas se pueden definir una cantidad importante de aplicaciones interesantes.

Actividad 12

Realice los siguientes ejercicios. En ellos se busca ejercitar el uso de expresiones numéricas.

Ejercicio 3.2.3. *Escribir una expresión sobre números que represente la suma de cuatro veces diez, más dos. Escribir una expresión sobre números que represente la suma de cuatro veces el número resultante de sumar diez más dos. ¿Qué diferencias se pueden observar en las expresiones? ¿Y en los resultados?*

¡Atención! No se pide el resultado de esta operación, sino una *expresión* que la represente. O sea, no es una respuesta correcta al ejercicio el número 42, si bien ese es el resultado de la expresión a escribir cuando se evalúe.

Ejercicio 3.2.4. *Escribir un procedimiento* `RespuestaVerde` *que ponga tantas bolitas verdes como la suma de cuatro veces diez, más dos, pero sin hacer la cuenta a mano.*
Ayuda: *reutilizar el procedimiento* `PonerN` *del ejercicio 3.1.3, parte 1 y alguna de las expresiones del ejercicio anterior.*

Ejercicio 3.2.5. *Escribir un procedimiento* `PonerDobleRojas` *que ponga el doble de bolitas rojas de lo que dice el parámetro.*
Ayuda: *reutilizar el procedimiento* `PonerN` *del ejercicio 3.1.3, parte 1, y combinarlo con alguna expresión adecuada que involucre al parámetro.*

G.3.9. Resultado de ejecutar `MostrarNumero(34)`

Ejercicio 3.2.6. *Escribir un procedimiento* `MostrarNumero` *que toma un parámetro de tipo número, y asumiendo que el mismo es menor que 100, lo muestra en el tablero en dos celdas contiguas utilizando bolitas negras. Para mostrar el número, la celda de la derecha debe contener tantas bolitas negras como unidades estén representadas por el número, y la de la izquierda debe contener tantas bolitas negras como decenas estén representadas por el número. Por ejemplo, en el gráfico G.3.9 puede observarse el resultado de ejecutar* `MostrarNumero(34)`*.*
Ayuda: *para obtener el número de unidades de un número se puede utilizar la operación de* `mod` *con uno de sus argumentos siendo el* `10`*; para obtener el número de decenas de un número menor a 100 se puede utilizar la operación* `div` *con uno de sus argumentos siendo* `10`*.*

Operaciones sobre colores

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`.

Operaciones sobre direcciones

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`.

Para Reflexionar

Estas operaciones por sí mismas no parecen muy útiles. La utilidad de estas operaciones se observa al utilizarlas en conjunción con comandos avanzados, como la repetición indexada. Reflexionar sobre el hecho de que ciertos elementos que por sí solos resultan poco interesantes, cobran sentido al combinarse con otros de formas no triviales. ¿Puede imaginarse cómo construir una repetición indexada que incluya a todos los colores en el rango que la define?

Las operaciones de mínimo y máximo sobre un tipo dado permiten armar repeticiones indexadas que recorren todos los valores del tipo, sin tener que recordar el orden particular. Por otra parte, dichos programas expresan con mayor adecuación el hecho de que se pretende recorrer *todos* los valores del tipo y no solo los que posea la versión actual del lenguaje. Y siempre es una buena idea que los programas expresen con mayor precisión el propósito del programador.

Actividad de Programación 13

Realice el ejercicio 3.2.7, y vuelva a considerar la utilidad de las operaciones de mínimo y máximo.

Ejercicio 3.2.7. *Realizar nuevamente el ejercicio 2.4.1, reescribiendo el procedimiento* `PonerUnaDeCadaColor`*, que pone una bolita de cada color en la celda actual, pero con las siguientes restricciones:*

- *solo se puede utilizar un único* `Poner` *en el cuerpo del procedimiento;*
- *no puede nombrarse explícitamente ningún color (o sea, no puede utilizar ninguna de las siguientes expresiones:* `Azul`*,* `Negro`*,* `Rojo` *ni* `Verde`*.)*

Ayuda: *piense en utilizar una repetición indexada con expresiones adecuadas en su rango.*

Booleanos y operaciones sobre ellos

El cuarto de los tipos, los booleanos, es un tipo especial, que merece ser considerado detenidamente. Sus valores son usados implícitamente por todos desde chicos, y son los que expresan la idea de verdad. Cualquier persona 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*.

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*.

El tipo de los booleanos trae las operaciones conocidas de lógica, a las cuales denominamos *conectivos lógicos* (que se explican más adelante en esta misma unidad), 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 uso de expresiones booleanas y el significado de los conectivos lógicos se explica y ejercita en la sección 3.2.3, al trabajar con alternativas.

Operaciones que combinan varios tipos

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

- las comparaciones por igualdad:
  - es igual a, ==
  - es diferente de, /=
- las comparaciones de orden:
  - es menor que, <
  - es menor o igual que, <=
  - es mayor que, >
  - es mayor o igual que, >=
- la operaciones de movimiento en el orden:
  - cálculo del siguiente, `siguiente`.
  - cálculo del previo, `previo`.
- operaciones especiales solo 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 una secuencia establecida para cada tipo. El orden de los números es el tradicional, midiendo la cantidad que representa. 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. 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 de 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 la expresión `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 $-n$.

Ejemplos de uso de estas expresiones son la comparación de que un cierto valor numérico pasado como parámetro es menor que un número fijo (e.g. `num < 9`), que dos colores pasados como parámetros son iguales (e.g. `color1 ==`

G.3.10. Resultado de ejecutar `ArribaAbajo(Norte)`

`color2`) y que la dirección siguiente de una dirección pasada como parámetro no es la última posible (e.g. `siguiente(dir) /= maxDir()`).

Actividad 14

Realice los ejercicios que se enuncian a continuación. Los mismos ejemplifican el uso de operaciones que combinan varios tipos.

Ejercicio 3.2.8. *Escribir una expresión que determine si dos parámetros de tipo dirección, llamados por ejemplo* `direccionOriginal` *y* `direccionNueva`*, son diferentes.*

Ejercicio 3.2.9. *Escribir una expresión que determine si un número está entre 0 y 100. Para ello, combinar dos expresiones relacionales mediante una conjunción booleana.*

Ejercicio 3.2.10. *Escribir un procedimiento* `ArribaAbajo` *que tome un parámetro de tipo Dirección, y ponga una bolita roja en la celda contigua a la inicial en la dirección indicada, y una bolita verde en la celda contigua a la inicial, pero en la dirección opuesta a la indicada. Por ejemplo, el resultado de llamar al procedimiento* `ArribaAbajo(Norte)` *debería ser como el mostrado en el gráfico G.3.10. Observar que la celda actual al terminar debe ser la misma que la celda inicial.*
Ayuda: *¿Cómo expresar el opuesto de una dirección dada por un parámetro? Pensar en las operaciones recién presentadas.*
Sugerencia: *Definir un procedimiento auxiliar con la parametrización adecuada para poner una bolita de algún color en alguna celda contigua. Este procedimiento debería invocarse dos veces con diferentes argumentos.*

Operaciones sobre el tablero

Finalmente, 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, cuántas hay, y si puede o no moverse de manera segura sin caerse del borde del tablero. Todas estas formas se expresan a través de expresiones. Las expresiones que tienen que ver con la operatoria del cabezal, y sirven para determinar ciertos valores asociados a la celda actual 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.

Por ejemplo, `hayBolitas(Rojo)` puede ser `True` o `False`, dependiendo del estado de la celda actual: si en la celda hay alguna bolita de color `Rojo`, entonces esta expresión valdrá `True`; si por el contrario, no hay ninguna, esta expresión valdrá `False`. Si `hayBolitas(Rojo)` vale `False`, entonces `nroBolitas(Rojo)` valdrá `0`, y si `nroBolitas(Rojo)` es distinto de `0`, entonces `hayBolitas(Rojo)` valdrá `True`. De manera similar a `hayBolitas`, `puedeMover(Sur)` valdrá `True` si el cabezal no se encuentra en la fila de más abajo, y `False` si se encuentra en dicha fila, y de manera análoga para las restantes direcciones. Ejercitaremos el uso de estas expresiones luego de ver una herramienta nueva para armar comandos complejos.

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

Actividad 15

Realice el ejercicio 3.2.11. (Todavía no tenemos suficientes herramientas para poder aprovechar este ejercicio. Su utilidad se verá en el siguiente apartado.)

Ejercicio 3.2.11. *Utilizando algunas de las expresiones recién presentadas, escribir una expresión que determine si la cantidad de bolitas en la celda actual es menor o igual que un número determinado (e.g.* `9`*). Recordar que una expresión para comparar por menor o igual se escribe, por ejemplo,* `16 <= 20`*, y que hay una expresión para comprobar el número de bolitas de una celda.*

Actividad de Programación 16

Realice los ejercicios 3.2.12 y 3.2.13 y utilícelos para quitar todas las bolitas rojas y negras de la celda actual, y la totalidad de las bolitas de la celda lindante al Norte.

Ejercicio 3.2.12. *Escribir un procedimiento* `SacarTodasLasDeColor` *que, dado un parámetro* `color`*, elimina todas las bolitas del color indicado de la celda actual. Para realizarlo, considere utilizar el procedimiento* `SacarN` *del ejercicio 3.1.3, parte 3, y combinarlo con alguna de las expresiones recién presentadas.*

Ejercicio 3.2.13. *Escribir un procedimiento* `VaciarCelda` *que elimine todas las bolitas de la celda actual. Para realizarlo, considere utilizar el procedimiento anterior, y una repetición indexada que recorra todos los colores.*

Leer con Atención

Los procedimientos `SacarTodasLasDeColor` y `VaciarCelda` son excelentes candidatos para ser colocados en la `Biblioteca`.

Actividad de Programación 17

Realice los ejercicios 3.2.14 y 3.2.15 y colóquelos en su `Biblioteca`.

Ejercicio 3.2.14. *Definir una función* `esCeldaVacia` *que retorne un booleano indicando si la celda actual no contiene bolitas de ningún color.*

Ayuda: *considere utilizar la combinación de 4 expresiones básicas sobre el tablero con las operaciones booleanas adecuadas.*

Ejercicio 3.2.15. *Definir una función* `esCeldaVaciaAl` *que dada una dirección, retorne un booleano indicando si la celda en esa dirección no contiene bolitas de ningún color. La precondición de esta función es que existe una celda en la dirección dada.*

Sugerencia: *reutilice adecuadamente la función del ejercicio anterior.*

3.2.3. Alternativas condicionales

Los booleanos pueden utilizarse para distinguir entre diferentes alternativas al momento de describir comandos para el cabezal. Para ello es necesario un comando que, con base en un booleano, decida entre otros dos comandos para saber cuál de ellos debe ejecutarse. Imaginemos que representamos una flor mediante una bolita roja, y fertilizante mediante una bolita negra, y que, si en la celda actual ya hay una flor, entonces le coloquemos fertilizante, y si no hay, entonces plantemos una nueva flor. Ninguna de las herramientas vistas hasta el momento permite describir esta situación. Entonces presentamos un nuevo comando, llamado `if-then-else`:

```
if (hayBolitas(Rojo)) -- Verdadero si hay flor en la celda actual
  { Poner(Negro) }    -- Solo agrega fertilizante
else
  { Poner(Rojo) }     -- Pone una flor, pues no hay ninguna
```

Para Reflexionar

¿Puede mejorar el programa anterior a través del uso de procedimientos simples que representen mejor el dominio del problema? (Por ejemplo, con el procedimiento `PonerFlor()`, etc.? ¿Qué sucederá con la condición? ¡No podemos utilizar un procedimiento allí para darle sentido a la misma! Es claro que hace falta una forma nueva de nombrar expresiones, la cual se verá en el próximo apartado.

En inglés, `if-then-else` significa *si-entonces-sino*, y es la idea de estructura alternativa con base en una condición. Por ejemplo, en este caso, el comando se leería "*si hay una bolita roja (flor) en la celda actual, entonces poner una bolita negra (fertilizante), y si no hay flores, poner una bolita roja (nueva flor)*".

La forma general del comando `if-then-else` está dada por la siguiente definición:

Definición 3.2.2. *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.

Actividad de Programación 18

Realice el ejercicio 3.2.16. Considere la cantidad de trabajo necesario, y reflexione sobre la importancia de reutilizar código definido con anterioridad (según se sugiere en dicho ejercicio).

Un dígito es un número entre 0 y 9.

Ejercicio 3.2.16. *Supongamos que una celda representa un dígito de un* indicador de kilometraje*, para lo cual utiliza bolitas negras. Escribir un procedimiento* `IncrementarDigCuentaKm()`*, que incremente en uno el kilometraje indicado por ese dígito (y solo ese). Tener en cuenta que el indicador, al llegar al límite de 9 debe volver a 0, pues es un dígito. Considerar el uso de una alternativa para detectar las dos situaciones posibles (¿cuáles son?), y la reutilización de la condición del ejercicio 3.2.11 y del procedimiento del ejercicio 3.2.12.*

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 solo 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` es una expresión que vale `True` siempre que haya bolitas rojas; o sea, la condición del `if-then` garantiza la precondición de la rama del `then`. De esta manera, la operación dada por la alternativa condicional es total incluso aunque el cuerpo de la misma es parcial, pues el comando `Sacar` solamente se invocará cuando haya una bolita roja en la celda actual; en caso de que no haya ninguna bolita roja, este comando compuesto no tendrá ningún efecto.

Ejercicio 3.2.17. *Escribir un procedimiento* `QuitarHastaTresFlores` *que, suponiendo que en la celda actual hay entre cero y tres flores representadas cada una por una bolita roja (pero no se sabe cuántas exactamente), las elimine de la celda. El procedimiento debe describir una operación total. Utilizar nombres adecuados para la subtarea de quitar una flor (pero no puede hacerse aún con la condición).*
Ayuda: *utilizar la variante de alternativa dada por el comando* `if-then`*.*

Volveremos sobre la alternativa condicional para profundizarla en la sección siguiente y para ver varios casos de usos inadecuados en el capítulo siguiente.

3.2.4. Funciones simples

Imaginemos ahora que un grupo de bolitas de varios colores representa a algún elemento de un juego, y que queremos representar su sombra en la celda vecina

al Este. Vamos a representar la sombra por una cantidad de bolitas negras igual a la cantidad total de bolitas del elemento. Queremos, entonces, escribir un procedimiento `HacerSombraBasica` que dibuja una sombra en la celda lindante al Este de la actual. Supondremos que contamos con un procedimiento `DibujarSombra` que, dado un parámetro numérico `valorIntensidad`, coloca esa cantidad de bolitas negras en la celda al Este, volviendo a la celda original antes de terminar. ¿Cómo sería el procedimiento pedido? Claramente, precisamos una expresión que calcule el número total de bolitas en la celda actual, y lo pase como argumento al procedimiento `DibujarSombra`.

```
procedure HacerSombraBasica()
 {- PROPÓSITO:
     * pone una sombra del elemento representado en la celda
       actual, en la celda lindante al Este
    PRECONDICIÓN:
     * que haya una celda al Este
 -}
 {
   DibujarSombra(nroBolitas(Azul) + nroBolitas(Negro)
                 + nroBolitas(Rojo) + nroBolitas(Verde))
 }
```

Sin embargo, la expresión que utilizamos, que suma los 4 usos de la función primitiva `nroBolitas`, es demasiado larga; podría decirse que es "operacional" en el sentido que no abstrae el concepto de contar el total de bolitas, sino que lo descompone en operaciones más elementales que deben combinarse.

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, abstrayendo su comportamiento. Tal mecanismo es conocido con el nombre de *función*. Una función simple es básicamente otra forma de nombrar a una expresión compuesta.

Definición 3.2.3. *Una* función simple *es una forma de asignar un nombre a una expresión. La forma de definir una función simple en* GOBSTONES *es*

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

siendo `<funName>` *un identificador que comienza con minúscula 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.

Leer con Atención

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, o combinado en expresiones complejas, etc., pero nunca secuenciada o como parte aislada en un bloque.

Definición 3.2.4. *Una función simple puede ser usada como una expresión. La forma de invocarla es escribir su nombre seguida por paréntesis, de la siguiente manera*

`<funcName>()`

donde < *funcName* > *es el nombre de alguna función declarada en el programa.*

En el ejemplo visto antes, podríamos definir una función simple para calcular el número total de bolitas de la celda actual de la siguiente manera.

```
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.

El procedimiento para hacer la sombra podría modificarse para aprovechar esta función de la siguiente manera

```
procedure HacerSombra()
 {- PROPÓSITO:
     * ilustrar el uso de funciones
     * pone una sombra del elemento representado en la celda
       actual, en la celda lindante al Este
    PRECONDICIÓN:
     * que haya una celda al Este
    OBSERVACIÓN:
     * esta es una manera adecuada de abstraer este problema
 -}
 {
   DibujarSombra(nroBolitasTotal())
   -- usa una función para calcular el número de bolitas de
   --  la celda actual y llama a DibujarSombra tal número
   --  como argumento
 }
```

Ejercicio 3.2.18. *Escribir el procedimiento* `DibujarSombra` *que recibe un parámetro* `valorIntensidad` *y coloca una sombra de la intensidad dada por el parámetro en la celda lindante al Este, retornando luego a la celda original.*

En este caso, llevar más cerca de las ideas del programador a través de nombrar adecuadamente los elementos.

Las funciones simples permiten abstraer la implementación efectiva de ciertas ideas expresadas mediante expresiones compuestas. Quizás las expresiones que resulten más útil abstraer de esta manera sean las determinadas por condiciones complejas expresadas mediante la combinación de varias expresiones booleanas simples mediante conectivos lógicos.

Condiciones booleanas más complejas

A partir de los valores de verdad básicos pueden construirse *condiciones* 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 3.2.5. *Un* conectivo lógico *es una operación entre booleanos. Los conectivos más comunes son la negación, la conjunción y la disyunción.*

Leer con Atención

La negación es un conectivo lógico que toma un booleano y devuelve otro booleano. Transforma `True` en `False` y viceversa. La manera de escribir la negación de una expresión booleana es con la operación `not`. En resumen, `not True` es igual a `False`, y `not False` es igual a `True`.

Definición 3.2.6. *La* negación *es un conectivo lógico unario. Su forma general es*

`not` < *expBooleana* >

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

Con esta operación, por ejemplo, podría escribirse una condición que verifique si en la celda actual no hay bolitas rojas como: `not hayBolitas(Rojo)`. Esta expresión valdrá `True` cuando el número de bolitas rojas en la celda actual sea `0`, y `False` en caso contrario, extactamente lo opuesto a `hayBolitas(Rojo)`.

Ejercicio 3.2.19. *Escribir una función* `noHayMarcianos()` *que retorne verdadero si en la celda actual no hay ningún marciano, y falso en caso de que haya alguno. Un marciano se representa mediante una bolita de color* `Azul`*.*

Ejercicio 3.2.20. *Escribir un ejercicio* `celdaVacia()` *que retorna verdadero si en la celda actual no hay ninguna bolita de ningún color, y falso en caso de que haya alguna. Para realizarlo, reutilizar la función* `nroBolitasTotal`*, y combinarla con un operador de igualdad y el conectivo lógico de negación.*

Leer con Atención

Otro 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 3.2.7. *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`. De esta manera se pueden construir condiciones complejas, como preguntar si en la celda actual hay bolitas rojas y verdes al mismo tiempo: `hayBolitas(Rojo) && hayBolitas(Verde)`.

Ejercicio 3.2.21. *Escribir una función* `hayFlores()` *que retorne verdadero si en la celda actual hay alguna flor, y falso en caso de que no haya ninguna. Una flor se representa mediante una bolita de color* `Rojo` *y una de color* `Verde`*. La condición de que haya flores puede verificarse, por tanto, verificando que haya bolitas de ambos colores en la celda. ¡Observar que no se pide verificar que la cantidad de bolitas verdes y rojas coincida, sino solo que haya de ambas! Y también que no molesta que haya bolitas de otros colores.*

Ejercicio 3.2.22. *Escribir una función* `floresSinMarcianos()` *que retorne verdadero si en la celda actual hay alguna flor, pero no hay ningún marciano. Considere reutilizar las funciones de los ejercicios anteriores, combinadas mediante algún conectivo adecuado.*

Ejercicio 3.2.23. *Escribir una función* `hayExactamenteUnaFlor()` *que retorna verdadero si en la celda actual hay exactamente una flor (representada igual que en el ejercicio 3.2.21). Tener en cuenta que no puede haber más de una bolita roja y una verde en la celda. ¿Cómo se puede expresar esta condición utilizando conjunciones y operaciones relacionales?*

Leer con Atención

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 solo si ambos son falsos. Para denotarla se utiliza la operación infija `||`.

Definición 3.2.8. *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`.

Las expresiones `True || True`, `True || False` y `False || True` tendrán como resultado `True`, y la expresión `False || False` tendrá como resultado `False`. Con la disyunción se puede pregunta si en la celda actual hay bolitas azules o hay bolitas negras (puede faltar cualquiera de las dos): `hayBolitas(Azul) || hayBolitas(Negro)`.

Actividad 19

¿Cómo construir una condición que verifique si hay alguna bolita en la celda actual? Considere el ejercicio 3.2.24.

Ejercicio 3.2.24. *Escribir una función* `hayBolitas` *que retorne verdadero si hay alguna bolita en la celda actual (no importa su color). Considerar la utilización de una combinación más complicada de* `hayBolitas` *y* `||`*, y que involucre a los 4 colores. ¿Puede expresar la condición de alguna otra forma?*

Ejercicio 3.2.25. *Escribir un procedimiento* `PintarCeldaMarron` *que pinte una celda de color marrón (agregando bolitas de los cuatro colores en la celda actual) solo si no hay bolitas de ningún color. Considerar la utilización de la función del ejercicio anterior y una alternativa condicional.*

Para Reflexionar

Esta forma de expresiones complicadas puede ser extremadamente engorrosa si tuviéramos que usarlas directamente. ¿Entiende la importancia de la habilidad de nombrar sus ideas? Este proceso, al que denominamos *abstracción* es súmamente importante en toda actividad de programación.

Leer con Atención

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 3.2.9. *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 hayPasto() && not hayMarciano() || haySangre()
```

significa lo mismo que

```
((not hayPasto()) && (not hayMaciano()))
   || haySangre()
```

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

```
not (hayPasto()
     && (not (hayMarciano() || haySangre()))
    )
```

Considerar que las funciones `hayPasto()`, `hayMarciano()` y `haySangre()` verifican la existencia de cada uno de estos elementos, representados respectivamente mediante bolitas verdes, azules y rojas.

Ejercicio 3.2.26. *Escribir* `hayPasto()`, `hayMarciano()` *y* `haySangre()`.

Actividad 20

¿Cuál será el valor de verdad de las distintas condiciones compuestas dadas antes? Para determinarlo, escriba una tabla que consigne las diferentes posibilidades de una celda para contener bolitas (por ejemplo, si contiene bolitas verdes pero no azules ni rojas, si contiene bolitas verdes y azules pero no rojas, etc., e indicar en cada caso el valor booleano de las expresiones.) ¿Qué diferencia se observa entre ambas condiciones (con diferentes precedencias)?

Ejercicio 3.2.27. *Escribir un procedimiento* `PlantarYFertilizar` *que, reutilizando la idea del ejemplo al comienzo del subsección 3.2.3, plante una flor si no hay ninguna, y fertilice las flores que haya (ya sea que fueron recién colocadas o ya existían). Nombre adecuadamente sus subtareas utilizando procedimientos y funciones auxiliares de manera conveniente.*

Ejercicio 3.2.28. *Revisar el ejercicio 3.2.16 para abstraer de manera adecuada la condición utilizada, dándole un nombre representativo a la misma mediante una función (por ejemplo,* `seVaAExceder()`*).*

3.3. Funciones avanzadas

En el apartado anterior aprendimos cómo escribir y utilizar funciones simples (definición 2.2.21), y trabajamos con ellas, aprendiendo cómo valernos adecuadamente para abstraer nuestras ideas. En este apartado vamos a extender nuestra concepción sobre las funciones, exhibiendo su uso junto con otros conceptos avanzados de programación.

3.3.1. Funciones con parámetros

Así como los procedimientos simples pueden generalizarse a esquemas de procedimientos mediante el uso de parámetros, las funciones simples pueden generalizarse a esquemas de funciones mediante el mismo recurso. La idea es la misma: un parámetro indica el nombre de un valor que cambia en cada uso de la función (como vimos, similar a un "agujero" que debe completarse al invocar su uso en el código). La sintaxis utilizada es similar a la utilizada para los procedimientos: el nombre de los parámetros de la función se coloca entre paréntesis después del nombre de la misma (indicando que esta función tiene dichos "agujeros").

Definición 3.3.1. *Una* función con parámetros *es similar a una función simple, pero agregando la noción de parámetros. La forma de definir una función con parámetros 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.*

Las funciones con parámetros se pueden invocar de manera similar a la vista para procedimientos con parámetros, mediante el uso de argumentos, con la diferencia, ya mencionada, de que por ser expresiones solo se pueden utilizar en lugares donde se espera una expresión.

Definición 3.3.2. *Una función con parámetros 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.*

Las nociones de concordancia de tipos entre cada argumento y los usos que se les da a los parámetros correspondientes es similar a la vista para los procedimientos.

Por ejemplo, podemos ver si hay más bolitas de un color que de otro, pero usando parámetros para indicar qué colores queremos.

```
function hayMasQue(color1, color2)
  {-
   PROPÓSITO:
     * retorna verdadero si hay más bolitas del color1
       que bolitas del color2 en la celda actual
   PRECONDICIÓN:
     * Ninguna (es una función total)
  -}
  {
   return (nroBolitas(color1) > nroBolitas(color2))
  }
```

Con esta función podría verificarse que hubiera más bolitas rojas que verdes y más negras que azules mediante la expresión `hayMasQue(Rojo,Verde) && hayMasQue(Azul,Negro)`.

Actividad de Programación 21

Realice el ejercicio 3.3.1. ¿Codificó un procedimiento auxiliar para abstrear la forma de huir? ¿O utilizó simplemente un comando básico?

Ejercicio 3.3.1. *Escribir un procedimiento* `HuirSiPintaMal` *que huya de la celda actual moviéndose al* `Norte` *si en la misma hay más enemigos que aliados. Los aliados se representan mediante bolitas verdes y los enemigos mediante bolitas rojas. Considerar la reutilización de la función* `hayMasQue`*, pero utilizando funciones adecuadas para abstraer la representación de aliados y enemigos (por ejemplo, con las funciones* `colorAliado()` *y* `colorEnemigo()`*).*

Actividad de Programación 22

Realice el ejercicio 3.3.2. Utilícelo en una función `esValorDigito` que dado un número determine si es un dígito o no, retornando un booleano.

Ejercicio 3.3.2. *Escribir una función* `enIntervalo` *que dados un número* `n` *y dos números* `inf` *y* `sup`*, retorne verdadero si* `n` *está dentro del intervalo* [`inf`,`sup`] *(o sea, es mayor o igual a* `inf` *y menor o igual a* `sup`*).*

Las funciones parametrizadas proveen una herramienta más poderosa que las funciones simples. Sin embargo, el verdadero poder de las funciones radica en poder realizar alguna forma de procesamiento antes de retornar su valor, permitiendo, por ejemplo, visitar otras zonas del tablero.

3.3.2. Funciones con procesamiento

Las funciones simples y parametrizadas son una forma importante de abstraer ideas que se expresan mediante expresiones complejas. Sin embargo, para poder devolver valores que dependan de otras celdas que no sea la celda actual, o expresar valores que dependan de procesar información dispersa en el tablero, es preciso contar con una forma más poderosa de funciones: las *funciones con procesamiento*. Este procesamiento se consigue agregando a las funciones, previo a expresar su valor de retorno, un grupo de comandos que realicen este procesamiento.

Definición 3.3.3. *Una* función con procesamiento *es una forma de función que además de nombrar a una expresión realiza previamente algún procesamiento. La forma de definir una función con procesamiento en* GOBSTONES *es*

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

siendo <*funName*>, <*params*> *y* <*expresion*> *como antes, y* <*comando*> *un comando (que puede ser un comando compuesto como una secuencia de comandos).*

Por ejemplo, si queremos expresar la cantidad de enemigos (representados como antes por una bolita roja cada uno) en una celda lindante a la actual, podemos utilizar la siguiente función

```
function nroEnemigosAl(dir)
 {-
    PROPÓSITO:
     * retorna la cantidad de enemigos en la celda lindante a la
       actual en la dirección dir
    PRECONDICIÓN:
     * hay una celda lindante en dirección dir
 -}
 {
   Mover(dir)
   return(nroBolitas(colorEnemigo()))
 }
```

La invocación de una función con procesamiento es exactamente la misma que la de una función con parámetros. La única diferencia es que, en GOBSTONES, el procesamiento que la misma realiza no se lleva a cabo de manera real sobre el tablero, sino solo de una forma simulada. Por eso, al retornar de una función con procesamiento, el estado del tablero no varió absolutamente en nada. Esto permite utilizar las funciones con procesamiento para retornar información, pero sin tener que preocuparse por las consecuencias del procesamiento.

Leer con Atención

Esta característica de GOBSTONES es única. Las funciones son *expresiones* y como tales, no deben denotar acciones (o sea, producir efectos) sino denotar valores. Por eso, el procesamiento se realiza de manera simulada. Sin embargo, en casi todos los demás lenguajes, se permite que las funciones además de denotar un valor, produzcan efectos. Técnicamente deberíamos decir que no son funciones, sino procedimientos que retornan valores; ¡pero es tan común utilizar el nombre de *función* para este tipo de procedimientos que es importante estar atentos a las diferencias, ya que la forma de pensar las "funciones" cambia!

Por ejemplo, imaginemos una situación donde queremos agregar en la celda actual un número de aliados igual a la cantidad de enemigos en la celda al `Norte`, más uno (por ejemplo, como parte de un programa que expresa nuestra estrategia de defensa en un juego), suponiendo la misma codificación que vimos antes (un enemigo se representa con una bolita roja y un aliado con una bolita verde). El procedimiento podría ser:

```
procedure EstrategiaDeDefensa()
 {-
    PROPÓSITO:
     * agregar tantos aliados en la celda actual como
       enemigos hay en la celda lindante al Norte,
       más uno (si hay enemigos)
    PRECONDICIÓN:
     * hay una celda lindante al Norte
  -}
 {
   if (hayEnemigosAl(Norte))
     { AgregarAliados(nroEnemigosAl(Norte)+1) }
 }
```

Observar que si bien la función `nroEnemigosAl` realiza un movimiento para retornar la cantidad de enemigos en la dirección dada, este movimiento no es reflejado por el procedimiento `EstrategiaDeDefensa`, que solo agrega los aliados en la celda actual.

Actividad de Programación 23

Realice los ejercicios 3.3.3 y 3.3.4, con el fin de completar el procedimiento anterior. Pruebe todos estos elementos en un programa sencillo y verifique que efectivamente las funciones no producen ningún efecto.

Ejercicio 3.3.3. *Escribir la función* `hayEnemigosAl` *que determina si hay enemigos en la celda lindante en la dirección dada. ¿Cuál es la precondición de la función?*

Ejercicio 3.3.4. *Escribir el procedimiento* `AgregarAliados` *que agrega tantos aliados como el número indicado como parámetro.*

Un procesamiento más complicado puede lograrse mediante el uso de comandos más complejos, combinados con la idea de usar bolitas de algún color para indicar resultados de éxito.

Por ejemplo, la siguiente función verifica si hay enemigos en alguna de las 3 celdas lindantes al `Norte` (supone el uso de una función `hayEnemigosAlEnRango`, similar a la función `hayEnemigosAl` vista, pero agregando un parámetro numérico para indicar el rango exacto donde deben buscarse enemigos).

```
function hayEnemigosCercaAlNorte()
 {-
    PROPÓSITO:
     * retorna verdadero si hay algún enemigo en las
       próximas 3 celdas al Norte de la actual
    PRECONDICIÓN:
     * hay 3 celdas lindantes al Norte
 -}
 {
   SacarTodas(Azul) -- Elimina todas las bolitas azules,
                    -- para estar seguro que no hay ninguna
   foreach i in [1..3]
    { if(hayEnemigosAlEnRango(Norte, i))
       { Poner(Azul) } -- Agrega una azul si hay enemigos
    }
   return(hayBolitas(Azul)) -- Si hay azules, es porque
                            -- uno de los rangos dio verdadero
 }
```

Observar que se utilizan bolitas azules para indicar el éxito de la búsqueda. Puesto que el procesamiento realizado por la función es solo simulado y no real, esta modificación de las bolitas azules no será reflejada en el procedimiento que utilice a `hayEnemigosCercaAlNorte`. En este ejemplo particular, la función podría haberse escrito simplemente como

```
  function hayEnemigosCercaAl()
    {-
       PROPÓSITO:
        * retorna verdadero si hay algún enemigo en las
          próximas 3 celdas al Norte de la actual
       PRECONDICIÓN:
     * hay 3 celdas lindantes al Norte
    -}
   {
    return(hayEnemigosAlEnRango(Norte,1) ||
           hayEnemigosAlEnRango(Norte,2) ||
           hayEnemigosAlEnRango(Norte,3)
   }
```

Sin embargo, esta forma no sería parametrizable en cuanto a la distancia a medir, quitando poder de generalidad a la solución propuesta y haciéndolo menos modificable a futuro. Por ejemplo, si modificásemos la definición de "cerca" para contemplar 10 celdas en lugar de 3, el resultado habría sido más engorroso. O si hubiéramos considerado agregar un parámetro numérico que estableciese el rango en el que nuestros exploradores pueden detectar enemigos, no habría sido posible expresar la función como una disyunción compleja, puesto que no sabríamos de antemano cuántos llamados debemos realizar.

Actividad de Programación 24

Realice los ejercicios 3.3.5 y 3.3.6 y pruébelos en algún programa.

Ejercicio 3.3.5. *Escribir una función* `hayEnemigosAlEnRango` *que, dado un parámetro* `d` *de tipo dirección y uno* `n` *de tipo número indique si hay enemigos en la celda distante exactamente* `n` *celdas en dirección* `d` *a partir de la celda actual. Utilizar la idea de la función* `hayEnemigosAl` *del ejercicio 3.3.3 pero utilizando el procedimiento* `MoverN` *en lugar del comando* `Mover`*.*

Ejercicio 3.3.6. *Escribir una función* `hayEnemigosCercaNAlNorte` *que, dado un parámetro* `n` *de tipo número indique si hay enemigos en alguna de las* `n` *celdas contiguas al* `Norte`*. Utilizar la idea de procesar con una repetición indexada y bolitas para indicar el resultado.*

El procesamiento que es posible realizar en las funciones con procesamiento se verá potenciado al incorporar formas de recordar valores. Esto se realizará en el siguiente capítulo.

3.4. Ejercitación

En esta sección se enuncian una serie de ejercicios adicionales a los ya dados. Al igual que en ejercitaciones previas, para su correcta resolución son necesarios todos los elementos aprendidos en las secciones y capítulos anteriores.

Actividad de Programación 25

Realice los ejercicios enunciados en esta sección. Nuevamente, le recordamos que debe utilizar todas las buenas prácticas que venimos estudiando.

Ejercicio 3.4.1. *Escribir una función* `nroBolitasNAl`*, que dados un color* `c`*, un número* `n` *y una dirección* `d`*, determine el número de bolitas de color* `c` *en la celda distante exactamente* `n` *lugares en dirección* `d` *a partir de la celda actual. Utilizar como inspiración la función* `hayEnemigosAlEnRango` *del ejercicio 3.3.5.*

El siguiente ejercicio se basa en una idea de Pablo Tobia. ¡Gracias Pablo!

Ejercicio 3.4.2. *En este ejercicio utilizaremos el display de un ecualizador representado en el tablero. El ecualizador tiene 2 canales (izquierdo y derecho) con 3 frecuencias cada uno (agudos, medios y graves). Cada frecuencia de cada canal posee 4 leds verdes y 2 leds rojos. Para representar el display en el tablero se utiliza una columna por cada frecuencia de cada canal (6 en total), a partir de la columna más al* `Oeste`*. Cada frecuencia se representa desde la celda base (la más al* `Sur`*), dejando dicha celda libre, y ubicando un led por celda hacia arriba.*

(a). Tablero inicial representando el display de un ecualizador.

(b). Tablero final representando el ecualizador con intensidades.

G.3.11. Tablero representando el display de un ecualizador (antes y después de calcular las intensidades)

Los leds encendidos se representan mediante una bolita del color correspondiente en la celda, y los apagados con una celda vacía. Se muestra un tablero representando al display de un ecualizador en el gráfico G.3.11a.

Escribir un procedimiento `CalcularIntensidades`*, que dado un tablero conteniendo la representación de un display de ecualizador cuente la intensidad de cada frecuencia, registrando el resultado en la celda de la base de cada columna con bolitas azules si la intensidad es menor o igual que 4, y negras si es mayor. El resultado para el ecualizador dado como ejemplo se muestra en el gráfico G.3.11b.*

Para realizar el ejercicio anterior, se recomienda dividir en subtareas y utilizar funciones y procedimientos según lo visto hasta el momento. Puede resultar útil contar con algunos de los nombres de los procedimientos y funciones en cuestión: `TotalizarFrecuenciaActual`, `ProcesarLedsVerdes`, `ProcesarLedsRojos`, `NormalizarColorIntensidad` y `nroBolitasNAl`. Tenga en cuenta que para totalizar todas las frecuencias puede usar una repetición indexada, y para totalizar la frecuencia actual puede utilizar la idea de `hayEnemigosCercaAlNorte`.

Los siguientes ejercicios se basan en una idea de Pablo Barenbaum. Utilizan su idea y su enunciado para un juego inspirado en el ajedrez, que él denominó *Procedrez*. Primero se describirá el juego, y luego se enunciarán los ejercicios que ayudarán a ir resolviendo distintos aspectos del juego de manera incremental. ¡Gracias Pablo!

El *Procedrez* es un juego en el que intervienen dos jugadores, que mueven piezas en un tablero de tamaño variable, dividido en escaques o casillas. Las piezas pueden ser de tres clases diferentes: visir, torre y rey. Un jugador puede tener varios visires y varias torres, pero solo un rey. Las piezas se mueven de la siguiente manera, de manera similar a los movimientos que se pueden encontrar en el ajedrez:

- Los visires y el rey se mueven una casilla en cualquier dirección, ortogonal o diagonal.
- Las torres se mueven en línea recta cualquier número de casillas, pero solo en dirección ortogonal. Las torres no pueden “saltar” las propias piezas ni las piezas enemigas.

Además, si una pieza se mueve sobre una casilla ocupada por una pieza del oponente, captura a dicha pieza, que se retira permanentemente del tablero, exactamente como en ajedrez.

También como en el ajedrez, se dice que el rey de un jugador está en *jaque* si puede ser capturado por una pieza del oponente. El objetivo del juego es dar *jaque mate*, que consiste en dar jaque al rey del oponente, de tal manera que le resulte imposible evitar el jaque.

El tablero del juego se representará con el tablero de GOBSTONES. Las piezas del primer jugador se representarán con bolitas de color `Rojo` y las del segundo con `Negro`; el número de bolitas dependerá de la clase de pieza en cuestión (1 = visir, 2 = torre, 3 = rey). Las bolitas de color `Verde` y `Azul` se reservarán para marcar posiciones del tablero.

Ejercicio 3.4.3. *Implementar las siguientes funciones auxiliares que se usarán en el resto de los ejercicios del* Procedrez*.*

- `visir()` *que devuelve 1.*
- `torre()` *que devuelve 2.*
- `rey()` *que devuelve 3.*
- `hayPieza()` *que devuelve* `True` *si hay una pieza en la casilla actual.*
- `clasePieza()` *que devuelve la clase de la pieza en la casilla actual, asumiendo que hay una.*
- `colorPieza()` *que devuelve el color de la pieza en la casilla actual, asumiendo que hay una.*
- `marcadaComoDestino()` *que devuelve* `True` *si la casilla actual está marcada como destino (con una bolita* `Azul`*).*
- `marcadaComoMovimiento()` *que devuelve* `True` *si la casilla actual está marcada como movimiento (con alguna bolita* `Verde`*).*

Este ejercicio simplemente prepara algunas funciones que resultarán útiles en el resto de los ejercicios. La idea de proponerlos es guiar en la forma de pensar cómo abstraer la representación de piezas de manera de poder pensar en términos del problema y no de la representación.

Ejercicio 3.4.4. *Implementar el procedimiento* `MarcarMovimientosVisir` *que recibe un color* `col` *y marca con una bolita de ese color cada uno de los potenciales movimientos del visir o el rey ubicado en la casilla actual. La precondición del procedimiento es que debe haber un visir o un rey en la casilla actual.*

Deben marcarse las casillas ocupadas por piezas del oponente, ya que estos son movimientos válidos (capturas). No deben marcarse las casillas que estén ocupadas por piezas del mismo color, ya que dos piezas no pueden ocupar la misma casilla simultáneamente.

El cabezal debe quedar ubicado en la casilla en la que estaba originalmente.

Algunos ejemplos de posibilidades de este ejercicio se muestran en los gráficos G.3.12 y G.3.13.

Ejercicio 3.4.5. *Suponiendo que ya están definidos* `MarcarMovimientosVisir` *y* `MarcarMovimientosTorre`*, procedimientos que marquen los movimientos de las piezas de manera similar a lo realizado en el ejercicio anterior, implementar el procedimiento* `MarcarMovimientosPieza` *que recibe un color* `col` *y marca con una bolita de ese color cada uno de los potenciales movimientos de la pieza ubicada en la casilla actual (cualquiera sea esta pieza).*

La precondición es que debe haber una pieza en la casilla actual.

Continuaremos con operaciones del *Procedrez* en el próximo capítulo, luego de aprender algunas herramientas avanzadas nuevas.

G.3.12. Los visires y el rey pueden moverse libremente

G.3.13. El rey negro puede capturar a la torre roja pero no puede pasar sobre su propia torre

4

Alternativa, repetición y memoria

En este capítulo vamos a profundizar las herramientas de alternativa y repetición que vimos en el capítulo anterior, mostrando otras formas posibles, y discutiendo las consecuencias de poseerlas. Además presentaremos la noción de *memoria*, como una forma de recordar valores a lo largo de la ejecución de un programa.

Si bien la memoria no es imprescindible para solucionar problemas (con parametrización se pueden lograr los mismos efectos), es una herramienta que permite expresar muchos problemas de forma más concisa, y es ampliamente utilizada en la mayoría de los lenguajes de programación. Sin embargo, su correcta utilización conlleva un número importante de dificultades que la hacen extremadamente compleja de manejar. Puesto que este libro se propone brindar un panorama sobre la programación de manera introductoria, no abundaremos ni en los problemas que puede ocasionar su uso, ni en la multitud de soluciones propuestas y sus consecuencias. Se discute un poco más sobre esta cuestión en el capítulo 6.

Además de las herramientas básicas (como alternativa, repetición o memoria), presentaremos una herramienta abstracta para organizar programas que utilizan repeticiones: la noción de *recorrido* de una secuencia. Cerramos el capítulo con ejercicios completos que utilizan todas las herramientas vistas.

4.1. Más sobre alternativas

Como dijimos, una *alternativa* es una forma de decidir entre otros dos elementos para saber cuál de ellos debe ejecutarse, o dicho de otra forma, una manera de describir algo que puede ser diferente dependiendo de ciertas condiciones. También vimos la forma más común de alternativas, el comando de *alternativa condicional*, comúnmente conocida en la jerga de programación como *condicional* o simplemente *sentencia* `if-then-else`, que es la manera de decidir entre dos comandos para describir una acción que puede ser diferente según la condición indicada por una condición.

Puede haber alternativa también para expresiones, aunque en GOBSTONES no aparece, por simplicidad.

En esta sección vamos a profundizar sobre algunas combinaciones de alternativas condicionales, y veremos una nueva forma de alternativa, la *alternativa indexada*, también conocida como *sentencia* `case` o *sentencia* `switch`.

Sentencia es sinónimo de *comando* en este caso. Es una denominación común, que corresponde con el término inglés *sentence* (oración).

4.1.1. Más sobre alternativa condicional

Existen varias cuestiones acerca de la forma de usar alternativa condicional que merecen atención. Es extremadamente común incurrir en abusos operacionales de la alternativa condicional. Debido a que en este libro propiciamos una visión denotacional, abstracta, de los programas, discutiremos estos casos que usualmente son ignorados en otros textos (o considerados como perfectamente adecuados).

Veremos tres formas de abuso de la alternativa condicional: su utilización para evitar la parcialidad de funciones en forma excesiva, su utilización anidada y su utilización en presencia de condiciones parciales.

Alternativas y parcialidad

Cuando presentamos la alternativa condicional, vimos que la misma nos permitía elegir alternativamente entre dos cursos de acción en base a una condición (de ahí el nombre de *alternativa condicional*). También vimos que una variante útil de este comando era utilizar solamente la rama del *then* (la alternativa verdadera), descartando la rama del *else* (la alternativa falsa), y que esto podía utilizarse para evitar la parcialidad de algunas operaciones. Sin embargo, no mencionamos nada sobre la conveniencia o no de utilizar la alternativa de esta manera.

Leer con Atención

La forma de evitar la parcialidad de ciertas operaciones mediante el uso de alternativas condicionales debe usarse con cuidado. Hay ocasiones donde es mejor utilizar la operación directamente, y asumir el requisito de que la precondición debe cumplirse trasladándolo al procedimiento que se está definiendo (por ejemplo, cuando la condición es demasiado compleja, o no queremos realizar la verificación).

Por ejemplo, si consideramos la operación `MoverN(n,dir)` del ejercicio 3.1.3, parte 2, vemos que la misma tiene como precondición que haya tantas celdas en dirección `dir` como la cantidad dada por `n`. Esto garantiza que la operación se mueve exactamente `n` lugares o falla. Si en cambio hubierámos hecho la variante donde el `Mover` interno se utiliza de manera segura, la operación habría sido total pero no habría garantías de que se movió `n` lugares (ver el ejercicio 4.1.1).

Actividad 1

Realice el ejercicio 4.1.1 y compare su comportamiento con el procedimiento `MoverN` del ejercicio 3.1.3, parte 2.

Ejercicio 4.1.1. *Realizar un procedimiento* `MoverNTotal` *que dados* `n` *de tipo número y* `dir` *de tipo dirección, se mueva* `n` *celdas en la dirección* `dir`*, si puede, o quede detenido en el borde, si hay menos de* `n` *celdas en dicha dirección.*

Ayuda: *reemplace el comando* `Mover` *en el procedimiento* `MoverN` *por uno llamado* `MoverSiSePuede` *que utilice una alternativa condicional para moverse solamente en el caso de que sea posible.*

Para Reflexionar

¿Es adecuado que el procedimiento para moverse `n` lugares nunca falle? Reflexione sobre la importancia de que una operación cumpla con su contrato, y la necesidad que tiene que ciertos requerimientos se cumplan para poder hacerlo. Si en cambio no le ponemos condiciones, ¿cómo podemos esperar que el procedimiento efectivamente cumpla?

Es muy común que al comenzar a programar se intente construir todas operaciones totales. El problema con esto es que en ese caso el propósito de los procedimientos construídos no es el mismo que el que se esperaba. Por ejemplo, el propósito de `MoverN` es moverse una cierta cantidad dada de lugares; pero si esto no es posible, ¡*no queremos* que se comporte de alguna otra manera! Es mucho mejor que en ese caso el procedimiento falle, y la precondición nos avise

de manera adecuada ese hecho, así al utilizarlo en nuestro propio procedimiento podemos decidir si verificamos que haya lugar, si eso no hace falta porque sabemos que lo habrá, o simplemente lo asumimos como válido y lo ponemos como parte de la precondición de nuestro propio procedimiento. El procedimiento `MoverNTotal` nunca falla, pero tampoco se mueve siempre la cantidad de lugares especificada, haciendo que sea compleja la verificación de cada uno de los casos, dificultando de esa manera su reutilización.

En conclusión, cuando se trabaja con efectos no necesariamente uno quiere cubrir todas las precondiciones de los comandos, sino que depende del problema condicionar o no la ejecución un comando. Si nos queremos mover 8 celdas, el propósito es moverse las 8, y no "moverse tantas celdas como pueda con máximo 8", que es otro procedimiento con un propósito distinto. Por eso la alternativa depende del objetivo de los procedimientos y no se usa siempre para cubrir las precondiciones de los comandos, aunque pueda hacerse así.

Alternativas anidadas

Los comandos de alternativa condicional pueden *anidarse* (o sea, usar un condicional en las ramas de otro), para tener diversas alternativas. Por ejemplo, suponer que el cabezal representa la posición de nuestro ejército y se desea representar la siguiente acción: si hay enemigos visibles, entonces nuestro ejército debe intentar huir al Norte y en caso de no poder, debe esconderse; pero si no hay enemigos, debe avanzar hacia el Este, y si no encuentra enemigos al llegar, debe armar una trinchera, pero si los encuentra, debe atacarlos. El código para resolver este problema podría ser:

```
if (not hayEnemigosAca()) -- 'Aca' es la celda inicial
  {
   Mover(Este) -- No hay enemigos, y puede avanzar
   if (not hayEnemigosAca())
       -- El nuevo 'Aca' es la celda al Este de la inicial!!
        { ArmarTrinchera() }
   else { Atacar() }
  }
else
  { -- Hay enemigos, y debe huir o esconderse
    if (elCaminoEstaDespejado(Norte)))
         { Mover(Norte) }
    else { Esconderse() }
  }
```

Leer con Atención

Observar que la precondición de este programa es que la celda actual no puede encontrarse en la fila más al `Este` ni en la fila más al `Norte`. También que no hace falta conocer la implementación exacta de las funciones `hayEnemigosAca`, `elCaminoEstaDespejado` o de los procedimientos `ArmarTrinchera`, `Atacar`, etc. para entender este código (alcanza con entender sus precondiciones y sus efectos).

Leer con Atención

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

Con procedimientos, quedaría de la siguiente manera

```
if (not hayEnemigosAca())
  { InvadirPosicionAlEste() }
else
  { HuirOEsconderse() }
```

donde los procedimientos utilizados expresan acciones condicionales

```
procedure InvadirPosicionAlEste()
 {- PROPÓSITO:
    * avanza 1 celda al Este y al llegar, decide si arma
      una trinchera (si no hay enemigos) o ataca al enemigo.
    PRECONDICIÓN:
    * hay una celda al Este
 -}
 {
  Mover(Este)
   if (not hayEnemigosAca())
     { ArmarTrinchera() }
   else
     { Atacar() }
 }
procedure HuirOEsconderse()
 {- PROPÓSITO:
    * decide si huir hacia el Norte o esconderse, en base
      a las condiciones del camino
    PRECONDICIÓN:
    * hay una celda al Norte
 -}
 {
  if (elCaminoEstaDespejado(Norte)))
    { Mover(Norte) }
  else
    { Esconderse() }
 }
```

Actividad de Programación 2

Realice el ejercicio 4.1.2 y observe cómo se manifiesta el uso de condicionales anidados. (La restricción de utilizar únicamente como condición `hayBolitas` fuerza a la utilización de condicionales anidados.)

Ejercicio 4.1.2. *Escribir un programa que decida entre un conjunto de acciones, basándose en un código representado en la celda actual con base en bolitas de colores. Las acciones se representan con los procedimientos* `DibujarVentana`, `RedimensionarVentana`, `MaximizarVentana` *y* `MinimizarVentana`. *Los códigos se indican según la presencia de bolitas en la celda actual. El código que indica que debe dibujarse una ventana es la presencia de bolitas verdes. El código que indica que la ventana debe redimensionarse es la presencia de bolitas azules (siempre y cuando no haya verdes). El código que indica que la ventana debe maximizarse es la presencia de bolitas negras (pero no verdes ni azules). Y el código que indica que la ventana debe minimizarse es la presencia de bolitas rojas (pero no de ninguno de los otros 3 colores). Si no hay bolitas, entonces no debe hacerse nada.*

Cada condición utilizada debe escribirse exclusivamente utilizando la función `hayBolitas`, *sin ningún otro conectivo. Además, las acciones son disjuntas; o sea solo debe realizarse una de ellas.*

Veremos en la próxima sección una manera de expresar este ejercicio de forma mucho más adecuada usando *alternativa indexada*. O sea, los condicionales anidados no siempre son necesarios. En muchas ocasiones pueden ser reemplazados por condiciones más complejas, a través del uso de *conectivos lógicos* y de abstracción a través de funciones, como se vio en el capítulo anterior.

Leer con Atención

El uso de condicionales anidados, entonces, es considerado por nosotros como una forma extrema que solo debe utilizarse cuando no es sencillo abstraer los comportamientos con procedimientos o mejorar las condiciones con el uso de expresiones complejas mediante conectivos y funciones, o cuando las condiciones de eficiencia lo requieran. Su abuso indica una forma altamente operacional de pensar, en contraste con las formas de alto nivel que proponemos en este libro.

Alternativas con condiciones parciales

La última de las formas de abuso de la alternativa condicional está vinculada con una forma operacional de pensar, y tiene que ver también con el uso de condiciones parciales. Se utiliza en el caso en que deben verificarse dos (o más) condiciones, pero donde la segunda de ellas es parcial, y depende de que la primera sea verdadera para poder funcionar adecuadamente. Por ejemplo, consideremos la función `esCeldaVaciaAl` del ejercicio 3.2.15; vemos que se trata de una operación parcial, que requiere que haya una celda en la dirección dada. Supongamos que la queremos utilizar para verificar que no hay enemigos en una dirección dada (suponiendo que los enemigos se codifican con bolitas de algún color) y colocar una bolita verde o roja en la celda actual en base a dicha condición. Claramente, existen dos situaciones donde no hay enemigos en la dirección dada: cuando no existe la celda en esa dirección, o cuando la celda existe y está vacía. Esto podría codificarse utilizando alternativas condicionales de manera excesivamente operacional de la siguiente forma

```
procedure CodificarSiHayEnemigoAlFeo(dir}
 /*
    PROPÓSITO: coloca una bolita roja o verde en la celda actual,
               si hay una celda ocupada por enemigos en la
               dirección dada
  */
 {
   if (puedeMover(dir))
     { if (not esCeldaVaciaAl(dir)) -- Es seguro preguntar,
                                    -- porque puede moverse
         { Poner(Rojo) } -- Hay celda y hay enemigos
       else
         { Poner(Verde) } -- Hay celda pero no enemigos
     }
   else
     { Poner(Verde) } -- No hay celda, entonces no hay enemigos
 }
```

Esta solución es innecesariamente complicada, puesto que debemos replicar en dos ramas la indicación de que no hay enemigos (una por cada una de las posibilidades de que no haya). En principo parecería que no puede hacerse de otra manera, pues para poder invocar la función `esCeldaVaciaAl` debemos estar seguros que se puede mover en dicha dirección.

Sin embargo es posible mejorar esta forma de codificar mediante el uso de una característica particular de las operaciones booleanas, conocida con el nom-

En algunas traducciones se utiliza como equivalente el término *cortocircuito*. Sin embargo, consideramos que dicha traducción no es adecuada. Un cortocircuito es una falla en un sistema eléctrico por error en la conexión de ciertos cables; si bien el término en inglés tiene esta acepción, también significa *evitar hacer algo*, y es esta la acepción que más se ajusta al uso de *short-circuit* cuando se habla de condiciones booleanas. Dado que el término castellano *cortocircuito* no tiene esta última acepción, preferimos la traducción de *circuito corto* como un neologismo que indica este comportamiento de las condiciones booleanas.

bre inglés de *short circuit* (en castellano, *circuito corto*). Una operación booleana se conoce como de *circuito corto* si detiene su análisis de condiciones ni bien puede otorgar una respuesta, independientemente de que haya verificado todas las condiciones. Esto sucede cuando la conjunción determina que su primera condición es falsa (porque sin importar el valor de la otra condición, la conjunción será falsa), y cuando la disyunción determina que su primera condición es verdadera (porque sin importar el otro valor, la disyunción será verdadera).

Mediante la característica de circuito corto de la conjunción, el ejemplo anterior podría haber escrito como

```
procedure CodificarSiHayEnemigoAlMejor(dir}
 /*
    PROPÓSITO: coloca una bolita roja o verde en la celda actual,
               si hay una celda ocupada por enemigos en la
               dirección dada
  */
 {
   if (puedeMover(dir)
    && not esCeldaVaciaAl(dir))
        -- Es seguro preguntar, porque puede moverse ya
        -- que el circuito corto de la conjunción garantiza
        -- que no llega aquí a menos que pueda hacerlo
     { Poner(Rojo) } -- Hay celda y hay enemigos
   else
     { Poner(Verde) } -- No hay celda, o hay celda pero no enemigos
 }
```

Observar como la condición es una sola hecha de una conjunción, y funciona a pesar de que la segunda parte de la misma es una operación parcial, puesto que si la primera parte es falsa, la segunda no se ejecutará (la conjunción devolverá falso sin mirar la segunda parte). Esta forma es mucho más adecuada para programar denotacionalmente, lo cual puede comprobarse en el siguiente ejercicio, que busca abstraer la condición anterior.

Actividad de Programación 3

Realice el ejercicio 4.1.3 y utilícelo para escribir una versión aún más adecuada del procedimiento `CodificarSiHayEnemigoAl` (adecuada en el sentido que es más clara y más legible, nombrando mejor sus partes).

Ejercicio 4.1.3. *Escribir una función* `hayCeldaVaciaAl` *que retorne verdadero si hay una celda en la dirección dada y la misma está vacía, y falso en caso contrario. ¿La operación será parcial o total?*

Ayuda: *utilice la capacidad de circuito corto de la conjunción.*

El uso de la característica de *circuito corto* de las operaciones booleanas es siempre preferible a la anidación de condiciones, en aquellos casos donde resulta equivalente su utilización.

4.1.2. Alternativa indexada

En el último ejemplo de la subsección anterior vimos la posibilidad de decidir entre varias acciones con base en el valor de un código. Esta forma de decisión se puede representar, tal cual se vio en dicho ejemplo, mediante condicionales anidados. Sin embargo, en muchos lenguajes existe otra forma de representar este tipo de decisiones: la *alternativa indexada*.

En esta forma de alternativa, se elige entre varios comandos con base en un índice, que indica de alguna forma cuál de los comandos es el elegido. La sintaxis

en GOBSTONES para alternativas indexadas utiliza la palabra clave `switch`, y se escribe de la siguiente manera:

```
switch (decodificarCelda()) to
  1 -> { DibujarVentana() }
  2 -> { RedimensionarVentana() }
  3 -> { MaximizarVentana() }
  4 -> { MinimizarVentana() }
  _ -> { NoHacerNada() }
```

En este caso, la función `decodificarCelda` devolverá `1` si en la celda actual hay bolitas verdes, `2` si hay bolitas azules pero no verdes, `3` si hay bolitas negras pero no azules ni verdes, `4` si hay solo bolitas rojas, y cualquier otro número (por ejemplo, `0`) si no hay bolitas en la celda actual (para definir la función `decodificarCelda` hace falta alguna forma de decidir mediante expresiones o alguna forma de recordar valores, por lo que volveremos a ella después de presentar la idea de *memoria*.) Finalmente, el procedimiento `NoHacerNada` no tiene ningún efecto.

En GOBSTONES existe un comando predefinido que no hace nada. Por razones históricas, el mismo se denomina `Skip`. También puede implementarse con otros comandos, por ejemplo, poniendo y sacando una bolita de cierto color de la celda actual, o con un bloque vacío.

Definición 4.1.1. *El comando de* alternativa indexada *(también conocido como sentencia* `switch`*) tiene la siguiente forma:*

`switch (`$<$ *indexExp* $>$`) to`
$<$ *val*$_1$ $>$ `->` $<$ *bloque*$_1$ $>$
$\vdots$
$<$ *val*$_n$ $>$ `->` $<$ *bloque*$_n$ $>$
`_ ->` $<$ *bloqueDefault* $>$

siendo $<$ *indexExp* $>$ *una expresión que describe un valor de alguno de los tipos básicos,* $<$ *val*$_1$ $>$ *a* $<$ *val*$_n$ $>$*, una serie de valores de ese tipo, y* $<$ *bloque*$_1$ $>$*,* $<$ *bloque*$_n$ $>$ *y* $<$ *bloquedefult* $>$*, una serie de bloques de código cualesquiera (que normalmente son diferentes, aunque podrían ser iguales). Al bloque* $<$ *var*$_i$ $>$ *se lo denomina* rama i-ésima *(o rama del valor* $<$ *val*$_i$ $>$*), para cada* i *entre* 1 *y* n*, y al bloque* $<$ *bloqueDefault* $>$ *se lo denomina* rama por defecto *(o rama* default*).*

El efecto de un comando `switch` es el de elegir uno de los bloques, en base al valor de la expresión $<$ *indexExp* $>$: si la expresión vale alguno de los $<$ *val*$_i$ $>$, entonces se elige el bloque $<$ *bloque*$_i$ $>$, y si su valor no es ninguno de los listados, entonces se elige el bloque $<$ *bloqueDefault* $>$.

Hay dos cuestiones que observar en un `switch`. La primera es que el orden en el que se elige es secuencial, al igual que en un `if`; o sea, si el mismo valor aparece más de una vez, solo se utilizará su primera aparición. La segunda es que la rama *default* siempre debe aparecer y captura todos los valores posibles que no se encuentren entre los listados. Esto puede llevar al caso anómalo donde esta rama nunca se elija, por no quedar valores disponibles.

Para ejemplificar esta situación, consideremos la necesidad de codificar una dirección en la celda actual, utilizando bolitas verdes. Un código posible sería

```
procedure CodificarDireccion(dir)
 {-
    SUPOSICION:
     * en la celda actual no hay bolitas verdes
 -}
 {
   switch (dir) to
     Norte -> { PonerN(1, Verde) }
     Sur   -> { PonerN(2, Verde) }
     Este  -> { PonerN(3, Verde) }
     _     -> { PonerN(4, Verde) }
 }
```

donde se observa que solo se pregunta por 3 direcciones, siendo innecesario preguntar por la dirección `Oeste` en el último caso, porque es la única opción posible. Si quisiésemos explicitar la pregunta sobre cada una de las direcciones, el `case` debería contener igualmente la rama default, pero con código que jamás se ejecutaría:

```
procedure CodificarDirAlternativo(dir)
 {
   switch (dir) to
     Norte -> { PonerN(1, Verde) }
     Sur   -> { PonerN(2, Verde) }
     Este  -> { PonerN(3, Verde) }
     Oeste -> { PonerN(4, Verde) }
     _     -> { Skip }
 }
```

Observar que no es imprescindible contar con el comando `switch`, puesto que el mismo puede ser representado mediante condicionales anidados y el uso del operador de igualdad ==, de la siguiente manera

```
if (<indexExp>==<val1>)        <bloque1>
else { if (<indexExp>==<val2>)<bloque2>
else { if (...)               ...
else                          <bloqueDefault>
}}
```

Sin embargo, es mucho más cómodo escribir esta construcción mediante el comando `switch`, y más representativo de las intenciones del programador de decidir entre varias alterantivas en base a un valor dado. En algunos lenguajes donde no existe el comando `switch`, existe una forma de alternativa condicional que permite varias ramas en el `if` mediante la palabra clave `elif` (abreviatura de `else if`), y que permite una construcción como la presentada recién con ifs anidados.

PYTHON por ejemplo.

Para Reflexionar

Cuando la alternativa indexada se utiliza sobre el tipo de los booleanos, y el único valor consultado es el `True`, la misma es equivalente a una alternativa condicional. Dicho de otro modo, la alternativa condicional puede entenderse como una alternativa que indexa sobre booleanos.

Actividad de Programación 4

Realice el ejercicio 4.1.4 utilizando un `switch`. Luego intente realizarlo utilizando un `if-then-else` y compare el código de ambas soluciones.

Ejercicio 4.1.4. *Escribir un procedimiento* `ProcesarTecla`*, que dado un número codificando a una tecla que fue presionada en algún tipo de interfase, determine la acción a seguir. Los códigos de teclas y las acciones que debe producir cada tecla son los siguientes:*

Tecla	*Código*	*Acción*
↑, *w*	*119*	`Acelerar()`
↓, *s*	*97*	`Frenar()`
←, *a*	*115*	`GirarIzquierda()`
→, *d*	*100*	`GirarDerecha()`
<*espacio*>	*32*	`Disparar()`
<*escape*>	*27*	`Terminar()`
otras	_	`MostrarError()`

4.2. Más sobre repeticiones

Como vimos, una *repetición* es un comando que expresa la ejecución reiterada de otro comando un cierto número de veces que se indica con un rango de valores.

En este apartado vamos a profundizar sobre algunas combinaciones de repeticiones indexadas, y veremos una nueva forma de repetición, la *repetición condicional*, también conocida como *sentencia* `while` o *sentencia* `do-while`. Además, veremos una manera de estructurar programas que utilizan repetición condicional, denominada por nosotros como *recorridos*, pensados como repeticiones que procesan *secuencias* de elementos. El objetivo de los recorridos es simplificar el uso de repeticiones condicionales, y evitar la multitud de problemas que pueden presentarse al utilizar esta poderosa forma de repetición.

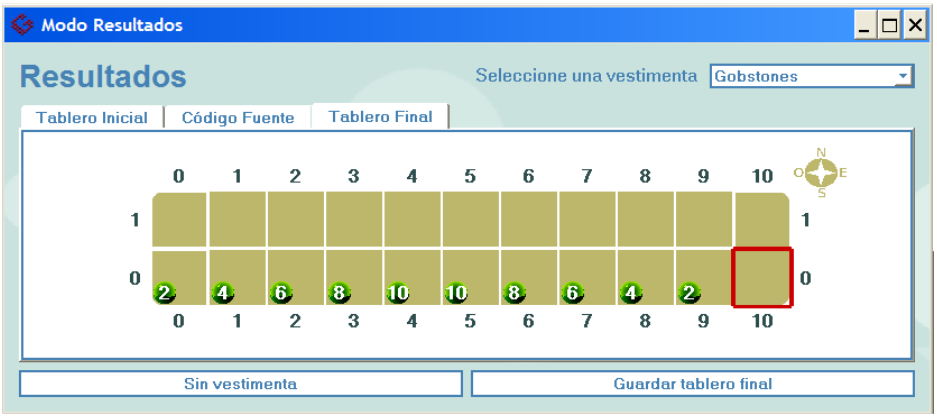
4.2.1. Más sobre repetición indexada

Al realizar repeticiones, puede suceder que algunos de los elementos del rango sobre el que se repite no pueda ser procesado de la misma manera que el resto. En ese caso, y acorde al enunciado del ejercicio mencionado, el procesamiento puede utilizar una alternativa condicional para evitar procesar elementos incorrectos. Para ejemplificar esto, consideremos un procedimiento al que podemos llamar `CodificarDireccionDelEnemigo`, que deje en la celda actual una marca que indique en cuál de las 4 direcciones se encuentra el enemigo al momento de poner la marca. Para este ejemplo vamos a suponer que el enemigo se encuentra en solo una de las celdas contiguas a la actual (o sea, no se ha dispersado...), y que la celda actual no contiene bolitas del color en el que codificaremos la respuesta. El código para dicho procedimiento podría ser

Imaginemos que somos los exploradores de nuestro ejército, y venimos siguiendo al enemigo, marcando el territorio para que nuestro ejército pueda seguirnos.

```
procedure CodificarDireccionDelEnemigo()
 {-
    PROPÓSITO:
     * Dejar en la celda actual una codificación de la
       posición en la que está el enemigo
    PRECONDICIÓN:
     * no hay bolitas verdes en la celda actual
     * solo una de las celdas contiguas tiene enemigos
 -}
 {
   foreach dir in [Norte, Este, Sur, Oeste]
     {
       if (puedeMover(dir))
         {
           if (hayEnemigosAl(dir))
             { CodificarDireccion(dir) }
         }
     }
 }
```

Se puede observar que hay una repetición sobre las 4 direcciones, para verificar por turnos en las 4 celdas contiguas. ¿Pero qué sucede si la celda actual no tiene 4 celdas contiguas? En ese caso, en esa dirección no se debe verificar, porque eso produciría la autodestrucción del cabezal; para eso se utiliza la alternativa condicional que pregunta si se puede mover en la dirección que se está procesando. La función `hayEnemigosAl` es la que se definió en el ejercicio 3.3.3, y el procedimiento `CodificarDireccion` se presentó en la sección de alternativa indexada. Observar que la precondición de `hayEnemigosAl` es que pueda moverse en la dirección suministrada como parámetro, y por lo tanto, no debe preguntarse si hay enemigos hasta no saber que puede moverse en dicha dirección.



G.4.1. Guarda de bolitas verdes en cantidades pares

Actividad de Programación 5

Pruebe la función `CodificarDireccionDelEnemigo` en un programa, luego de colocar convenientemente todos los procedimientos y funciones necesarias y armar correctamente un procedimiento principal.

Otra detalle a observar en la repetición indexada es que la forma básica de la misma utiliza todos rangos que incluían todos los números entre 2 dados (por ejemplo, de 1 a 10) de forma creciente, y como máxima complicación, hacer que el rango dependiese de un parámetro, para poder modificar la cantidad (como en el ejercicio 3.1.3). Sin embargo, a veces es necesario realizar repeticiones que requieren secuencias de números no consecutivos, e incluso secuencias descendentes o arbitrarias.

Por ejemplo, uno puede querer realizar una guarda de bolitas que solo incluya bolitas pares, como en el gráfico G.4.1. Observar que la guarda tiene 2 partes claramente diferenciadas: una parte creciente, con bolitas en cantidad par (2,4,6,8 y 10), y otra parte decreciente, también con bolitas en cantidad par (10,8,6,4 y 2). ¿Cómo podemos lograr esto mediante una repetición indexada? GOBSTONES provee una forma de variar el incremento al especificar un rango; para ello deben escribirse los dos primeros elementos en lugar de solo el primero. Por ejemplo, la parte creciente puede escribirse como sigue

```
foreach i in [2,4..10]
  { PonerN(i, Verde); Mover(Este) }
```

Observar que como la diferencia entre 4 y 2 es de 2, la secuencia especificada será 2 4 6 8 10, como se deseaba. Para la segunda parte el número debe *decrecer*. Para lograr que una secuencia decrezca, el segundo valor debe ser menor que el primero (y por supuesto, el valor final debe ser menor que el inicial). Entonces, la segunda parte quedaría

```
foreach i in [10,8..2]
  { PonerN(i, Verde); Mover(Este) }
```

Una alternativa más complicada, pero que muestra el poder de la matemática al programar, sería conservar los índices creciendo de a uno, pero utilizar alguna relación entre los números 1,2,3,4 y 5, y los números 2,4,6,8 y 10, primero, y los números 10,8,6,4 y 2, luego. En el primer caso, la relación es sencilla: al 1 le corresponde el 2, al 2 le corresponde el 4, al 3, el 6... claramente al número `i` le debe corresponder el número `2*i`. Esto se expresa como:

```
foreach i in [1..5]
  { PonerN(2*i, Verde); Mover(Este) }
```

donde podemos observar que la cantidad de bolitas a poner responde a la fórmula deducida. Para clarificar la forma en que la fórmula establece la correspondencia, podemos usar la siguiente tabla:

i	1	2	3	4	5
2*i	2*1	2*2	2*3	2*4	2*5
=	2	4	6	8	10

Para la segunda relación, vemos que la relación es inversa, porque el número debe *decrecer*. Para lograr que una secuencia decrezca, podemos restar una secuencia creciente a partir de un número fijo. Si pensamos en la secuencia de pares calculada primero, y queremos darla vuelta, deberíamos pensar una fórmula que permita obtener un 10 a partir de un 2, un 8 a partir de un 4, un 6 a partir de un 6, etcétera. Esto se puede obtener restando `12-(2*i)`. Nuevamente, una tabla puede ayudar a clarificar la forma en que la fórmula establece la correspondencia:

i	1	2	3	4	5
2*i	2	4	6	8	10
12-(2*i)	12-2	12-4	12-6	12-8	12-10
=	10	8	6	4	2

Estas formas no es necesario aprenderlas en un lenguaje donde los rangos pueden variar distinto que de a uno, pero resulta imprescindible para otros lenguajes donde esta facilidad no existe.

Actividad de Programación 6

Realice el ejercicio 4.2.1, usando las ideas explicadas hasta aquí y obtenga una guarda como la del gráfico G.4.1. Hágalo con la variante de rango con variación distinta de uno y con la variante que usa fórmulas.

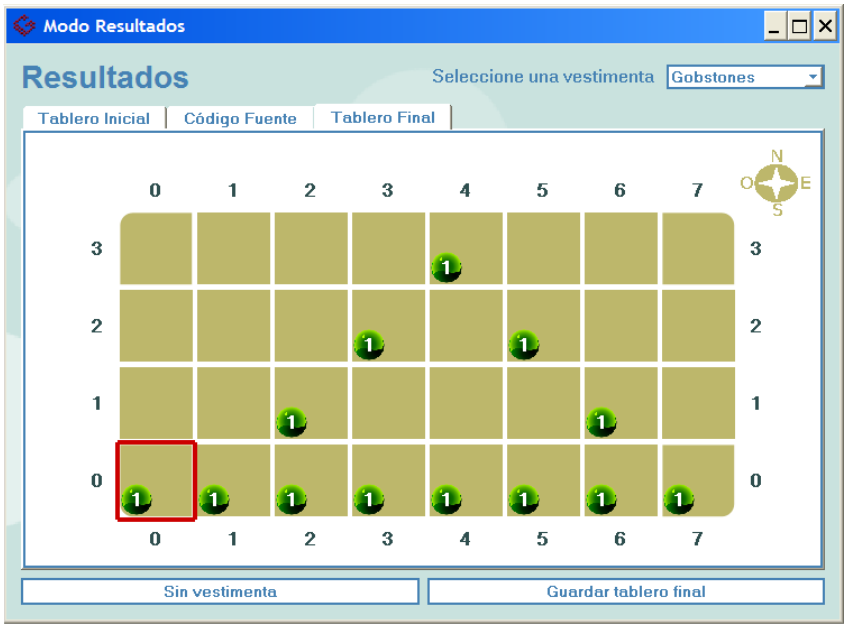
Ejercicio 4.2.1. *Escribir un procedimiento* `GuardaDePares`*, que permita realizar una guarda de bolitas verdes similar a la del gráfico G.4.1. Utilizar dos repeticiones indexadas, una creciente y otra decreciente.*

Resumiendo lo analizado hasta aquí, podemos obtener secuencias regulares que no sean crecientes o de a uno a través de dos maneras: rangos especiales o fórmulas aplicadas a los índices. En el caso de las fórmulas, para secuencias con incrementos mayores a uno se utilizan multiplicaciones; para secuencias decrecientes, se resta una secuencia creciente a un tope dado.

Para Reflexionar

Si en algún lenguaje las repeticiones indexadas son siempre de uno en uno, pueden obtenerse otras secuencias a partir del uso de fórmulas. Aunque este curso no hace uso de una matemática intensiva, reflexione sobre la importancia de tener un buen manejo matemático y numérico para realizar programas más complejos. Piense en las facilidades que podrían proveer los lenguajes para simplificar la matemática necesaria.

Mediante estas repeticiones indexadas con diferentes secuencias podemos revisar algunos de los ejercicios vistos en el capítulo 2.



G.4.2. Resultado de invocar `DibujarTrianguloBase(8)` (sin cumplir la precondición): observar que no es exactamente un triángulo

Actividad de Programación 7

Realice los ejercicios 4.2.2 y 4.2.3, y verifique que obtienen los mismos resultados que los originales que mejoran.

Ejercicio 4.2.2. *Rehacer el ejercicio 2.4.12 para generalizar el tamaño de la base del triángulo. Observar que se requiere como precondición que el tamaño de la base sea un número impar. Completar la precondición con la cantidad de celdas que debe haber disponibles para el dibujo.*

Leer con Atención

Hasta el momento habíamos visto ejemplos donde la precondición indicaba los requisitos necesarios para que el programa no se autodestruyese. Sin embargo, las precondiciones se usan en general para establecer los requisitos necesarios para que el programa funcione como se espera. De esta manera, no es siempre necesario que si la precondición no se cumple, el programa debe autodestruirse; muchas veces el programa parece funcionar, pero hace cosas que no son lo que se esperaba del mismo (como dibujar de forma incompleta una figura). Esta forma de utilizar precondiciones ofrece gran flexibilidad a la hora de diseñar procedimientos y ayuda a evitar controles innecesarios de condiciones.

En el caso del triángulo, según la solución que uno adopte, al invocar el procedimiento con un número par (o sea, no cumpliendo la precondición), el dibujo no resultará en un triángulo. Por ejemplo, en la solución que pensamos nosotros, al invocar `DibujarTrianguloBase(8)` queda un dibujo como el que mostramos en el gráfico G.4.2, que no es exactamente un triángulo de base 8, sino uno de base 7 con una “pata”.

Ejercicio 4.2.3. *Rehacer el ejercicio 2.4.13 utilizando una repetición de 1 a 3. Hacerlo utilizando un rango con incremento distinto de uno y también con una*

fórmula. Observar que los números impares obedecen, en este caso, a la fórmula $(2 * i + 3)$*, siendo* i *el índice de la repetición, y que si uno los quiere en orden inverso, debe restarlos del tope de* 14*.*

Leer con Atención

Se pueden conseguir fórmulas mucho más complejas que permitan obtener secuencias no regulares a través del uso de funciones que calculen los resultados esperados, por ejemplo, una secuencia de números primos, o secuencias con pasos variables en distintas partes, etcétera.

Para el caso en que la secuencia de elementos no sea regular, donde no es simple expresarlas mediante fórmulas lo suficientemente generales, GOBSTONES provee una forma de especificar la secuencia enumerando cada uno de los elementos. Para hacer esto, sencillamente escribimos los elementos entre llaves y separados por comas. Por ejemplo, la secuencia 4 8 15 16 23 42 puede especificarse como `[4, 8, 15, 16, 23, 42]`.

4.2.2. Repetición condicional

Sabemos que la repetición indexada repite un bloque de código un número fijo de veces, cantidad que es controlada por una secuencia de valores. Ahora bien, ¿qué debemos hacer si no conocemos de antemano la cantidad de veces que debemos repetir un comando? Por ejemplo, imaginemos que queremos movernos hasta el borde del tablero. ¡Puesto que no tenemos en GOBSTONES forma de conocer las dimensiones del mismo, no hay manera de saber cuántas veces debemos movernos! Situaciones similares a esta se presentan numerosas veces. Considerar por ejemplo que cuando uno busca sus llaves repite la acción de buscar en lugares, pero no sabe de antemano en cuántos lugares debe buscar.

Actividad 8

Considere realizar un procedimiento que lleve el cabezal hasta el borde del tablero, pero sin utilizar el comando `IrAlBorde`. La solución debería involucrar una repetición del comando `Mover`, pero, como la posición inicial del cabezal es arbitraria, ¡no sabemos cuántas celdas debemos movernos! Piense de qué otras maneras podría realizar esto en GOBSTONES con los elementos presentados hasta el momento. (**Atención:** no hay forma de hacerlo con esos elementos. La idea de la actividad es pensar alternativas hasta convencerse de que efectivamente ninguno de los elementos vistos hasta ahora (salvo el comando primitivo que lo hace de manera predefinida) sirve para esa tarea.)

Claramente no se trata de adivinar una cantidad, sino de proveer alguna herramienta que permita realizar una repetición controlada de otra forma, y no mediante un índice. Esta forma se denomina *repetición condicional*, pues la misma permite repetir un comando 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 4.2.1. *La* repetición condicional *es una forma de repetición que depende de una condición booleana para indicar cuándo 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 <*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 vuelve 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 que la condición llegue a resultar falsa en algún momento.

Leer con Atención

En la repetición indexada, la cantidad de repeticiones se conoce de antemano, por lo que no hace falta controlar el final en cada iteración. En cambio, en la repetición condicional, la condición puede volverse falsa luego de cualquiera de las iteraciones, pero no se sabe anticipadamente de cuál de ellas. Por eso en esta forma de repetición la condición se evalúa cada vez.

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 una secuencia sobre la cual realizar la iteración). El ejemplo más sencillo de uso de repetición condicional es llevar el cabezal hasta un borde del tablero, como se propuso pensar en la actividad del inicio de esta sección. 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) }
  }
```

Si realizó concienzudamente la actividad inicial, ya comprobó que no hay manera de realizar esta tarea con una repetición indexada pues no se conoce de antemano el número de celdas que deberá moverse el cabezal. Debe observarse 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. Pero sí podría parametrizarse, lo cual se evidencia en el próximo ejercicio.

Esto es así porque el tablero es finito.

Actividad de Programación 9

Realice el ejercicio 4.2.4, y pruébelo con distintas direcciones.

Ejercicio 4.2.4. *Escribir un procedimiento* `IrHastaElBorde`*, que dada una dirección, se mueve en esa dirección hasta que llega al borde del tablero. Este procedimiento realiza la misma tarea que el comando* `IrAlBorde`*, comprobando que*

con las herramientas adecuadas, algunos comandos básicos pueden ser reemplazados.

Leer con Atención

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.

Como ejemplo, considerar el siguiente código:

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

Donde los dos procedimientos internos son cualesquiera que definan operaciones totales. Es claro que como la condición es siempre la misma (es independiente de la repuesta obtenida o del estado del tablero), 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 MoverseHastaLaEsquina(d1, d2)
  {- PROPÓSITO:
      * ilustrar el uso de precondiciones para
        evitar no terminación
      * moverse simultaneamente en dos direcciones,
        hasta alcanzar una esquina del tablero
     PRECONDICIÓN:
      * d1 y d2 no pueden ser direcciones opuestas
  -}
  {
    while (puedeMover(d1) && puedeMover(d2))
      { Mover(d1); Mover(d2) }
  }
```

Observar que la condición establece que el valor de los parámetros debe ser tal que no sean direcciones opuestas. Si las direcciones son opuestas (por ejemplo, una es `Norte` y la otra es `Sur`), los efectos de ambos `Mover` se anularán, y la repetición no terminará nunca, provocando la ejecución infinita. En cualquier otro

caso los efectos se irán acumulando haciendo que a la larga se alcance una esquina del tablero, por lo que el procedimiento se dentendrá.

Para controlar este poder de la repetición condicional, presentamos a continuación la idea de *recorrido*.

4.2.3. Recorridos simples

La idea de recorrido es una herramienta conceptual que ofrece una forma de ordenar los elementos en una repetición condicional de manera de evitar los problemas más comunes que aparecen al intentar utilizar este tipo de repetición. Creemos que es importante para comenzar a aprender que se respeten los esquemas brindados por esta herramienta, como forma de ordenar las ideas; cuando esta forma fue completamente dominada es más sencillo pasar a formas más libres y complejas de manejar de la repetición condicional.

Es usual que muchas veces al pensar una tarea que involucra repetición, tengamos en mente alguna secuencia de elementos. Por ejemplo, si buscamos procesar todas las celdas de una columna pintando cada una de ellas de un color determinado, la secuencia en cuestión es la secuencia de celdas de la columna actual.

Actividad de Programación 10

Intente escribir el código de un procedimiento `PintarColumna` que pinte de un color dado todas las celdas de la columna. ¡El código se ofrece a continuación, pero es más interesante si lo intenta solo primero y luego contrasta su solución con la nuestra!

El programa pedido podría escribirse como:

```
procedure PintarColumna(color)
 {- PROPÓSITO:
     * pinta una columna entera con bolitas de color
    PRECONDICIÓN:
     * ninguna, es una operación total
    OBSERVACIÓN:
     * se estructura como un recorrido por celdas
  -}
 {
   -- empieza arriba de todo para estar seguro
   --  que las procesa todas
   CabezalAlTopeDeLaColumna()

   while (not llegoALaBase())
      -- todavía falta procesar celdas
    {
      PintarCeldaDe(color) -- procesar la celda
      Mover(Sur)    -- pasar a la siguiente celda
    }

   -- pinta la última celda, pues no entró al while
   --  si la celda estaba en la base (y por lo tanto,
   --  no la pintó)
   PintarCeldaDe(color)
 }
```

Aquí podemos observar varios puntos interesantes en referencia al uso de la repetición condicional. El primero es que se comienza llevando el cabezal al tope de la columna para asegurarse que todas las celdas resultarán pintadas. Otro más es que al finalizar el ciclo hace falta pintar la última celda a mano, pues el

pintar la celda dentro del ciclo no se realiza sobre la última celda (ya que desde ella no se puede mover al Sur). Finalmente, vemos que dentro de la repetición hay una parte que procesa la celda actual, y otra que se ubica en el siguiente elemento de la secuencia. Aparte, podemos observar el uso de todas las buenas prácticas de programación que venimos aprendiendo en los capítulos anteriores: uso de procedimientos y funciones auxiliares para describir subtareas, comentarios adecuados en el código para remarcar los puntos interesantes, buen uso de nombres, indentación para indicar la estructura del programa, etcétera.

Actividad de Programación 11

Realice el ejercicio 4.2.5. Pruebe el procedimiento `PintarColumna` en un programa completo. Recuerde copiar el código de `CabezalATopeDeLaColumna` en su programa.

Ejercicio 4.2.5. *Completar las operaciones* `llegoALaBase` *y* `PintarCeldaDe`*.*
La función `llegoALaBase` *verifica si el cabezal se encuentra en la base de la columna, i.e. en la celda que se encuentra más al* `Sur` *(o sea,* no puede moverse más *hacia el sur). El procedimiento* `PintarCeldaDe` *simplemente pone una o más bolitas del color indicado.*

Esta forma de repetición condicional aparece una y otra vez, según cual sea la secuencia de elementos que consideremos. Por ejemplo, si en lugar de pintar una columna de un color determinado, quisiéramos pintar todo el tablero, podríamos hacer un recorrido sobre todas las columnas, pintando cada una de ellas. El procedimiento quedaría muy parecido al de `PintarColumna`, pues la estructura del problema es la misma:

```
procedure PintarTableroPorColumnas(color)
 {- PROPÓSITO:
     * pinta todo el tablero con bolitas de color
    PRECONDICIÓN:
     * ninguna, es una operación total
    OBSERVACIÓN:
     * se estructura como un recorrido por columnas
  -}
 {
   -- va a la primera columna para saber que las hace todas
   CabezalALaPrimeraColumna()

   while (not llegoALaUltimaFila())
     -- todavía falta procesar columnas
    {
      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`.

Actividad de Programación 12

Realice el ejercicio 4.2.6. Utilice el procedimiento `PintarTableroPorColumnas` en un programa. ¡Recuerde incluir todo el código en su programa!

Ejercicio 4.2.6. *Completar el código anterior. Para ello, debe escribir la función* `llegoALaUltimaFila`*, que establece si el cabezal se encuentra en la fila más al* `Este` *(o sea,* no puede moverse más *hacia el este) y también el procedimiento* `CabezalALaPrimeraColumna` *que lleva el cabezal a la primera columna del tablero (similar al procedimiento* `CabezalATopeDeLaColumna`*).*

Este esquema de repetición donde se realiza una tarea para cada uno de una serie de elementos se denomina *recorrido*, pues recorre la secuencia de elementos de a uno, procesando de alguna forma cada uno de ellos. Este concepto está basado en ideas tales como el uso de invariantes de ciclo [Mannila, 2010], la idea de folds [Hutton, 1999] (especialmente *foldr*) y generalizaciones asociadas [Meijer *et al.*, 1991], y el enfoque de esquemas básicos de algoritmos [Scholl and Peyrin, 1988]. Todos estos textos realizan presentaciones avanzadas de la idea y son extremadamente interesantes, pero no son fáciles de comprender cuando uno recién comienza. Por eso, no debe asustarse si intenta mirarlos y no los comprende; la programación abarca un mundo fascinante y vasto de ideas, que no es fácil de dominar en poco tiempo.

El esquema que siguen todos los recorridos es el siguiente:

```
procedure RecorridoGenerico()
 {- PROPÓSITO:
     * recorre una secuencia de elementos genéricos, según
       las definiciones que se den a las partes
    PRECONDICIÓN:
     * depende de las definiciones que se den a las partes
  -}
 {
   IniciarRecorrido()
      -- todo lo necesario para recorrer los elementos:
      --  ubicarse en el primero de ellos,
      --  preparar lo que haga falta, etc.

   while (not llegoAlUltimoElemento())
      -- todavía falta procesar elementos
    {
      ProcesarElementoActual() -- procesa de alguna forma
                               -- el elemento actual
      PasarAlSiguiente()       -- pasar al siguiente elemento
    }

   FinalizarRecorrido()
     -- termina el recorrido, cerrando lo necesario
     -- y completando el trabajo
 }
```

Los procedimientos utilizados en este código son operaciones genéricas (al igual que sus nombres), y deben reemplazarse por las operaciones correspondientes a la secuencia de elementos que quiere recorrerse. Por ejemplo, en el caso de `PintarColumna`, la correspondencia entre las operaciones genéricas y las reales es la siguiente:

Operación genérica	Operación real
`IniciarRecorrido()`	`CabezalAlTopeDeLaColumna()`
`ProcesarElementoActual()`	`PintarCeldaDe(color)`
`PasarAlSiguiente()`	`Mover(Sur)`
`FinalizarRecorrido()`	`PintarCeldaDe(color)`
`llegoAlUltimoElemento()`	`llegoALaBase()`

Como se puede observar, puede haber variaciones entre el procedimiento genérico y el real, por ejemplo, agregando parámetros, o realizando una (o varias) operaciones elementales (como `Mover(Sur)`), o incluso no realizando ninguna operación. Sin embargo, lo importante del recorrido es que se encarga de recorrer y procesar cada uno de los elementos de la secuencia considerada.

Definición 4.2.2. *Un* recorrido *es un esquema de programa que permite estructurar una solución a un problema de procesamiento de cierta secuencia de elementos, sugiriendo el tipo de subtareas a considerar. Las subtareas genéricas de un recorrido se pueden denominar* iniciar del recorrido*,* controlar si es último elemento*,* procesar elemento actual*,* pasar al siguiente elemento *y* finalizar recorrido*.*

Los recorridos más simples tendrán su complejidad en el procesamiento de los elementos. Sin embargo, veremos en breve un caso de recorrido donde la mayor complejidad se encuentra en pasar al siguiente elemento.

Actividad 13

Realice la correspondencia entre operaciones genéricas y operaciones reales en el procedimiento `PintarTableroPorColumnas`. Luego realice la correspondencia para el procedimiento `CabezalAlTopeDeLaColumna`. Tenga en cuenta que en este último caso 3 de las operaciones no realizan nada.

Una cuestión interesante a considerar es que pueden existir varias formas distintas de considerar un problema con recorridos, según cuál sea la secuencia de elementos que consideremos, y cómo dividamos las subtareas. Por ejemplo, en el caso del problema de pintar el tablero, podemos elegir la opción que acabamos de presentar, donde estructuramos la solución como un recorrido por columnas (en cada columna pintamos como un recorrido por filas), pero también podemos elegir otras opciones. Una alternativa sencilla sería considerar un recorrido por filas en lugar de uno por columnas. Otra variante, algo más compleja, pero que resulta en menor cantidad de código, es utilizar un recorrido por celdas.

Actividad de Programación 14

Modifique el código del programa del ejercicio 4.2.6 para que en lugar de hacer el recorrido por columnas, lo haga por filas. ¡No olvide modificar los nombres de los procedimientos y los comentarios adecuadamente!

Para abordar el problema de pintar el tablero mediante un recorrido por celdas, las preguntas pertinentes son cuál será la primera celda a pintar y cómo determinamos cuál es la siguiente en cada caso. Las respuestas a estas preguntas determinarán los restantes elementos del recorrido. Supongamos que elegimos que la primera celda será la de la esquina suroeste, y que las celdas serán recorridas de Oeste a Este y de Sur a Norte, en ese orden. Entonces, el código quedaría así

```
procedure PintarTableroPorCeldasNE(color)
 {- PROPÓSITO:
```

```
    * pinta todo el tablero con bolitas de color
   PRECONDICIÓN:
    * ninguna, es una operación total
   OBSERVACIÓN:
    * se estructura como un recorrido por celdas, de
      Oeste a Este y de Sur a Norte
 -}
{
  -- va a la primera celda para saber que las hace todas
  IrALaEsquina(Sur,Oeste)

  while (not llegoALaEsquina(Norte,Este))
     -- todavía falta procesar celdas
   {
     -- procesar la celda
     PintarCeldaDe(color)
     -- pasar a la siguiente celda
     AvanzarASiguienteDelRecorridoDeCeldas(Norte,Este)
   }

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

Aquí el procedimiento verdaderamente interesante es el que pasa a la siguiente celda. Para realizarlo debe considerarse primero si se puede mover al Este, y en ese caso, moverse; pero si no puede moverse al Este, entonces debe moverse al Norte *y volver al extremo Oeste* para continuar con las celdas de arriba. El código para esto sería

```
procedure AvanzarASiguienteDelRecorridoDeCeldas(dirExterna
                                                ,dirInterna)
 {- PROPÓSITO:
     * pasa a la siguiente celda en un recorrido por celdas
       de todo el tablero
    PRECONDICIÓN:
     * la celda actual no es la esquina dada por los parámetros
       dirExterna y dirInterna (o sea, puede moverse en una de
       las dos direcciones)
  -}
 {
   if (puedeMover(dirInterna))
      -- Todavía quedan celdas en la fila (o columna) actual
    { Mover(dirInterna) }
   else
    {
      -- Pasa a la siguiente fila (o columna)
      Mover(dirExterna)
      -- Vuelve al extremo opuesto de la fila (o columna)
      -- para seguir desde ahí
      IrAlExtremo(dirOpuesta(dirInterna))
    }
 }
```

Suponiendo que la `dirExterna` sea `Norte` y la `dirInterna` sea `Este`, el recorrido se realiza primero en cada fila de Oeste a Este y las filas se recorren de Sur a Norte. La precondición indica que aún quedan celdas por recorrer; en caso de encontrarse en la esquina final, este procedimiento fallará con un boom, cayéndose del tablero.

Actividad de Programación 15

Realice el ejercicio 4.2.7 y coloque las operaciones allí solicitadas y el código del procedimiento `AvanzarASiguienteDelRecorridoDeCeldas` en la `Biblioteca`.

Ejercicio 4.2.7. *Definir un procedimiento* `IniciarRecorridoDeCeldas` *y una función* `esFinDelRecorridoDeCeldas` *que reciban como parámetros dos direcciones y abstraigan adecuadamente las operaciones genéricas de iniciar recorrido y verificar el fin de recorrido utilizadas en el código de* `PintarTableroPorCeldasNE`*.*

Actividad de Programación 16

Realice los ejercicios 4.2.8 y 4.2.9 y pruébelos con varios colores y direcciones.

Ejercicio 4.2.8. *Completar los procedimientos y funciones necesarios para completar el procedimiento* `PintarTableroPorCeldasNE`*.*

Ejercicio 4.2.9. *Realizar un procedimiento* `PintarTableroPorCeldas` *que generalice el recorrido por celdas, tomando como parámetros las direcciones en las que deben recorrerse las celdas. ¿Cuál será la precondición de este procedimiento?*

Si pensamos en variaciones de los recorridos genéricos, vemos que el recorrido no tiene por qué procesar todos los elementos. Por ejemplo, si consideramos que una columna es un cantero, y queremos recorrer cada sector del cantero (representado cada uno con una celda), colocando fertilizante (bolitas negras) solo en aquellos sectores donde haya flores (bolitas rojas), el código podría ser el siguiente

```
procedure FertilizarFlores()
 {- PROPÓSITO:
     * Poner fertilizante en las celdas de la
         columna que contengan flores
    PRECONDICIÓN:
     * ninguna, es una operación total
  -}
 {
   -- va al tope para saber que las hace todas
   IrAlExtremo(Norte)

   while (puedeMover(Sur))
      -- la condición dice "no llegó a la base"
    {
      if (hayFlor())     -- solo se agrega en las celdas
        { Fertilizar() } --  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 (hayFlor())
     { Fertilizar() }
 }
```

Podemos observar que la operación de procesar elemento actual, en este caso, es condicional a que se cumpla la existencia de una flor en la celda. De esta manera, solo las celdas con flores serán fertilizadas. Esto podría abstraerse y considerar un procedimiento `ProcesarCeldaDelCantero` que implemente la acción condicional de fertilizar, logrando

G.4.3. Sendero simple con forma aproximada de “52”

de esa manera una visión uniforme del recorrido. Sin embargo, muchas veces elegimos explicitar la condicionalidad de la forma mostrada.

Actividad de Programación 17

Pase el procedimiento anterior, complete sus subtareas, y pruébelo en un programa con varios tableros diferentes.

Otra cuestión interesante es que la secuencia de elementos no tiene por qué aparecer de manera lineal en la visualización en el tablero. Por ejemplo, si tomamos un *sendero* y lo seguimos, podemos considerar a las posiciones del sendero como los elementos a procesar. Veamos cómo sería, definiendo bien la idea de *sendero* y haciendo un programa que lo procese de alguna manera.

Un *sendero* simple se indica en el tablero mediante una serie de bolitas negras contiguas una de las otras, de manera que cada bolita no tiene nunca más de dos vecinas (excepto la primera y la última). La primera celda del sendero se indica con 2 bolitas negras y es siempre la celda en la esquina suroeste, y la última con 3 bolitas negras y puede ser cualquiera. Es importante remarcar que la celda inicial con 2 bolitas y la celda final con 3 bolitas son las únicas de todo el tablero que tienen esta cantidad de bolitas negras. Además también es importante remarcar que no todas las celdas del tablero que tengan 1 bolita negra serán parte del sendero. Podemos ver un ejemplo de sendero en el gráfico G.4.3. Observar que cada celda interna del sendero tiene a lo sumo dos celdas contiguas (en las cuatro direcciones cardinales) que contienen bolitas. De esta manera, si comenzamos en la celda inicial del sendero, es sencillo saber cuál es la próxima celda a visitar: la única de las contiguas que tiene bolitas negras y en la que no estuvimos antes. También vemos que en este tablero hay 3 celdas con bolitas negras que no pertenecen al

sendero.
Ahora supongamos que la intención es colocar una rosa (representada mediante una bolita roja) en cada celda del sendero. Así, podemos realizar esta tarea definiendo las siguientes operaciones, y luego armando un recorrido con ellas:

Operación genérica	Operación real
`IniciarRecorrido()`	`BuscarInicioSendero()`
`ProcesarElementoActual()`	`ColocarRosa()`
`PasarAlSiguiente()`	`MoverASiguienteSector()`
`FinalizarRecorrido()`	`ColocarRosa()` `VolverAlInicioDelSendero()`
`llegoAlUltimoElemento()`	`esFinDelSendero()`

La operación de `ColocarRosa` es trivial. La condición de `esFinDelSendero` también es simple, pues alcanza con mirar el número de bolitas negras de la celda actual. Las operaciones dadas por los procedimientos `BuscarInicioSendero` y `VolverAlInicioDelSendero` son, en esta forma simplificada de senderos, nada más que `IrALaEsquina(Sur,Oeste)`. Entonces, la operación interesante es`MoverASiguienteSector`, que es una búsqueda por las 4 celdas contiguas de aquella que tenga bolitas negras pero no tenga flores y luego moverse allí. Se puede realizar con dos subtareas, de la siguiente manera

```
procedure MoverASiguienteSector()
 {-
    PRECONDICIÓN:
     * La celda actual no tiene bolitas verdes
     * La celda actual es parte de un camino simple
     * Una de los dos sectores contiguos ya fue visitado
        y tiene una flor
  -}
 {
   -- Elegir la dirección para seguir y codificarla
   foreach dir in [Norte..Oeste]
    {
      if (puedeMover(dir))
        {
          if (haySectorNoVisitadoAl(dir))
            { CodificarDireccion(dir) }
        }
    }
   DecodificarDireccion()
 }
```

La primera parte trabaja de manera similar al procedimiento de la sección 4.2.1 denominado `CodificarDireccionDelEnemigo`, codificando la dirección en la que hay bolitas negras pero no rojas, teniendo en cuenta qué sucede en las celdas que no tienen 4 contiguas. Y luego una segunda parte decodifica la dirección, mirando el número de bolitas verdes (usando, por ejemplo, una alternativa indexada), y con base en ese número, saca todas las verdes y se mueve en la dirección correspondiente. ¡Observar que debe hacerse en dos partes, porque si no, no habría forma de saber si debemos parar de mirar las celdas vecinas! También hay que observar que el procedimiento `DecodificarDireccion` debe eliminar las bolitas verdes *después* de "leerlas", así no quedan en el tablero.

Una vez codificadas todas las partes, podemos definir el procedimiento como

```
procedure PonerFloresEnSenderoSimple()
 {- PROPÓSITO:
     * Poner flores en todas las celdas de un sendero simple
    PRECONDICIÓN:
     * debe haber codificado un sendero simple en el tablero
  -}
 {
   BuscarInicioSendero() -- iniciar recorrido
   while (not esFinDelSendero())
     -- no terminó el recorrido
    {
      ColocarRosa()           -- procesar elemento
```

```
      MoverASiguienteSector() -- pasar al siguiente
    }
  ColocarRosa()          -- finalizar recorrido
  VolverAlInicio()       --  "
}
```

Actividad de Programación 18

Pase el código del procedimiento `PonerFloresEnSenderoSimple`, junto con todas sus partes (codificando las que faltan según el ejercicio 4.2.10), y pruébelo en el tablero de ejemplo mostrado en el gráfico G.4.3. El resultado debería verse como el gráfico G.4.4, donde se puede observar que solo las celdas del sendero tienen rosas.

Ejercicio 4.2.10. *Escribir las operaciones*

- `ColocarRosa`,
- `esFinDelSendero`,
- `BuscarInicioSendero`,
- `VolverAlInicioDelSendero` *y*
- `DecodificarDireccion` *(del procedimiento* `MoverASiguienteSector`*),*

según las ayudas ofrecidas previamente. Una sutileza interesante es que en el caso del procedimiento `VolverAlInicioDelSendero`*, se debería considerar simplemente llamar a* `BuscarInicioSendero`*, para no tener que rehacer aquella en caso de que esta última cambie.*

Una forma de mejorar la definición de sendero es quitar la restricción de que la celda inicial se encuentre en la esquina suoreste. Al hacer esto, deberemos modificar el procedimiento `BuscarInicioSendero` para que sea un recorrido sobre todas las celdas del tablero, buscando aquella que tenga exactamente 2 bolitas negras. Es interesante pensar en un recorrido sobre todas las celdas que se detenga al encontrar la celda buscada. La estructura sería la siguiente

```
procedure BuscarInicioSendero()
 {-
    PRECONDICIÓN:
      * hay una única celda que es el inicio del sendero
       (indicada con 2 bolitas negras)
  -}
 {
   IrALaEsquina(Sur,Oeste)
   while(not esInicioDelSendero())
     {
       AvanzarASiguienteDelRecorridoDeCeldas(Norte,Este)
     }
 }
```

Este es un *recorrido de búsqueda*, y podemos observar varias cosas en él. La primera, más obvia, es que no hay que procesar la celda (más que para visitarla) pues estamos buscando una, y las que no son, no precisan procesamiento, ni tampoco hay que finalizar el recorrido (porque al llegar, ya está lo que había que hacer). La segunda, más sutil, es que el recorrido corta al encontrar la celda buscada; o sea, en rigor, es un recorrido sobre las celdas *previas* a la que buscamos. La última cosa a observar es que el procedimiento `AvanzarASiguienteDelRecorridoDeCeldas` no es trivial, porque las celdas están dispuestas en un cuadriculado; este procedimiento fue presentado cuando hablamos de pintar el tablero recorriendo por celdas.

Definición 4.2.3. *Un* recorrido de búsqueda *es un esquema de recorrido que no recorre la secuencia de elementos en su totalidad, sino que se detiene al encontrar el primer elemento que cumple cierta condición. En este caso, las tareas de* procesar elemento actual *y* finalizar recorrido *no son necesarias, y la tarea de* controlar si es último elemento *se modifica para determinar si el elemento cumple la propiedad buscada.*

G.4.4. Sendero simple del gráfico G.4.3 con rosas en toda su extensión

Si queremos ser precisos, en realidad un recorrido de búsqueda procesa la secuencia de elementos más larga (de una secuencia base) que *no cumple* la propiedad buscada, deteniéndose en el primer elemento que sí cumple. Sin embargo, es más común considerarlo un recorrido sobre toda la secuencia base, con el control de finalización modificado.

Actividad de Programación 19

Modifique el código del procedimiento `BuscarInicioDelSendero` en el ejemplo anterior para que sea como el que acabamos de mostrar (un recorrido de búsqueda), y luego vuelva a probar el procedimiento `PonerFloresEnSenderoSimple` en un tablero que contenga un sendero simple diferente al del ejemplo anterior (y con el inicio en un lugar diferente a la esquina suroeste).

Volveremos a revisar la idea de recorrido una vez que hayamos presentado una de las herramientas más poderosas (y más complejas de utilizar) que tienen los lenguajes de programación: la memoria. Mediante la combinación de recorridos y memoria podremos resolver varios problemas que de otra forma serían extremadamente engorrosos (o directamente imposibles) de realizar. ¡Nuevamente se observará la necesidad de herramientas poderosas en los lenguajes de programación, para lograr descripciones adecuadas a los propósitos buscados!

4.3. Memorización de datos

Es notable, hasta el momento, que cuando el cabezal se mueve por el tablero, no tenemos una forma de "recordar" información, tal como el número de bolitas de las celdas que ya vimos. Esta dificultad hace que ciertas tareas se vuelvan complicadas de resolver, o directamente imposibles. Por ejemplo, para contar la cantidad de enemigos que están cerca, en el ejercicio 3.3.5 utilizábamos como "memoria" la cantidad de bolitas de cierto color en la celda actual; o también, para determinar en qué dirección está el próximo segmento del sendero, debíamos "codificar" la dirección con bolitas verdes en la celda actual. Estas tareas serían mucho más sencillas si el cabezal tuviese *memoria* para recordar valores específicos.

4.3.1. Variables

Para conseguir que el cabezal recuerde cosas mediante una memoria, se puede utilizar la misma idea de dar nombres a expresiones que ya fueron utilizadas en los parámetros de un procedimiento y en el índice de una repetición indexada. En ambos casos usamos una forma limitada de memoria, donde el cabezal recuerda por cierto tiempo un valor dado (por ejemplo un argumento, mediante el nombre del parámetro, mientras dura la ejecución de un procedimiento). Esta misma idea de nombrar valores puede utilizarse de manera independiente y controlada por el progamador. 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. O sea, la forma de proveer *memoria* es mediante la capacidad de nombrar valores y sabiendo qué nombre le dimos, "recordar" el valor; o sea, establecemos una *correspondencia* entre un nombre y un valor dado.

Definición 4.3.1. *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 palabra *variable* sufre de un problema: en distintos contextos tiene distintos significados. Así, un matemático entenderá una cosa por *variable*, mientras que un programador puede entender otra. Por eso debemos ser cuidadosos en respetar las formas de entender un término de cada disciplina, y de cada enfoque distinto dentro de la misma disciplina. En el caso de este libro, el término *variable* se utiliza de una forma **diferente a** la que es usual en los lenguajes de programación más tradicionales donde variable es sinónimo de *posición de memoria*.

Leer con Atención

Las *variables* en GOBSTONES son meramente *nombres* con los que se recuerdan ciertos valores. Esta idea tiene apenas una relación vaga con la noción de *posición de memoria* con la que tradicionalmente se asocia la idea de variable en los lenguajes de programación tradicionales. La noción de variables que manejamos en GOBSTONES tiene mucha más relación con la idea que se tiene en matemáticas, donde las variables son meramente valores que se desconocen en un momento dado.

Leer con Atención

La idea tradicional de variable como *posición de memoria* podría verse como una forma de *implementación* (o sea, concreta, operacional) de la noción de recordar valores (o sea *memorizar*, *tener "memoria"*. La noción de *recordar valores* sería la forma abstracta, denotacional, y por lo tanto más cercana a lo que pensamos los programadores.

Para establecer la correspondencia entre una variable y el valor que la misma denota se utiliza un comando particular, conocido como *asignación*. La asignación de un nombre a un valor determinado se realiza escribiendo el nombre de la variable, luego el signo `:=`, y luego la expresión que describe el valor que la variable debe nombrar. Por ejemplo

```
cantidadRojas := nroBolitas(Rojo)
```

establece la correspondencia entre la variable llamada `cantidadRojas` y el número de bolitas de la celda en la que se encuentra el cabezal al ejecutar este comando. También decimos que "*asigna* el valor actual de `nroBolitas(Rojo)` *a* la variable `cantidadRojas`".

Leer con Atención

Dicho de otra forma, `cantidadRojas` nombra a la cantidad de bolitas de la celda actual *en ese momento de la ejecución*; se transforma, temporalmente (hasta la próxima asignación de dicha variable) en otra forma de describir a ese valor. El valor descrito es el número de bolitas rojas en **una celda determinada en un instante determinado**, y no en la celda actual en otros momentos, ni siquiera el número de bolitas de esa celda en cualquier momento anterior o posterior, donde la cantidad puede haber variado. Esto es importante recordarlo, pues es fuente de mucha confusión. Si luego el cabezal se mueve o la cantidad cambia, `cantidadRojas` seguirá nombrando al valor de aquella celda en aquel momento.

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

Definición 4.3.2. *La* asignación *es un comando mediante el cual se establece la correspondencia entre un nombre y un cierto valor. Su forma es*

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

siendo <variable> un identificador con minúsculas, y <expresion> una expresión cualquiera que describe (o denota) al valor que se quiere asociar (asignar) al nombre de la variable.

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. Debe destacarse que la expresión puede ser cualquiera de las vistas, de cualquier complejidad.

Leer con Atención

Una cuestión importante sobre la forma de hablar de variables es que usaremos como sinónimos "el valor *descrito por* la variable" y "el valor *de* la variable", y también "la variable *describe a un* valor" y "la variable *toma un* valor". Si bien las segundas formas en cada caso no son totalmente correctas desde el punto de vista del castellano, son terminologías ampliamente difundidas y utilizadas. Es importante no tomar literalmente estas formas descuidadas de hablar de variables, pues pueden conducir a confusiones.

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 volver a remarcar que el valor que la variable nombra será el mismo aun 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 CopiarVerdesAlNorte()
 {- 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:
     * debe haber una celda al Norte
    SUPOSICIÓN:
     * la celda al Norte se encuentra vacía
  -}
 {
   cantVerdes := nroBolitas(Verde)
     -- Recuerda el nro de bolitas de la celda inicial
   Mover(Norte)
     -- Se mueve al Norte
   PonerN(cantVerdes, Verde)
     -- Pone en la nueva celda la cantidad recordada
     --  (observar que se supone que esta celda no
     --   tiene bolitas)
   Mover(Sur)
 }
```

La celda inicial es la celda actual cuando inicia el procedimiento.

El valor que se asocia a `cantVerdes` es el número de bolitas que hay en la celda *inicial*, e incluso después de que el cabezal se mueva, la variable seguirá teniendo el valor de la celda inicial, aunque la nueva celda no tenga dicha cantidad. En este ejemplo, suponemos que la celda al `Norte` no tiene bolitas, y que la inicial sí las tiene. Podemos observar el efecto de este procedimiento en el gráfico G.4.5.

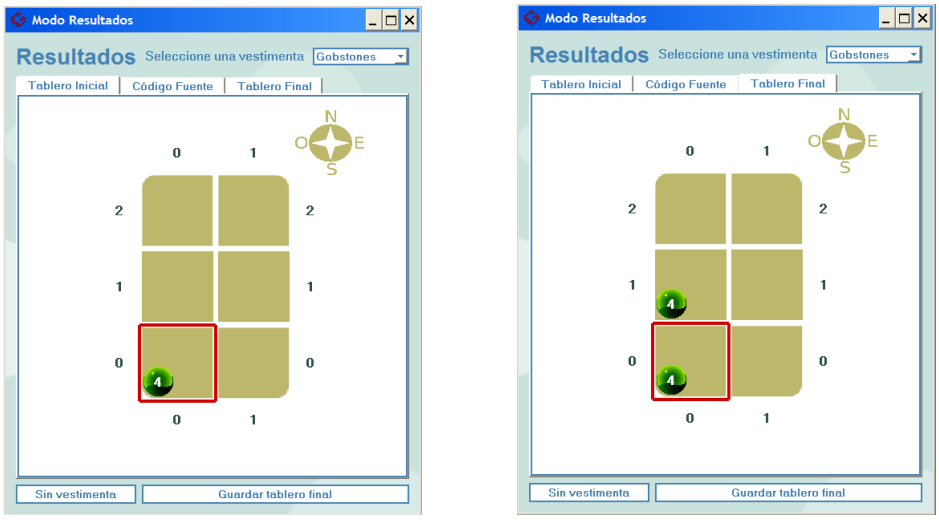
Actividad de Programación 20

Copie el código del procedimiento `CopiarVerdesAlNorte` en un programa, complételo con las definiciones faltantes, y pruébelo con varias configuraciones de tableros diferentes.

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`.

Actividad de Programación 21

Realice el ejercicio 4.3.1 y pruébelo con varias celdas con distinto número de bolitas de diversos colores.



(a). Tablero inicial (b). Tablero final

G.4.5. Prueba del procedimiento `CopiarVerdesAlNorte`

Ejercicio 4.3.1. *Escribir un procedimiento* `CopiarCeldaAl` *que, dada una dirección* `dir`*, y suponiendo que la celda lindante en esa dirección no tiene bolitas, haga que dicha celda termine con el mismo número de bolitas de cada color que la celda inicial. El procedimiento debe realizar solo dos movimientos del cabezal, y debe dejar el cabezal en el lugar de inicio.*

Sugerencia: *considerar el uso de cuatro variables que podrían denominarse respectivamente* `cantRojas`*,* `cantVerdes`*,* `cantAzules` *y* `cantNegras`*.*

Las variables pueden nombrar valores de cualquier tipo. Por ejemplo, supongamos que las bolitas en una celda cuentan votos, y que queremos indicar en la celda de la esquina suroeste el ganador de la votación con una bolita del color correspondiente (suponemos además que en caso de empate, gana el color más chico, o sea, el que esté más cerca del inicio del alfabeto). Podría, entonces, usarse el procedimiento que se presenta a continuación

```
procedure RecuentoDeVotos()
 {- PROPÓSITO:
     * colocar en la celda de la esquina suroeste, una bolita
        del color de las que haya más en la celda actual
  -}
 {
   colorAPoner := colorConMasCantidad()
   IrALaEsquinaS()
   Poner(colorAPoner)
 }
```

donde la función `colorConMasCantidad` retorna el color del que hay más bolitas en la celda actual (y si hay más de uno, retorna el menor de todos); esta función será definida más adelante. Se observa que el valor descrito por la variable `colorAPoner` es un color (pues se usa como argumento del comando `Poner`). También pueden guardarse direcciones y booleanos en variables, además de números y colores.

Leer con Atención

En GOBSTONES las variables deben asignarse siempre con valores de un tipo específico (determinado por la primera asignación que se realice de la variable). Hay lenguajes que permiten que las variables recuerden cualquier cosa, pero otros que solo permiten que una variable guarde elementos de un único tipo. Estas diferencias exceden el alcance de este libro, pero es importante que al aprender nuevos lenguajes se preste atención a esta clase de restricciones.

Ahora bien, ¿qué sucederá si se intenta usar una variable a la que no se le asignó ningún valor? Sencillamente, eso provoca la autodestrucción del cabezal. La forma más común de cometer este error es equivocar el nombre de una variable (por ejemplo, si asignamos `colorAPoner` y luego intentamos usar `coloraPoner` en el ejemplo anterior; observar que en el primer caso la letra `A` es mayúscula, y en el segundo, minúscula). Otra forma común es utilizar una variable fuera de la zona donde la misma tiene sentido (el *alcance* de la variable, noción que se definirá en breve). Finalmente, una forma más compleja y sutil es realizar una asignación en alguna de las ramas de un condicional, y olvidarlo en la otra.

Por ejemplo, supongamos que queremos enviar al cabezal a que encienda una luz roja en la celda de la esquina suroeste (representando la luz con una bolita de ese color) si el cultivo en la celda actual (representado también con distintas bolitas) ha comenzado a ponerse negro, y que encienda una luz verde en aquella esquina, si el cultivo se conserva verde. El código propuesto para esta tarea es

```
procedure AvisarEstadoDelCultivo()
 {-
    PRECONDICIÓN:
      * debe haber bolitas negras o azules
        en la celda actual
  -}
 {
   if (hayBolitas(Negro))
     -- El cultivo se empezó a poner Negro
     { colorAEncender := Rojo }
   else
     { if (hayBolitas(Verde))
         -- El cultivo conserva su color Verde
         { colorAEncender := Verde }
     }
   IrALaEsquina(Sur,Oeste)
   Poner(colorAEncender)
    -- Encender la luz correspondiente
  }
```

Sin embargo, el código del ejemplo contiene un error potencial. En el caso de que el cultivo se haya muerto (o sea no haya bolitas de ningún color), la variable `colorAEncender` no tendrá valor, pues no se habrá ejecutado ninguna de las asignaciones en las ramas de los condicionales. Puesto que la variable no tiene valor, el llamado a `Poner` de la última línea producirá la destrucción del cabezal.

Actividad de Programación 22

Escriba el código de `AvisarEstadoDelCultivo` en un programa, y probarlo con tres celdas diferentes: una que contenga bolitas negras, una que contenga bolitas azules pero no negras, y una que no contenga bolitas ni negras ni azules. Observe lo que sucede en cada caso, y compruebe qué sucede cuando la variable no toma valor.

Una forma de corregir el error es darnos cuenta de que la acción de prender la luz no siempre debe ejecutarse. En caso de que el cultivo no esté ni azul ni negro, no debe encenderse ninguna luz. Entonces podemos indicar la necesidad de prender la luz con una nueva variable, esta vez de valor booleano. Esta variable debe contener verdadero si la luz debe encenderse y falso en caso contrario.

```
procedure AvisarEstadoDelCultivo()
 {-
    PRECONDICIÓN: ninguna, es una operación total
  -}
 {
   if (hayBolitas(Negro))
    -- El cultivo se empezó a poner Negro
     {
       colorAEncender := Rojo
       encenderLuz := True
```

```
        -- Indicamos que hay que encender la luz de color rojo
    }
  else
    { if (hayBolitas(Azul))
       -- El cultivo conserva su color Azul
       {
         colorAEncender := Verde
         encenderLuz := True
           -- Indicamos que hay que encender
           --   la luz de color verde
       }
      else
       { encenderLuz := False }
           -- Indicamos que no hay que encender
           --  la luz
    }

  IrALaEsquina(Sur,Oeste)
  if (encenderLuz) { Poner(colorAEncender) }
    -- Encender la luz correspondiente,
    --  si fue indicado
}
```

Podemos ver que el comando de encender la luz (el `Poner` del final) es ahora condicional, y la condición es el valor de la variable, que habrá cambiado según el color del cultivo. Este código no produce la destrucción del cabezal (o sea, es una operación total).

Otra forma de provocar la autodestrucción mediante el uso de una variable que no fue definida es utilizar dicha 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 4.3.3. *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 solo dentro del procedimiento que las asigna.*

El efecto de que el alcance de las variables sea entre procedimientos es que la única manera de comunicar información entre procedimientos sea a través de los parámetros o de los valores de retorno de las funciones. 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 DuplicarRojasAlNorteMal()
 {- PROPÓSITO:
     * ilustrar el uso INCORRECTO de variables
    PRECONDICIÓN:
     * falla siempre, por invocar un procedimiento que
       no puede tener éxito
    OBSERVACIÓN:
     * pretende mostrar lo que NO DEBE hacerse
  -}
 {
   cantRojas := nroBolitas(Rojo)
   Mover(Norte)
   CompletarDuplicarRojasAlNorteMal()
 }

procedimiento CompletarDuplicarRojasAlNorteMal()
 {- PROPÓSITO:
     * ilustrar el uso INCORRECTO de variables
```

```
    PRECONDICIÓN:
     * falla siempre, pues cant no es una variable definida,
       ni un parámetro
    OBSERVACIÓN:
     * pretende mostrar lo que NO DEBE hacerse
  -}
  {
    PonerN(cantRojas,Rojo)
        -- cantRojas ES UNA VARIABLE SIN ASIGNAR!
        --  (La asignación de cantRojas en el otro
        --   procedimiento no tiene validez en este)
  }
```

La variable `cantRojas` asiganada con el número de bolitas rojas en el procedimiento `DuplicarRojasAlNorteMal`, solo tiene alcance dentro del cuerpo de ese procedimiento (o sea, solo tiene sentido utilizar dicha variable en ese procedimiento, y en ningún lugar más). Por ello, el uso de `cantRojas` en el procedimiento `CompletarDuplicarRojasAlNorteMal` es incorrecto, puesto que no fue asignada ninguna variable con ese nombre. La forma correcta de comunicar ambos procedimientos es utilizar un parámetro, de la siguiente manera:

```
procedimiento DuplicarRojasAlNorteBien()
 {- PROPÓSITO:
     * ilustrar el uso correcto de variables y parámetros
    PRECONDICIÓN:
     * debe haber una celda al norte
  -}
 {
   cantRojas := nroBolitas(Rojo)
   Mover(Norte)
   CompletarDuplicarRojasAlNorteBien(cantRojas)
 }

procedimiento CompletarDuplicarRojasAlNorteBien(cantAPoner)
 {- PROPÓSITO:
     * ilustrar el uso correcto de variables y parámetros
    PRECONDICIÓN:
     * siempre funciona
  -}
 {
   PonerN(cantAPoner,Rojo)
       -- cantAPoner ES UN PARÁMETRO!
 }
```

Se puede observar que `cantRojas` se pasa como argumento al invocar el procedimiento `CompletarDuplicarRojasAlNorteBien`, y este lo utiliza mediante el parámetro `cantAPoner`.

¿Cómo se relacionan las variables con los parámetros y los índices? La respuesta a esta pregunta es que no se mezclan. O sea, si bien es cierto que los parámetros y los índices son identificadores al igual que lo son las variables, **no pueden asignarse** ni parámetros ni índices, puesto que su forma de tomar valores es otra. La asignación es una operación que se realiza solamente con variables. Entonces, no pueden existir, en un procedimiento dado, variables con el mismo nombre que un parámetro o un índice. Si esto sucediese, se producirá un error de sintaxis.

```
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
    foreach 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?
      }
  }
```

Actividad de Programación 23

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 los elementos de manera diferente para producir otro efecto distinto?

Leer con Atención

GOBSTONES separa de manera clara las nociones de variable, parámetro e índice. Esto es así con el objetivo de que resulte clara la distinción entre las 3 ideas. Sin embargo, en otros lenguajes se utilizan los nombres de maneras indistintas, ya sea para denotar parámetros, variables o índices, y puede resulta confuso para quien no tiene las ideas claras.

Utilizando variables hay ciertas operaciones que son más sencillas de expresar, o más fáciles de generalizar. En los próximos ejemplos revisaremos algunas de las operaciones vistas en secciones o capítulos anteriores, para modificarles a través del uso de variables.

El primero de los ejemplos a revisar es la función `hayEnemigosCercaAlNorte` de la subsección 3.3.2. En dicho ejemplo, la función debía devolver un booleano indicando si había enemigos en alguna de las 3 celdas hacia el norte de la actual. Para realizarlo, utilizamos el truco de codificar la existencia de enemigos con bolitas azules en la celda actual y al terminar retornamos si había o no bolitas azules en dicha celda. Pero este ejercicio podría hacerse sin codificar con bolitas, recordando si hay o no enemigos en una variable. El código sería:

```
function hayEnemigosCercaAlNorte()
 {-
    PROPÓSITO:
     * retorna verdadero si hay algún enemigo en las
       próximas 3 celdas al Norte de la actual
    PRECONDICIÓN:
     * hay 3 celdas lindantes al Norte
  -}
 {
   hayEnemigos := False -- Al comienzo no sabemos si hay enemigos
   foreach i in [1..3]
    { if(hayEnemigosAlEnRango(Norte, i))
       { hayEnemigos := True } -- Si ve un enemigo, lo recuerda!
    }
   return(hayEnemigos) -- Se retorna el booleano recordado
 }
```

Podemos observar cómo se hace innecesaria la codificación con bolitas, gracias a que el mecanismo de memoria provisto por las variables nos permite recordar si en algún punto de las celdas controladas encontramos enemigos. También podemos observar en este ejemplo que la asignación recuerda un valor fijo (por ejemplo, en este caso, el `False`) hasta la siguiente asignación, donde recuerda al nuevo valor (en este caso, el `True`).

Actividad de Programación 24

Realice los siguientes ejercicios, modificando adecuadamente los programas que realizó antes. Pruebe si las nuevas versiones funcionan de igual manera.

Ejercicio 4.3.2. *Definir una función* `direccionDelEnemigo` *que cumpla el mismo propósito que el procedimiento* `CodificarDireccionDelEnemigo` *de la subsección 4.2.1 (o sea, indicar la dirección en la que se encuentra el enemigo), pero que en lugar de codificarlo con bolitas, lo retorne como resultado.*

Ayuda: *considerar la utilización de una variable para recordar en qué dirección se encontró al enemigo, y luego retornar el valor de dicha variable.*

Ejercicio 4.3.3. *Reescribir el procedimiento* `MoverASiguienteSector`*, del ejemplo del sendero completado en el ejercicio 4.2.10, para utilizar variables en lugar de codificar las direcciones con bolitas verdes. Para asegurarse que el cambio fue correcto, probar nuevamente el procedimiento* `PonerFloresEnSenderoSimple`*.*

Ayuda: *reemplazar* `CodificarDireccion(dir)` *por una asignación de* `dir` *a la variable* `dirSectorSig`*, y luego reemplazar* `DecodificarDireccion()` *por un comando simple que utilice dicha variable.*

El lenguaje GOBSTONES provee además de la asignación simple una forma de asignación simultánea. Esta forma de asignación se utiliza exclusivamente con funciones que retornan múltiples resultados. El significado es el mismo que el de la asignación simple, con la única salvedad que todas las variables involucradas toman valor al mismo tiempo. Por ejemplo, se puede hacer una función que retorne simultáneamente la división entera y el resto de dividir dos números

(para lo cual simplemente indican en el comando `return`varias expresiones)

```
function divMod(n,m)
 /*
    PROPÓSITO: devolver simultáneamente el resultado de la división
               entera y el resto, al calcular n sobre m.
  */
 { return (n div m, n mod m) }
```

Observamos que en el return aparecen *dos* expresiones en lugar de una. Para poder utilizar esta función se requiere de la asignación simultánea, como se observa en este comando

```
(q,r) := divMod (23, 10)
```

Luego de esta asignación, el valor de la variable `q` será `2`, y el de la variable `r` será `3`.

Un punto de importancia con respecto a la asignación (sea esta simple o simultánea) es que la misma variable puede asignarse varias veces a valores diferentes, dentro del mismo procedimiento (pero NO dentro de la misma asignación simultánea). 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. En la próxima subsección profundizaremos sobre este aspecto, al complementar la herramienta de recorridos con la de variables.

4.3.2. Recorridos más complejos

La idea de recorrido, presentada en la subsección 4.2.3 se ve potenciada al contar con la posibilidad de recordar información mediante el uso de variantes.

Actividad 25

Piense cómo escribiría una función que cuente todas las bolitas de la fila actual.

La solución al problema planteado es utilizar un *recorrido de totalización* o *recorrido de*

acumulación, una idea que podemos definir combinando recorridos y variables. Esta forma de recorrido procesa una secuencia de elementos, acumulando el total de cierta información que cada elemento posee. Por ejemplo, si las celdas de la fila actual representan a los productos de un changuito de supermercado, siendo la cantidad de bolitas azules el precio a pagar por cada producto, y quisiéramos averiguar el precio total que deberemos pagar por la compra, podríamos utilizar el siguiente código, que hace uso de una variable `libreta`, donde llevamos la cuenta:

```
function valorTotalDeLaCompra()
 {-
   PROPÓSITO:
    * retorna el precio total de los productos de un changuito
      de supermercado
   PRECONDICIÓN:
    * el fin del changuito está indicado con una celda sin
       productos
 -}

 {
   -- Iniciar el recorrido
   libreta := 0          -- Preparar algo para llevar la cuenta
   IrAlPrimerProducto()  -- Ubicarse en el primer elemento

   while (not finChango())  -- No se acabaron los productos
     {
       libreta := libreta + precioProducto()
                            -- Incorporar el precio actual a la
                            --  cuenta
       PasarAlSiguienteProducto()
                            -- Pasar al siguiente
     }

   -- No hace falta procesar la última celda,
   --   por la precondición

   -- Finalizar el recorrido, informando el precio total
   return (libreta)
 }
```

La línea más interesante en este código es la de la asignación dentro de la repetición condicional. Puede considerarse como la idea de leer lo que tenemos anotado en nuestra `libreta`, sumarle a eso el precio del producto que estamos poniendo en el chango, y luego anotar el resultado nuevamente en la `libreta`. Esta asignación funciona *incrementando* el valor de la `libreta` con el precio del producto actual, y entonces se la suele llamar un *incremento* de la variable. El inicio de recorrido debe asegurarse que la variable exista, y como hasta el momento no se han visto productos, debe anotarse un 0 en la misma. Al terminar el recorrido debe retornarse el valor anotado en la libreta, que será el del total de la suma de todos los productos en el chango. La operación `IrAlPrimerProducto` debe ir al borde Oeste de la fila, la operación `finChango` debe verificar que el cabezal se encuentra en la fila de más al Este, y la operación `PasarAlSiguienteProducto` debe moverse al Este.

Definición 4.3.4. *Un* recorrido de totalización *es un esquema de recorrido que utiliza variables para acumular el total de cierta información que posee cada elemento de la secuencia. En esta variante de recorrido, la subtarea de* iniciar el recorrido *incluye la inicialización de la variable de totalización en 0, la subtarea de* procesar elemento *incluye la acumulación de la información del elemento en dicha variable, y la subtarea de* finalizar el recorrido *incluye la utilización de la variable que contiene el total acumulado.*

Es usual que los recorridos de totalización se utilicen en funciones, puesto que de esta forma la cantidad total se retorna como resultado de la función.

(a). Tablero inicial

(b). Tablero final

G.4.6. Prueba de la función `valorTotalDeLaCompra`

Actividad de Programación 26

Pase el código de la función `valorTotalDeLaCompra`, complete las operaciones faltantes, y pruebe el código, colocando tantas bolitas negras como el total de la compra en la celda de la esquina suroeste. Podría quedarle un tablero como el del gráfico G.4.6, donde se ven 7 productos (uno de los cuales es gratuito).

La idea de recorrido de totalización se puede utilizar para cualquier tipo de elementos, y para cualquier recorrido. Lo único que precisa es inicializar una variable con 0, y luego, al procesar cada elemento, agregar el valor de cada uno al total, incrementando la variable en cuestión.

Actividad de Programación 27

Realice los ejercicios 4.3.5 y 4.3.4, teniendo en cuenta en ambos casos la idea de recorrido de totalización. Pruebe los programas resultantes.

Ejercicio 4.3.4. *Definir una función* `distanciaALaBase` *que cuente y retorne el número de celdas que hay entre la celda actual y la base de la columna actual. Pensar en estructurarlo como un recorrido de totalización, donde cada celda tiene valor 1. Verificar que la cantidad retornada sea exactamente la pedida, pues es común retornar uno más o uno menos que el valor solicitado.*

Este error se denomina *off-by-one* (en castellano se podría traducir como "errado-en-uno"), y proviene del hecho de que el cabezal se mueve menos veces de las que debe contar.

Ejercicio 4.3.5. *Revisar la definición de* `TotalizarFrecuenciaActual` *del ejercicio 3.4.2, para rehacerla como un recorrido de totalización sobre el valor de cada led.*

Actividad de Programación 28

Realice el ejercicio 4.3.6 y coloque el código resultante en la `Biblioteca` para poder probarlo desde varios programas.

Ejercicio 4.3.6. *Definir una función* `medirDistanciaAlBorde` *que tome una dirección y cuente y retorne el número de celdas que hay entre la celda actual y el borde del tablero en la dirección dada. Para guiarse, considerar modificar la función del ejercicio 4.3.4 de manera adecuada.*

Una variante del recorrido de totalización es la del recorrido que permite calcular el máximo o el mínimo de cierta cantidad. Para ello, se recuerda el máximo visto hasta el momento en una variable, y al procesar cada elemento, se verifica si es mayor que el máximo guardado, y en caso de serlo, se lo reemplaza.

Por ejemplo, para verificar de cuál color hay más bolitas en la celda actual, se puede hacer un recorrido de cálculo de máximo sobre colores.

```
function colorConMasCantidad()
 {-
    PROPÓSITO:
      * retorna el color del que hay más bolitas.
        Si hay igual cantidad de más de un color,
        retorna el menor color posible
   PRECONDICIÓN:
      * ninguna, es una operación total
  -}
 {
   -- Inicializar el recorrido, recordando el
   --  número de bolitas del color mínimo
   maxColor := minColor()
   maxCant := nroBolitas(minColor())

   -- Al recorrerse, puede usarse una repetición indexada
   --  si se conocen todos los elementos a recorrer!
   foreach colorActual in [siguiente(minColor())..maxColor()]
     {
       -- Procesar viendo si el actual es mayor
       if (nroBolitas(colorActual) > maxCant)
         {
           maxColor := colorActual
           maxCant  := nroBolitas(colorActual)
         }
       -- Pasar al siguiente está implícito en el repeat!
     }

   -- Finalizar el recorrido, informando
   return (maxColor)
 }
```

Podemos observar varias cosas. La primera es que si conocemos todos los elementos a recorrer, puede utilizarse una repetición indexada en lugar de una condicional. La segunda es que procesamos por separado el primer color, pues es necesario tener un valor inicial para el máximo posible. La tercera, y más importante para nuestro ejemplo de recorrido de cálculo de máximo, es que son necesarias 2 variables: una para recordar el máximo número hasta el momento, y otra para recordar a qué color correspondía dicho número.

Definición 4.3.5. *Un* recorrido de cálculo del máximo *(o mínimo) es un esquema de recorrido en el que se utiliza una variable para recordar el elemento de la secuencia que tiene el valor más grande (o más chico) para cierta información entre todos los elementos de la secuencia. En esta variante, la subtarea de* iniciar el recorrido *incluye la consideración del valor del primer elemento recordándolo como el más grande (más chico) visto hasta el momento, la de* procesar elemento actual *incluye la comparación entre el valor del elemento actual y el máximo (mínimo) visto hasta el momento, y si corresponde, la modificación del elemento recordado como máximo (mínimo), y la de* finalizar el recorrido *incluye la utilización del valor máximo (mínimo) obtenido.*

Puede suceder que no nos interese el valor máximo o mínimo obtenido, sino el *elemento* que posee ese valor. En ese caso, deberán usarse variables adicionales para recordar

cuál fue el elemento que contribuyó a maximizar o minimizar la cantidad, y la modificación acorde de las subtareas correspondientes.

Por ejemplo, si queremos saber cuál es el máximo número de bolitas rojas de una celda en la columna actual, usaríamos un recorrido sobre la columna de la siguiente manera:

```
function maxBolitasEnColumna()
 {-
    PROPÓSITO:
      * calcular el máximo número de bolitas rojas en
        una celda en la columna actual
    PRECONDICIÓN:
      * ninguna, es una operación total
  -}
 {
   -- Iniciar el recorrido
   IrAlExtremo(Norte)
   maxCantidad := nroBolitas(Rojo)

   -- Falta procesar alguna celda?
   while (puedeMover(Sur))
     {
       Mover(Sur)
       if (nroBolitas(Rojo) > maxCantidad)
         {
           -- Si ahora hay más, se reemplaza
           --  el máximo recordado
           maxCantidad := nroBolitas(Rojo)
         }
     }

   -- Procesamos la última celda por separado,
   --  al finalizar el recorrido
   if (nroBolitas(Rojo) > maxCantidad)
     {
       -- Si ahora hay más, se reemplaza
       --  el máximo recordado
       maxCantidad := nroBolitas(Rojo)
     }

   -- Finalizar el recorrido
   return (maxCantidad) -- Se informa al terminar
 }
```

Aquí podemos observar que como los elementos a recorrer son celdas, entonces debemos usar una repetición condicional, y puesto que la pregunta para seguir es si se puede mover, el último elemento debe procesarse por separado. Sin embargo, el procesamiento del elemento actual sigue el mismo principio que en el recorrido anterior, para calcular el máximo.

Pero si en lugar de querer saber la cantidad, quisiéramos saber de qué celda se trata (por ejemplo, para marcarla con bolitas de color `Azul`), deberíamos recordar la cantidad de veces que nos movemos y usar ese número al terminar el recorrido.

```
procedure MarcarCeldaConMaxBolitasEnColumna()
 {-
    PROPÓSITO:
      * marcar con bolitas azules la celda con el máximo
        número de bolitas rojas en la columna actual
    PRECONDICIÓN:
      * ninguna, es una operación total
    SUPOSICIÓN:
      * las celdas de la columna actual no contienen
        bolitas azules
  -}
 {
```

```
-- Iniciar el recorrido
IrAlBorde(Norte)
maxCantidad := nroBolitas(Rojo)
movimientosAlMax := 0
cantMovimientos := 0

-- Falta procesar alguna celda?
while (puedeMover(Sur))
 {
   -- Se mueve y cuenta el movimiento
   Mover(Sur)
   cantMovimientos := cantMovimientos + 1
    if (nroBolitas(Rojo) > maxCantidad)
      {
        -- Si ahora hay más, se reemplaza
        --  el máximo recordado
        maxCantidad := nroBolitas(Rojo)
        movimientosAlMax := cantMovimientos
      }
 }

-- Procesamos la última celda por separado,
--   al finalizar el recorrido
if (nroBolitas(Rojo) > maxCantidad)
  {
    -- Si ahora hay más, se reemplaza
    --   el máximo recordado
    maxCantidad := nroBolitas(Rojo)
    movimientosAlMax := cantMovimientos
  }

-- Para finalizar el recorrido, marcamos la
--  celda que corresponde
IrAlExtremo(Norte)
MoverN(Sur, movimientosAlMax)
Poner(Azul)
}
```

En este código podemos observar cómo se utilizan 3 variables para recordar, por un lado, la máxima cantidad de bolitas vistas hasta el momento (variable `maxCantidad`), por otra parte, la cantidad de movimientos necesarios hasta donde fue vista dicha cantidad (variable `movimientosAlMax`, que es similar a recordar qué color es el que tenía más bolitas en el primer ejemplo, pues se trata de un dato asociado al máximo buscado), y finalmente, la cantidad de movimientos hechos hasta el momento (variable `cantMovimientos`, necesaria para saber cuánto nos movimos y poder recordarlo si es el máximo). Al finalizar el recorrido, en lugar de informar la cantidad de movimientos, se realiza el marcado de la celda correspondiente, con base en la cantidad calculada. Se debe observar que a diferencia de un recorrido de búsqueda, que finaliza apenas encuentra lo buscado, en este caso deben procesarse todos los elementos, pues no se sabe de antemano cuál es el máximo.

Un recorrido para calcular el mínimo elemento de una secuencia es exactamente igual en estructura a uno de calcular el máximo, solo que en lugar de recordar el mayor, se recuerda el menor. Tener esto en cuenta al realizar los siguientes ejercicios.

Actividad de Programación 29

Realice los ejercicios 4.3.7 y 4.3.8, y pruébelos adecuadamente.

Ejercicio 4.3.7. *Definir un procedimiento* `IndicarPuntoDebil` *que marque con una bengala (representada con una bolita azul) la celda en la que el enemigo es más débil (haya menor cantidad de enemigos, representados con bolitas rojas).*

Sugerencia: *realizar un recorrido por celdas (similar al visto en la forma mejorada de* `BuscarInicioSendero`*), y durante este recorrido, recordar por separado los movimientos al este y los movimientos al norte a hacer desde la esquina suroeste para llegar a dicha celda (de manera similar a como se mostró en* `MarcarCeldaConMaxBolitasEnColumna`*).*

Ejercicio 4.3.8. *Utilizando la representación del display de ecualizador del ejercicio 3.4.2, realizar un procedimiento* `PosicionarseEnFrecuenciaMaxima` *que ubique al cabezal en la base de la columna que tenga la frecuencia con mayor intensidad.*

Otra forma de recorrido donde la capacidad de recordar valores es fundamental es cuando se debe recorrer una secuencia pero no se deben procesar todos los elementos. Por ejemplo, imaginemos que queremos colocar una progresión de bolitas verde en `n` celdas, tal que la cantidad de bolitas en cada celda sean solo los múltiplos de 2 o de 3 (o sea, 2, 3, 4, 6, 8, 9, 10, 12, etc.). Entonces, podría usarse un recorrido con memoria de la siguiente manera

```
procedure ProgresionVerdeDosOTres(n)
 {- PROPÓSITO:
     * armar una progresión de n celdas con cantidad de
       bolitas que sea múltiplo de 2 o de 3
    PRECONDICIÓN:
     * que haya n celdas al Este
    OBSERVACIÓN:
     * se estructura como un recorrido sobre los
       números naturales mayores que 2 necesarios,
       los cuales se recuerdan con la variable
       cantAColocar
  -}
 {
   -- Iniciar el recorrido
   procesadas := 0      -- la cantidad de celdas ya tratadas
   cantAColocar := 2    -- el inicio de la progresión

   -- Mientras no se haya terminado el recorrido
   while (procesadas < n)
     {
       -- Procesar el elemento actual
       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.
        }

       -- Pasar al siguiente número a considerar
       cantAColocar := cantAColocar + 1 -- probar un nuevo nro.
     }

   -- Finalizar el recorrido, volviendo al punto de partida
   MoverN(Oeste, n)
 }
```

Puede observarse cómo la pregunta para terminar de procesar responde a la cantidad de celdas procesadas (que en este caso debe ser el parámetro `n`), mientras que el pasar a un nuevo elemento consiste en probar un nuevo número. El procesamiento de elementos consiste en verificar si el número actual corresponde a una cantidad a poner, y en ese caso, ponerlo en la celda correspondiente. Variando el tratamiento de estos dos parámetros se pueden lograr diferentes combinaciones.

Actividad de Programación 30

Realice el ejercicio 4.3.9, y verifíquelo en el sendero del gráfico G.4.3. En el gráfico G.4.7 pueden verse un par de invocaciones del procedimiento, con parámetros diferentes.

Ejercicio 4.3.9. *Modificar el procedimiento* `PonerFloresEnSenderoSimple` *presentado en la subsección 4.2.3, para que tome un parámetro* `distancia`*, y coloque flores en el sendero separadas por la* `distancia` *dada.*

Sugerencia: *agregar una variable* `proximoSectorEn` *que cuente cuánto falta para el próximo sector donde poner flores. Al procesar, si esta variables está en 1, poner la flor y volver a poner la variable en la distancia máxima, y si no, decrementar la variable.*

Ayuda: *una complicación que tiene este ejercicio es que el control de cuál era el siguiente sector a mover asumía que en el anterior ya se había colocado una flor. Puesto que en esta variante no se coloca una flor, debe marcarse un sector visitado (por ejemplo, reemplazando las bolitas negras por bolitas verdes), para indicar que efectivamente ya fue visitado. Al finalizar el recorrido habría que hacer una subtarea que vuelva a recorrer el sendero cambiando bolitas verdes por negras.*

Actividad de Programación 31

Realice el ejercicio 4.3.10, y pruébelo con el sendero del gráfico G.4.3. El resultado debería quedar como el gráfico G.4.8.

Ejercicio 4.3.10. *Modificar el procedimiento* `PonerFloresEnSenderoSimple` *de la subsección 4.2.3 para que "numere" los sectores, colocando tantas flores en un sector como la distancia de ese sector al inicio del sendero. O sea, en el inicio debe colocar 1 flor, en el primer sector, 2 flores, en el tercero, 3 flores, etcétera.*

Actividad de Programación 32

Realice el ejercicio 4.3.11, y pruébelo en varias de las frecuencias del ecualizador del gráfico G.3.11a.

Ejercicio 4.3.11. *Escribir un procedimiento* `IncrementarFrecuenciaActual`*, que, teniendo en cuenta la representación del ejercicio 3.4.2, incremente en uno la frecuencia representada en la columna actual. Puede suponerse que la celda actual se encuentra en la base de la columna. Tener en cuenta que si la frecuencia está al máximo, no se incrementa, y que al incrementar, debe respetarse el color de cada led.*

Ayuda: *estructurarlo como un recorrido sobre los 6 leds de la frecuencia, incrementando una variable* `intensidad` *cada vez que se encuentra un led prendido. Al finalizar recorrido debe posicionarse sobre el led siguiente a la intensidad (si esta es menor que 6), y encenderlo (ya sea verde o rojo, según la posición).*

4.4. Ejercitación

Este apartado culmina la introducción al lenguaje GOBSTONES, con algunos ejercicios avanzados que utilizan todas las herramientas vistas hasta el momento.

Actividad de Programación 33

Realice los ejercicios de este apartado, y pruébelos en cada caso, sobre los tableros adecuados.

Continuando con el juego de *Procedrez* que se describió en la sección 3.4, completamos los ejercicios que requerían herramientas avanzadas.

(a). Resultado de invocar `PonerFloresEnSenderoSimpleCada(4)`

(b). Resultado de invocar `PonerFloresEnSenderoSimpleCada(5)`

G.4.7. Sendero simple con flores colocadas cada algunos sectores

G.4.8. Sendero simple “numerado”

El procedimiento del ejercicio que se presenta a continuación sigue la idea del ejercicio 3.4.4, y fue utilizado en el ejercicio 3.4.5.

Ejercicio 4.4.1. *Implementar el procedimiento* `MarcarMovimientosTorre`*, que recibe un color* `col`*, y marca con una bolita de ese color cada uno de los potenciales movimientos de la torre ubicada en la casilla actual. La precondición del procedimiento es que debe haber una torre en la casilla actual.*

Igual que en el caso del visir, deben marcarse las casillas ocupadas por piezas del oponente, pero no las casillas ocupadas por piezas del mismo color. No se deben marcar casillas “saltando” por sobre piezas propias ni del oponente.

El cabezal debe quedar ubicado en la casilla en la que estaba originalmente.

Se muestra un ejemplo para una torre negra en el gráfico G.4.9a.

Ejercicio 4.4.2. *Implementar el procedimiento* `MarcarMovimientosJugador` *que recibe un color denominado* `jugador`*, y marca con una bolita de color* `Verde` *cada uno de los potenciales movimientos de ese jugador. La precondición es que* `jugador` *debe ser* `Rojo` *o* `Negro`*.*

Por ejemplo, si el jugador tiene un rey y una torre, marca con bolitas de color `Verde` *todos los potenciales movimientos del rey, y todos los potenciales movimientos de la torre. Una casilla puede quedar marcada con varias bolitas, si hay varias piezas que potencialmente puedan moverse allí.*

En el gráfico G.4.9b se muestran todos los movimientos marcados del jugador negro, que tiene un rey y dos torres. Observar que varias casillas reciben más de una amenaza.

Ejercicio 4.4.3. *Implementar la función* `esMovimientoPosible`*, que devuelve un booleano que indica si la pieza ubicada en la casilla actual podría moverse a la casilla marcada como destino, indicada con una bolita de color* `Azul`*.*

La precondición de la función es que debe haber una pieza en la casilla actual, que no debe haber casillas del tablero marcadas con bolitas de color `Verde` *y que exactamente una casilla del tablero debe estar marcada como destino (en todo el tablero debe haber exactamente una bolita* `Azul`*).*

Sugerencia: *pensar en cómo reutilizar los procedimientos definidos en los ejercicios anteriores.*

(a). Movimientos de la torre.

(b). Movimientos del jugador negro.

G.4.9. Movimientos de piezas del Procedrez

Leer con Atención

De aquí en más asumiremos siempre que el tablero es un *juego válido* de Procedrez, o sea, que siempre hay exactamente un rey de cada color en el tablero, que puede haber a lo sumo una casilla marcada como destino (con una bolita `Azul`) y que no puede haber casillas marcadas con bolitas de color `Verde`.

Ejercicio 4.4.4. *Definir la función* `reyEnJaque` *que toma un parámetro* `jugadorDefensor` *que indica el color del jugador defensor (*`Rojo` *o* `Negro`*), y retorna si el rey del jugador defensor está en jaque. Es decir, devuelve* `True` *si alguna pieza del jugador oponente podría moverse a la casilla donde se encuentra el rey del jugador defensor. Devuelve* `False` *en caso contrario.*

Ayuda: *realizar un recorrido sobre las piezas del oponente, marcando al rey defensor como destino.*

Ejercicio 4.4.5. *Definir el procedimiento* `MoverPieza` *que mueve la pieza ubicada en la casilla actual a la casilla marcada como destino, sacando la bolita* `Azul` *del destino. Si en dicha casilla hay una pieza del oponente, la captura.*

La precondición es que debe haber una pieza en la casilla actual, y exactamente una casilla del tablero marcada como destino. Además, la casilla marcada como destino debe ser un movimiento válido para la pieza que se está moviendo. Finalmente, la casilla marcada como destino debe estar vacía (sin piezas propias) o con una pieza del oponente.

En el gráfico G.4.10 se pueden ver dos momentos del movimiento de un visir negro.

Ejercicio 4.4.6. *Definir la función* `puedeMoverPiezaAlDestino`*, que indica si la pieza ubicada en la casilla actual puede moverse a la casilla marcada como destino* sin que su rey quede en jaque*. O sea, devuelve* `True` *si la condición se cumple, y* `False` *en caso contrario.*

La precondición es que debe haber una pieza en la casilla actual. Además, debe haber exactamente una casilla del tablero marcada como destino.

Ejercicio 4.4.7. *Definir la función* `puedeMoverAlgunaPiezaAlDestino` *que toma un parámetro* `jugadorDefensor` *(*`Rojo` *o* `Negro`*) y retorna si el jugador defensor dispone de* alguna *pieza que pueda moverse al destino sin que su rey quede en jaque. En caso contrario, devuelve* `False`*.*

La precondición es que debe haber exactamente una casilla del tablero marcada como destino.

El último tema para presentar ejercicios se basa en una idea de Pablo Barenbaum. Nuevamente agradecemos a Pablo por su generosidad.

La Piedra Rosetta es una losa del Antiguo Egipto que contiene inscripciones del mismo texto en tres sistemas de escritura distintos: griego, egipcio demótico y jeroglíficos egipcios. En el siglo XIX, Jean-François Champollion se valió de esta información para descifrar la escritura jeroglífica. Para auxiliarnos en el análisis de lenguas muertas, haremos una versión simplificada de este modelo.

El tablero de GOBSTONES se dividirá en dos mitades, separadas por una columna marcada con bolitas negras en todas sus celdas. Las columnas de la mitad izquierda representarán un texto de referencia. Las columnas de la mitad derecha representarán un texto a descifrar. Notar que las mitades no necesariamente tienen el mismo ancho.

Dentro de cada mitad, las bolitas azules representarán símbolos de un sistema de escritura desconocido. Las bolitas rojas representarán símbolos de un sistema de escritura conocido. Por ejemplo, nueve bolitas azules podrían representar un jeroglífico con forma de lechuza, y dos bolitas rojas la letra ”B”.

En cada posición del texto de referencia habrá un símbolo desconocido, que representa el texto original, y uno conocido, que representa su traducción. En cambio, cada posición del texto a descifrar tendrá solamente un símbolo desconocido, que es lo que se quiere traducir.

Supondremos, de forma poco realista, que las traducciones se realizan símbolo a símbolo, y que un símbolo se traduce siempre de la misma manera. Asumiremos también que todos los símbolos del texto a traducir ya figuran en el texto de referencia. En ambas partes del texto puede haber espacios, que no se traducen ni afectan la traducción.

(a). El visir negro está a punto de moverse a la casilla marcada como destino.

(b). El visir negro se ha movido, capturando a la torre roja y dando así jaque al rey rojo.

G.4.10. Visir negro moviéndose

Ejercicio 4.4.8. *Escribir el procedimiento* `DescifrarJeroglificos` *que complete la traducción del texto a descifrar.*

Este ejercicio puede realizarse de tres maneras diferentes, cada una con su propia complejidad.

1. Recorrer primero el texto de referencia, recordando la traducción de cada símbolo (digamos, armando un diccionario), y después recorrer el texto a traducir, usando el diccionario ya armado.

   Esta solución es conceptualmente correcta, pero las herramientas que permitirían estructurar la solución de esta manera escapan completamente al alcance de este curso (se mencionan algunas en el capítulo 6. En GOBSTONES, con las herramientas vistas, esto no es factible, porque solo se dispone de un número *fijo* de variables que recuerdan un único valor, y no se pueden usar para "recordar" un número desconocido de símbolos del alfabeto.

2. Recorrer el texto de referencia y, a medida que se encuentra un par clave/valor, traducir todas las ocurrencias de ese símbolo.

   Esta solución tiene la dificultad de que una vez visitado el texto a traducir, hay que volver a la posición del texto de referencia de donde había partido el cabezal. Esto puede hacerse dejando marcas sobre el tablero. Sin embargo, esta solución es compleja.

3. Esta posibilidad es la más simple de programar (y quizás la más "natural" en algún sentido). La idea consiste en hacer una función `traducción` que dado un símbolo representado como un parámetro de tipo número que sería el código del "jeroglífico" correspondiente, busque en el texto de referencia su traducción, devolviendo el código de la "letra conocida" correspondiente.

   Hecho esto, el problema se puede resolver haciendo un único recorrido sobre el texto a traducir, traduciendo símbolo a símbolo mediante la función recién definida. La ventaja es que no hay que preocuparse por dejar marcas para volver, aprovechando que las funciones deshacen todos sus efectos.

La solución más simple del problema de la Piedra Rosetta muestra el poder que tiene la división de un problema complejo en subtareas más sencillas, siempre que la descomposición se haga pensando en simplificar el proceso completo. Este, más que ningún otro concepto, es el que deben llevarse de esta introducción a la programación.

5

Un ejemplo completo: ZILFOST

En este capítulo mostramos cómo los conceptos presentados en los capítulos anteriores pueden utilizarse para programar un juego sencillo, para el que explicamos completamente su diseño y su codificación en GOBSTONES. El juego es una variante del clásico juego de TETRIS™, y se denomina ZILFOST como regla mnemotécnica para recordar las formas de las diferentes piezas (ver gráfico G.5.1).

Además presentamos una pequeña extensión a GOBSTONES que permite una forma básica de interacción con el usuario, lo que habilita la posibilidad de usar la herramienta que implementa GOBSTONES para probar el juego, y por supuesto, para permitir la codificación de otros juegos similares (basados en turnos). Esta forma de interacción está pensada solamente como recurso para mostrar el poder de las ideas presentadas en el libro, y no como una herramienta real de programar interfaces de entrada/salida para aplicaciones reales. Sin embargo, este mecanismo rudimentario, cuando se combina con otro par de detalles de presentación (como la posibilidad de aplicar *"skins"* o *vestimentas* a las celdas del tablero), puede mostrar de manera certera las posibilidades de los conceptos tratados en el libro.

Tetris es un videojuego de ubicación de piezas diseñado y programado por Alexey Pajitnov en la Unión Soviética en 1984. El nombre deriva del prefijo griego *tetra* (pues todas las piezas se componen de cuatro secciones) y del deporte del tenis, deporte favorito del autor.

https://en.wikipedia.org/wiki/Tetris

Para probar el juego, puede usarse el sitio oficial

http://www.tetris.com/

o el sitio de la versión libre

http://www.freetetris.org/

5.1. Representación del ZILFOST

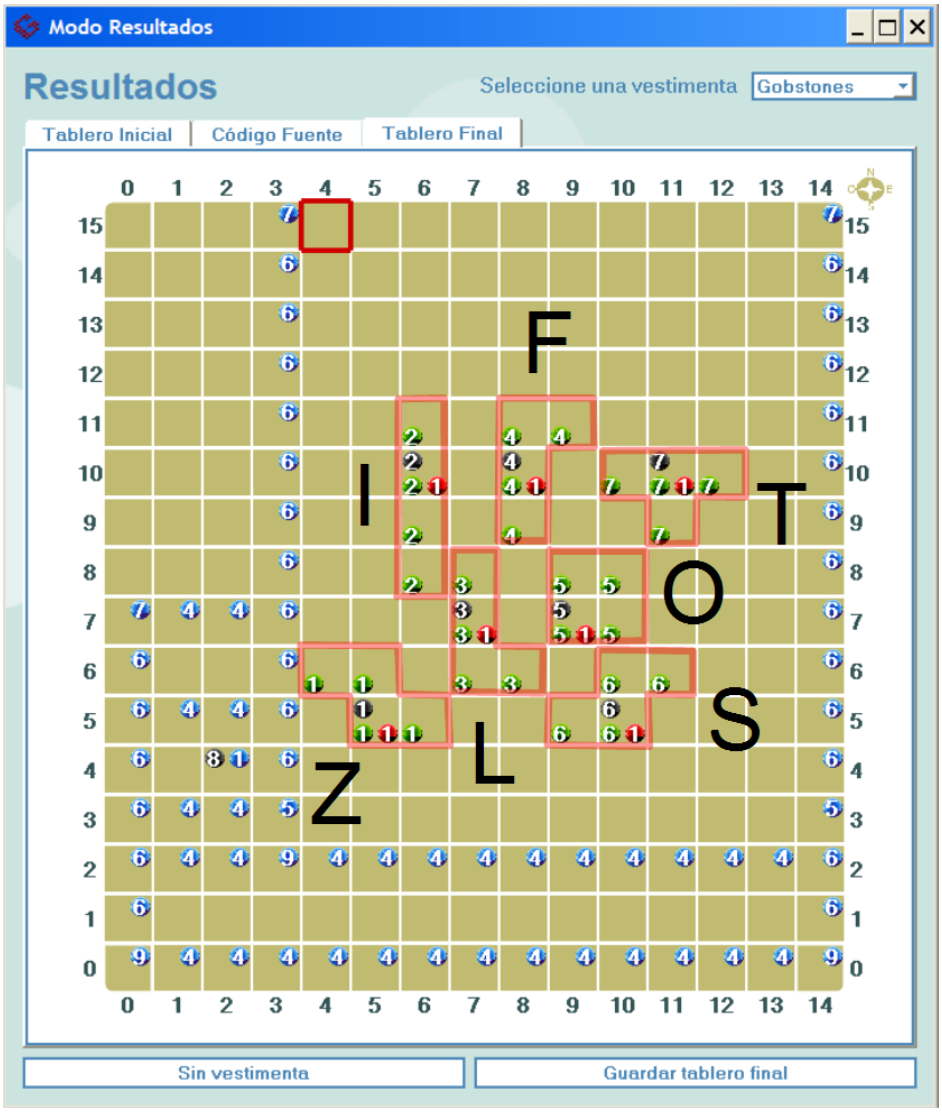
El juego de ZILFOST está basado en el popular juego TETRIS™. En este juego hay un grupo de piezas de varios formatos básicos que "caen" en una zona de juego rectangular. Decimos que las piezas "caen", porque pensamos a la zona de juego como un espacio vertical y a las piezas como objetos físicos, sujetos a las leyes de la gravedad. De esta manera, las piezas aparecen por arriba y van moviéndose hacia la base a medida que transcurre el tiempo.

La idea de cómo codificar este juego en GOBSTONES surgió de un parcial de la materia Introducción a la Programación de la carrera Tecnicatura Universitaria en Programación Informática de la Universidad Nacional de Quilmes, donde se utiliza GOBSTONES como primer lenguaje de programación. En dicho parcial solo se consideraba un tipo único de piezas, y se generaba la mecánica básica del juego. Posteriormente el autor extendió esta idea con las restantes piezas, y completó la mecánica del juego y los restantes elementos del mismo.

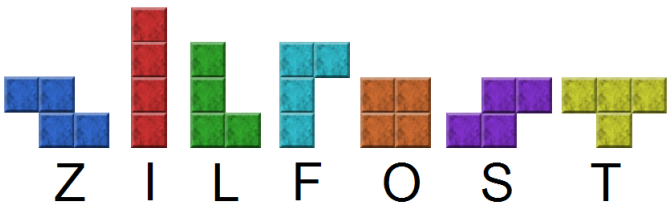
El nombre ZILFOST hace referencia a la regla mnemotécnica utiliza para recordar los formatos de las piezas básicas. Las mismas son piezas compuestas de 4 secciones cuadradas, dispuestas de manera contigua de diferentes formas. Así, a las piezas las denominamos Z, I, L, F, O, S y T, por la forma que rememora la disposición de sus secciones (ver gráfico G.5.1).

El tablero de GOBSTONES se utilizará para representar las diferentes partes del juego. Las piezas se representarán en 4 celdas contiguas utilizando bolitas verdes; la cantidad de bolitas indicará el número de pieza que se trate. Además las piezas tendrán una sección especial, denominada *pivote*, que contendrá información adicional (codificada con bolitas negras y rojas); adicionalmente, la rotación de las piezas será respecto del pivote.

El tablero codificará las diferentes partes del juego de ZILFOST; habrá una zona principal, la *zona de juego*, donde se representarán las piezas que van cayendo, y dos zonas especiales: la *zona de semilla* y la *zona de datos*. La zona de semilla se utiliza para almacenar la semilla de un generador de números seudoaleatorios, y podrá ser utilizada en

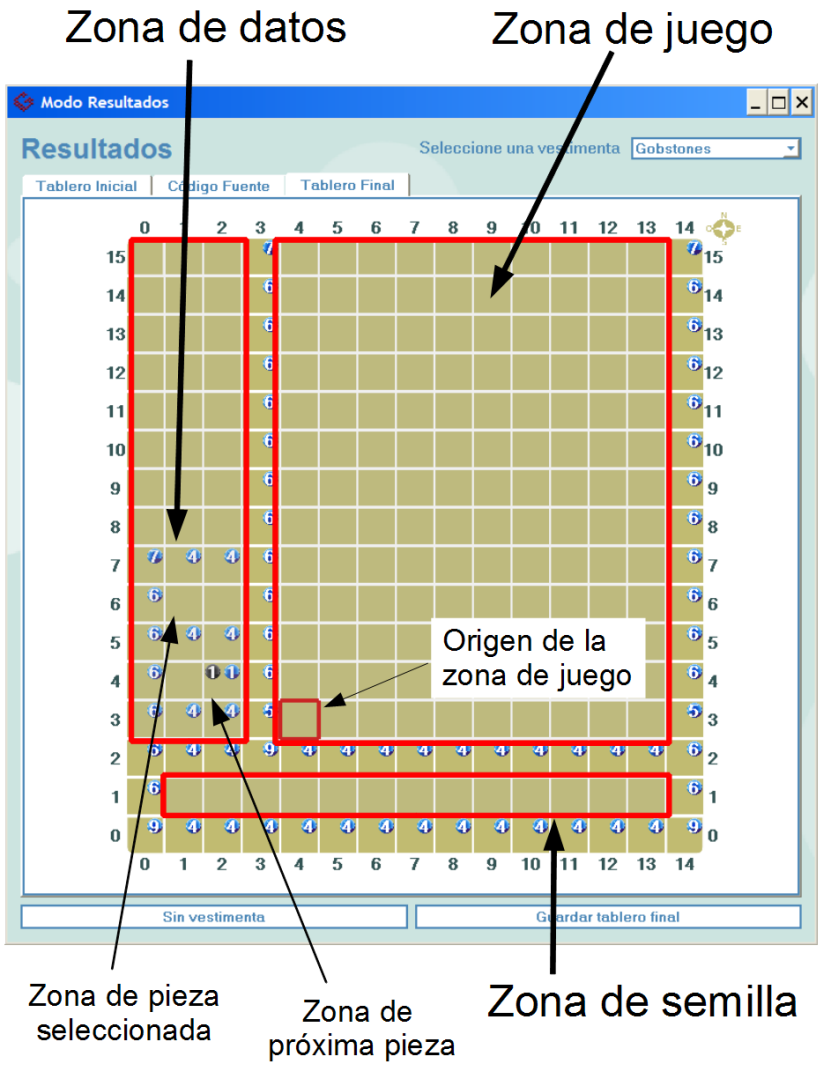


(a). Las piezas de ZILFOST sobre el tablero



(b). Las piezas de ZILFOST desplegadas para formar el nombre

G.5.1. Las piezas de ZILFOST, donde puede apreciarse cómo el nombre es inspirado por la forma de las piezas



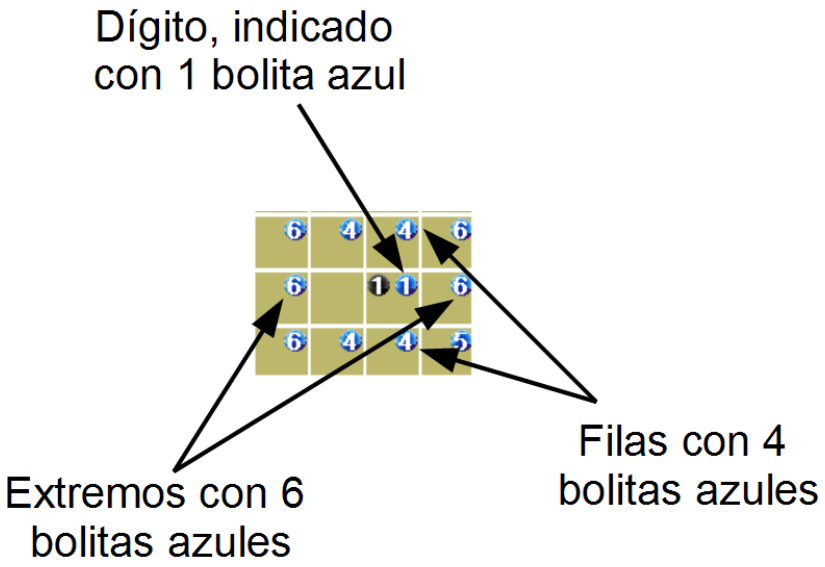
G.5.2. Las diferentes zonas de un tablero de ZILFOST

el tablero inicial para variar la secuencia de piezas del juego y otras particularidades. La zona de datos se utilizará para representar el número de la próxima pieza a ser jugada y la pieza seleccionada actualmente; por el momento además posee un espacio no utilizado (que podría usarse, por ejemplo, para representar la próxima pieza que aparecerá, u otros elementos). En el gráfico G.5.2 se pueden ver un tablero inicial de ZILFOST, con las diferentes zonas indicadas específicamente.

Para codificar un juego de ZILFOST se utilizarán las siguientes convenciones.

- La *zona de juego* está delimitada en sus extremos inferior izquierdo e inferior derecho con dos celdas con exactamente 5 bolitas azules, de manera que la de la izquierda es la primer celda en recorrido noreste que tiene exactamente esa cantidad de bolitas. Además desde esas dos celdas, hacia arriba, hay dos columnas de bolitas azules con 6 bolitas en cada celda del cuerpo de la columna y 7 bolitas en la celda de tope; este tope se encuentra en el borde Norte (ver gráfico G.5.2). La primer celda vacía a la derecha de la celda inferior izquierda de la zona de juego (con 5 bolitas azules) es el *origen de la zona de juego*.
- A la izquierda de la zona de juego se encuentra la *zona de datos*, que tiene la misma altura que la zona de juego y el ancho necesario para abarcar hasta borde oeste del tablero. La zona de datos contendrá en su parte inferior dos zonas de números (el código de la próxima pieza a ingresar a la zona de juego y el código de la pieza seleccionada actualmente).

Recordemos que un recorrido noreste arranca en la esquina suroeste y avanza por filas hacia el este y el norte, según se viera en la subsección 4.2.3.



G.5.3. Codificación de una zona de números de 2 dígitos, representando al número 1 (1 bolita negra en el dígito)

- Abajo de las zonas de juego y de datos se encuentra la zona de semilla, abarcando el mismo ancho que ambas zonas. Esta zona es una zona de números.
- Las *zonas de números* tienen 1 fila de alto y están delimitadas arriba y abajo por filas de 4 bolitas azules en la mayoría de las celdas (algunas de las celdas pueden variar la cantidad en función de otras condiciones); además, los extremos izquierdo y derecho de una fila de la zona de números estarán delimitados por una celda con 6 bolitas azules (ver gráfico G.5.3).
- Las zonas de números utilizan numeración posicional decimal estándar, de derecha a izquierda desde las unidades. Cada dígito d en una zona de números se codifica con 1 bolita azul para indicar su presencia, y d bolitas negras para indicar de qué dígito se trata ($0 <= d <= 9$). Una celda vacía indica que no hay dígito en ese lugar.
- Las piezas se codifican en 4 celdas contiguas utilizando bolitas verdes, rojas y negras.
- El piso se codifica con exactamente 8 bolitas azules (y puede llevar marcas especiales con bolitas azules extra).

Debemos explicar también cómo se codifican las piezas. Una pieza se codifica mediante las siguientes convenciones.

- Cada pieza tiene 4 secciones contiguas que ocupan celdas identificadas con la misma cantidad de bolitas verdes (el *código de la pieza*).
- Una sola de las celdas de la pieza es el pivote, identificado con bolitas negras y rojas; las negras indican el tipo de pieza y las rojas la rotación.
- Las celdas de la pieza están dispuestas según indican el tipo y la rotación
- Un tipo válido va entre 1 y 7, y la codificación sigue el orden mnemotécnico dado por el nombre del juego (1-Z, 2-I, 3-L, 4-F, 5-O, 6-S, 7-T). Cada tipo tiene una rotación "natural" que corresponde con la mnemotecnia.
- Una rotación válida va entre 1 y 4 y la codificación sigue el sentido de las agujas del reloj (1 - rotación natural, 2 - $1/4$ de giro horario, 3 - $1/2$ giro, 4 - $1/4$ de giro antihorario)
- El pivote de una pieza puede además llevar una marca de exactamente 7 bolitas rojas adicionales.

En el gráfico G.5.1a se observan las 7 piezas, numeradas del 1 al 7 (bolitas verdes), cada una con su tipo (bolitas negras, que en este caso coinciden con las verdes por el orden en

que fueron ingresadas las piezas) y su rotación (bolitas rojas, en este caso todas con 1, puesto que todas están en su rotación natural).

Para considerar que el tablero codifica de manera válida un juego de ZILFOST deben cumplirse las siguientes condiciones.

- El tablero está dividido en zonas a través del uso de bolitas azules como fuera explicado.
- En la zona de juego solo puede haber piezas válidas o piso, ambos sin marcar.
- No hay dos piezas con el mismo código en la zona de juego.
- En las zonas de números solo puede haber dígitos válidos o celdas vacías.

La mayoría de las operaciones de ZILFOST (procedimientos y funciones) tendrán como precondición la existencia de una codificación válida de un juego de ZILFOST en el tablero (algunos aceptarán como válida la presencia de marcas de pieza o de piso).

Habiendo completado la manera de representar un juego de ZILFOST en el tablero, estamos en condiciones de empezar a codificar algunas partes del juego.

5.2. Código GOBSTONES para las zonas

Para empezar a codificar el juego de GOBSTONES, empezaremos por las operaciones concernientes a las diferentes zonas del juego. Trataremos primero las de la geometría de la zona de juego, luego las de las zonas de números y finalmente las específicas de la geometría de las zonas de semilla y las de las zonas particulares de números de la zona de datos.

5.2.1. Zona de juego

Para la geometría de la zona de juego, empezaremos escribiendo los procedimientos y funciones necesarios para ubicarse en la zona de juego y poder luego movernos dentro de ella sin invadir otras zonas.

Recordemos que la zona de juego es una zona rectangular, delimitada en sus esquinas inferiores por celdas con 5 bolitas azules, en sus esquinas superiores por celdas con 7 bolitas azules y rodeada por celdas con 6 bolitas azules a ambos lados.

Definiremos los siguientes procedimientos y funciones concernientes a la zona de juego

```
// * Geometría de la zona de juego
//   procedure IrAlOrigenDeZonaDeJuego()
//   function desplazamientoXDeZonaDeJuego()
//   function desplazamientoYDeZonaDeJuego()
//   function anchoDeZonaDeJuego()
//   function altoDeZonaDeJuego()
//
// * Movimiento dentro de la zona de juego
//   function puedeMoverEnZonaDeJuego(dir)
//   procedure IrACoordenadaDeZonaDeJuego(x,y)
//   procedure IrAlBordeDeZonaDeJuego(dir)
//   function esFinDelRecorridoNEDeZonaDeJuego()
//   procedure AvanzarEnRecorridoNEDeZonaDeJuego()
```

El procedimiento `IrAlOrigenDeZonaDeJuego` mueve el cabezal hasta la celda denominada *origen de la zona de juego* (ver gráfico G.5.2), la cual se ubica al este de la celda con 5 bolitas azules que se encuentra en la esquina inferior izquierda de esta zona. Puesto que en una codificación válida de ZILFOST dicha celda es la primera en un recorrido noreste del tablero que cumple esa propiedad, podemos escribir este procedimiento como

```
procedure IrAlOrigenDeZonaDeJuego()
 /*
    PROPÓSITO: ir al origen de la zona de juego del Zilfost
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
```

```
    * El origen esta al Este de la primer celda
      de 5 bolitas en un recorrido NE de las celdas
*/
{ IrAPrimerCeldaNEConBolitas(Azul,5); Mover(Este) }
```

El procedimiento `IrAPrimerCeldaNEConBolitas` es un recorrido de búsqueda (ver definición 4.2.3), y es similar a la segunda versión de `BuscarInicioSendero` en el ejercicio 4.2.10. No debe dejar de observarse el uso de los comentarios para establecer el contrato del procedimiento, tal cual fuera explicado en la subsección 2.3.4. Esto se mantendrá en cada uno de los procedimientos presentados.

Actividad de Programación 1

Realice el ejercicio 5.2.1 y verifique que el código resultante es similar al que aparece en el código completo de ZILFOST presentado en el anexo B. Tener en cuenta que este procedimiento aparece en la `Biblioteca`, pues no es específico del juego, sino general.

Ejercicio 5.2.1. *Escribir el procedimiento* `IrAPrimerCeldaNEConBolitas`*, que toma un color* `c` *y un número* `n` *y posiciona el cabezal en la primer celda en un recorrido noreste que tenga exactamente* `n` *bolitas de color* `c`*. Considerar utilizar el procedimiento* `BuscarInicioSendero` *del ejercicio 4.2.10 como guía (pueden reutilizarse varias de las subtareas definidas en ese ejercicio), y reutilizar los procedimientos definidos en el ejercicio 4.2.7.*

El procedimiento `IrAlOrigenDeZonaDeJuego` es extremadamente útil para ubicarnos en la zona de juego, y desde allí realizar otras operaciones. Sin embargo, para simplificar controles posteriores es más sencillo si conocemos la cantidad de celdas que hay desde la esquina suroeste hasta el origen de la zona de juego. Para esto definiremos funciones `desplazamientoXDeZonaDeJuego` y `desplazamientoYDeZonaDeJuego`, de la siguiente manera

```
function desplazamientoXDeZonaDeJuego()
 /*
    PROPÓSITO: retorna la cantidad de celdas al Este
               a moverse desde la esquina suroeste para
               ubicarse en la 1era columna de la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
 */
 {
   IrAlOrigenDeZonaDeJuego()
   return (medirDistanciaAlBorde(Oeste))
 }
```

Podemos observar que reutilizamos la función `medirDistanciaAlBorde` que realizamos en el ejercicio 4.3.6, que en este caso totaliza la cantidad de celdas que hay que moverse desde el origen de la zona de juego hasta alcanzar el borde Oeste. La función `desplazamientoYDeZonaDeJuego` es similar, pero se mueve hacia el borde Sur.

Leer con Atención

Es de destacar la forma en que se van reutilizando las operaciones que ya fueron definidas con anterioridad. De esta manera la cantidad de código que debe realizarse disminuye, y consecuentemente la cantidad de trabajo necesario para escribirlo y mantenerlo.

Para Reflexionar

Reflexione sobre la importancia de que las operaciones que uno va definiendo sean lo suficientemente generales (por ejemplo mediante el uso de parámetros) como para que puedan aplicarse en diversos problemas con ningún o pocos cambios. Reflexione sobre la importancia de contar con un mecanismo (como la `Biblioteca`) que permita no replicar el código de estas operaciones entre diversas aplicaciones diferentes.

Un elemento de utilidad para movernos dentro de la zona de juego sin salirse de la misma (y sin tener que controlar cada vez mediante la codificación de bolitas si estamos en el borde) será conocer el ancho y alto de la zona de juego. Para eso definiremos funciones `anchoDeZonaDeJuego` y `altoDeZonaDeJuego`, de la siguiente manera

```
function anchoDeZonaDeJuego()
 /*
    PROPÓSITO: retorna la cantidad de celdas de ancho
               de la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
 */
 {
   IrAlOrigenDeZonaDeJuego()

   // Contar la distancia hasta el otro borde de la zona
   anchoActual := 0
   while (not nroBolitas(Azul)==5)
    {
      anchoActual := anchoActual + 1
      Mover(Este)
    }
   return (anchoActual)
 }
```

Observamos que se trata de un recorrido de totalización, pero que no podemos usar una función general (de biblioteca), porque la condición es específica de este caso. El código para la función `altoDeZonaDeJuego` es similar pero se mueve hacia el Norte, y controla de manera diferente el fin del recorrido.

Definimos una función, `puedeMoverEnZonaDeJuego`, de verificación de movimiento y dos procedimientos, `IrAlBordeDeLaZonaDeJuego` e `IrACoordenadaDeZonaDeJuego`, para efectivizar ese movimiento.

La función de control de movimiento, `puedeMoverEnZonaDeJuego`, cumple una función similar a la de la función primitiva `puedeMover`, pero restringida a la zona de juego. La misma se define como

```
function puedeMoverEnZonaDeJuego(dir)
 /*
    PROPÓSITO: determina si puede moverse en la dirección
               dada sin caerse de la parte de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * optimiza la pregunta (no usan cuentas que deban
        recorrer mucho el tablero)
      * la verificación usa el formato de codificación
        de un tablero Zilfost
 */
 {
   switch (dir) to
    Norte ->
           // Si dir es Norte, se puede mover dentro de la
           // zona si hay lugar al norte
       { puede := puedeMover(Norte) }
```

```
    Este, Oeste ->
          // Si dir es Este u Oeste, se puede mover dentro
          // de la zona si al moverse se topa con el borde
       {
         Mover(dir)
         puede := not (nroBolitas(Azul)==5
                    || nroBolitas(Azul)==6
                    || nroBolitas(Azul)==7
                     )
       }
    _   -> // Solo hay 4 direcciones!!!
           // Si dir es Sur, se puede mover dentro de la
           // zona si al moverse al sur se topa con el borde
       {
         Mover(Sur)
         puede := not (nroBolitas(Azul)==4)
       }

   return (puede)
 }
```

Para detectar si se trata de un borde se utilizan las convenciones en la representación de la zona. Al Sur el límite contiene 4 bolitas azules; a Este y Oeste, contiene 5, 6 ó 7 bolitas azules; y al Norte el límite de la zona es es límite del tablero. Esto permite realizar una consulta eficiente de la posibilidad de moverse en la zona de juego, lo que resulta crucial para la eficiencia de las operaciones del juego.

El primer procedimiento se define como

```
procedure IrAlBordeDeZonaDeJuego(dir)
 /*
    PROPÓSITO: ir al borde en dirección dir
               dentro de la zona de juego del Zilfost
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
 */
 { while (puedeMoverEnZonaDeJuego(dir)) { Mover(dir) } }
```

Observamos que el mecanismo es simplemente un recorrido de búsqueda que busca el borde (la celda donde no se puede mover dentro de la zona de juego). Para ir a una coordenada específica dentro de la zona de juego seguimos una estrategia diferente: se utilizan las cantidades previamente obtenidas.

```
procedure IrACoordenadaDeZonaDeJuego(x,y)
 /*
    PROPÓSITO: ir a la coordenada (x,y) en la zona
               de juego del SeudoTetris
    PRECONDICIONES:
     * (x,y) indica una coordenada válida
              dentro de la zona de juego de Zilfost
    OBSERVACIONES:
     * la zona de juego de Zilfost está desplazada
       al Este por la parte de datos
 */
 {
   IrACoordenada(x + desplazamientoXDeZonaDeJuego()
               , y + desplazamientoYDeZonaDeJuego())
 }
```

En este caso, traducimos la geometría de la zona de juego en la del tablero, y reutilizamos un procedimiento de biblioteca para ir a una coordenada específica. Dicho procedimiento no lo hemos construído aún, por lo cual se deja como ejercicio. Observar, además, que se asume como precondición que la coordenada indicada es válida dentro de la zona de juego (o sea, que los números no son negativos, ni mayores que el ancho o alto de la zona de juego).

Actividad de Programación 2

Realice el ejercicio 5.2.2 y ubíquelo en la `Biblioteca`. Compare el código producido con el que se presenta en el anexo B y verifique si reutilizó las mismas subtareas o no y si definió una precondición similar. En caso que la respuesta sea negativa, reflexione sobre las diferencias, y sobre si son o no significativas desde el punto de vista de los conceptos impartidos en el libro.

Ejercicio 5.2.2. *Definir un procedimiento* `IrACoordenada` *que dados dos parámetros numéricos* `x` *e* `y` *ubique el cabezal en la celda que se encuentra* `x` *celdas al Este e* `y` *celdas al Norte de la esquina suroeste. Establecer la precondición de este procedimiento de manera adecuada.*

Las últimas dos operaciones de la zona de juego son las que permiten completar un recorrido de las celdas de la zona de juego en dirección noreste. Son similares a las operaciones del ejercicio 4.2.7, pero utilizando de manera fija los parámetros, y cambiando la pregunta sobre si se puede mover para que solo consulte dentro del tablero.

```
function esFinDelRecorridoNEDeZonaDeJuego()
 /*
    PROPÓSITO: determina si puede moverse a la celda
               siguiente en un recorrido Noreste de
               las celdas de la zona de juego
  */
 {
   return (not puedeMoverEnZonaDeJuego(Norte)
        && not puedeMoverEnZonaDeJuego(Este))
 }

//-----------------------------------------------------
procedure AvanzarEnRecorridoNEDeZonaDeJuego()
 /*
    PROPÓSITO: avanza a la celda siguiente en un
               recorrido Noreste de las celdas de
               la zona de juego
    PRECONDICIONES: no está en el final del recorrido
  */
 {
   if (puedeMoverEnZonaDeJuego(Este))
     { Mover(Este) }
   else
     { IrAlBordeDeZonaDeJuego(opuesto(Este)); Mover(Norte) }
 }
```

5.2.2. Zonas de números

Para las zonas de números tenemos tres tipos de operaciones: las de movimiento, las de lectura y las de modificación. Recordemos que las zonas de números consisten en una única fila, delimitada arriba y abajo por celdas con 4 bolitas azules, y a izquierda y derecha por celdas con 6 bolitas azules.

Definiremos los siguientes procedimientos y funciones concernientes a la zona de juego

```
// Operaciones de movimiento en la zona de números actual
//   function puedeMoverEnZonaDeNumeroAl(dir)
//   procedure IrAlBordeDeZonaDeNumeros(dir)
//
// Operaciones para leer y grabar un número de una
// zona de números
//   function leerZonaDeNumeros()
//     function hayDigito()
```

```
//     function leerDigito()
//   procedure GrabarNumeroEnZonaDeNumeros(numero)
//     procedure GrabarDigitoEnCelda(dig)
//
// Operaciones para modificar una zona de números
//   procedure BorrarZonaDeNumeros()
//   procedure AgregarDigitoAZonaDeNumerosPorIzq(dig)
//   procedure IncrementarZonaDeNumeros()
//     procedure IncrementarDigitoDeCelda()
```

La primera función a definir permite controlar si es posible moverse en una dirección determinada dentro de la zona de números. Dado que no es posible moverse en dirección Norte o Sur, la pregunta carece de sentido en estos casos, por lo que supondremos como precondición que la dirección no es ninguna de esas dos. Además, para que la función tenga sentido la celda actual debe encontrarse dentro de una zona de números. El código de la función será entonces muy simple

```
function puedeMoverEnZonaDeNumeroAl(dir)
 /*
    PROPÓSITO:
      devuelve si hay más lugar en dirección dir en
      la zona de números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * la celda actual está en la zona de números a
       determinar si se puede mover
     * dir es Este u Oeste (no tiene sentido que sea
       Norte o Sur, porque las zonas de números tienen
       altura 1)
    OBSERVACIONES:
     * la zona termina con 6 azules al Este y Oeste
 */
 { return(nroBolitasAl(Azul,dir)/=6) }
```

Observar cómo se establece la precondición de manera que el código tenga sentido. Esta precondición deberá garantizarse cada vez que se invoque esta función.

La siguiente operación de movimiento consiste en el procedimiento para ir al borde de la zona. Al igual que en el caso de la función anterior, supondremos como precondición que la dirección es Este u Oeste y que la celda actual está dentro de una zona de números. El código entonces es simplemente un recorrido de búsqueda

```
procedure IrAlBordeDeZonaDeNumeros(dir)
 /*
    PROPÓSITO:
      ir al borde dir de la zona de números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * se encuentra dentro de una zona de números
     * dir es Este u Oeste (no tiene sentido que sea
       Norte o Sur, porque las zonas de números tienen
       altura 1)
 */
 {
   while(puedeMoverEnZonaDeNumeroAl(dir))
    { Mover(dir) }
 }
```

Dado que la zona de números codifica un número, es necesario contar con una forma de obtener de qué número se trata. La función `leerZonaDeNumero` es la que da sentido a la zona de números; la misma utiliza dos funciones auxiliares, `hayDigito` y `leerDigito`, que capturan la operatoria elemental. El código se estructura como un recorrido de totalización sobre los dígitos, leyéndolos de derecha a izquierda, para lo cual asumimos como precondición que la celda actual se encuentra en el borde derecho de la zona a leer. El código de la función será entonces

```
function leerZonaDeNumeros()
 /*
    PROPÓSITO:
      devuelve un número codificado en la zona de números
      actual, si tal número existe, o cero si no
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * está en el borde derecho de la zona de números
       a leer
    OBSERVACIONES:
     * se estructura como un recorrido sobre los dígitos
       codificados en la zona de números
     * total guarda el número leído hasta el momento
     * posDig guarda la próxima unidad a leer
 */
 {
   total := 0
   posDig := 1
   while(hayDigito() && puedeMoverEnZonaDeNumeroAl(Oeste))
    {
      // cada digito contribuye según su posición
      total := leerDigito() * posDig + total
      posDig := posDig * 10  // base 10
      Mover(Oeste)
    }
   // si no pudo mover al Oeste y hay dígito, no leyó el
   //  último dígito
   if (hayDigito() && not puedeMoverEnZonaDeNumeroAl(Oeste))
    {
      // cada digito contribuye según su posición
      total := leerDigito() * posDig + total
      posDig := posDig * 10  // base 10
    }
   return(total)
 }
```

Es interesante observar que existen dos formas de finalizar: cuando deja de haber dígitos pero aún hay espacio (se encuentra una celda vacía donde no hay dígito), o se encuentra el borde izquierdo de la zona. En este último caso el último dígito debe procesarse por separado. La variable `posDig` almacena si el próximo dígito a leer es de unidades, decenas, centenas, etcétera. De ahí que sea inicializado en uno y que en cada repetición se multiplique por 10 (la base de numeración). Las funciones auxiliares `hayDigito` y `leerDigito` se dejan como ejercicio.

Actividad de Programación 3

Realice el ejercicio 5.2.3. Recuerde indicar adecuadamente las precondiciones correspondientes. Luego de realizarlo, verifique con el código del anexo B que realizó el ejercicio correctamente.

Ejercicio 5.2.3. *Definir las siguientes funciones auxiliares*

1. `hayDigito`*, que indique si en la celda actual hay o no un dígito. Recordar que la presencia de un dígito se indica con una bolita azul, la cantidad se indica con bolitas negras, y que el dígito debe estar entre 0 y 9.*
2. `leerDigito`*, que, suponiendo que en la celda actual hay un dígito correctamente codificado, retorna la cantidad representada por ese dígito.*

Por otra parte, el procedimiento `GrabarNumeroEnZonaDeNumeros` se encarga de codificar el número dado dentro de la zona de números actual. El código se estructura como un recorrido sobre los dígitos del número, desde las unidades. En cada repetición se divide al número por 10 (la base de numeración), lo cual deja disponible el siguiente dígito para la

siguiente repetición. Además, al iniciar debe borrarse la zona de números actual para evitar que se superpongan las representaciones de más de un número. El código resultante será

```
procedure GrabarNumeroEnZonaDeNumeros(numero)
 /*
    PROPÓSITO: guardar el número dado en la zona de
               números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * está en la zona de números donde debe grabarse
 */
 {
   BorrarZonaDeNumeros()
   IrAlBordeDeZonaDeNumeros(Este)
   aGuardar := numero
   while (aGuardar > 0)
     {
       GrabarDigitoEnCelda(aGuardar mod 10)
       aGuardar := aGuardar div 10
       Mover(Oeste)
     }
 }
```

El procedimiento utiliza una operación auxiliar para grabar un dígito en una celda, que se deja como ejercicio.

Actividad de Programación 4

Relice el ejercicio 5.2.4. Constate con la implementación presentada en el anexo B que su solución sea correcta.

Ejercicio 5.2.4. *Escribir el procedimiento* `GrabarDigitoEnCelda` *que toma un número representando a un dígito y codifica dicho dígito en la celda actual. Puede asumir como precondiciones que la celda está vacía y que el número recibido es efectivamente un dígito (un número entre 0 y 9).*

Las operaciones para modificar las zonas de números permiten en primer lugar borrar cualquier número codificado en la zona, luego agregar un dígito a la zona de izquierda a derecha (que servirá para aceptar los dígitos de a uno al ingresar un código), y finalmente, incrementar el número representado en la zona.

La operación de borrado de zona de números simplemente debe recorrer cada celda de la zona de números, vaciándola. El código resultante será

```
procedure BorrarZonaDeNumeros()
 /*
    PROPÓSITO:
      borra el contenido de la zona de números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * se encuentra en la zona de números a borrar
    OBSERVACIONES:
     * se estructura como un recorrido en la zona
       de números
 */
 {
   IrAlBordeDeZonaDeNumeros(Este)
   while(puedeMoverEnZonaDeNumeroAl(Oeste))
    {
      VaciarCelda()
      Mover(Oeste)
    }
   VaciarCelda()
 }
```

donde la operación `VaciarCelda` es la que se definió en el ejercicio 3.2.13 y corresponde con la subtarea de *procesar elemento* del esquema genérico de recorridos. Observar también el uso de las operaciones de zona, el procedimiento `IrAlBordeDeZonaDeNumeros` como la subtarea de iniciar el recorrido, y la función `puedeMoverEnZonaDeNumeroAl` como la subtarea de verificar el fin del recorrido actual.

Otra de las operaciones de modificación específica de las zonas de números es la operación de `AgregarDigitoAZonaDeNumerosPorIzq`. Esta operación se utiliza para ir leyendo los dígitos de un número de izquierda a derecha (por ejemplo, por teclado), para especificar un número completo. La lectura de un número por teclado de esta manera se asemeja a la función de un control remoto cuando colocamos los números de canales para cambiar de canal: apretamos primero el 0 y después el 7 para poner el canal 7. En cada momento debe ubicarse cuál es la próxima posición libre en la zona de números y agregar el dígito pasado como parámetro a esa posición. En caso que la zona esté llena, la borra y vuelve a empezar. El código resultante es el siguiente

```
procedure AgregarDigitoAZonaDeNumerosPorIzq(dig)
 /*
    PROPÓSITO:
      agrega el dígito dig codificado en la zona de
      números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * dig es un dígito (entre 0 y 9)
     * se encuentra en la zona de números donde debe
       agregar el dígito
    OBSERVACIONES:
     * los dígitos entran a la zona de izquierda a derecha
     * recorre los dígitos de izq a der para
       encontrar dónde poner el dígito
     * los espacios libres solo pueden estar
       a la derecha y si no hay, borra el número
       completo y pone el dígito como el primero
       (alternativamente podría ignorar el dígito)
 */
 {
   IrAlBordeDeZonaDeNumeros(Oeste)
   while(hayDigito()) { Mover(Este) }

   if (nroBolitas(Azul)==0)
     { GrabarDigitoEnCelda(dig) }
   else
     {
       // Si no hay espacio, borra el número anterior
       BorrarZonaDeNumeros()
       IrAlBordeDeZonaDeNumeros(Oeste)
       GrabarDigitoEnCelda(dig)
     }
 }
```

Alternativamente esta operación podría ingresar los dígitos siempre en la posición menos significativa (el extremo derecho), corriendo los demás dígitos para hacer lugar. Esta modificación se deja como ejercicio.

Actividad de Programación 5

Realice el ejercicio 5.2.5, reemplácelo en el código provisto en el anexo B y pruébelo. ¿Cuál comportamiento le resulta más satisfactorio?

Ejercicio 5.2.5. *Escribir un procedimiento cuyo propósito sea similar al del procedimiento* `AgregarDigitoAZonaDeNumerosPorIzq`*, pero donde cada dígito nuevo ingresa siempre en la posición menos significativa, "empujando" a los demás dígitos hacia la izquierda (si hubiera lugar), o borrando los demás dígitos (si no lo hubiera). Nombrarlo correctamente.*

Ayuda: *se recomienda verificar primero si hay lugar, chequeando que la celda en el extremo izquierdo está libre, y en base a esa condición, elegir entre la alternativa de correr todos los dígitos a la izquierda, o borrar la zona, para terminar colocando el dígito en la celda menos significativa.*

La última de las operaciones de modificación sobre una zona de números es la de incrementar el número representado en dicha zona. Esta operación se utiliza, por ejemplo, para aumentar el código de la próxima pieza a introducir en el tablero una vez que se introdujo una. Se realiza como un recorrido sobre los dígitos a incrementar, desde el menos significativo hasta el más significativo, recordando en cada iteración si hay que "llevarse uno" (en inglés, *carry*); se detiene cuando ya no hay que "llevarse uno". Para poder contar con un *carry* inicial, en lugar de procesar por separado el *último elemento* de la secuencia, se comienza procesando por separado el *primero*.

```
procedure IncrementarZonaDeNumeros()
 /*
    PROPÓSITO: incrementa el número codificado en la
               zona de números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * el número máximo codificado no excede la cantidad
       de dígitos disponibles
     * se encuentra en la zona de números a incrementar
    OBSERVACIONES:
     * usa el algoritmo usual de incremento con carry
       ("llevarme uno"), o sea un recorrido sobre los
       dígitos a incrementar
     * puede fallar si excede el máximo representable
 */
 {
   IrAlBordeDeZonaDeNumeros(Este)
   IncrementarDigitoDeCelda()
   carry := (leerDigito() == 0)
   while (carry && puedeMoverEnZonaDeNumeroAl(Oeste))
    {
      Mover(Oeste)
      IncrementarDigitoDeCelda()
      carry := (leerDigito() == 0)
    }
   if (carry) // Se excedió del máximo permitido de piezas!
     {
       BorrarZonaDeNumeros()
       IncrementarDigitoDeCelda()
     }
 }
```

Observemos el uso de las operaciones de movimiento en la zona. Además, al finalizar se verifica que no haya carry; en caso de haberlo, significa que el número excedió el máximo representable en la zona. En ese caso, vuelve el número a 1.

Al volver el número a 1, si la zona de código de próxima pieza tuviese pocos dígitos, podría suceder que apareciesen 2 piezas con el mismo código en la zona de piezas. Esta situación es inválida, por lo que se recomienda no tener pocos dígitos en esta zona.

Esta última operación utiliza el procedimiento `IncrementarDigitoDeCelda`, cuyo comportamiento es similar al del ejercicio 3.2.16 pero con una codificación levemente diferente.

```
procedure IncrementarDigitoDeCelda()
 /*
    PROPÓSITO: incrementa el dígito actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * o bien no hay dígito, o bien es un dígito válido
       (entre 0 y 9)
    OBSERVACIONES:
     * si no hay dígito, lo agrega
     * si se excede, lo vuelve a 0
 */
```

```
{
  // Agrega un dígito si no lo había
  if (not hayDigito()) { Poner(Azul) }
  // Incrementa dicho dígito
  Poner(Negro)
  // Si se excede, vuelve a 0
  if (leerDigito() == 10) { SacarN(Negro, 10) }
}
```

Las operaciones sobre zonas de números se utilizarán en la definición de las zonas específicas: zonas de códigos de piezas y zona de semilla.

5.2.3. Zonas de números específicas

Las zonas de específicas de números son 3: las dos zonas de código de piezas, la zona de pieza seleccionada y la zona de próxima pieza, y la zona de semilla. Para las 3 zonas tendremos operaciones para ir al origen de la zona (su extremo derecho), leer el número de la zona y modificar la zona. Además para las dos zonas de códigos de pieza tendremos operaciones de borrar la zona. Todas estas operaciones usarán las operaciones de zonas de números definidas en la subsección anterior. En el caso de las operaciones de modificación de zona, la zona de pieza seleccionada se modifica agregando dígitos de a uno (ya sea por izquierda o por derecha, según el procedimiento de la zona de números que se elija), mientras que la zona de próxima pieza se modifica incrementando la zona, y la zona de semilla se modifica grabando completamente una semilla nueva. Las operaciones para estas zonas serán las siguientes

```
// procedure IrAlOrigenDeZonaDeProximaPieza()
// procedure IrAlOrigenDeZonaDeSeleccion()
// procedure IrAlOrigenDeZonaDeSemilla()
//
// function leerZonaDeProximaPieza()
// function leerZonaDeSeleccion()
// function leerSemilla()
//
// procedure BorrarZonaDeProximaPieza()
// procedure BorrarZonaDeSeleccion()
//
// procedure AgregarDigitoASeleccion(dig)
// procedure IncrementarZonaDeProximaPieza()
// procedure GrabarSemilla(semilla)
```

Las operaciones de ir al origen se hacen asumiendo la posición relativa de estas zonas con respecto al origen de la zona de juego. De esta manera, resultan sencillas.

```
procedure IrAlOrigenDeZonaDeProximaPieza()
 /*
    PROPÓSITO:
      ir al origen de la zona de próxima pieza
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * esta zona está 2 al Oeste del origen de la
       zona de juego y 1 al Norte
 */
 {
   IrAlOrigenDeZonaDeJuego()
   MoverN(Oeste,2); MoverN(Norte, 1)
 }

procedure IrAlOrigenDeZonaDeSeleccion()
 /*
    PROPÓSITO:
      ir al origen de la zona de selección
    PRECONDICIONES:
```

```
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * 2 al Norte del origen de la zona de
       próxima pieza
 */
 {
   IrAlOrigenDeZonaDeProximaPieza()
   MoverN(Norte,2)
 }

procedure IrAlOrigenDeZonaDeSemilla()
 /*
    PROPÓSITO: ir al origen de la zona de semilla
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * la semilla se codifica debajo de la zona
        de piezas, todo a lo ancho
 */
 {
   IrAlOrigenDeZonaDeJuego()
   MoverN(Sur, 2)
   IrAlBordeDeZonaDeNumeros(Este)
 }
```

La lectura de estas zonas es tan sencillo como ir al origen de la zona correspondiente (mediante las subtareas expresadas en los procedimientos anteriores), y retornar el resultado de leer la zona (con la operación de lectura de zona de números definida en la sección anterior). Por ejemplo, la lectura de la zona de próxima pieza será la siguiente

```
function leerZonaDeProximaPieza()
 /*
    PROPÓSITO:
      devuelve un número codificado en la zona de
      próxima pieza, si existe, o cero si no
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * va a la zona de próxima pieza y lee el
       número allí codificado
 */
 {
   IrAlOrigenDeZonaDeProximaPieza()
   return(leerZonaDeNumeros())
 }
```

Las operaciones para leer las zonas restante son prácticamente idénticas y las omitimos.

Las operaciones de borrado y de modificación de estas zonas son prácticamente idénticas a las de lectura: se mueven al origen de la zona, y la modifican utilizando la operación correspondiente de la sección anterior. Debido a su simplicidad, omitimos también el código.

Para Reflexionar

Quizás resulte oportuno volver a reflexionar sobre la importancia de dividir en subtareas. Puesto que las subtareas de operaciones sobre zonas de números fueron bien elegidas, las operaciones para las zonas específicas son extremadamente sencillas. ¡Es tan importante este concepto que en nuestra propuesta de enseñanza consideramos que debe ser *de los primeros* conceptos que aprenda una persona que se inicia en programación!

5.3. Código para expresar piezas

Las piezas del ZILFOST se representan con bolitas verdes, rojas y negras en la zona de juego. Se definen para las piezas una serie de operaciones con el objetivo de expresar su geometría y detectarlas a ellas o su ausencia.

En primer lugar consideraremos las operaciones relativas a la geometría de las piezas, luego las operaciones que permiten detectar piezas o lugar para ellas.

5.3.1. Geometría de las piezas

Cada una de las piezas tienen una forma específica, y para dibujarla o borrarla, debemos conocer esa forma. En lugar de proveer una operación de dibujo y una de borrado por cada una de las piezas, lo cual haría que codificásemos varias veces la forma de la pieza en lugares diferentes, vamos a separar el problema de dibujarlas o borrarlas en, por un lado, representar la forma de la pieza mediante datos, y por el otro en utilizar dichos datos para el dibujo o borrado efectivo.

En esta subsección presentamos el código necesario para representar la forma de las distintas piezas mediante datos. Las funciones que definiremos para esta tarea son las siguientes

```
// function rotar(rotacion,sentidoHorario)
//
// function esClaseA(tipoPieza)
// function esClaseB(tipoPieza)
//
// function diresDePiezaClaseA(tipoPieza,rotPieza)
// function diresDePiezaClaseB(tipoPieza,rotPieza)
//   function diresDePiezaZ(rotPieza)
//   function diresDePiezaI(rotPieza)
//   function diresDePiezaL(rotPieza)
//   function diresDePiezaF(rotPieza)
//   function diresDePiezaO(rotPieza)
//   function diresDePiezaS(rotPieza)
//   function diresDePiezaT(rotPieza)
//
//   function id(dA,dB,dC1,dC2)
//   function ajustarDires(dA,dB,dC1,dC2,rotPieza)
```

Iremos viendo el significado y uso de cada una de ellas. Tener en cuenta que las funciones que se usarán luego son solo las 5 primeras; las restantes son funciones auxiliares de ellas.

La primera de las funciones no está directamente relacionada con las piezas, sino que concierne exclusivamente a la rotación. Dado que las rotaciones se representan con números de 1 a 4, cuando queremos rotar una pieza debemos calcular el código de la nueva rotación. Lo que hace la función `rotar` es justamente este cálculo.

```
function rotar(rotacionBase,sentidoHorario)
 /*
    PROPÓSITO: calcular la rotacion siguiente en
               sentido horario o antihorario, según
               lo indique el booleano sentidoHorario
    PRECONDICIONES:
      * rotacion es una rotación válida
        (de 1 a 4 o con marca de 7 rojas)
    OBSERVACIONES:
      * el cálculo se realiza de la siguiente forma
        rotacion  1 2 3 4
        . mod 4   1 2 3 0
        . +1      2 3 4 1 // En sentido horario

        rotacion  1 2 3 4
        . + 2     3 4 5 6
        . mod 4   3 0 1 2
        . + 1     4 1 2 3 // En sentido antihorario
```

```
*/
{
  // Recordar si tiene o no marca
  if (rotacionBase>=1 && rotacionBase<=4)
    { marca := 0 }
  else { if (rotacionBase>=8 && rotacionBase<=11)
    { marca := 7 }
  else { }} -- La rotación es inválida

  // Ajusta la rotación si tiene marcas
  rotacion := rotacionBase - marca

  // Calcula la nueva rotación
  if (sentidoHorario)
    { nuevaRotacion := (rotacion mod 4) + 1 }
  else
    { nuevaRotacion := (rotacion+2) mod 4 + 1 }

  // Retorna, restaurando la marca si corresponde
  return (nuevaRotacion+marca)
}
```

La función asume que la rotación puede venir marcada (algo que se presentará después, durante la codificación de la mecánica del juego), por lo que la primer parte se encarga de detectar si hay tal marca, y quitarla temporalmente. La rotación propiamente dicha se realiza mediante una fórmula, cuyo sentido se explica en el comentario. Finalmente, al retornar, se restaura la marca (si es que la misma existía). Una observación interesante es cómo, en caso de que la rotación no sea válida, la función falla por no haber inicializado correctamente la variable `marca`.

Para las piezas, como dijimos, cada una consta de 4 secciones contiguas y tal que una de ellas es el *pivote* de la pieza. El pivote será el eje de rotación de la pieza, y además lo usaremos como el centro desde el cual se trabajará sobre la pieza. Entonces es posible codificar una pieza dando las direcciones desde el pivote en el que se encuentran las otras 3 secciones. Ahora bien, en 6 de las piezas (la Z, la I, la L, la F, la O y la S), dos de las tres secciones restantes se encuentran inmediatamente al lado del pivote, y la otra a mayor distancia. En cambio en la última de las piezas (la T), las tres secciones restantes están contiguas al pivote. Este hecho lo representaremos hablando de la *clase* de la pieza: una pieza será de *clase A* cuando tenga dos secciones contiguas y una distante, y será de *clase B* cuando tenga las tres secciones contiguas al pivote. Con esta definición, es fácil escribir el código de las primeras dos funciones

```
function esClaseA(tipoPieza)
  /*
     PROPÓSITO: indica si la pieza del tipo dado
                es de clase A
   */
  { return(tipoPieza >= 1 && tipoPieza <= 6) }

function esClaseB(tipoPieza)
  /*
     PROPÓSITO: indica si la pieza del tipo dado
                es de clase B
   */
  { return(tipoPieza >= 7 && tipoPieza <= 7) }
```

Vemos que la primera función retorna un booleano que indica si el código del tipo de pieza es de una de las primeras 6 piezas (Z, I, L, F, O ó S) y la segunda retorna un booleano que indica si el código del tipo de pieza de la última de las piezas (T). Es interesante utilizar las funciones de esta manera, para abstraer la definición específica del concepto a expresar. Estas funciones se usarán posteriormente para decidir cuál de las operaciones genéricas de piezas deben invocarse (si los de clase A o los de clase B).

Una vez definida la noción de clase, vemos que las piezas de clase A pueden ser representadas mediante 4 direcciones. Las dos primeras direcciones indican en qué dirección desde el pivote se encuentran las dos secciones contiguas. Las dos últimas direcciones indican hacia dónde se encuentra la tercer sección (que siempre estará a distancia

2 del pivote, por ser una pieza de clase A). Por otra parte, las piezas de clase B puede ser representadas mediante 3 direcciones. Además, las direcciones en las que se encuentran las secciones contiguas también dependen de la rotación. Con estas ideas, estamos en condiciones de definir las siguientes dos funciones, que se encargan de obtener las direcciones necesarias de cada tipo de pieza, dado el tipo y la rotación.

```
function diresDePiezaClaseA(tipoPieza,rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una pieza
               de clase A, ajustadas según la rotación
    PRECONDICIONES:
     * tipoPieza es un tipo válido
     * rotPieza es una rotación válida
    OBSERVACIONES:
     * realiza una selección alternativa en base al
       tipo de pieza
 */
 {
   switch (tipoPieza) to
     1 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaZ(rotPieza) }
     2 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaI(rotPieza) }
     3 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaL(rotPieza) }
     4 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaF(rotPieza) }
     5 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaO(rotPieza) }
     6 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaS(rotPieza) }
     _ -> { }

   return (dirA,dirB,dirC1,dirC2)
 }

//-------------------------------------------------------
function diresDePiezaClaseB(tipoPieza,rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una pieza
               de clase B, ajustadas según la rotación
    PRECONDICIONES:
     * tipoPieza es un tipo válido
     * rotPieza es una rotación válida
    OBSERVACIONES:
     * realiza una selección alternativa en base al
       tipo de pieza
 */
 {
   switch (tipoPieza) to
     7 -> { (dirA,dirB,dirC) := diresDePiezaT(rotPieza) }
     _ -> { }

   return (dirA,dirB,dirC)
 }
```

Hay varios detalles para observar. El primero es que ambas funciones asumen como precondición que tanto el tipo de pieza como la rotación son válidos; en caso de recibir datos inválidos, fallarán con un error (en este caso, por indefinición de variables). El segundo es que ambas realizan una selección alternativa en base al tipo de pieza, e invocan a una función específica correspondiente a esa pieza; esta función toma como parámetro el código de rotación de la pieza y devuelve 4 (ó 3) direcciones, las cuales se asignan a variables en forma simultánea. El último detalle a observar es que la rotación de la pieza se utiliza como argumento para la función específica invocada.

La asignación simultánea es una forma extendida de la asignación que posee GOBSTONES, y que permite que una función retorne varios resultados al mismo tiempo. Puede verse como la operación opuesta de recibir varios argumentos. Ver el final de la subsección 4.3.1.

Las funciones específicas usadas en `diresDePiezaClaseA` y `diresDePiezaClaseB` sirven para expresar la forma de cada pieza utilizando direcciones. Para ello reciben como parámetro la rotación, y ajustan las direcciones "naturales" (las que tiene la pieza cuando está en rotación 1) según este parámetro. Veamos el código de una de estas funciones para la clase A, la de la pieza Z

```
function diresDePiezaZ(rotPieza)
 /*
    PROPÓSITO:
      * devolver las direcciones de una Z
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
      * en rotación 1, Z es
         Norte,Oeste ->  ZZ  <- Norte
                         XZ <- Este
        donde la X representa al pivote y
        las Zs a las demás secciones
 */
 {
   (dA,dB,dC1,dC2) := id(Este,Norte,Norte,Oeste)
   (dA,dB,dC1,dC2) := ajustarDires(dA,dB,dC1,dC2,rotPieza)
  return (dA,dB,dC1,dC2)
 }
```

Vemos que la primera línea contiene las direcciones que codifican la pieza: `Este` para una sección, `Norte` para la segunda, y `Norte` y `Oeste` para la tercera. La función `id` retorna simplemente los 4 valores, y la usamos para poder usar la asignación simultánea. La función `ajustarDires` se encarga de modificar las direcciones según la rotación. Las restantes funciones específicas de clase A son similares y las omitiremos. Pero la función específica para la pieza de clase B, `diresDePiezaT` merece cierta atención.

```
function diresDePiezaT(rotPieza)
 /*
    PROPÓSITO:
      * devolver las direcciones de una T
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
      * en rotación 1, T es
               Oeste ->  TXT  <- Este
                          T   <- Sur
        donde la X representa al pivote y
        las Ts a las demás secciones
      * se usa una dirección dD como dummy
        para reutilizar ajustarDires
 */
 {
   (dA,dB,dC,dD) := id(Oeste,Este,Sur,Sur)
                                    --^--DUMMY!!
   (dA,dB,dC,dD) := ajustarDires(dA,dB,dC,dD,rotPieza)
  return (dA,dB,dC)
 }
```

En inglés la palabra *dummy* usada como adjetivo significa *ficticio* y como sustantivo significa *imitación*. Sin embargo, el uso en computación es más bien técnico, y se utiliza para un valor que no tiene significado alguno en el cómputo (o sea, una *imitación*, un valor *ficticio*.

A primera vista parece casi idéntica a la anterior. Pero si recordamos bien, ¡para las piezas de clase B bastaba con 3 direcciones! Sin embargo, estamos dando 4... Esto es así para proveer uniformidad en los accesos y poder reutilizar la función `ajustarDires` y también otras más adelante. Sin embargo, la cuarta dirección es innecesaria para el trabajo (en inglés se utiliza el término *dummy* para indicar un valor que no se utilizará). Puede observarse que el valor de la variable `dD` NO se retorna.

Solo falta establecer el código de las funciones `id` y `ajustarDires`. La primera de ellas es extremadamente sencilla y solo sirve para retornar varios valores juntos (y deberá ser usada en una asignación múltiple)

```
function id(dA,dB,dC1,dC2)
```

```
 /*
    PROPÓSITO: retornar varios valores simultáneamente
 */
 { return (dA,dB,dC1,dC2) }
```

La segunda requiere verificar cada una de las 4 rotaciones posibles, y alterar sus parámetros de manera acorde con la rotación indicada.

```
function ajustarDires(dirA,dirB,dirC1,dirC2,rotPieza)
 /*
    PROPÓSITO: ajustar las direcciones en base a la rotación
    PRECONDICIONES:
      * la rotación es válida (y puede llevar una
          marca de 7 bolitas rojas)
 */
 {
   switch (rotPieza) to
    1,8 -> -- La rotación ‘‘natural’’
      {
        ndirA := dirA
        ndirB := dirB
        ndirC1 := dirC1
        ndirC2 := dirC2
      }
    2,9 -> -- 1/4 de giro en sentido horario
      {
        ndirA := siguiente(dirA)
        ndirB := siguiente(dirB)
        ndirC1 := siguiente(dirC1)
        ndirC2 := siguiente(dirC2)
      }
    3,10 -> -- 1/2 giro
      {
        ndirA := opuesto(dirA)
        ndirB := opuesto(dirB)
        ndirC1 := opuesto(dirC1)
        ndirC2 := opuesto(dirC2)
      }
    4,11 -> -- 1/4 de giro en sentido antihorario
      {
        ndirA := previo(dirA)
        ndirB := previo(dirB)
        ndirC1 := previo(dirC1)
        ndirC2 := previo(dirC2)
      }
    _ -> { }

   return (ndirA,ndirB,ndirC1,ndirC2)
 }
```

Observar el uso de la alternativa indexada para elegir el caso, y de las operaciones primitivas `siguiente`, `opuesto` y `previo` para calcular las nuevas direcciones.

5.3.2. Detección de piezas

Para la detección de piezas (o su ausencia), definiremos varias funciones que permitirán mirar desde la existencia de una sección específica, las de una pieza entera, e incluso si hay lugar para una pieza (que verificaría las mismas direcciones que se usan para verificar la existencia de la pieza). Las funciones son

```
// function esSeccionDeAlgunaPieza()
// function esSeccionPivoteDeAlgunaPieza()
// function esSeccionPivoteDePieza(codPieza)
// function hayPiezaActual()
```

```
//
// function leerCodigoDePiezaActual()
// function leerTipoDePiezaActual()
// function leerRotacionDePiezaActual()
//
// function hayLugarParaPiezaTipo(tipoPieza, rotPieza)
//    function hayLgPzClaseAEnDires(dirA,dirB,dirC1,dirC2)
//    function hayLgPzClaseBEnDires(dirA,dirB,dirC)
//    function hayCeldaLibreAl(dir)
//    function hayCeldaLibreAlY(dir1,dir2)
```

Las cuatro finales son solamente auxiliares para `hayLugarParaPiezaTipo`.

El primer grupo de funciones permite detectar si el cabezal se encuentra sobre una sección de pieza, y en algunos casos, si se trata del pivote. La primera simplemente detecta si hay una sección en la celda actual, lo cual es extremadamente simple

```
function esSeccionDeAlgunaPieza()
 /*
    PROPÓSITO: determinar si la celda actual es
               sección pivote de alguna pieza
 */
 { return (hayBolitas(Verde)) }
```

La siguiente función refina a la primera, agregando la condición de que no sea cualquier sección, sino que debe ser el pivote de alguna pieza; dada la codificación propuesta, también es muy simple

```
function esSeccionPivoteDeAlgunaPieza()
 /*
    PROPÓSITO: determinar si la celda actual es la
               sección pivote de una pieza
 */
 {
   return (esSeccionDeAlgunaPieza()
        && hayBolitas(Negro)
        && hayBolitas(Rojo))
 }
```

La tercera refina aún más a las anteriores, especificando que no debe tratarse de cualquier pieza, sino de la indicada por el parámetro; para hacerla también se sigue la codificación propuesta

```
function esSeccionPivoteDePieza(codPieza)
 /*
    PROPÓSITO: determinar si la celda actual es la
               sección pivote de la pieza codPieza
 */
 { return (esSeccionPivoteDeAlgunaPieza()
          && nroBolitas(Verde)==codPieza) }
```

Puesto que durante el juego usaremos la convención de que todo tratamiento para las piezas (moverlas, rotarlas, etc.) se hace desde el pivote, estas dos funciones resultan ser muy importantes para encontrar las piezas a tratar. Para facilitar la lectura del código posterior, se provee una pequeña abstracción para decidir si hay una pieza en la celda actual (en lugar de tener que preguntar por su pivote, preguntamos por toda la pieza)

```
function hayPiezaActual()
 /*
    PROPÓSITO: establecer si la celda actual determina una pieza
               seleccionada, lo cual por convención quiere decir
               que está sobre la sección pivote de una pieza
 */
 { return (esSeccionPivoteDeAlgunaPieza()) }
```

Una vez que se ha establecido que existe una pieza, nos hará falta determinar de qué pieza se trata y cuáles son sus características (tipo y rotación). Para ello definimos funciones

de lectura con la precondición de que para utilizarla ya se debe estar sobre el pivote de la pieza, lo cual es fácilmente detectable con las funciones presentadas antes. Las funciones de lectura de piezas se dejan como ejercicio.

Actividad de Programación 6

Realice el ejercicio 5.3.1 y constate que el código que escribió es correcto contrastándolo con el código del anexo B.

Ejercicio 5.3.1. *Escribir funciones para leer el código,* `leerCodigoDePiezaActual`*, el tipo,* `leerTipoDePiezaActual`*, y la rotación,* `leerRotacionDePiezaActual` *de la pieza actual.*
Ayuda: *en cada caso, es tan simple como retornar el número de ciertas bolitas.*

Leer con Atención

¿Por qué de definir funciones tan simples cuando podríamos utilizar directamente la expresión retornada? La razón es que de esta forma estamos *abstrayendo* la solución, separándola de la manera específica que elegimos para codificar. Esto nos permitirá formular el problema en términos de lo que queremos expresar (por ejemplo, que `hayPiezaActual()`) en lugar de tener que formularlo en términos de una codificación prefijada (`hayBolitas(Verde) && hayBolitas(Negro) && hayBolitas(Rojo)`), pues aunque sus ejecuciones son equivalentes, la forma en que las personas lo imaginamos no lo es. O sea, estamos *elevando el nivel* de abstracción, alejándolo de la máquina subyacente y acercándolo a nuestra forma de pensar.

Para Reflexionar

Resulta interesante volver a reflexionar sobre la naturaleza dual de los programas, que al mismo tiempo son descripciones de soluciones de *alto nivel* (legibles y entendibles por los seres humanos) y establecen un mecanismo de ejecución de *bajo nivel* (que permite que una máquina pueda llevar adelante un cómputo específico). También se debe reflexionar sobre el impacto que conocer y entender este hecho tiene sobre la calidad del código producido.

La última de las funciones importantes de esta sección es la que permite determinar si la celda actual indica una posición donde pueda colocarse una determinada pieza. Para definirla se usan varias funciones auxiliares que describiremos. Esta función, que denominamos `hayLugarParaPiezaTipo`, toma como parámetros el tipo y la rotación de una pieza, y retorna un booleano que es verdadero cuando hay lugar para la pieza (hay celdas libres en los lugares necesarios). Para esto utiliza las funciones que obtienen las direcciones de una pieza, y dos funciones auxiliares para procesar las celdas de clase A y B.

```
function hayLugarParaPiezaTipo(tipoPieza, rotPieza)
 /*
     PROPÓSITO: informar si hay lugar en el tablero
                para colocar una pieza de tipo tipoPieza
                y rotación rotPieza con pivote en la
                celda actual
     PRECONDICIONES:
       * tipoPieza es un tipo de pieza válido
       * rotPieza es una rotación válida
     OBSERVACIONES:
       * puede no haber lugar porque se acaba el tablero
         o porque está ocupada
 */
 {
   if (esClaseA(tipoPieza))
     {
```

```
       (dirA,dirB,dirC1,dirC2)
            := diresDePiezaClaseA(tipoPieza,rotPieza)
       hayL := hayLgPzClaseAEnDires(dirA,dirB,dirC1,dirC2)
     }
   else // Si no es clase A, es clase B
     {
       (dirA,dirB,dirC)
            := diresDePiezaClaseB(tipoPieza,rotPieza)
       hayL := hayLgPzClaseBEnDires(dirA,dirB,dirC)
     }

   return (hayL)
 }
```

Observar que la función simplemente decide, en base a la clase del tipo de pieza, cuál de las funciones auxiliares debe usarse. Resulta entonces interesante mirar el código de las funciones auxiliares. Para la clase A tendremos el siguiente código

```
function hayLgPzClaseAEnDires(dirA,dirB,dirC1,dirC2)
 /*
    PROPÓSITO: completar el trabajo de hayLugarParaPiezaTipo
               para las piezas de clase A
    PRECONDICIONES:
      * está parado sobre el lugar en el que va el pivote
        de la pieza
      * las direcciones dadas codifican la pieza
        correctamente
 */
  {
    return(esCeldaVacia()
           && hayCeldaLibreAl(dirA)
           && hayCeldaLibreAl(dirB)
           && hayCeldaLibreAlY(dirC1,dirC2))
  }
```

y para la clase B tendremos el siguiente

```
function hayLgPzClaseBEnDires(dirA,dirB,dirC)
 /*
    PROPÓSITO: completar el trabajo de hayLugarParaPiezaTipo
               para las piezas de clase B
    PRECONDICIONES:
      * está parado sobre el lugar en el que va el pivote
        de la pieza
      * las direcciones dadas codifican la pieza
        correctamente
 */
  {
    return(esCeldaVacia()
           && hayCeldaLibreAl(dirA)
           && hayCeldaLibreAl(dirB)
           && hayCeldaLibreAl(dirC))
  }
```

Se puede observar que en cada caso se verifica que las 4 celdas que alojarán a las secciones (indicadas por las direcciones obtenidas de los parámetros) están libres. Para eso se utilizan tres funciones auxiliares: `esCeldaVacia`, `hayCeldaLibreAl` y `hayCeldaLibreAlY`. Consideraremos cada una de ellas por separado, puesto que requieren cierto cuidado.

La función `esCeldaLibre` simplemente verifica que la celda actual esté vacía. Para ello abstrae a la función `esCeldaVacia` que es una función de `Biblioteca` que verifica si en la celda no hay bolitas de ningún color; esta función se definió en el ejercicio 3.2.14.

```
function esCeldaLibre()
 /*
    PROPÓSITO: determinar si la celda actual es una
```

```
                celda libre
    PRECONDICIONES:
      * la celda actual se encuentra en la zona de
        juego
    OBSERVACIONES:
      * abstrae a la función de Biblioteca que verifica
        si una celda está vacía
 */
 { return (esCeldaVacia()) }
```

La precondición establece que la celda actual está en la zona de juego, porque esta función se utiliza para verificar si se puede colocar la pieza en esa celda, y las piezas solo se pueden colocar en la zona de juego.

La función `hayCeldaLibreAl` debe indicar si en la dirección dada existe una celda donde se pueda colocar la pieza y si dicha celda está libre. Dado que para colocar la pieza la celda debe estar en la zona de juego, la función debe verificar que efectivamente *hay* una celda en la zona de juego *y* que la misma esté libre. Para esto se sigue una idea similar a la de la función `hayCeldaVaciaAl`, del ejercicio 4.1.3, pero adaptando la condición para que realice la pregunta dentro de la zona de juego.

```
function hayCeldaLibreAl(dir)
 /*
    PROPÓSITO: determinar si hay una celda libre en la
               zona de juego, en la dirección indicada
               por dir
    PRECONDICIONES:
      * la celda actual se encuentra en la zona de
        juego
    OBSERVACIONES:
      * es libre si hay celda y no tiene pieza
 */
 {
   return (puedeMoverEnZonaDeJuego(dir)
           && esCeldaVaciaAl(dir))
 }
```

Vale recordar que estamos usando la característica de *circuito corto* (*short circuit*) de la conjunción booleana (ver subsección 4.1.1) para evitar la parcialidad de la función `esCeldaVaciaAl`. Además, la precondición de la función establece que al comenzar la celda actual se encuentra en la zona de juego, para garantizar que estamos por colocar la pieza en un lugar válido.

La última de estas funciones auxiliares es `hayCeldaLibreAlY`, que en principio parecería similar a las anteriores. Sin embargo, esta función tiene una pequeña complicación. La función recibe dos direcciones, `dir1` y `dir2`, y verfica que la celda que se encuentra a dos posiciones de distancia de la actual está libre. La complicación es que dicha celda puede no existir por *dos* razones: no existe la celda en dirección `dir1`, o bien dicha celda existe pero la celda en dirección `dir2` *desde esta nueva celda* no existe. Por esta razón debemos pedir como precondición que las direcciones no se cancelen mutuamente (o sea, que no sean una `Norte` y la otra `Sur`, o una `Este` y la otra `Oeste`)), y debemos controlar la existencia de la segunda celda *después de habernos movido a la primera, si existe*. El código queda entonces como sigue

```
function hayCeldaLibreAlY(dir1,dir2)
 /*
    PROPÓSITO: determinar si hay una celda libre en la
               zona de juego, en la dirección indicada
               por las direcciones dir1 y dir2
    PRECONDICIONES:
      * la celda actual se encuentra en la zona de
        juego
      * las direcciones dadas no se "cancelan" mutuamente
        (por ejemplo, no son Norte y Sur o Este y Oeste)
    OBSERVACIONES:
      * es libre si hay celda y no tiene pieza
```

```
*/
{
  if (puedeMoverEnZonaDeJuego(dir1))
    { Mover(dir1)
      if (puedeMoverEnZonaDeJuego(dir2))
        { Mover(dir2)
          celdaLibre := esCeldaLibre() }
      else { celdaLibre := False }} -- No existe la celda2
  else { celdaLibre := False } -- No existe la celda1

  return (celdaLibre)
}
```

Podríamos haber utilizado una expresión compleja basándonos en la característica del *circuito corto* (*short circuit*) de la conjunción booleana, pero habría sido innecesariamente complicado.

5.4. Código para operaciones básicas sobre piezas

Una vez definidas las funciones que expresan a las piezas, podemos empezar a ver cuáles son las operaciones básicas sobre piezas. Las vamos a dividir en tres partes: la operación de localizar una pieza, las operaciones de colocar y quitar una pieza del tablero y las operaciones de movimiento de piezas (bajar, desplazar y rotar). A cada una de estas partes le dedicaremos una de las próximas subsecciones.

5.4.1. Localizar una pieza

La primera de las operaciones básicas de piezas es la de localizar una pieza determinada en la zona de juego. Esta operación se utiliza para que cuando el jugador digita el código de una pieza en la zona de selección, el programa pueda encontrarla y así poder operar sobre ella. Si la pieza no existe, se ubica en una esquina y no hace nada más. Al procedimiento que la codifica lo denominamos `IrAPiezaSiExiste`, y se estructura como un recorrido de búsqueda sobre las celdas de la zona de juego, terminando o bien cuando encuentra la pieza buscada, o bien cuando se acaban las celdas.

```
procedure IrAPiezaSiExiste(codPieza)
 /*
    PROPÓSITO: va a la celda pivote de la pieza de
               código codPieza, si existe
    PRECONDICIONES:
      * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * se estructura como un recorrido NE sobre las
        celdas de la zona de juego
      * si la pieza no existe, queda en un extremo
        de la zona de juego
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while (not esFinDelRecorridoNEDeZonaDeJuego()
          && not esSeccionPivoteDePieza(codPieza))
    { AvanzarEnRecorridoNEDeZonaDeJuego() }
 }
```

Observar el uso de las operaciones de recorrido sobre la zona de juegos que fueron definidas en la sección 5.2.1.

¡Una vez más podemos observar el poder de contar con subtareas adecuadas!

5.4.2. Colocar y quitar una pieza

Para colocar una pieza definiremos un procedimiento `ColocarPieza` que se encargue de la tarea. Este procedimiento seguirá un principio de trabajo similar al utilizado para verificar

si hay lugar para una pieza: en base a la clase del tipo de pieza elegirá uno de dos procedimientos auxiliares, uno para colocar piezas de clase A y otro para piezas de clase B. Éstos a su vez utilizaran operaciones auxiliares para colocar cada una de las secciones. Las operaciones serán las siguientes

```
// procedure ColocarPieza(codPieza, tipoPieza, rotPieza)
//   procedure ColocarPzClaseA(codPieza,tipoPieza,rotPieza
//                             ,dirA,dirB,dirC1,dirC2)
//   procedure ColocarPzClaseB(codPieza,tipoPieza,rotPieza
//                             ,dirA,dirB,dirC)
//     procedure ColocarSeccionDePieza(codPieza)
//     procedure ColocarPivote(codPieza,tipoPieza,rotPieza)
//     procedure ColocarSeccionDePiezaEn(codPieza,dir)
//     procedure ColocarSeccionDePiezaEnY(codPieza,dir1,dir2)
```

El código de `ColocarPieza` es sencillo una vez que entendemos que solo determina a cual de las operaciones auxiliares debe invocar.

```
procedure ColocarPieza(codPieza, tipoPieza, rotPieza)
 /*
    PROPÓSITO: coloca la pieza codPieza en el tablero
    PRECONDICIONES:
      * hay lugar para colocar la pieza
      * no hay otra pieza codPieza en el tablero
      * tipoPieza es un tipo válido
      * rotPieza es una rotación válida
    OBSERVACIONES:
      * la celda actual será el centro de la pieza
        codPieza
      * la pieza codPieza será de tipo tipoPieza
      * la rotación estará dada por rotPieza
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   if (esClaseA(tipoPieza))
     {
       (dirA,dirB,dirC1,dirC2)
             := diresDePiezaClaseA(tipoPieza,rotPieza)
       ColocarPzClaseA(codPieza,tipoPieza,rotPieza
                       ,dirA,dirB,dirC1,dirC2)
     }
   else // Si no es clase A, es clase B
     {
       (dirA,dirB,dirC)
             := diresDePiezaClaseB(tipoPieza,rotPieza)
       ColocarPzClaseB(codPieza,tipoPieza,rotPieza
                       ,dirA,dirB,dirC)
     }
 }
```

Observar las precondiciones, y el uso de las operaciones que obtienen las direcciones de las piezas de cada clase.

Las operaciones auxiliares simplemente colocan las secciones en los lugares correspondientes utilizando más operaciones auxiliares.

```
procedure ColocarPzClaseA(codPieza,tipoPieza,rotPieza
                          ,dirA,dirB,dirC1,dirC2)
 /*
    PROPÓSITO: completar el trabajo de ColocarPieza
               para las piezas de clase A
    PRECONDICIONES Y OBSERVACIONES:
      * las mismas que para ColocarPieza
      * las direcciones coinciden con el tipoPieza
```

```
 */
  {
    ColocarPivote(codPieza,tipoPieza,rotPieza)
    ColocarSeccionDePiezaEn(codPieza,dirA)
    ColocarSeccionDePiezaEn(codPieza,dirB)
    ColocarSeccionDePiezaEnY(codPieza,dirC1,dirC2)
  }

//-----------------------------------------------------
procedure ColocarPzClaseB(codPieza,tipoPieza,rotPieza
                          ,dirA,dirB,dirC)
 /*
    PROPÓSITO: completar el trabajo de ColocarPieza
               para las piezas de clase B
    PRECONDICIONES Y OBSERVACIONES:
      * las mismas que para ColocarPieza
      * las direcciones coinciden con el tipoPieza
 */
  {
    ColocarPivote(codPieza,tipoPieza,rotPieza)
    ColocarSeccionDePiezaEn(codPieza,dirA)
    ColocarSeccionDePiezaEn(codPieza,dirB)
    ColocarSeccionDePiezaEn(codPieza,dirC)
  }
```

Las operaciones de colocar secciones son similares a las de verificar si hay lugar, con la excepción que pueden asumir como precondición que las celdas donde deben ubicarse las secciones existen, lo cual simplifica un poco el código.

```
procedure ColocarSeccionDePieza(codPieza)
 /*
    PROPÓSITO: coloca una sección de la pieza codPieza
    PRECONDICIONES Y OBSERVACIONES:
      * las mismas que para ColocarPieza
 */
 { PonerN(Verde,codPieza) }

//-----------------------------------------------------
procedure ColocarPivote(codPieza, tipoPieza, rotPieza)
 /*
    PROPÓSITO: coloca el pivote de la pieza codPieza
    PRECONDICIONES Y OBSERVACIONES:
      * las mismas que para ColocarPieza
 */
 {
    ColocarSeccionDePieza(codPieza)
    PonerN(Negro,tipoPieza)
    PonerN(Rojo,rotPieza)
 }

//-----------------------------------------------------
procedure ColocarSeccionDePiezaEn(codPieza,dir)
 /*
    PROPÓSITO: coloca una sección de la pieza codPieza
               en la celda lindante al dir
    PRECONDICIONES Y OBSERVACIONES:
      * las mismas que para ColocarPieza
 */
 {
   Mover(dir)
   ColocarSeccionDePieza(codPieza)
   Mover(opuesto(dir))
 }
```

```
//-----------------------------------------------------
procedure ColocarSeccionDePiezaEnY(codPieza,dir1,dir2)
 /*
    PROPÓSITO: coloca una sección de la pieza codPieza
               en la celda lindante al dir1 y dir2
    PRECONDICIONES Y OBSERVACIONES:
      * las mismas que para ColocarPieza
 */
 {
   Mover(dir1);Mover(dir2)
   ColocarSeccionDePieza(codPieza)
   Mover(opuesto(dir1));Mover(opuesto(dir2))
 }
```

Para quitar una pieza del tablero se procede de manera similar que para colocarla, pero cambiando los `Poner` por `Sacar`. Adicionalmente definiremos un procedimiento denominado `QuitarSeccionDePiezaActual` para quitar una sección independientemente de si se trata del pivote o de otra de las secciones.

```
// procedure QuitarSeccionDePiezaActual()
// procedure QuitarPiezaActual()
//   procedure QuitarPzClaseA(codPieza,tipoPieza,rotPieza
//                            ,dirA,dirB,dirC1,dirC2)
//   procedure QuitarPzClaseB(codPieza,tipoPieza,rotPieza
//                            ,dirA,dirB,dirC)
//     procedure QuitarSeccionDePieza(codPieza)
//     procedure QuitarPivote(codPieza,tipoPieza,rotPieza)
//     procedure QuitarSeccionDePiezaDe(codPieza,dir)
//     procedure QuitarSeccionDePiezaDeY(codPieza,dir1,dir2)
```

La operación de `QuitarPiezaActual` y todas su auxilares se omiten porque son idénticas en estructura a las operaciones explicadas para colocar una pieza, excepto dos diferencias ínfimas. La primera es que todas estas operaciones asumen como precondición que la pieza no está marcada. La segunda es que la operación de `QuitarPiezaActual` no recibe parámetros, por lo que debe leerlos desde la celda actual (y para ello se utilizarán las operaciones definidas en la sección 5.3). Solo daremos el código de la operación `QuitarSeccionDePiezaActual`.

```
procedure QuitarSeccionDePiezaActual()
 /*
    PROPÓSITO: quita una sección de la pieza actual
    PRECONDICIONES:
      * la celda es una sección de pieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   SacarTodasLasDeColor(Verde)
   SacarTodasLasDeColor(Negro)
   SacarTodasLasDeColor(Rojo)
 }
```

Vemos que utiliza tres veces el procedimiento del ejercicio 3.2.12, y de esta manera no tiene que preguntar si es o no una celda pivote.

5.4.3. Movimientos de una pieza

Las operaciones de movimiento de piezas se dividen en tres grupos: las que sirven para mover la pieza, las que sirven para bajar la pieza y las que sirven para rotar la pieza. En los dos primeros grupos tenemos una función que verifica si es posible llevar a cabo el movimiento y un procedimiento que efectivamente lo realiza; en el tercero el procedimiento realiza el movimiento si puede, y si no lo deja sin realizar.

```
// function puedeMoverPiezaActual(dir)
// procedure MoverPiezaActual(dir)
//
// function puedeBajarPiezaActual()
// procedure BajarPiezaActual()
//
// procedure RotarPiezaActual(sentidoHorario)
```

Es **tan importante** este concepto que no nos vamos a cansar de repetirlo.

Para codificar estas operaciones se utilizan muchas de las operaciones definidas en secciones anteriores, mostrando una vez más la utilidad de pensar por subtareas y aprender a usar de manera adecuada la abstracción provista por procedimientos y funciones.

Veamos el código de la función `puedeMoverPiezaActual`.

```
function puedeMoverPiezaActual(dir)
 /*
    PROPÓSITO: determina si la pieza actual se puede
               mover en la dirección dir
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * para saber si puede mover, se la quita y se
        determina si hay lugar en la celda
        correspondiente (si es que existe)
      * si no hay pieza, no la puede mover...
 */
 {
   if (esSeccionPivoteDeAlgunaPieza()
    && puedeMoverEnZonaDeJuego(dir))
    {
     tipoPieza := leerTipoDePiezaActual()
     rotPieza  := leerRotacionDePiezaActual()
     QuitarPiezaActual() // Se puede pues hay pieza
     Mover(dir)          // Se puede pues hay lugar
     puedeM := hayLugarParaPiezaTipo(tipoPieza,rotPieza)
    }
   else
    { puedeM := False }

   return (puedeM)
 }
```

Observar como las precondiciones de las operaciones de lectura, quitar pieza, etc. quedan satisfechas por la condición ya establecida.

Primero se verifica si la pieza actual existe y si hay celda destino para moverla; si eso no sucede, entonces informa que no puede moverla. Si existe, procede a leerla y a verificar que hay lugar para ella en el destino. Para esta verificación quita la pieza para poder invocar la función que determina si hay lugar para una pieza. El hecho de quitar la pieza es necesario porque al moverse, la misma no estará más en la posición anterior, y entonces la función de detección de lugar debe encontrar esas celdas libres.

El procedimiento de `MoverPiezaActual` procede de manera muy similar a la de la función recién explicada.

```
procedure MoverPiezaActual(dir)
 /*
    PROPÓSITO: mover la pieza actual en dirección dir,
               si se puede o nada si no se puede
    PRECONDICIONES:
      * la celda actual está sobre el pivote de una
        pieza
    OBSERVACIONES:
      * para mover una pieza se la quita toda y se la
        pone de nuevo en el nuevo lugar, si hay lugar
      * la celda actual queda en el pivote de la pieza
        ya sea que se movió o no
 */
 {
```

```
   codPieza  := leerCodigoDePiezaActual()
   tipoPieza := leerTipoDePiezaActual()
   rotPieza  := leerRotacionDePiezaActual()
   if (puedeMoverEnZonaDeJuego(dir))
     {
       QuitarPiezaActual()
       Mover(dir) // Puede, porque se verificó
       if (hayLugarParaPiezaTipo(tipoPieza,rotPieza))
         {
           // Coloca la pieza en la nueva posición
           ColocarPieza(codPieza, tipoPieza, rotPieza)
         }
       else
         {
           // Coloca la pieza en la celda inicial
           Mover(opuesto(dir))
           ColocarPieza(codPieza, tipoPieza, rotPieza)
         }
     }
 }
```

La única diferencia apreciable es que si la pieza no se podía mover, entonces la vuelve a colocar en el mismo lugar inicial. La celda actual queda siempre en el pivote de la pieza, tanto si la pieza se pudo mover como si no.

Las operaciones de bajar pieza simplemente especializan las dos operaciones anteriores especificando que el parámetro no es cualquier dirección si no simplemente el valor `Sur`. El código se omite por ser extremadamente simple.

Finalmente, el código de rotar pieza es similar al de mover pieza, salvo que debe calcularse y usarse la nueva rotación.

```
procedure RotarPiezaActual(sentidoHorario)
 /*
    PROPÓSITO: rotar la pieza actual, si se puede
               o nada si no se puede
    PRECONDICIONES:
      * la celda actual está sobre el pivote de una
        pieza
    OBSERVACIONES:
      * para rotar una pieza se la quita toda y se la
        pone rotada,si hay lugar
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   codPieza  := leerCodigoDePiezaActual()
   tipoPieza := leerTipoDePiezaActual()
   rotPieza  := leerRotacionDePiezaActual()
   nuevaRot  := rotar(rotPieza, sentidoHorario)
   QuitarPiezaActual()
   if (hayLugarParaPiezaTipo(tipoPieza,nuevaRot))
     { ColocarPieza(codPieza, tipoPieza, nuevaRot) }
   else
     { ColocarPieza(codPieza, tipoPieza, rotPieza) }
 }
```

Con esto culmina la presentación de las partes básicas del juego. Ahora estamos en condiciones de empezar a establecer la mecánica del mismo.

5.5. Código para la mecánica del juego

La mecánica del juego consta de cuatro partes principales: la operación de agregar una nueva pieza en la zona de juego, la operación que hace que todas las piezas de la zona de juego bajen un lugar (si pueden hacerlo), la operación que calcula cuáles de las piezas

deben volverse parte del piso (porque no pueden bajar más) y la operación que elimina aquellas filas del piso que están llenas. Para cada una de estas operaciones dedicaremos una subsección. Además brindaremos una subsección final con un ejemplo del uso de estas operaciones.

5.5.1. Colocar nueva pieza

La operación para colocar una nueva pieza en el tablero es la más simples de las operaciones de mecánica del juego. Simplemente recibe como parámetros el código y tipo de la pieza a poner y un número que indica en qué columna de la zona de juego debe colocar la pieza, e intenta colocar la pieza allí. Por simplicidad, se asume que las piezas se colocan en la zona de juego en rotación "natural" (o sea, código de rotación 1). El código es el siguiente

```
procedure ColocarNuevaPieza(codPieza,tipoPieza,ubicacion)
 /*
    PROPÓSITO: coloca una nueva pieza de código
               codPieza y tipo tipoPieza en la zona de
               juego, en la columna indicada por
               ubicacion
    PRECONDICIONES:
      * la pieza codPieza no existe en la zona de juego
      * ubicación es mayor o igual a 0 y menor que el
        ancho de la zona de juego
    OBSERVACIONES:
      * se coloca la pieza en la 2da fila desde arriba
        en rotación 1 (posición original)
        (ATENCION: para colocarla en otra rotación hay
          que analizar dónde quedaría el pivote, para
          bajar acorde. En rotación 1 todos los pivotes
          van en fila 2 desde arriba)
      * si no hay lugar, no hace nada
 */
 {
   IrACoordenadaDeZonaDeJuego(ubicacion
                              ,altoDeZonaDeJuego()-1)
   if (tipoPieza /= 7) { Mover(Sur) }
          // Ajuste para estar 1 más abajo
          // pues los pivotes van acá siempre en
          // rotación 1, excepto en la pieza 7 (T)
   if (hayLugarParaPiezaTipo(tipoPieza,1))
     { ColocarPieza(codPieza,tipoPieza,1) }
 }
```

Para generalizar a otras rotaciones habría que agregar a la geometría de las piezas una función que permita calcular la diferencia entre la distancia del pivote a la parte superior en rotación 1 y la distancia del pivote a la parte superior en la nueva rotación, y usar dicha función para ajustar la posición donde se coloca la pieza.

5.5.2. Bajar las piezas

Como se explicó al principio, las piezas "caen" en la zona de juego como si la zona fuera vertical y las piezas estuvieran sujetas (aproximadamente) a las leyes de la gravedad. Decimos que es aproximado porque la caida se produce lentamente, de a una fila por vez, y porque no se utiliza un verdadero algoritmo para simular gravedad (que sería muchísimo más complejo de codificar).

Para simular la acción de "caer", definiremos una operación que baja en un lugar todas las piezas de la zona de juego, `BajarPiezasDeZonaDeJuego`. Esta operación requiere de varias operaciones auxiliares. Las mismas son las siguientes

```
// procedure BajarPiezasDeZonaDeJuego()
//   procedure UnicamenteBajarPiezas()
//   procedure IrAPiezaBajableNoMarcadaSiExiste()
```

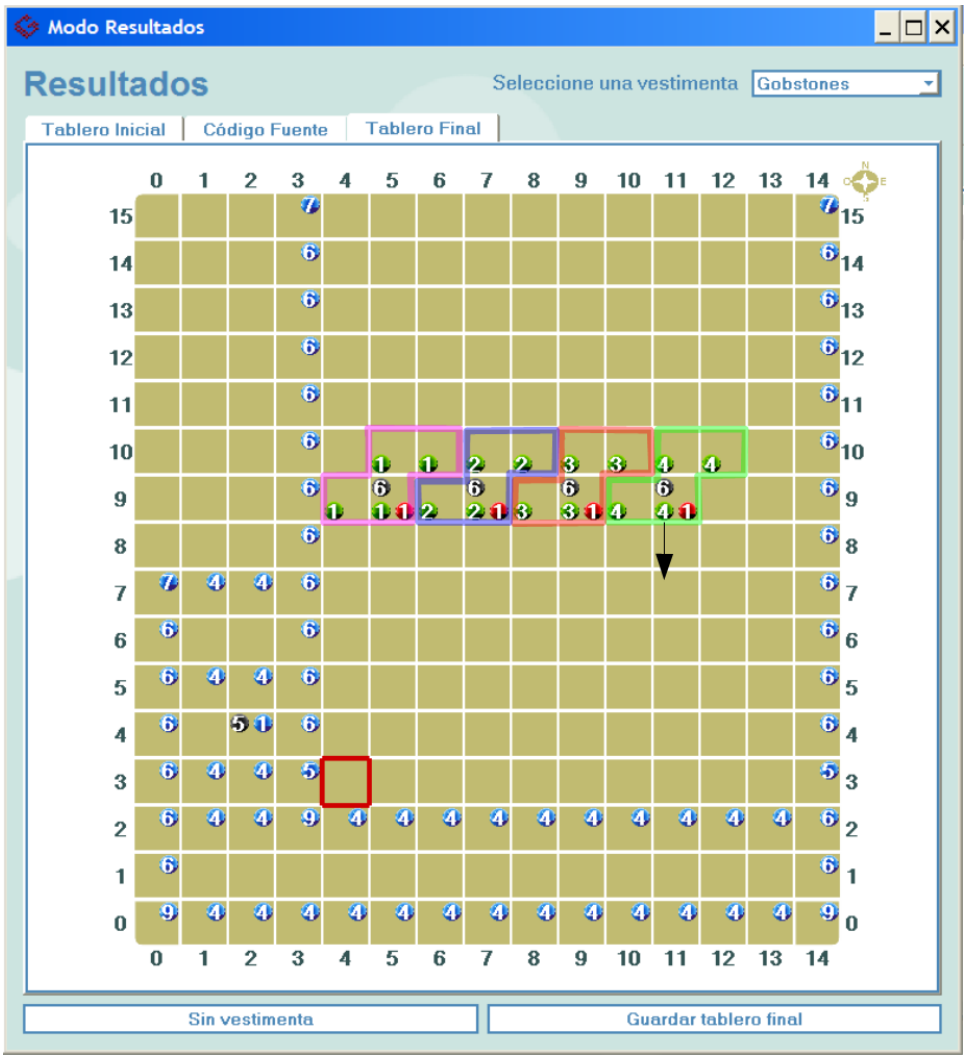
```
//     function esSeccionPivoteDePiezaNoMarcadaBajable()
//     function esCeldaConMarcaDePieza()
//   procedure MarcarPiezaActual()
//   procedure QuitarMarcasDePiezas()
//     procedure DesmarcarPiezaActualSiHay()
```

La operación principal se divide en dos partes: bajar las piezas y extender el piso. El código será el siguiente

```
procedure BajarPiezasDeZonaDeJuego()
 /*
    PROPÓSITO: bajar un lugar todas las piezas que
               pueden bajar
    OBSERVACIONES:
      * al bajar piezas se puede generar nuevo piso
        y por eso se invoca a la operación de extender
        el piso
 */
 {
   UnicamenteBajarPiezas()
   ExtenderElPiso()
 }
```

La operación de bajar únicamente las piezas es bastante menos simple de lo que se espera: debemos recorrer las piezas e irlas bajando, pero puede suceder que unas piezas bloqueen temporariamente a otras. Entonces si hacemos un recorrido simple sobre las piezas no obtendremos los resultados esperados: algunas piezas no habrán bajado a pesar de poder hacerlo. En el gráfico G.5.4 puede observarse una cadena de piezas de manera tal que si hacemos un único recorrido noreste de la zona, solo bajará la pieza más a la derecha, aún cuando sería esperable que bajasen todas juntas. No es solución utilizar otros recorridos simples, porque simplemente podemos utilizar otras piezas simétricas para reproducir el problema. La solución es entonces bajar las piezas de a una, y marcar las piezas para que no bajen más de una vez. Se estructura como un recorrido sobre las piezas no marcadas.

```
//-----------------------------------------------------
procedure UnicamenteBajarPiezas()
 /*
    PROPÓSITO: bajar un lugar todas las piezas que
               pueden bajar
    OBSERVACIONES:
      * luego de bajar una pieza, la misma se marca
        para no volver a procesarla
      * esto es necesario cuando hay varias piezas
        bajables que se enciman y unas bloquean a
        otras para bajar
        Ej: (las piezas son 1,2,3 y 4, todas S)
           11223344
          11223344
        Todas las piezas deberían bajar, pero solo
        la 4 lo hace en una primera vuelta; la
        marca garantiza que solo bajará un lugar
      * se estructura como un recorrido de piezas
        sin marcar
      * el proceso de pasar al siguiente puede
        excederse si no hay más, terminando siempre
        en el final de un recorrido de celdas (sin
        pieza); de ahí que se llame
        "IrAPiezaBajableNoMarcadaSiExiste"
 */
 {
   IrAPiezaBajableNoMarcadaSiExiste()
   while (puedeBajarPiezaActual())
     {
```



G.5.4. Grupo de piezas que impiden que para bajar se use un recorrido simple sobre las piezas: solo la pieza verde puede bajar en una primera pasada

```
      IrAPiezaBajableNoMarcadaSiExiste()
      BajarPiezaActual()
      MarcarPiezaActual()
    }
  QuitarMarcasDePiezas()
}
```

Al iniciar el recorrido, en lugar de preguntar si existe una pieza, se usa un procedimiento que va a la primera que cumpla la condición, si la misma existe. Esto se repite en el procedimiento de avanzar al siguiente, lo cual hace que el recorrido siempre se "caiga" más allá de la última pieza que cumple. De esta forma se optimiza la pregunta, ahorrando un recorrido para verificar que la próxima pieza exista.

Para ir a una pieza no marcada bajable, se recorre toda la zona de juego buscando el pivote de una pieza que no tenga marca y que pueda ser bajada. Se estructura como un recorrido de búsqueda sobre las celdas de la zona de juego; pero dado que puede terminar sin que haya ninguna, debe preguntarse al terminar si efectivamente terminó sobre una pieza como la buscada.

```
procedure IrAPiezaBajableNoMarcada()
 /*
    PROPÓSITO: va a un pieza de la zona
               de juego que puede ser bajada
               y no tiene marca
    OBSERVACIONES:
     * se estructura como un recorrido de búsqueda
       sobre las celdas de la zona de juego
     * si no hay pieza, no hay marca y no puede bajar
       con lo cual sigue buscando
     * si para, es porque encontró una pieza no
       marcada que puede bajar o porque se terminaron
       las celdas sin que haya ninguna
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while(not esFinDelRecorridoNEDeZonaDeJuego()
         && not esSeccionPivoteDePiezaNoMarcadaBajable())
     { AvanzarEnRecorridoNEDeZonaDeJuego() }
 }
```

En esta última operación se usa una función de detección del pivote de una pieza marcada, que verifica tres condiciones: que haya una sección de pivote, que esté marcada, y que la pieza en esa posición pueda bajar.

```
function esSeccionPivoteDePiezaNoMarcadaBajable()
 /*
    PROPÓSITO: informa si la celda actual es celda
               pivote de una pieza no marcada que
               puede bajar
 */
 {
   return (esSeccionPivoteDeAlgunaPieza()
           && not esCeldaConMarcaDePieza()
           && puedeBajarPiezaActual())
 }
```

Observar como se usa el circuito corto de las operaciones booleanas para estar seguros que la precondición de la última condición se cumple. La operación para verificar si una celda tiene marca de piezas es simple

```
function esCeldaConMarcaDePieza()
 /*
    OBSERVACIONES:
      la marca de 7 se superpone con la
      codificación de giro (de 1 a 4), por eso
      pregunta por mayor
```

```
 */
 { return (nroBolitas(Rojo)>7) }
```

Solo faltan las operaciones de marcado para completar las operaciones de esta sección. Como vimos en la presentación, las piezas se marcan con 7 bolitas rojas. Es importante que las marcas no interfieran con la codificación de otras partes del juego. El procedimiento de marcar piezas es extremadamente simple

```
procedure MarcarPiezaActual()
 /*
    PROPÓSITO: marcar la pieza actual
    PRECONDICIONES:
      * la celda actual es el pivote de una pieza
        y la misma no está marcada
    OBSERVACIONES:
      * se marca con 7 bolitas rojas (verificar que
          otras marcas no usen esta misma codificación)
 */
 { PonerN(Rojo,7) }
```

Para quitar las marcas de todas las piezas ya marcadas se usa un recorrido de las celdas de la zona de juego

```
procedure QuitarMarcasDePiezas()
 /*
    PROPÓSITO: quita todas las marcas de las piezas
    OBSERVACIONES:
      * se estructura como un recorrido sobre las
        celdas de la zona de juego, quitando todas
        las marcas de pieza
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while (not esFinDelRecorridoNEDeZonaDeJuego())
    {
      // Procesar es sacar la marca de pieza, si existe
      DesmarcarPiezaActualSiHay()
      AvanzarEnRecorridoNEDeZonaDeJuego()
    }
   // Procesar último
   DesmarcarPiezaActualSiHay()
 }
```

La operación de procesar la celda actual debe simplemente verificar si hay una pieza marcada y retirar la marca; no hace nada en caso que no haya marca.

```
procedure DesmarcarPiezaActualSiHay()
 /*
    PROPÓSITO: desmarcar la pieza actual, si existe
               y está marcada; si no, no hacer nada
    OBSERVACIONES:
      * se marca con 7 bolitas rojas (verificar que
          otras marcas no usen esta misma codificación)
 */
 { if (nroBolitas(Rojo)>7) { SacarN(Rojo,7) } }
```

5.5.3. Extender el piso

Cuando las piezas no pueden bajar más, se vuelven parte del piso. El piso se representa con 8 bolitas azules. La operación de extender el piso controla todas las celdas de la zona de juego, para verificar si contienen una pieza que debe transformarse en piso. Se utilizan varias operaciones auxiliares para esta operación.

```
// procedure ExtenderElPiso()
//   procedure MarcarTodasLasCeldasDelPiso()
```

```
//   procedure ExtenderElPisoEnLaFilaBase()
//     procedure MarcarLaPosicionDeBase()
//     procedure IrAMarcaDePosicionDeBaseYDesmarcar()
//     function esPiso()
//     function hayPisoAl(dir)
//     procedure PonerPiso()
//     function esCeldaDePisoMarcada()
//     procedure MarcarElPiso()
//     procedure DesmarcarElPiso()
//     procedure IrACeldaDePisoMarcadaSiExiste()
//
//   procedure TransformarEnPisoSiEsPieza(marca)
//     procedure TransformarEnPisoPiezaActual(marca)
//     procedure TransfPzClaseA(codPieza,tipoPieza,rotPieza
//                              ,dirA,dirB,dirC1,dirC2, marca)
//     procedure TransfPzClaseB(codPieza,tipoPieza,rotPieza
//                              ,dirA,dirB,dirC, marca)
//     procedure TransformarCeldaEnPiso(marca)
//     procedure TransformarCeldaEnPisoAl(dir, marca)
//     procedure TransformarCeldaEnPisoAlY(dir1,dir2, marca)
```

La operación de extender el piso es una de las más complicadas del juego, ya que debe detectar todas las piezas que están en contacto con el piso y transformarlas a su vez en piso (y si, luego de transformar una pieza, el nuevo piso está a su vez en contacto con otra pieza, esta última debe transformarse también, y continuando si hay más piezas en contacto con esta, etcétera). Para esto se utiliza un mecanismo de marcas de la siguiente manera: se marcan todas las piezas y luego se van procesando de a una; al transformar en piso una pieza se la transforma en piso marcado, así se garantiza que sus celdas aún no han sido procesadas. El código es el siguiente

```
procedure ExtenderElPiso()
 /*
    PROPÓSITO: analiza si hay nuevas posiciones que pueden
               estar en el piso, y las convierte en piso
    OBSERVACIONES:
      * no hace falta procesar el último elemento pues
        seguro está en la fila más al Norte y no puede
        tener nada encima
      * se estructura como un recorrido NE sobre celdas
      * para cada celda de piso, se transforma la pieza
        al Norte en piso (este nuevo piso puede producir
        la transformación de más celdas)
 */
 {
   ExtenderElPisoEnLaFilaBase()
   MarcarTodasLasCeldasDelPiso()
   IrACeldaDePisoMarcadaSiExiste()
   while (esCeldaDePisoMarcada())
     {
       DesmarcarElPiso()
       if (puedeMoverEnZonaDeJuego(Norte))
         {
           Mover(Norte)
           TransformarEnPisoSiEsPieza(True)
                                  //-^- Piso marcado
               // La pieza se transforma en piso marcado!
               // Esta operación mueve el cabezal de lugar
         }
       IrACeldaDePisoMarcadaSiExiste()
     }
   EliminarFilasLlenas()
 }
```

Antes de realizar el marcado de las celdas del piso, se procede a transformar las piezas que estén en contacto con el borde Sur de la zona, ya que deben transformarse en piso pero no necesariamente están en contacto con alguna celda del piso. Para detectar si aún hay celdas marcadas para recorrer se utiliza un esquema similar al que se usó para detecar si hay piezas marcadas bajables: se mueve a la próxima si existe *antes* de preguntar; si pudo moverse a tal pieza, se ingresa en el cuerpo de la repetición, y si no, quiere decir que no hay más celdas marcadas, por lo que debe terminar.

El procesamiento de una celda de piso marcada consiste en considerar la celda inmediatamente al norte de la misma, y transformar la pieza que pueda encontrarse allí en piso (si existe). Esta pieza se debe transformar en piso marcado, para que dichas celdas sean consideradas en posteriores iteraciones. El hecho de que el procedimiento de transformar en piso una pieza desplace el cabezal no afecta el recorrido, pues el procedimiento de pasar al siguiente busca cualquier celda con marca de piso que falte procesar.

El procedimiento `MarcarTodasLasCeldasDelPiso` consiste en un recorrido sobre las celdas de la zona de juego, marcando cada una de las celdas que son del piso. Para detectar esto utiliza la función `esPiso` y el procedimiento `MarcarPiso`. El código es el siguiente

```
procedure MarcarTodasLasCeldasDelPiso()
 /*
    PROPÓSITO: marca todas las celdas del piso en
               la zona de juego
    OBSERVACIONES:
      * se estructura como un recorrido sobre las celdas
        de la zona de juego
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while (not esFinDelRecorridoNEDeZonaDeJuego())
    {
      if (esPiso()) { MarcarElPiso() }
      AvanzarEnRecorridoNEDeZonaDeJuego()
    }
   if (esPiso()) { MarcarElPiso() }
 }
```

Las operaciones auxiliares de esta operación, al igual que las demás operaciones sobre el piso, son extremadamente sencillas y se dejan como ejercicio.

Actividad de Programación 7

Realice el ejercicio 5.5.1 y constate que las definiciones son correctas cotejando con el código del anexo B.

Ejercicio 5.5.1. *Escribir las siguientes operaciones*

1. *funciones* `esPiso` *y* `hayPisoAl` *que detectan si hay una celda de piso (sin marcar) en la celda actual, y en la celda lindante en la dirección dada (suponiendo que existe), respectivamente;*
2. *un procedimiento* `PonerPiso` *que ponga piso en la celda actual, suponiendo que está vacía;*
3. *procedimientos* `MarcarPiso` *y* `DesmarcarPiso` *que marquen el piso (con 60 bolitas azules) y lo desmarquen (suponiendo que está marcado), respectivamente;*
4. *una función* `esCeldaDePisoMarcada` *que informa si la celda actual es una celda de piso con marca.*

El procedimiento `IrACeldaDePisoMarcadaSiExiste` consiste en un recorrido de búsqueda sobre las celdas del piso, deteniéndose o bien cuando se acaban las celdas (porque ninguna contiene marca) o bien cuando encuentra una celda marcada. El código no reviste ninguna complejidad y por lo tanto lo omitimos.

El procedimiento `ExtenderElPisoEnLaFilaBase` consiste en un recorrido sobre las celdas de la fila base de la zona de juego (la fila que contiene el origen de dicha zona), procesando cada pieza con `TransformarEnPisoSiEsPieza`. En este caso no debe transformarse en piso marcado porque se utiliza en `ExtenderElPiso` justo antes de marcarlo. Un problema que se presenta es que el procedimiento `TransformarEnPisoSiEsPieza` mueve el cabezal de lugar y altera entonces el recorrido de las celdas de la base; por ello, se hace necesario marcar el piso *antes* de transformar la pieza, y luego de haberla transformado volver a esa marca. Sin embargo, dado que la celda actual del recorrido está ocupada por una pieza que va a ser reemplazada por piso, la marca no puede ser colocada en dicha celda; para solucionar esto se marca en la celda inmediatamente inferior, que por tratarse de la base, es una celda del borde del tablero. El código es el siguiente

```
procedure ExtenderElPisoEnLaFilaBase()
 /*
    PROPÓSITO: analiza si hay piezas en la fila
               base que puedan estar en el piso, y las
               convierte en piso
    OBSERVACIONES:
      * se estructura como un recorrido sobre las
        celdas de la fila base
      * las piezas se transforman a piso sin marcar
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while (puedeMoverEnZonaDeJuego(Este))
     {
       MarcarLaPosicionDeBase()
          // Necesario para volver al mismo lugar

       TransformarEnPisoSiEsPieza(False)
                              //-^- Piso sin marcar
       // Esta operación mueve el cabezal de lugar

       IrAMarcaDePosicionDeBaseYDesmarcar()
          // Vuelve al mismo lugar para continuar
          //  el recorrido
       Mover(Este)
     }
   TransformarEnPisoSiEsPieza(False)
      // Esta operación mueve el cabezal de lugar
 }
```

Los procedimientos `MarcarPosicionDeLaBase` para poner una marca a la que volver, y `IrAMarcaDePosicionDeBaseYDesmarcar` para volver a ella son sencillos:

```
procedure MarcarLaPosicionDeBase()
 /*
    PROPÓSITO: marca el piso DEBAJO de la línea de base
               para poder volver.
    OBSERVACIÓN: ¡no marca en la misma posición, porque
                 la misma va a cambiar!
 */
 {
   Mover(Sur)
   MarcarElPiso()
   Mover(Norte)
 }

//-----------------------------------------------------
procedure IrAMarcaDePosicionDeBaseYDesmarcar()
 /*
    PROPÓSITO: ejecuta la acción de volver a una posición
               de piso marcada en la fila base
    OBSERVACIÓN: la marca está en la base y no en la misma
```

```
                posición, porque el piso cambió ahí
 */
 {
  IrAPrimerCeldaNEConBolitas(Azul, 64)
  DesmarcarElPiso()
  Mover(Norte)
 }
```

Simplemente tienen en cuenta moverse al sur para marcar (para no marcar en el mismo lugar) y al norte al volver y por lo demás utilizan las subtareas de marcado ya definidas.

El procedimiento `TransformarEnPisoSiEsPieza` recibe un booleano que indicará si el piso en el que se transforme la pieza debe marcarse o no; si recibe True quiere decir que debe colocarse una marca. Este parámetro se irá pasando hasta el momento de transformar la pieza en piso.

```
procedure TransformarEnPisoSiEsPieza(marca)
 /*
    PROPÓSITO: transforma en piso a la pieza que
               interseccíona con la celda actual,
               si existe, y agrega la marca si
               corresponde
    OBSERVACIONES:
      * si no hay pieza, entonces no hace nada
 */
 {
   if (esSeccionDeAlgunaPieza())
    {
      IrAPiezaSiExiste(leerCodigoDePiezaActual())
      TransformarEnPisoPiezaActual(marca)
    }
 }
```

El procedimiento `TransformarEnPisoPiezaActual` trabaja de manera similar a los procedimientos de `ColocarPieza` y `QuitarPieza`: determina la clase de la pieza, obtiene sus direcciones y utiliza funciones auxiliares que completan el trabajo transformando cada sección de la pieza indicada por las direcciones obtenidas. El código se omite pues es prácticamente idéntico al de los otros procedimientos similares (con la excepción de que debe avisar si el piso debe transformarse marcado o no). El único procedimiento que tiene alguna variación es el de `TransformarCeldaEnPiso`, cuyo código es el siguiente

```
procedure TransformarCeldaEnPiso(marca)
 /*
    PROPÓSITO: transforma en piso la celda actual
               y agrega la marca si corresponde
    PRECONDICIONES:
      * la celda es sección de una pieza
    OBSERVACIONES:
      * el piso se indica con 8 bolitas azules
 */
 {
   QuitarSeccionDePiezaActual()
   PonerPiso()
   if (marca) { MarcarElPiso() }
 }
```

La última acción luego de extender el piso es eliminar las filas que hayan quedado llenas. Esta operación se describe a continuación.

5.5.4. Eliminar filas llenas

Una vez que todas las piezas que debían transformarse en piso lo hicieron, puede resultar que alguna de las filas de piso quede llena. Una de las reglas del ZILFOST es que las filas llenas desaparecen, haciendo que las restantes celdas de piso desciendan un lugar (pero no así las piezas). El propósito del procedimiento `EliminarFilasLlenas` es exactamente

el de hacer desaparecer todas las filas de piso llenas que haya en la zona de juego. Para su tarea utiliza varias operaciones auxiliares.

```
// procedure EliminarFilasLlenas()
//   procedure EliminarMientrasSigaLlena()
//   function esFilaLlena()
//   procedure BajarFilasSobreEsta()
//   procedure BajarFilaSuperior()
//   procedure VaciarDePisoLaFilaActual()
//   procedure QuitarPiso()
```

El código de `EliminarFilasLlenas` se estructura como un recorrido de las filas de la zona de juego, desde el sur hacia el norte. Cuando encuentra una fila llena, baja todas las superiores sobre ella y continúa bajando hasta que la fila actual no está más llena. El código es entonces bastante directo

```
//-----------------------------------------------------
procedure EliminarFilasLlenas()
 /*
    PROPÓSITO:
      eliminar todo el piso de las filas llenas de
      la zona de juego, bajando el piso de las superiores
      a la eliminada
    PRECONDICIONES:
      * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * debe ir de abajo-arriba para no repetir trabajo
      * se estructura como un recorrido sobre filas
  */
 {
   IrAlOrigenDeZonaDeJuego()
   while(puedeMoverEnZonaDeJuego(Norte))
     {
       EliminarMientrasSigaLlena()
       Mover(Norte)
     }
   // Procesa la fila más al Norte
   EliminarMientrasSigaLlena()
 }
```

El único procedimiento nuevo es `EliminarMientrasSigaLlena`, que procesa la fila actual. En este caso su código es muy simple: baja las filas superiores hasta que la actual no esté llena.

```
procedure EliminarMientrasSigaLlena()
 /*
    PROPÓSITO:
      eliminar las secciones de piso de la fila
      actual bajando las de piso que están sobre ella,
      hasta que la fila actual no esté llena
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * al bajar las filas sobre la actual, se borra
        la que estaba, y la condición puede cambiar
      * al terminar, la fila actual no está llena
  */
 {
   while (esFilaLlena())
    { BajarFilasSobreEsta() }
 }
```

Es importante observar que puesto que la cantidad de filas es finita, y la fila superior desaparece cuando está llena, entonces este procedimiento debe terminar.

La función `esFilaLlena` es un recorrido de totalización donde se utiliza un booleano como acumulador, el cual cambia a False apenas detecta que hay una celda que no es piso en la fila actual (deteniendo la búsqueda). Cuando termina, si el booleano sigue en True significa que la fila no contiene celdas diferentes del piso (está llena).

```
function esFilaLlena()
 /*
    PROPÓSITO:
      determinar si la fila actual está llena de piso
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * la variable esLlena indica si hasta el momento
        se encontró evidencia que la fila no está
        llena
      * se estructura como un recorrido de búsqueda
        sobre las celdas de la fila actual
  */
 {
   IrAlBordeDeZonaDeJuego(Oeste)
   esLlena := True
   while(esLlena && puedeMoverEnZonaDeJuego(Este))
     {
       esLlena := esLlena && esPiso()
       Mover(Este)
     }
   esLlena := esLlena && esPiso()
   return(esLlena)
 }
```

La pregunta sobre si es llena antes de saber si puede mover optimiza levemente el código. La operación de conjunción produce el resultado deseado de volver False el valor de la variable en cuánto una celda no es piso.

El procedimiento para bajar las filas encima de la actual, borrando la actual, se estructura como un recorrido de las filas de la zona de juego que están al norte de la actual. Debe bajar *todo* el piso que se encuentre al norte de la fila actual. Puesto que el cabezal debe permanecer en la misma fila en la que empezó (aunque no necesariamente en la misma columna), y que para bajar otras filas debe moverse, entonces se guarda un número de cuántas filas se desplazó hacia el norte para poder volver al final. Además, el procesamiento de la última fila debe hacerse diferente, pues ella no tiene filas arriba; consiste simplemente en borrar todas las celdas del piso de dicha fila.

```
procedure BajarFilasSobreEsta()
 /*
    PROPÓSITO:
      bajar todas las filas de piso sobre la actual,
      sobreescribiendo la fila actual,
      solo en la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * al bajar, se quita todo el piso de la fila
        más al Norte
      * el cabezal no se debe mover de la fila
        actual (por lo que debe devolvérselo a su
        lugar al terminar)
      * la variables desplazamiento guarda
        cuántos lugares debe volverse al Sur
      * se estructura como un recorrido sobre
        las filas arriba de la actual
  */
 {
   desplazamiento := 0
   while (puedeMoverEnZonaDeJuego(Norte))
```

```
    {
      BajarFilaSuperior()
      desplazamiento := desplazamiento + 1
      Mover(Norte)
    }
  VaciarDePisoLaFilaActual()
  MoverN(Sur, desplazamiento)
}
```

Para bajar la fila superior sobre esta, simplemente se recorren las celdas de la fila actual de izquierda a derecha y se baja la celda al norte. Es un recorrido sencillo y se omite su código. El código que sí se ofrece es el de procesar una celda, bajando la que está sobre ella, pues resulta interesante.

```
procedure BajarCeldaAlNorte()
 /*
    PROPÓSITO:
      baja la celda al norte de la actual,
      si corresponde
    PRECONDICIONES:
      * no está en la fila más al Norte
    OBSERVACIONES:
      * solo baja las celdas de piso; las de piezas
        no se bajan
      * Si al bajar el piso puede romper una pieza,
        entonces es absorvido (no rompe la pieza)
  */
 {
   if (not esSeccionDeAlgunaPieza())
     // Con este if, las piezas no resultan destruidas
     //  por el piso
     {
       if (hayPisoAl(Norte))
         // con este if, las piezas arriba de este
         //  piso no bajan
         { if (not esPiso()) { PonerPiso() } }
       else
         // si arriba no hay piso (está vacío o hay
         //  pieza), debe vaciarse porque las celdas
         //  de pieza no bajan
         { VaciarCelda() }
     }
 }
```

Como está establecido en las observaciones del contrato del procedimiento, al bajar la celda al norte, debemos verificar varias condiciones. Primero que nada, si la celda actual contiene una sección de pieza, entonces no debe alterarse (o sea, si hay piso sobre ella, el piso será absorvido). En caso que la celda no tenga una sección de pieza, debe verificarse la celda inmediatamente al norte (que sabemos que existe, por la precondición), y si hay piso al norte, entonces la celda actual debe terminar siendo piso, ya sea que ya había uno, o porque agregamos uno si no había; si en cambio la celda al norte está vacía o hay pieza (o sea, no hay piso), entonces la celda actual debe vaciarse.

Para completar las operaciones de eliminar filas de piso llenas, vemos el procedimiento `VaciarDePisoLaFilaActual`. El mismo se estructura como un recorrido de las celdas de la fila actual, quitando el piso de aquellas celdas que lo contengan. Dado que tanto este código como el de la operación auxiliar `QuitarPiso` son extremadamente simples, los mismos serán omitidos.

5.5.5. Generación del logo de ZILFOST

Todas las operaciones definidas hasta el momento nos permitirán controlar la mecánica de las piezas en la zona de juego. Antes de continuar con las operaciones de interfaz que permitirán que un usuario juegue (o simular el juego de un usuario ficticio), mostraremos el

procedimiento utilizado para generar el gráfico G.5.1. Este procedimiento es muy sencillo, pues simplemente combina operaciones anteriormente definidas de manera secuencial.

```
procedure GenerarLogoZILFOST()
 /*
    PROPÓSITO: Dibuja ZILFOST con piezas
    PRECONDICIONES:
     * la zona de juego tiene un ancho mínimo de 9
 */
 {
    VaciarZonaDeJuego()
    ColocarNuevaPieza(1,1,1)
    ColocarNuevaPieza(6,6,6)
    BajarPiezasDeZonaDeJuego()
    BajarPiezasDeZonaDeJuego()
    ColocarNuevaPieza(3,3,3)
    ColocarNuevaPieza(5,5,5)
    BajarPiezasDeZonaDeJuego()
    BajarPiezasDeZonaDeJuego()
    ColocarNuevaPieza(7,7,7)
    BajarPiezasDeZonaDeJuego()
    ColocarNuevaPieza(2,2,2)
    ColocarNuevaPieza(4,4,4)
    BajarPiezasDeZonaDeJuego()
    BajarPiezasDeZonaDeJuego()
    BajarPiezasDeZonaDeJuego()
}
```

La operación de `VaciarZonaDeJuego` es simplemente un recorrido sobre las celdas de la zona de juego, donde el procesamiento consiste en vaciarlas.

5.6. Código para las operaciones de interfaz

Toda la mecánica del juego se accede con una interfaz específica. Estas operaciones determinan las acciones posibles que se pueden realizar en el juego, que serán combinación de acciones básicas de la mecánica y algunos otros elementos. En el caso de este juego encontramos dos grupos de operaciones de interfaz: las necesarias para proveer cierto grado de "aleatoriedad" en la secuencia de piezas y las operaciones específicas de interacción con el usuario. Describiremos cada grupo en una subsección diferente.

5.6.1. Determinar la próxima pieza

Para que el juego resulte interesante es necesario que la secuencia de piezas no sea siempre predecible. Para lograr esto, y puesto que GOBSTONES no tiene primitivas para generar números seudoaleatorios, se hace necesario codificar operaciones para esta tarea.

Números seudoaleatorios

Un *generador de números aleatorios* es un algoritmo o dispositivo que sirve para procudir una secuencia de números que no siguen ningún esquema prefijado, o sea, que son, o parecen ser, aleatorios. Existen muchos métodos para generar números aleatorios, pero aunque muchos consiguen varias características de imprevisibilidad, la mayoría a menudo falla en ofrecer verdadera alteatoriedad. En aplicaciones que requieren de aleatoriedad de manera fundamental (como por ejemplo, aplicaciones de seguridad informática) se prefieren generadores de números verdaderamente aleatorios, los cuales involucran hardware además de software. Pero en aplicaciones de simulación y en algunos juegos, esto es demasiado costoso.

Un *generador de números seudoaleatorios* es un procedimiento computacional que, basado en un dato inicial (conocido como *semilla*), produce una secuencia larga de números que aparecen a primera vista como aleatorios. No es un verdadero generador de

números aleatorios, pues la secuencia puede ser totalmente predecida conociendo la semilla; pero eligiendo bien el algoritmo pueden dar la impresión de aleatoriedad en muchas aplicaciones, como por ejemplo juegos y simulaciones. Y puesto que dadas las mismas condiciones iniciales (la misma semilla), la secuencia de números resulta la misma, los generadores de números seudoaleatorios ofrecen la ventaja de que se puede volver a reproducir un escenario dado, lo cual constituye una característica deseable en este tipo de aplicaciones.

Para Ampliar

Para saber más sobre generación de números aleatorios, su importancia, la comparación entre generadores de verdadera aleatoriedad y generadores seudoaleatorios existen diversas fuentes. Para este libro utilizamos dos fuentes principales:

La organización RANDOM.org, donde puede consultarse el siguiente link: http://www.random.org/randomness/

La Wikipedia, especialmente las entradas https://en.wikipedia.org/wiki/Random_number_generation, https://en.wikipedia.org/wiki/Pseudorandom_number_generator y https://en.wikipedia.org/wiki/Linear_congruential_generator.

Uno de los métodos más comunes de generación de números seudoaleatorios se conoce como *generador lineal congruente*, y utiliza una ecuación de recurrencia con la siguiente forma:

$$X_{i+1} = (aX_i + b) \mod m$$

donde X_0 es la semilla original, y a, b y m son los parámetros del generador. Este tipo de generadores es capaz de generar un máximo de m números diferentes en una secuencia que aparentemente carece de patrón preestablecido.

El generador de números seudoaleatorios que utilizamos para el juego de ZILFOST se debe a Lewis, Goodman y Miller, quiénes sugirieron los valores $a = 7^5$, $b = 0$ y $m = 2^{31} - 1$ para los parámetros de un generador lineal congruente. Este generador ha sido ampliamente estudiando desde entonces, y en 1988, Stephen Park and Keith Miller propusieron este algoritmo como un estándar [Park and Miller, 1988], y discutieron un par de implementaciones en C y BASIC. La importancia del algoritmo reside en que puede ser implementado con aritmética de 32 bits, e igualmente produce excelentes resultados. Nosotros entramos en contacto con este algoritmo gracias a un artículo de Mark Jones [Jones, 1996], que lo implementa en el lenguaje funcional HASKELL. Allí muestra además cómo implementarlo con aritmética de 32 bits y cómo obtener números en un rango de 0 a un máximo dado. El código resultante es el siguiente

```
function randomEntre0YConSemilla(maximo,semilla)
 /*
    PROPÓSITO: calcula en base a una semilla dada (x_i),
               un número seudoaleatorio entre 0 y el
               máximo dado, y una nueva semilla (x_{i+1})
               a ser usada en invocaciones futuras
    OBSERVACIONES:
      * este código fue copiado del artículo "Functional
        Programming with Overloading and Higher-Order
        Polymorphism" de Mark Jones, publicado en "Advanced
        Functional Programming", J.Jeuring y E.Meijer, editores,
        LNCS 925, Springer Verlag, 1995.
      * para usarlo correctamente, la nuevaSemilla debe ser
        usada como siguiente semilla en posteriores llamados,
        como en
         semilla := 42
```

```
        repeat(17)
         {
           (n,semilla) := randomEntre0YConSemilla(10,semilla)
           PonerN(Verde,n)
           Mover(Norte)
         }
 */
 {
   nuevaSemilla := min_stand(semilla)
   return(nuevaSemilla mod maximo, nuevaSemilla)
 }

//-----------------------------------------------------
function min_stand(semilla)
 /*
    PROPÓSITO: calcula un número seudoaleatorio según una
               semilla dada
    OBSERVACIONES:
      * auxiliar para randomEntre0YConSemilla(maximo, semilla)
      * Mark Jones lo atribuye a "Random Number Generators: Good
        Ones are Hard to Find" de S.K.Park y K.W.Miller, publicado
        en la revista "Communications of the ACM", 31(10):1192-1201,
        en octubre de 1988.
      * este artículo en realidad lo toma de una propuesta de 1969
        por Lewis, Goodman and Miller y lo propone como estándar
        mínimo de generación de números seudoaleatorios
      * el comentario sobre su funcionamiento fue agregado por mí,
        en base a alguna lectura que encontré alguna vez que lo
        explicaba, y de la que no recuerdo la cita:
          x_{i+1} = a*x_i mod m
        donde
          a = 7^5 = 16807
          m = 2^31 - 1 = 2147483647
          q = m div a = 127773
          r = m mod a = 2836
        y entonces
          x_{i+1} = a*(x_i mod q) - r*(x_i div q) + delta*m
        siendo
          delta = 1 si (a*(x_i mod q) - r*(x_i div q) > 0)
          delta = 0 si no
 */
 {
   hi := semilla div 12773  -- semilla div (2^31 mod 7^5)
   lo := semilla mod 12773  -- semilla mod (2^31 mod 7^5)
   preresultado := 16807 * lo - 2836 * hi
                --  7^5  * lo - (2^31 mod 7^5) * hi
   if (preresultado > 0) { delta := 0 }
   else                  { delta := 1 }
   return (preresultado + delta * 2147483647)
                     --  delta * 2^31
 }
```

Puede observarse que para utilizarlo correctamente, la primera vez debe suministrarse una semilla inicial, y luego, en cada nuevo uso, debe suministrarse la semilla devuelta por la iteración anterior. Esto es así para satisfacer el requerimiento de que cada nueva semilla se calcula en base a la anterior, y es necesario para garantizar el correcto funcionamiento del algoritmo.

Actividad de Programación 8

Agregue los dos procedimientos de generación de números seudoaleatorios a su `Biblioteca`, puesto que constituyen un recurso importante y general.

Generación de piezas

Habiendo presentado las ideas de generación de números seudoaleatorios, estamos en condiciones de codificar la generación de nuevas piezas. Este código tomará como parámetro una semilla que servirá para alimentar al generador de números de la biblioteca, y producirá 4 datos:

- el código de la próxima pieza,
- el tipo de la próxima pieza,
- la columna en la que debe introducirse la próxima pieza y
- la nueva semilla para continuar la generación de números en el futuro.

El primer dato se puede leer simplemente de la sección de próxima pieza del tablero (ver sección 5.1), para lo que pueden usarse las operaciones definidas en la subsección 5.3.1. Para los siguientes 2 datos se utilizará el generador de números seudoaleatorios, y el último de ellos será producido por este mismo proceso. El código resultante es el siguiente

```
function determinarNuevaPieza(semilla)
 /*
    PROPÓSITO: devolver código, tipo y ubicación para
               una nueva pieza de manera seudorandómica
               en base a una semilla. También devuelve
               la nueva semilla
    OBSERVACIONES:
      * la semilla de parámetro se usa como semilla inicial
      * el resultado de cada número seudorandómico generado
        se usa como próxima semilla, y el final se retorna
 */
 {
   (ubicacion,nuevaSemilla)
      := randomEntre0YConSemilla(anchoDeZonaDeJuego()-1
                                 ,semilla)
   (tipoPieza,nuevaSemilla)
      := randomEntre0YConSemilla(6
                                 ,nuevaSemilla)
   return(leerZonaDeProximaPieza(), tipoPieza+1
          ,ubicacion                , nuevaSemilla)
 }
```

Puede verse que el tipo de pieza se acota entre 0 y 6, y luego se ajusta sumándole 1. Lo mismo con respecto a la ubicación, que se acota por el ancho de la zona de juego, salvo que en este caso no hace falta ajustarlo, pues la primera columna de la zona de juego se numera como 0. La variable `nuevaSemilla` guarda los valores intermedios de las semillas producidas por el generador, siendo el último valor de la misma el que debe devolverse para futuros usos. Es importante observar como se aplica el esquema correcto de uso del generador de números seudoaleatorios para garantizar que la semilla suministrada siempre es la última producida por el generador. Este patrón debe repetirse al utilizar esta operación.

5.6.2. Operaciones de interacción

Las operaciones de interacción son aquellas que se invocan desde la interfaz de usuario y que sirven para expresar las alternativas de interacción que el usuario tiene con el sistema. Expresan las opciones posibles con las que el usuario se encuentra cuando interactúa con el sistema, y constituyen los puntos de acceso al resto del programa, facilitando la separación de las diferentes partes en unidades autocontenidas.

En el caso del ZILFOST, las operaciones de interacción son cinco

```
// procedure OperacionColocarNuevaPieza()
// procedure OperacionMoverPiezaAl(dir)
// procedure OperacionRotarPieza(sentidoHorario)
// procedure OperacionAgregarDigitoASeleccion(dig)
// procedure OperacionBajarPiezas()
```

La operación de colocar nueva pieza combina las operaciones de determinar nueva pieza con la mecánica de la semilla para números seudoaleatorios, y luego coloca efectivamente la pieza en la zona de juego. Las operaciones de mover y rotar pieza sirven para interactuar con una pieza específica, que deben encargarse de determinar cuál es. La operación de agregar dígitos a la selección se encarga de ir completando la lectura de dígitos para saber sobre cuál pieza deben trabajar las operaciones anteriores. Finalmente la opción de bajar piezas se encarga de ir bajando las piezas ante determinados eventos.

Para la operación de colocar una nueva pieza, tenemos que determinar cómo se mantendrá de manera global la semilla de números seudoaleatorios, y utilizar esta forma para interactuar con las operaciones de la mecánica del juego. Dado que el tipo de interacción que imaginamos para el juego no involucra recordar valores de variables entre operaciones de interfaz, debemos encontrar una manera de recordar este valor que no utilice variables. Dicha manera viene provista por el tablero. Puesto que los números a recordar son grandes (del orden de los 10 dígitos) y en las implementaciones actuales de GOBSTONES la operación de poner muchas bolitas toma un tiempo directamente proporcional al número de bolitas a poner, se elige utilizar una representación por dígitos que pueda ser grabada y leída en el tablero de manera eficiente. Este es el propósito de la zona de semilla que fuera descrita en la sección 5.1. Entonces, el código para esta operación resulta simplemente de la combinación adecuada de operaciones anteriores

```
procedure OperacionColocarNuevaPieza()
 /*
    PROPÓSITO:
      combinar las acciones necesarias para la aparición
      de una nueva pieza en la zona de juego (si entra)
  */
 {
   semilla := leerSemilla()
   (codPz,tipoPz,ubicPz,semilla) := determinarNuevaPieza(semilla)
   GrabarSemilla(semilla)
   ColocarNuevaPieza(codPz,tipoPz,ubicPz)

   // Al exceder el máximo, vuelve a 1
   IncrementarZonaDeProximaPieza()
   BorrarZonaDeSeleccion()
 }
```

Vemos que se utilizan las operaciones de `leerSemilla` de la zona de semilla para invocar a la operación de `determinarNuevaPieza`, y de `GrabarSemilla` en dicha zona una vez que se utilizó la semilla y se tiene la nueva. Vemos también que los datos obtenidos por `determinarNuevaPieza`, accesibles a través de la asignación simultánea, se utilizan en la operación de `ColocarNuevaPieza`. Finalmente se incrementa la cuenta de la cantidad de piezas sobre el tablero y se borra la zona de selección. De esta manera, podemos ver que modificando en el tablero inicial el número representado en la zona de semilla podemos variar la secuencia de piezas que aparecen en el juego. Además vemos que luego de que una nueva pieza es colocada, la zona de selección queda vacía.

Las operaciones de movimiento y rotación de piezas utilizan el número codificado en la zona de selección para determinar sobre qué pieza del tablero deben trabajar, y luego usan la operación correspondiente para hacerlo, suministrando los parámetros correspondientes. El código de ambas es muy similar; vemos solo la de mover pieza

```
procedure OperacionMoverPiezaAl(dir)
 /*
    PROPÓSITO:
      combinar las acciones necesarias para mover la
      pieza indicada en la zona de selección (si existe)
  */
 {
   IrAPiezaSiExiste(leerZonaDeSeleccion())
   if (hayPiezaActual())
     { MoverPiezaActual(dir); ExtenderElPiso() }
   BorrarZonaDeSeleccion()
 }
```

Vemos que simplemente lee la zona de selección y utiliza el código leído para ir a la pieza con ese código; si consigue llegar a esa pieza, entonces la mueve. Además, luego de mover una pieza resulta necesario verificar si debe extenderse el piso. Finalmente, se borra la zona de selección, forzando a que para cada movimiento de una pieza deba volver a ingresarse su código.

La operación de agregar dígito a la selección utiliza la operación correspondiente de la codificación del juego. La lectura de un código en la zona de selección se realiza de a un dígito por vez, puesto que la interacción que estamos considerando se realiza de a una tecla por vez. Cuando los dígitos completan un código válido, esta operación baja todas las piezas.

```
procedure OperacionAgregarDigitoASeleccion(dig)
 /*
    PROPÓSITO:
      agregar un dígito a la selección actual, y bajar
      las piezas al terminar de ingresar un código válido
    OBSERVACIONES:
      * como los dígitos se ingresan de izquierda a derecha
        pero se leen de derecha a izquierda, determinar si
        se completó el ingreso de un código válido se puede
        realizar leyendo el número y viendo que es distinto
        de cero
  */
 {
   AgregarDigitoASeleccion(dig)
   if (leerZonaDeSeleccion()/=0) // Al terminar de seleccionar
                                 //  una pieza, baja todas
     { BajarPiezasDeZonaDeJuego() }
 }
```

Como se puede observar en el comentario del contrato, la manera de determinar si se completó la lectura de un código válido es simplemente verificar que el código es distinto de cero; esto es así porque los dígitos se ingresan en la zona de selección de izquierda a derecha y se leen de derecha a izquierda, con lo cual mientras que no haya un dígito en la celda menos significativa de la zona, el número leído dará cero.

Finalmente la última de las operaciones de interacción es la de bajar todas las piezas en simultáneo.

```
procedure OperacionBajarPiezas()
 /*
    PROPÓSITO:
      baja todas las piezas y resetea la zona de selección
  */
 {
   BajarPiezasDeZonaDeJuego()
   BorrarZonaDeSeleccion()
 }
```

Esta operación es la más sencilla de todas, combinando simplemente la operación correspondiente de la mecánica del juego con el borrado de la zona de selección.

El hecho de que varias piezas coexistan en la zona de juego al mismo tiempo es una decisión de diseño del juego, al igual que la acción de todas las operaciones de interacción de borrar la zona de selección una vez que completaron su trabajo y que las piezas bajen luego de que se completa una selección. Estas decisiones son las que diferencian el ZILFOST del TETRIS™, puesto que en el último solo puede haber una pieza por vez y no hace falta seleccionar piezas, y además la bajada de piezas está determinada por el paso del tiempo. Puesto que en GOBSTONES no tenemos otra manera de medir el paso del tiempo que las acciones del usuario, el ZILFOST se transforma más en un juego de mecánica fija, del tipo *puzzle*, donde el objetivo es acomodar las piezas que uno mismo va solicitando. En caso de querer buscar alternativas en esta mecánica, son las operaciones de interacción las principales candidatas a sufrir modificaciones.

5.7. El programa principal

Todas las operaciones definidas previamente deben combinarse en un programa principal. Como vimos, en GOBSTONES el programa principal se establece a través de un bloque especial indicado con la palabra reservada `program`, donde ese bloque describe completamente la solución planteada por el programa. Para programas sencillos esto resulta suficiente, pero para juegos, donde el usuario debe interactuar con el programa, no. Entonces, en GOBSTONES diseñamos una variante de programa principal que provee una forma básica de interacción y permite la definición de juegos sencillos.

Todas las variantes de programa asumen que en el tablero inicial se encuentra codificado un tablero de ZILFOST como se describió en la sección 5.1. Esto se puede conseguir o bien codificando un tablero inicial mediante un procedimiento que lo dibuje, o, como elegimos en este libro, mediante algún mecanismo externo para proveer un tablero ya dibujado.

En esta sección describimos primero cómo sería un programa simple, y luego describimos cómo es el mecanismo de interacción provisto por GOBSTONES y cómo el mismo se utilizó para completar la programación del juego de ZILFOST.

5.7.1. Un programa simple

Un programa simple invoca sencillamente a una o más operaciones de interfaz. Por ejemplo, el tablero final que muestra el nombre ZILFOST escrito con piezas del gráfico G.5.1 fue obtenido con el siguiente programa

```
program
 /*
   PRECONDICIÓN:
     * tiene que haber un juego de Zilfost válido
       codificado en el tablero con una zona de
       juego de ancho mínimo 9
 */
 { GenerarLogoZILFOST() }
```

Vemos que se asume que ya hay un juego de ZILFOST codificado sobre el tablero; si no, este programa fallará.

Otros programas pueden mostrar el uso de diferentes operaciones básicas y diferentes combinaciones. Por ejemplo, podría pensarse en utilizar la zona de datos para codificar de alguna manera combinaciones de operaciones básicas y luego simular dichas operaciones leyendo los códigos e invocando adecuadamente a las operaciones codificadas. Pero la gran desventaja de este enfoque para probar el juego es que solo podemos visualizar el tablero final, perdiéndonos toda posibilidad de interacción.

5.7.2. Programas interactivos

Al pensar en ejemplificar el uso de GOBSTONES mediante juegos sencillos, surgió la necesidad de imaginar un mecanismo de interacción simple que respetase los principios de diseño del lenguaje. La forma más básica de interacción consiste en un ciclo donde la computadora muestra al usuario cierto estado, le solicita una interacción elemental (por ejemplo mediante la digitación de alguna tecla), y luego procesa el resultado de dicha interacción, mostrando el resultado y volviendo a repetir este accionar. Esta forma de interacción se suele conocer como ciclos *read-eval-print* (o también *read-execute-print*), pues comienzan con la lectura de cierta información provista por el usuario (la etapa de *read*, lectura de información), luego continúan con la ejecución (la etapa *eval* o *exectue*, de evaluación o ejecución del program), y se completan con la muestra de la información resultante al usuario (la etapa *print*, de impresión o muestra de resultados). En el caso de GOBSTONES, el mecanismo de muestra de información sería sencillamente permitir visualizar el estado del tablero luego del cómputo de cada ciclo, y la lectura de información podría consistir en recibir una tecla presionada por el usuario. (Podrían también considerarse eventos más complejos, como interacción a través del *mouse* u otros dispositivos, pero preferimos mantener la simplicidad en este lenguaje.)

Para expresar este ciclo de *read-eval-print* en forma de programa, GOBSTONES acepta una variante de programa llamada *programa interactivo*, e identificada con la palabra clave `interactive` usada como modificador de la palabra clave `program`. El cuerpo de un

programa interactivo será simplemente una lista de asociaciones entre códigos de tecla y bloques del programa a ser ejecutados cuando la tecla asociada sea presionada. En esta variante, cada bloque mantiene sus propias variables, no habiendo variables comunes entre dos ciclos de ejecución diferentes; lo común es que cada bloque consista en unos pocos llamados a procedmientos (idealmente, uno).

Los códigos de tecla se indican mediante constantes preestablecidas, y la forma de la asociación entre teclas y bloques sigue el mismo principio sintáctico que el de las diferentes opciones de una alternativa indexada. Los códigos de teclas que se aceptan actualmente comienzan con la letra `K` (por `K`ey, tecla en inglés), y son los siguientes:

- los códigos de las letras del alfabeto, `K_A`, ..., `K_Z`;
- los códigos de los dígitos, `K_0`, ..., `K_9`;
- los códigos de las flechas para arriba, abajo, izquierda y derecha, `K_ARROW_UP`, `K_ARROW_DOWN`, `K_ARROW_LEFT` y `K_ARROW_RIGHT`, respectivamente;
- los códigos de las teclas especiales, `K_ENTER`, `K_SPACE`, `K_ESCAPE`, `K_TAB`, `K_DELETE` y `K_BACKSPACE`; y
- el código de la tecla de fin de ciclo, `K_CTRL_D`.

El caracter de `CTRL-D` es históricamente usado en telecomunicaciones como el caracter de fin de transmisión (*end-of-transmission*, o EOT). En el sistema operativo UNIX se utiliza para indicarle a una terminal que terminó el ingreso de datos. En el GOBSTONES lo usaremos para indicar que deber terminar el ciclo de interacción con el usuario. Esta información se obtuvo de la Wikipedia.

Recurso Web

https://en.wikipedia.org/wiki/End-of-transmission_character

Cada herramienta (IDE) será responsable de implementar la manera en que se debe expresar la interacción, proveyendo quizás herramientas adicionales (como podría ser un texto inicial previo al inicio de la ejecución, la capacidad de redibujar celdas para que aparezcan de otras maneras, etcétera).

5.7.3. El juego interactivo

Habiendo presentado la capacidad de GOBSTONES de expresar programas interactivos, completaremos nuestra presentación del ZILFOST mostrando un programa principal interactivo, donde fueron elegidas teclas específicas para llevar adelante la interacción. Cada tecla se asociará con una operación de interfaz específica, como se ve en el siguiente código

```
interactive program
 /*
    PRECONDICIÓN:
      * tiene que haber un juego de
        Zilfost válido codificado en el tablero
  */
 {
   K_B, K_ESCAPE, K_DELETE, K_BACKSPACE
       -> { BorrarZonaDeSeleccion() }
   K_0 -> { OperacionAgregarDigitoASeleccion(0) }
   K_1 -> { OperacionAgregarDigitoASeleccion(1) }
   K_2 -> { OperacionAgregarDigitoASeleccion(2) }
   K_3 -> { OperacionAgregarDigitoASeleccion(3) }
   K_4 -> { OperacionAgregarDigitoASeleccion(4) }
   K_5 -> { OperacionAgregarDigitoASeleccion(5) }
   K_6 -> { OperacionAgregarDigitoASeleccion(6) }
   K_7 -> { OperacionAgregarDigitoASeleccion(7) }
   K_8 -> { OperacionAgregarDigitoASeleccion(8) }
   K_9 -> { OperacionAgregarDigitoASeleccion(9) }
   K_J, K_ARROW_LEFT
```

```
        -> { OperacionMoverPiezaAl(Oeste) }
    K_L, K_ARROW_RIGHT
        -> { OperacionMoverPiezaAl(Este) }
    K_K, K_ARROW_DOWN
        -> { OperacionMoverPiezaAl(Sur) }
    K_I, K_ARROW_UP
        -> { OperacionColocarNuevaPieza() }
    K_D, K_ENTER
        -> { OperacionRotarPieza(True) -- Sentido horario }
    K_A, K_SPACE
        -> { OperacionRotarPieza(False) -- Sentido antihorario }
    _   -> { OperacionBajarPiezas() }
  }
```

Podemos observar que los dígitos sirven para ir completando el código de selección en la zona correspondiente, las flechas a izquierda y derecha sirven para mover las piezas hacia los costados, la flecha hacia abajo hace descender la pieza un lugar y la flecha para arriba permite colocar una nueva pieza en el tablero. Las teclas de ENTER y espacio se usan para rotar las piezas en sentido horario y antihorario respectivamente, y las teclas de ESCAPE, DELETE y BACKSPACE vacían la zona de selección. Todas las teclas especiales (flechas y comandos) tienen su contraparte en forma de letras. Cualquier tecla que se toque que no figure entre las especificadas provocará el decenso de las piezas.

5.8. Ejercitación

Habiendo completado la codificación básica del juego de ZILFOST, resulta interesante mostrar la capacidad de realizar modificaciones que tiene un programador a mano, y cómo los conocimientos impartidos en este libro sirven para tal propósito. Para ello proponemos una serie de cambios de diferente complejidad que permitirán practicar las habilidades obtenidas. Al realizar estos cambios seguramente será necesario ir haciendo ajustes en el código que se va escribiendo, puesto que al realizar modificaciones que requieren alterar diversas partes del código es común olvidar alguna y tener que irlas corrigiendo a medida que se descubren.

El primero de los cambios propuestos es sencillo: modificar el momento en el que las piezas bajan. En el código que presentamos en el anexo B, las piezas bajan después de que se completa el ingreso de un código de selección. Sin embargo, esta opción puede ser fácilmente cambiada modificando los lugares donde aparece la invocación a `BajarPiezasDeZonaDeJuego`.

Actividad de Programación 9

Realice el ejercicio 5.8.1 y codifíquelo en una copia del código, así puede probar ambas versiones por separado y comprarlas.

Ejercicio 5.8.1. *Modificar el código del* ZILFOST *de manera que en lugar de que las piezas bajen después de completar el ingreso del código de selección, lo hagan después de cada movimiento de pieza (rotar, mover a izquierda o derecha o ingresar una nueva pieza).*

Como dijimos, es un ejercicio sencillo, puesto que no involucra modificar ninguna de las operaciones de base, sino solamente las operaciones de interacción (de la sección B.6 del anexo B).

El siguiente cambio tiene una complejidad apenas mayor. Consiste en modificar las primitivas que establecen el formato del tablero, de tal manera de lograr que la zona de semilla ocupe solo el ancho de la zona de juego, y las zonas de números del sector de datos estén más abajo, de manera tal que la zona de próxima pieza se encuentre a continuación de la zona de semilla, hacia la izquierda.

Actividad de Programación 10

Realice el ejercicio 5.8.2 y pruébelo en el código del ZILFOST. Recuerde que para este ejercicio debe modificar los tableros iniciales que tuviera guardados ya que si no lo hace, no se cumplirían las precondiciones correspondientes y el programa fallaría. ¡Y recuerde cambiar todos los comentarios correspondientes!

Ejercicio 5.8.2. *Realizar los cambios de codificación necesarios para lograr que la zona de semilla sea solo tan ancha como la zona de juego, la zona de código de próxima pieza esté a la izquierda de la de semilla y la zona de código de selección de pieza siga estando justo arriba de la zona de código de próxima pieza.*

Ayuda: *el procedimiento* `IrAlOrigenDeZonaDeProximaPieza` *es el único que requiere modificaciones.*

La complejidad de este cambio reside en que se están modificando las propiedades que el tablero debe cumplir para representar a un juego válido, y por ello, las operaciones de base, lo cual requiere modificar la totalidad de los tableros preexistentes para adaptarlos a este nuevo uso. En particular, hay que preservar las propiedades de las zonas de números, acerca de la forma de codificar sus límites. Estos cambios no deberían repercutir de ninguna manera en las operaciones de más alto nivel del juego (las que tienen que ver con piezas, movimientos o interacción).

Una cambio de complejidad media consiste en agregar el código necesario para que la próxima pieza que vaya a ser colocada en la zona de juego se muestre previamente en la zona de datos. Se plantea en varios ejercicios, para ir guiando la solución.

Actividad de Programación 11

Realice los ejercicios desde el ejercicio 5.8.3 hasta el ejercicio 5.8.6 y pruébelos. Recuerde utilizar una copia del código del ZILFOST, para poder comparar ambas versiones.

Ejercicio 5.8.3. *Escribir un procedimiento* `IrAZonaDeEsperaProximaPieza` *que se ubique en la zona de datos en el lugar elegido para visualizar la próxima pieza.*

Para este ejercicio debe decidirse dónde será el lugar para visualizar la pieza, de manera que todas las piezas posibles quepan en el mismo.

Ejercicio 5.8.4. *Escribir un procedimiento* `ColocarPiezaEnZonaDeEspera` *que reciba los mismos parámetros que el procedimiento* `ColocarNuevaPieza`*, pero que en lugar de dibujarla en la zona de juego la dibuje en la zona de espera. Tener en cuenta que, además de codificar el tipo de pieza (su pivote), se hace necesario codificar el parámetro de ubicación (para su uso futuro).*

Sugerencia: *una forma posible de codificar el parámetro de ubicación es utilizar piezas azules en el pivote.*

Para este ejercicio se hace necesario reutilizar los procedimientos `ColocarPieza` y el recién definido `IrAZonaDeEsperaProximaPieza`, y pensar un procedimiento auxiliar para codificar el parámetro de ubicación.

Ejercicio 5.8.5. *Escribir una función* `leerPiezaZonaDeEspera` *que retorne el código, el tipo y la ubicación de la pieza en la zona de espera.*

Tener en cuenta que esta función debe devolver varios valores, al igual que la función `determinarNuevaPieza`.

Ejercicio 5.8.6. *Modificar el procedimiento* `OperacionColocarNuevaPieza` *para que ubique la pieza en la zona de espera en la zona de juego, y una nueva pieza en la zona de espera. Tener en cuenta que si no hay pieza en la zona de espera (por tratarse de la primera pieza), deberá comenzarse poniendo una allí.*

Para este ejercicio será necesario utilizar `QuitarPiezaActual` y `ColocarPieza`, verificando que se puedan utilizar en la zona de espera. Además, puede que la zona de código de próxima pieza indique un número más de las piezas en la zona de juego, por contabilizar la pieza en la zona de espera; sería interesante que esto no fuera así, para lo cual deberá ajustarse adecuadamente el código.

El último de los cambios que proponemos es el de mayor complejidad, pues involucra la interactividad. Consiste en que la pieza seleccionada sea destacada (por ejemplo, mediante el sencillo mecanismo de hacer de su pivote la celda actual al mostrar un tablero de juego), y en modificar el procedimiento de selección de piezas para que en lugar de ingresar un número por dígitos en la zona de selección, se utilice una tecla (por ejemplo, `K_TAB`) para ir seleccionando alternativamente diferentes piezas (también puede ser un agregado y no una modificación). Es deseable para este cambio que siempre haya una pieza seleccionada, y que el código de la misma se visualice en la zona de selección (luego de cada cambio de pieza debe actualizarse la zona de selección de manera acorde). Para este cambio delineamos las modificaciones principales en los siguientes ejercicios, pero es posible que hagan falta mayores ajustes que deben detectarse a medida que se prueban los cambios.

Actividad de Programación 12

Realice los ejercicios entre el ejercicio 5.8.7 y el ejercicio 5.8.16 y pruébelos en una copia del código. Recuerde que puede ser necesario que tenga que realizar varios ajustes antes de que la versión final se comporte como es deseado, ya que todos estos procedimientos deben trabajar de manera sincronizada.

Ejercicio 5.8.7. *Modificar todas las operaciones de interacción para que luego de terminar ubiquen el cabezal en la pieza seleccionada y no borren la zona de selección (puesto que ahora la zona de selección debe siempre contenter un código de pieza seleccionada en caso que haya piezas en la zona de juego).*

Para ir a la pieza seleccionada puede reutilizarse `IrAPiezaSiExiste`, teniendo en cuenta leer el código correspondiente de la zona de selección.

Sin embargo, no es suficiente con estas modificaciones, pues existen operaciones que modifican la cantidad de piezas y consecuentemente pueden producir que la pieza seleccionada deje de existir. Por ello, deben realizarse modificaciones adicionales.

Ejercicio 5.8.8. *Escribir un procedimiento* `GrabarZonaDeSeleccion` *que reciba un código de pieza válido y lo grabe en la zona de selección de piezas.*

Este procedimiento será necesario cuando la pieza seleccionada cambie por efecto de alguna de las operaciones.

Ejercicio 5.8.9. *Modificar el procedimiento* `OperacionColocarNuevaPieza` *para que al colocar una pieza en la zona de juego, esta se transforme en la nueva pieza seleccionada. Recordar que para esto, deben grabarse su código en la zona de selección y luego ir a la pieza seleccionada.*

Esta modificación simplifica la necesidad de controlar si hay o no piezas en la zona de juego al agregar una pieza, ya que la misma siempre resulta seleccionada. Pero de todas maneras, puede que en algunas situaciones la zona de juego se quede sin piezas por lo que deben hacerse más operaciones.

Ejercicio 5.8.10. *Escribir* `EncontrarPiezaParaSeleccionarSiExiste`*, un procedimiento que recorra la zona de juego y se ubique en alguna de las piezas de la misma, si existe alguna.*

Estructurarlo como un recorrido que arranque en el origen de la zona de juego.

Ejercicio 5.8.11. *Escribir un procedimiento* `RenovarPiezaSeleccionada` *que, si la pieza indicada actualmente en la zona de selección no existe más en la zona de juego, encuentre una nueva pieza para seleccionar, si existe, y actualice la zona de juego de manera correspondiente (puede ser que deba grabar un nuevo código en ella, o borrar dicha zona, según el caso).*

Ayuda: *reutilizar el procedimiento del ejercicio anterior.*

Este procedimiento de renovación será útil luego de que las piezas se transforman en piso, para que no quede desactualizada la zona de juego.

Ejercicio 5.8.12. *Modificar el procedimiento* `ExtenderElPiso` *para que luego de terminar con la extensión del piso, verifique si la pieza seleccionada no desapareció (por haber sido transformada en piso) y en dicho caso, la renueve.*

De esta manera, se contribuye a garantizar que, de haber piezas, una de ellas está seleccionada.

Solo falta realizar las operaciones necesarias a asociar con la tecla `K_TAB` para realizar la rotación en la selección de piezas.

Ejercicio 5.8.13. *Escribir el procedimiento* `EncontrarProximaPieza` *que, suponiendo que hay una pieza seleccionada, encuentra la pieza siguiente a la pieza seleccionada actual (si la misma existe). Tener en cuenta que este procedimiento, en lugar de arrancar del origen, arranca desde la pieza seleccionada; también debe considerarse el hecho de que el final del recorrido no es el final de la zona de juego, sino haber retornado a la pieza seleccionada (o sea, debe rotarse por todas las piezas de la zona de juego y no solo por las que están después de la seleccionada).*

Ejercicio 5.8.14. *Escribir un procedimiento* `CambiarPiezaSeleccionada` *que, reutilizando el procedimiento del ejercicio anterior, cambie la pieza seleccionada y actualice de manera acorde la zona de selección.*

Este procedimiento se utilizará en una operación de interfaz para cambiar la pieza seleccionada.

Ejercicio 5.8.15. *Escribir* `OperacionCambiarPiezaSeleccionada`*, un procedimiento que provea la interfaz adecuada con el procedimiento anterior.*

Esta operación de interfaz consiste en simplemente invocar al procedimiento previamente definido. Se recomienda realizara para proveer una adecuada separación de niveles, así el programa principal solo utiliza operaciones de interfaz.

El cambio se completa con la modificación del programa interactivo para que la tecla `K_TAB` se asocie a la operación correspondiente.

Ejercicio 5.8.16. *Modificar el programa interactivo para asociar la tecla* `K_TAB` *con la operación de cambio de pieza seleccionada.*

Otros cambios posibles quedan librados a la imaginación del programador. Algunos que son conocidos de otros juegos tipo TETRIS™ son: la posibilidad de contar con una "pieza de reserva" que se puede intercambiar con la próxima pieza a aparecer; la posibilidad de tener piezas especiales (como por ejemplo, piezas con la capacidad de borrar el piso cuando lo tocan, etcétera); y, de mayor complejidad, la posibilidad de que las piezas no se transformen en piso apenas entran en contacto con el piso, sino un turno después, lo que permitiría acomodarlas a último momento. En cada caso, debe diseñarse la solución (decidiendo cómo se representará cada concepto en términos de bolitas de manera que no entre en conflicto con otras representaciones existentes) y agregar o modificar código según sea necesario.

5.9. Comentarios Finales

Codificar una aplicación completa, y más cuando la misma se trata de un juego, es una tarea para la que GOBSTONES no fue pensado inicialmente. Sin embargo puede verse el poder de los conceptos que maneja este lenguaje en el hecho de que es posible codificar tal aplicación.

En el diseño y codificación de una aplicación completa entran muchos conceptos, algunos presentados en este libro y otros que exceden completamente esta presentación. Si bien la aplicación que mostramos utiliza todos los conceptos presentados (con el objetivo de ejemplificar su uso), hay varios conceptos adicionales, especialmente los relacionados con el diseño general, que no fueron tratados por el libro, pero que son necesarios para la construcción correcta de programas de cierta envergadura.

Uno de tales conceptos es la separación completa entre diferentes partes del programa. Por ejemplo, todo lo que tenga que ver con interfaz de usuario está claramente

separado de la lógica del programa. Como se puede ver, la mayoría del código presentado no hace ningún tipo de consideración con respecto a cómo se comunica el usuario con el programa, o cómo se visualiza gráficamente el juego (aunque la codificación en el tablero tiene en cuenta también algunos aspectos visuales). Otra separación interesante es la que se da entre la representación de la geometría del juego y las operaciones que establecen la mecánica del mismo. Finalmente, otra separación más puede encontrarse entre la representación de las piezas y diferentes componentes y su utilización en el programa (aunque en este caso dicha separación no siempre es total). Todos estos conceptos tienen que ver con procesos de abstracción y se estudian en cursos más avanzados de programación, como ser cursos de estructuras de datos, de interfases de ususario, etcétera.

Otro de los conceptos fundamentales que se utilizó en el diseño de esta aplicación es la utilización del tablero para representar diferentes estructuras. En este caso, las zonas y las piezas se representan directamente en el tablero y se proveen operaciones para manipular zonas y piezas que tienen en cuenta esta representación. En lenguajes más avanzados no se utilizarían este tipo de codificaciones de bajo nivel, sino representaciones que usen estructuras de datos más avanzadas.

Finalmente, otro de los conceptos importantes utilizados tiene relación con el concepto de precondición, pero en este caso aplicada a los datos. Al representar los elementos en el tablero asumimos determinados requerimientos que debían cumplirse de manera específica (por ejemplo, la propiedad de que la primera celda en un recorrido noreste con 5 bolitas azules sea la que se encuentra a la izquierda del origen de la zona de juego, la forma de rodear las diferentes zonas, la propiedad de que en la zona de juego solo hay piezas completas o piso, la propiedad de que no existen dos piezas con el mismo código en la zona de juego, la propiedad de que todas las piezas quedan identificadas por su pivote, etcétera). Estas propiedades se cumplen en cada uno de los tableros que se le presentan al usuario para interactuar, y muchas de ellas se cumplen a lo largo de todo el programa (incluso cuando se está realizando una operación de modificación). Así como los requerimientos de un procedimiento o función para funcionar se llaman *precondiciones*, a los requerimientos que los datos deben cumplir para ser considerados válidos se los llama *invariantes de representación*. Esta noción no fue presentada formalmente en este libro, y sin embargo se utilizó de manera extensiva en la representación del ZILFOST, y en las precondiciones de prácticamente todas las operaciones del juego.

Entre los conceptos que sí fueron tratados en este libro y que fueron fundamentales para el diseño de esta aplicación podemos mencionar la separación en subtareas y la adecuada parametrización. También el concepto de recorridos fue utilizado extensivamente. Finalmente, las cuestiones de estilo, como comentar adecuadamente el código, nombrar adecuadamente los diferentes elementos (procedimientos y funciones, parámetros, variables, etcétera) y la indentación correcta, resultaron de excelente ayuda para la comprensión del código producido.

Constituye una buena forma de aprender el intentar detectar cada uso de los conceptos presentados en el libro dentro de la aplicación, y su replicación conciente a la hora de construir aplicaciones y juegos propios. Y también el intentar deducir los otros conceptos presentados que mencionamos en esta reflexión final de la aplicación.

6

¿Cómo continuar aprendiendo a programar?

El aprendizaje de la programación no se agota en los conceptos básicos que trabajamos en este libro. Hace falta conocer muchas herramientas abstractas de programación más, diversas complejidades, paradigmas, lenguajes, herramientas de soporte, cuestiones de arquitectura, y una multitud de etcéteras antes de poder decir que uno *es* un programador. Sin embargo, la base provista en este libro es lo suficientemente sólida como para que todo eso sea posible.

La pregunta que surge naturalmente es la que utilizamos para el título del capítulo: ¿cómo continuar aprendiendo a programar, dada tal cantidad de cosas a conocer para ser programador? Nosotros creemos que hay algunos caminos que son mejores que otros y los discutimos en este capítulo de cierre.

6.1. Estructuras de datos, algorítmica y lenguajes

El primer lugar en esta lista de continuaciones lo tienen las estructuras de datos. Como vimos en el capítulo anterior, programar una aplicación que no sea trivial utilizando exclusivamente el tablero requiere muchísimo esfuerzo y resulta en código complejo y no siempre lo suficientemente eficiente. Las estructuras de datos son formas de organizar los datos en unidades complejas, mucho mayores y con mayor flexibilidad que simplemente números, colores, booleanos o direcciones. Un elemento de un tipo de datos que representa a una estructura es un valor compuesto por muchos datos individuales, con diferentes alternativas y accesibles de diferentes maneras según de la estructura de que se trate.

Existen numerosos enfoques para el aprendizaje de estructuras de datos. Sin embargo, al igual que sucede con el aprendizaje inicial de programación, muchos de esos enfoques fallan en focalizarse adecuadamente en un conjunto de conceptos fundamentales, y normalmente abruman al estudiante con una cantidad de detalles que no hacen a la base conceptual. Por esa razón, creemos que se hace necesario diseñar un enfoque específico para dar los primeros pasos en estructuras de datos. Junto con el material presentado en este libro, se diseñó tal enfoque para el curso de Introducción a la Programación de la carrera de Tecnicatura en Programación Informática de la UNQ, pero no fue incluído aquí por problemas de alcance y extensión. Dicho enfoque se centra en aprender a modelar entidades mediante agrupaciones de datos heterogéneos, concepto denominado con el término `registros`, y a manipular colecciones de datos mediante secuencias de entidades, concepto denominado con el término `listas`. En ese enfoque tanto los registros como las listas se entienden desde el punto de vista de las operaciones que es posible realizar sobre ellos, y no desde el punto de vista de su representación de memoria. De esta manera el propósito de centrarse en los aspectos denotacionales por sobre los aspectos operacionales vuelve a ser centro en esta continuación.

Una vez aprendidos los conceptos de registros y listas desde un punto de vista denotacional, nuestra sugerencia es continuar con estructuras recursivas algebraicas (diversos tipos de árboles y otros tipos algebraicos, por ejemplo en el lenguaje HASKELL), también desde un punto de vista denotacional, para continuar luego con la teoría de tipos abstractos de datos. Recién luego de manejar ampliamente los conceptos de abstracción de

datos recomendamos incluir aspectos de implementaciones de bajo nivel utilizando memoria de manera explícita (por ejemplo en el lenguaje C, utilizando memoria dinámica y punteros), y profundizar en aspectos de eficiencia. Para completar el estudio básico de estructuras de datos recomendamos agregar estructuras de datos estáticas (arreglos y matrices) y sus aplicaciones (heaps binarias, quicksort y binsort, hashing, etcétera).

El aprendizaje de estructuras de datos involucra al mismo tiempo el aprendizaje de principios de algorítmica. En este libro hemos iniciado los rudimentos del aprendizaje de algoritmos al estudiar recorridos. Otros temas clásicos de algorítmica involucran el aprendizaje de soluciones recursivas (conocido como "divide y vencerás" – *divide&conquer*), con énfasis en su aplicación al problema de ordenamiento de una secuencia de datos, y otras técnicas como algoritmos glotones o golosos (algoritmos *greedy*), programación dinámica, heurísticas, etcétera. Sin embargo, la mayoría de los enfoques clásicos hacen demasiado hincapié en los detalles de bajo nivel, privilegiando un enfoque mayormente operacional.

Otro tema importante asociado al estudio de estructuras de datos y de algoritmos es el de la complejidad. La eficiencia de un programa queda determinada por diversos factores. Reducir el estudio de la eficiencia a simplemente medir cuánto tiempo lleva un programa en algunos ejemplos particulares es un error común, que debe evitarse pues puede conducir a falsas impresiones y conclusiones sobre lo bueno que puede ser cierto programa. Es por ello que el correcto estudio de la temática de complejidad es tan importante. El más importante de todos los factores que inciden en la eficiencia es la complejidad teórica propia de la estructura de datos o algoritmo utilizado. Para estudiar esta complejidad inherente, la técnica más difundida y estudiada es la de la matemática de funciones de orden para análisis de cotas superiores asintóticas (O grande) utilizada principalmente en el análisis del peor caso de algoritmos. También existen otras formas de análisis como funciones de recurrencia, análisis de cotas inferiores asintóticas (Omega grande), análisis de cotas ajustadas asintóticas (Theta grande), pero todas estas son menos comunes que la primera mencionada, que alcanzaría para iniciarse en estudios de complejidad. Otros factores que inciden en menor medida (pero no despreciable) al estudiar eficiencia son la cantidad de veces que ciertas operaciones se realizan (que en el estudio asintótico se abordan mediante las denominadas *constantes de proporcionalidad*), la cantidad de memoria consumida (especialmente en lenguajes con manejo automático de memoria), la velocidad del procesador, y en el caso de sistemas con concurrencia o paralelismo, el grado de carga del procesador en número de procesos simultáneos. Todos estos aspectos deben ser tenidos en cuenta a estudiar temas de eficiencia de programas.

Otro aspecto que debe tenerse en cuenta a la hora de aprender a programar es ver cómo los conceptos aprendidos se manifiestan en diferentes lenguajes. Por ello, es importante aprender diversos lenguajes y practicar los conceptos en cada uno de ellos. Pero no debe confundirse el aprender un lenguaje y ver cómo se manifiestan en él los conceptos de programación con aprender a programar **en** un lenguaje. Cada lenguaje tiene su propio conjunto de particularidades y trucos para realizar ciertas tareas, y no debe confundirse aprender esos trucos con aprender a programar. Y además, no deben mezclarse los paradigmas de programación. Algunos lenguajes modernos combinan varios paradigmas, pero siempre es conveniente aprender los conceptos en lenguajes puros, que manejan exclusivamente los conceptos de un único paradigma. Creemos que para aprender verdaderamente a programar deben comprenderse ciertas nociones básicas que aparecen en casi todos los lenguajes, puesto que los lenguajes particulares aparecen y desaparecen con el tiempo.

Lenguajes puros son HASKELL (paradigma funcional), SMALLTALK (paradigma de orientación a objetos) y C (paradigma imperativo). Una vez aprendidos los conceptos en dichos lenguajes/paradigmas, es conveniente ver cómo los mismos se manifiestan en otros lenguajes híbridos.

6.1.1. Programación orientada a objetos

Diseñar software a nivel industrial es más que simplemente programar y no es una tarea trivial. Además, se hace más y más complejo cuanto mayor es el tamaño y alcance del software desarrollado. Para abordar esta complejidad surgió, entre otras soluciones, el paradigma de Programación Orientada a Objetos (POO), que aporta elementos básicos que benefician el diseño de sistemas de gran escala.

El paradigma de Programación Orientada a Objetos (POO) es uno de los más predominantes en la actualidad, y aparece en la mayoría de los lenguajes industriales. En este tipo de programación se ven involucrados ciertos conceptos que podrían ser considerados únicos para este paradigma, pero de todas formas se utilizan todos aquellos conceptos que presentamos con GOBSTONES, al igual que los conceptos provenientes del estudio de las estructuras de datos.

Uno de los conceptos más importantes en POO es el de *encapsulamiento*, que permite abstraer estado (estructura) y comportamiento (procedimientos y funciones), dentro de lo que se conoce como un *objeto*. Un objeto en este paradigma es, entonces, un concepto teórico que permite relacionar un estado y un comportamiento específicos, abstrayéndolos para su reutilización. Esta idea está relacionada con la idea de tipos abstractos de datos, que son accedidos por interfaz, permitiendo ocultar detalles internos de implementación al usuario. Sin embargo, introduce nociones específicas y ciertas formas de pensar los programas que hacen más sencillo la producción de sistemas de software en gran escala.

Los objetos se comunican entre sí mediante el envío de *mensajes*, que son similares a la invocación de procedimientos, aunque no iguales. Al enviarse un mensaje se desencadena un mecanismo para determinar qué método específico de qué objeto responderá ese mensaje (algo así como decir que se determinará durante la ejecución qué procedimiento exacto será ejecutado al encontrar una invocación). Para ordenar este proceso de determinar cómo responder a los mensajes, muchos lenguajes utilizan la idea de *clase* de objetos, aunque esta noción no es esencial para el paradigma.

La mejor manera de comenzar a estudiar POO es a través de un lenguaje puro, que solo tenga la noción de objetos y mensajes. Uno de tales lenguajes es SMALLTALK. Recién luego de manejar los conceptos de objetos adecuadamente es conveniente aprender lenguajes que combinan objetos con otras nociones, tales como JAVA, PYTHON, etcétera.

6.1.2. Programación funcional

Otro de los paradigmas importantes actualmente es el paradigma de programación funcional. En este paradigma no existe, en principio, la noción de acción ni de comando: todas las expresiones del lenguaje referencian valores abstractos. Esta ausencia de entidades con efectos se conoce con el nombre de *transparencia referencial*, e implica que el lenguaje es más apto para manipulaciones algebraicas y para determinadas formas de combinación de programas donde los efectos producen numerosas complejidads (por ejemplo, formas de computación paralelas o concurrentes). En ese marco conceptual, las funciones son todo lo necesario para representar programas.

Decimos en principio porque existen varios mecanismos para incorporar la noción de acción y de comando en el lenguaje manteniendo la pureza conceptual del paradigma.

Un concepto importante dentro del paradigma funcional es conocido como *alto orden*, que implica que las funciones pueden utilizarse al igual que otros valores, pasándolas como argumentos, retornándolas como resultados e incluso almacenándolas en estructuras de datos (y en el extremo, ¡utilizándolas como estructuras de datos!) Este concepto es de central importancia, y permite expresar esquemas de programas en forma de código, lo cual tiene muchísimas ventajas a la hora de pensar programas. Además, el uso de esquemas de programas fomenta la visión abstracta del código, favoreciendo el enfoque utilizado en este libro.

Muchos de los desarrollos de herramientas y conceptos que hoy consideramos fundamentales fueron desarrollados originalmente en el paradigma funcional; por ejemplo, la idea de manejo de memoria automática (*garbage collection*), la idea de funciones como valores (actualmente presente en lenguajes como PYTHON, SCALA, etcétera), la idea de polimorifismo paramétrico (lo que en JAVA se conoce con el nombre de *generics*), y muchas otras. Sin embargo, el verdadero auge moderno de la programación funcional proviene de la facilidad con la que el paradigma se adapta a la producción de programas paralelos, puesto que la ausencia de efectos dada por la pureza referencial del lenguaje lo hace ideal para esta tarea.

Creemos que un programador moderno no puede dejar de conocer los conceptos fundamentales de este paradigma.

6.2. Disciplinas asociadas

Finalmente, el diseño de software no se agota en la programación. Existen otras nociones relacionadas con la programación, como ser las técnicas para representar grandes volúmenes de información, las técnicas utilizadas para desarrollar en equipos grandes de programadores teniendo en cuenta las necesidades de los usuarios del software, las problemáticas asociadas al soporte físico de las máquinas y dispositivos involucrados, etcétera. Cada una de estas temáticas se vincula de alguna forma con la programación, pero tiene su propia lógica, sus propios conceptos y sus propios desarrollos tanto teóricos como prácticos. No se puede pretender estudiar programación hoy día sin tener conocimientos de estas temáticas.

El estudio de las técnicas para representar grandes volúmenes de información se engloban bajo el estudio de *bases de datos*, que comprenden desarrollos teóricos y prácticos para esta tarea. Aprender de cómo modelar entidades y las relaciones entre ellas, cómo representar esos modelos en formato relacional, cómo comprender y mejorar las características de estas representaciones para evitar problemas comunes y cómo consultar la información almacenada de manera consisa y expresiva son todas nociones básicas vinculadas con esta temática. Además, existen otras nociones más avanzadas como la persistencia de estructuras de datos u objetos en gran escala, el acceso en gran escala a los datos en forma concurrente o paralela, la implementación de herramientas que plasmen todas estas cuestiones, etcétera. Para poder desarrollar programas reales se hace necesario tener al menos una idea de los temas de bases de datos, por lo que iniciarse en estos temas es importante para la tarea de programación.

Otros de los temas imprescindibles son los que se relacionan con el ambiente en el que los programas se ejecutan. Y a este respecto hay tres temas importantes: arquitectura de computadoras, sistemas operativos y redes de computadoras. La *arquitectura de computadoras* trata de cómo se compone una computadora en el nivel físico, qué partes tiene (memoria, dispositivos de almacenamiento, procesador, etcétera), cómo interactúan y cómo se representa la información en bajo nivel mediante codificación binaria o hexadecimal. También trata sobre cómo se realiza el ciclo de ejecución de los programas en bajo nivel (assembler). Además, todas las computadoras modernas poseen un programa base, llamado *sistema operativo*, que es el encargado de comunicar a todos los demás programas y aplicaciones con la arquitectura de la máquina. Todos los programas interactúan de una forma u otra con el sistema operativo, y es por eso que conocer la forma de trabajo de este sistema es fundamental. Al aprender sobre sistemas operativos se aprende sobre sistemas de almacenamiento de archivos, sobre maneras de administrar la memoria, sobre la manera de administrar múltiples procesos concurrentes, administrar diversos usuarios con diferentes niveles de acceso, y varios otros temas que son importantes para entender cómo ejecutan nuestros programas en un entorno real. Finalmente, las computadoras modernas no trabajan aisladas, sino que se comunican con otras a través de *redes de datos*. Entender cómo son las conexiones físicas, las reglas de codificación de la información, los protocolos de comunicación en los diversos niveles y la arquitectura general de las redes de computadoras son conocimientos escenciales para poder escribir programas que se puedan comunicar con otros, u ofrecer servicios en la red.

El último de los temas vinculados con la programación tiene que ver con el hecho de que al desarrollar programas o sistemas de gran envergadura hacen falta técnicas para desarrollo en gran escala. Los temas cubiertos en este libro, y los que tienen que ver con estructuras de datos y algorítmica pueden ser vistos como la base conceptual de la programación. Pero para poder abarcar las situaciones y problemas que se presentan al desarrollar programas en gran escala y en equipos de mucha gente (usualmente multidisciplinarios), hace falta tener conocimientos de *ingeniería de software*. La ingeniería de software abarca los problemas de comunicación entre equipos y con los usuarios del programa que se desarrolla, la planificación de los tiempos, la evaluación de costos, la planificación para posibles contingencias, etcéterca, y todo en un marco de trabajo donde los requerimientos que se realizan sobre los programas son muy dinámicos y cambiantes. Tener conocimientos básicos de ingeniería de software es necesario en la medida en que se quiera programar algo más que pequeños programas para uno mismo.

6.3. Palabras finales

En este libro hemos cubierto los conceptos fundamentales para comenzar a entender de programación, pero desde una concepción innovadora con respecto a otros enfoques más tradicionales. Esto implicó por un lado un recorte de muchos temas que usualmente nadie se cuestiona como parte de una introducción a la programación, como ser mecanismos de entrada/salida de información, estructuras de datos elementales, etcétera, y por el otro una profundización en los temas considerados esenciales, como ser los procesos de abstracción y las herramientas para expresarlos, la identificación de la naturaleza de los elementos que componen un programa y su expresión en forma pura, la uniformidad de las formas de combinación de elementos, etcétera.

Creemos que este enfoque, o uno con similares características, es imprescindible para que la programación no quede restringida a un círculo pequeño de iniciados y pueda enseñarse como materia básica junto con la matemática o la literatura. Claramente, este es solo el comienzo de la enorme aventura que representa aprender a programar, pero

sentando bases sólidas, que permitan construir todos los demás conceptos y herramientas que hay que conocer para poder decir que uno sabe programar. Y aunque uno no tenga intenciones de volverse un programador, los conocimientos básicos aquí brindados son necesarios para comprender mejor el mundo actual, donde los programas son ubicuos, proveyendo una forma de “alfabetización” moderna.

A

La herramienta PyGobstones

La herramienta PyGobstones es un entorno de programación básico para el lenguaje Gobstones. El programa está implementado en Python y se compone de varios módulos que pueden utilizarse para analizar y ejecutar programas escritos en Gobstones. En este apartado se detallan los pasos necesarios para poder ejecutar la herramienta PyGobstones.

A.1. Instalación

Para la instalación de PyGobstones hay que seguir varios pasos: obtener el código de la herramienta, el lenguaje para ejecutarlo y varias bibliotecas de funciones gráficas. Describimos cada paso por separado.

A.1.1. La herramienta

El primer paso es obtener los *scripts* que conforman el entorno de programación PyGobstones (este libro describe la herramienta en su versión 1.0). Se encuentran en un archivo comprimido que está disponible para descargar desde el sitio de Gobstones.

Recurso Web

http://www.gobstones.org/descargas/

Allí se debe puede descargar el archivo cuyo nombre comienza con `PyGobstones`, luego contiene la versión correspondiente, una fecha y luego una extensión, que puede ser `.zip` o `.tar.gz`, indicando que se trata de un archivo comprimido. Por ejemplo, la versión que utilizamos en este libro es la 1.0, y los archivos se llaman `PyGobstones_v_1.0_10-12-13.zip` y `PyGobstones_v_1.0_10-12-13.tar.gz`.

El archivo comprimido debe descomprimirse en una carpeta. Allí se encontrarán varios archivos de extensión `.py` que conforman la implementación de PyGobstones. Los restantes pasos dependen del sistema operativo que se esté utilizando, e incluyen instalar el lenguaje Python y algunas bibliotecas.

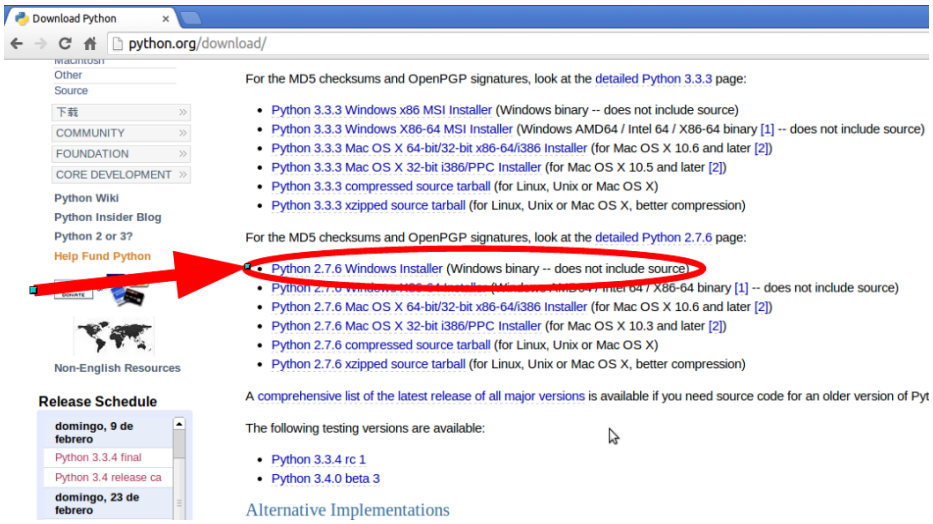
A.1.2. Usuarios de Windows

Para los usuarios de Windows, el intérprete tradicional de Python se puede descargar desde el sitio de la organización Python.

Recurso Web

http://www.python.org/download

Allí debe buscarse el archivo para instalar en Windows, que para la versión 2.7.6 es *Pyt-*



G.A.1. Descarga de PYTHON

hon 2.7.6 Windows Installer (`python-2.7.6.msi`). En el gráfico G.A.1 puede observarse la página mencionada con la opción correcta resaltada.
Una vez obtenido el archivo se lo debe abrir para proceder con la instalación. Alcanza con realizar la instalación con todas las opciones predeterminadas.

El paso siguiente es descargar e instalar las bibliotecas gráficas PyQt4, ya que son las que se utilizaron para desarrollar la interfaz gráfica de PYGOBSTONES. El sitio para descargarlas está en `sourceforge`, en el proyecto `pyqt`.

Recurso Web

http://sourceforge.net/projects/pyqt/files/PyQt4/PyQt-4.10.2/

El archivo específico depende del sistema operativo y la versión de la biblioteca. Por ejemplo, para un WINDOWS estándar de 32 bits y la versión 4.8.4 de la biblioteca, el archivo es `PyQt4-4.10.2-gpl-Py2.7-Qt4.8.4-x32.exe`. Al igual que con el paquete de PYTHON, se procede a ejecutar el instalador con las opciones predeterminadas.

Con estos tres pasos, ya estamos en condiciones de ejecutar PYGOBSTONES. Para cargar la herramienta debe ejecutarse PYTHON sobre el script `PyGobstones_Windows.py` que se encuentra en la carpeta de PYGOBSTONES. Esto se puede hacer utilizando el botón derecho sobre dicho archivo y eligiendo la opción “Abrir con”, desde donde debe seleccionarse el link al programa ejecutable `C:\Python27\pythonw.exe` (o la ruta en la que esté instalado – ver el gráfico G.A.2). Alternativamente, en la mayoría de las configuraciones de Windows se puede hacer doble click directamente sobre el archivo `PyGobstones_Windows.py`.

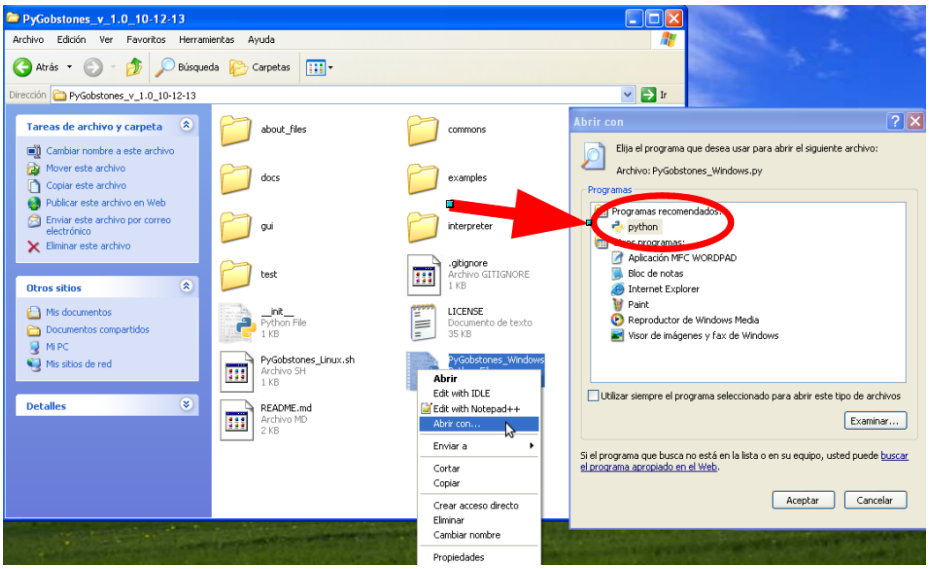
A.1.3. Usuarios de GNU/LINUX

Las distribuciones de LINUX generalmente ya traen el intérprete de PYTHON instalado. Además de esto, para ejecutar PYGOBSTONES se requiere instalar el paquete `python-qt4`. Para instalarlo, se puede utilizar el administrador de paquetes synaptic, o bien abrir una terminal y ejecutar el comando

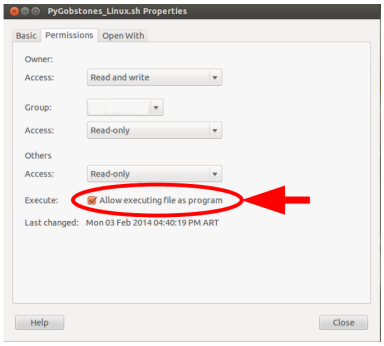
```
sudo apt-get install python-qt4
```

(para lo cual se precisan permisos de administrador).

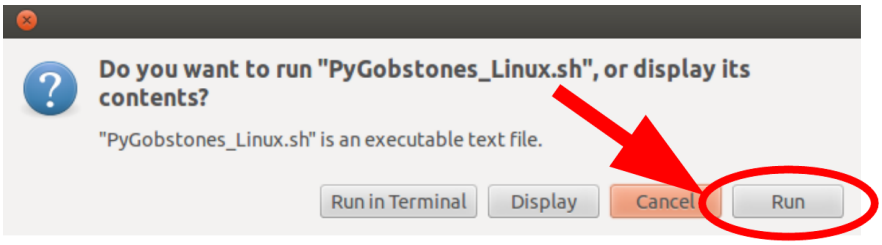
Para iniciar PYGOBSTONES en LINUX primero se debe abrir la carpeta que descargada anteriormente (por ejemplo `PyGobstones_v_1.0_10-12-13`), dar click derecho sobre el archivo `PyGobstones_Linux.sh`, ir a la opción propiedades, ir a la solapa de permisos y



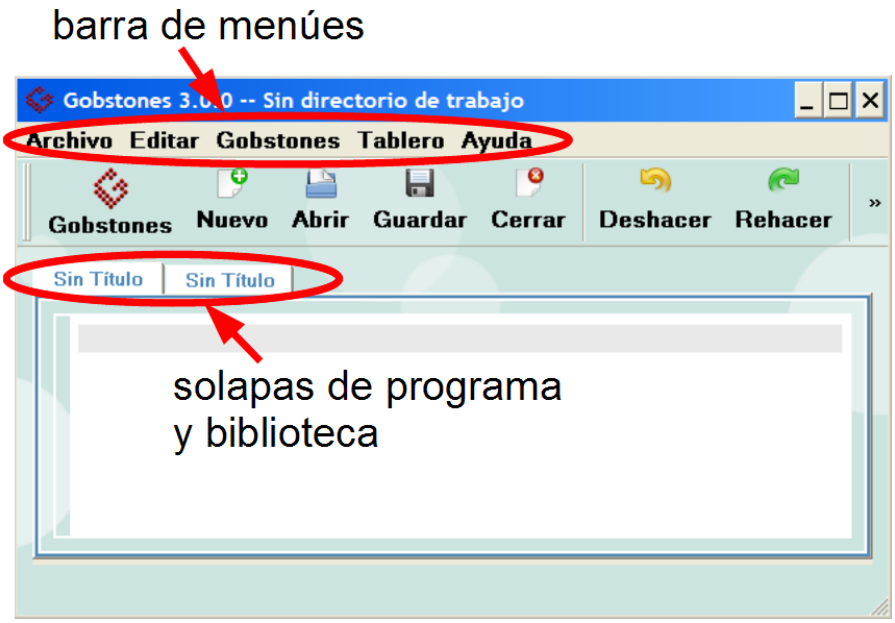
G.A.2. Abrir la herramienta PyGobstones



G.A.3. Cambiar el permiso de ejecución de `PyGobstones_Linux.sh`



G.A.4. Seleccionar la opción de ejecución



G.A.5. Ventana principal de PyGobstones, indicando la barra de menúes y las solapas

tildar la opción que permite ejecutar ese archivo como un programa (puede verse el resultado de esta acción en el gráfico G.A.3). Una vez hecho esto, basta con hacer doble click sobre el archivo en cuestión y seleccionar la opcion `Run` o `Ejecutar`, como se muestra en el gráfico G.A.4.

A.2. Primeros pasos en PyGobstones

Al arrancar la herramienta, y luego de visualizar el logo de Gobstones, la herramienta muestra su ventana principal. En primer lugar conviene reconocer algunos componentes importantes que van a ser de utilidad para el desarrollo de programas Gobstones usando PyGobstones.

A.2.1. Barra de menúes

La barra de menúes se encuentra en la parte superior de la ventana principal. Su aspecto puede observarse en el gráfico G.A.5. En esta barra están agrupadas la mayoría de las opciones que provee la herramienta.

- En el menú `Archivo` se encuentran las opciones para `Abrir`, `Guardar` y `Cerrar` un archivo de programa. Además se puede cerrar la aplicación desde allí.
- En el menú `Editar` se encuentran las opciones clásicas de `Buscar` y `Buscar y Reemplazar` palabras, de `Rehacer`, `Deshacer`, `Copiar`, `Cortar` y `Pegar`. Además se

encuentra la opción `Preferencias`, donde se pueden establecer algunas preferencias generales de la aplicación. Las preferencias se explican más adelante.

- En el menú GOBSTONES se encuentran las opciones para ejecución de un programa. Desde aquí se puede `Ejecutar` y `Chequear` el código fuente y también detener la ejecución del mismo antes de que culmine. Estas opciones pueden devolver cierta información, la cual puede encontrarse en el `Logger`, ubicado en el sector inferior de la pantalla.
- En el menú `Tablero` se encuentran las opciones para interactuar con el tablero. Desde aquí se puede cargar un archivo `.gbb` (un tablero) desde el disco. Además se puede acceder a `Opciones del tablero` y al `Editor del tablero`. Finalmente se cuenta con una opcion `Elegir vista`, que permite elegir el tipo de vista que va a tener el resultado, es decir la vestimenta con la que se va a vestir a los tablero inicial y final. Cada una de las alternativas se explica más adelante.
- En el menú `Ayuda` se encuentra una opción para acceder al manual de GOBSTONES (para lo que se requiere acceso a internet), la `Licencia` de la herramienta y finalmente la opción `Acerca de`, con información sobre herramientas utilizadas, miembros del equipo de GOBSTONES y algo de la historia de este lenguaje.

Varias de las opciones de la barra de menúes aparecen de manera individual en forma de botones inmediatamente debajo de la barra.

A.2.2. Editor de textos de programa y biblioteca

Debajo de los botones de menúes se encuentra un recuadro editable en el cual se puede escribir y editar un programa GOBSTONES (o su biblioteca), que puede verse indicado en el gráfico G.A.6. El editor es un editor tradicional, con opciones para copiar, cortar y pegar, deshacer, moverse por el texto, etcétera. Permite crear y modificar programas de GOBSTONES, que se guardan en archivos de texto con extensión `.gbs` con las opciones de menú correspondientes.

La zona del editor posee dos solapas en la parte superior, inicialmente con la leyenda `Sin Título`, como puede observarse en los gráficos G.A.5 y G.A.6a. Estas solapas corresponden a las ventanas de edición del programa (la de la izquierda) y de la biblioteca (la de la derecha).

Mientras que no se ha creado y salvado un archivo de programa, las solapas muestran la opción `Sin Título`. Una vez que un archivo de programa fue editado y salvado, o cargado desde el disco, la primera solapa muestra el nombre del archivo, como puede observarse en el gráfico G.A.6b. La segunda solapa contendrá el archivo de nombre `Biblioteca.gbs` que guarda las operaciones que querramos preservar en más de un programa.

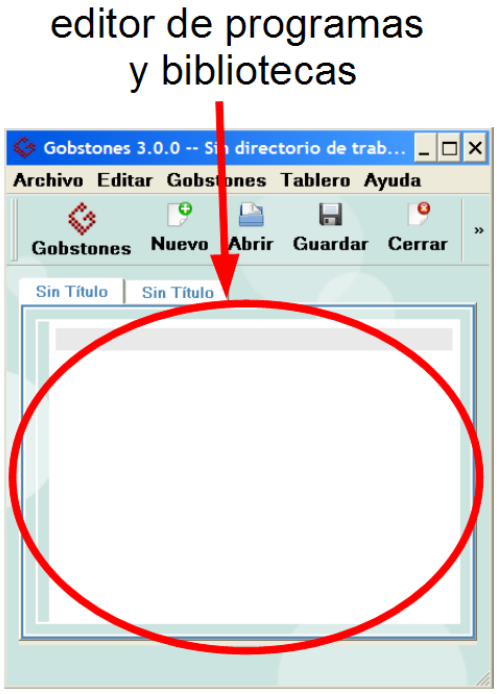
Un detalle importante a tener en cuenta con el manejo de la biblioteca es que, en esta herramienta, la misma se asocia a una carpeta de trabajo. De esta manera, varios archivos de programa en la misma carpeta pueden compartir la misma biblioteca. La carpeta de trabajo actual es aquella en la que se encuentra el programa que esté cargado en la solapa de programa (indicado en la barra de título de la ventana principal como se indica en el gráfico G.A.6b), y en ese caso, la solapa de biblioteca mostrará la biblioteca correspondiente; si la biblioteca aún no existe en la carpeta de trabajo, la herramienta crea una biblioteca vacía. Si la carpeta donde se guarda el programa ya contiene un archivo `Biblioteca.gbs`, la aplicación cargará el existente, eliminando la biblioteca que estuviese en dicha solapa.

Una opción adicional de este editor es que es posible elegir mostrar o no los números de línea; esta opción se puede establecer utilizando la opción correspondiente del menú `Editar`→`Preferencias`.

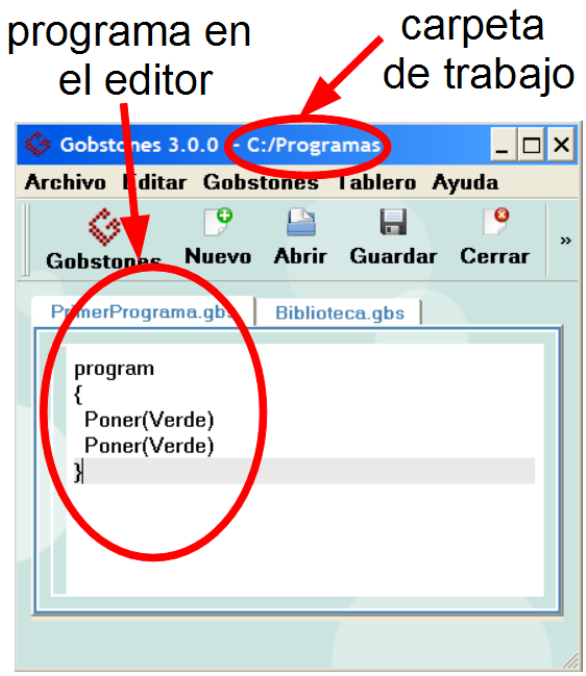
A.2.3. Ejecutar un programa y ver su resultado

El propósito del programa escrito en el editor es ejecutarse en GOBSTONES. PYGOBSTONES provee tres formas de iniciar la ejecución de un programa: usar la opción de menú `Gobstones` → `Ejecutar`, presionar la tecla `F5`, o presionar el botón `Ejecutar` de la barra de herramientas. En el gráfico G.A.7 se muestran dos de las opciones.

Una vez que la ejecución del programa finalizó, y si lo hizo de manera exitosa, la herramienta abre automáticamente una nueva ventana llamada *"Modo de resultados"* que permite inspeccionar 3 elementos: el tablero inicial, el código fuente y el tablero final. En

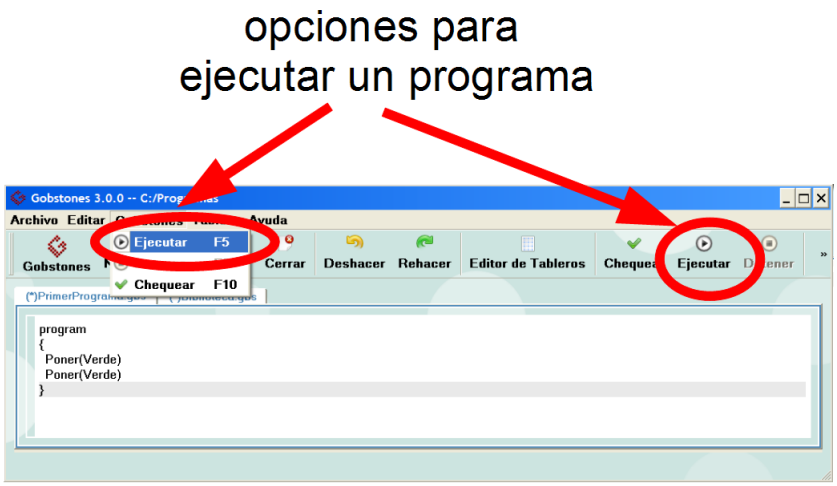


(a). Sin archivo.



(b). Con un programa salvado.

G.A.6. Editor de textos de programas y bibliotecas



G.A.7. Ejecutar un programa

el gráfico G.A.8 puede verse esta ventana, con el tablero final como elemento visualizado. Estos elementos son accesibles invidivualmente a través de solapas y permiten comprobar que el programa efectivamente cumplió con su cometido. Su significado es el siguiente:

Tablero Inicial: el estado del tablero *antes* de ejecutar el programa.

Código Fuente: el código que efectivamente se ejecutó.

Tablero Final: el estado del tablero *después* de haber ejecutado el programa.

Por defecto el tablero inicial tiene un tamaño de 8 filas y 8 columnas, y se encuentra vacío, con el cabezal sobre en la esquina suroeste; esto puede luego ser modificado a través del uso de opciones de tablero.

Existen algunas funcionalidades asociadas a la ventana de *Modo de resultados* que se describen más adelante.

A.2.4. Visualizar información adicional

Distintas funcionalidades de la herramienta PyGobstones retornan una cierta cantidad de información útil para el programador. Por ejemplo, la funcionalidad de ejecución informa, en caso de éxito, la hora de inicio y fin de la ejecución, o en caso de error, el tipo de error con información extra para ayudar a su corrección. Otras funcionalidades proporcionan otras informaciones.

Esta información se presenta en una zona específica, denominada *"logger"*. El logger se encuentra en la zona inferior de la pantalla, aunque inicialmente la mayor parte del mismo está oculto. Para visualizar más información puede deslizarse la línea inferior del editor hacia arriba, o encender la opción `Mostrar logger` de las preferencias (que se acceden desde `Editar`→`Preferencias`). En el gráfico G.A.9 se puede visualizar el resultado de la ejecución de un programa exitoso.

Log, en inglés, significa *bitácora*. Entonces, el término "logger" indica un lugar donde se muestra la bitácora, o sea, el registro de información resultante de cada una de las funcionalidades.

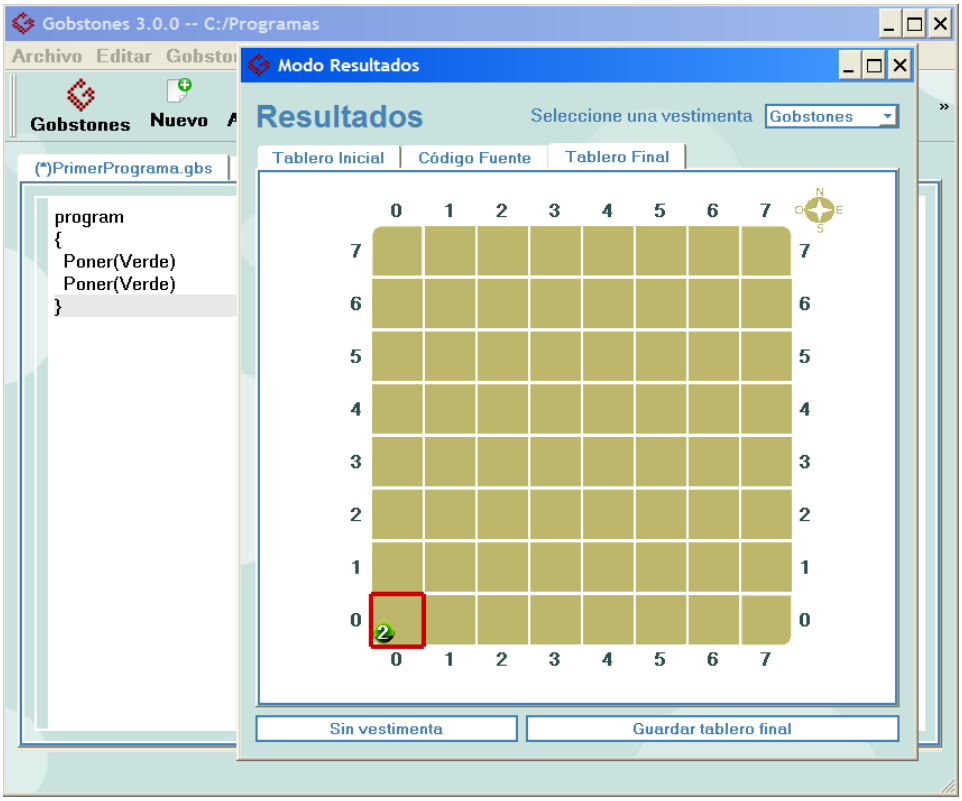
El logger puede ocultarse en cualquier momento, y volverlo a visualizar con las opciones mencionadas.

A.2.5. Chequear un programa

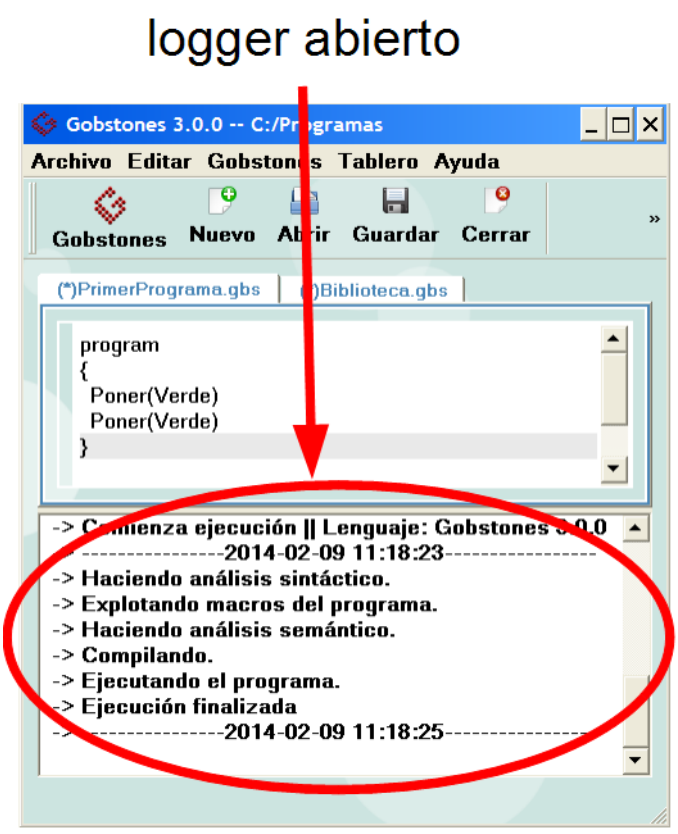
Una funcionalidad nueva de esta versión de PyGobstones es la opción de chequear (o validar) si el programa que se está escribiendo es correcto, desde el punto de vista de la coincidencia de tipos, antes de ejecutarlo. Existen 3 formas de acceder a esta característica: presionando la tecla F10, desde el menú `Gobstones`→`Chequear`, o presionando el botón con un tilde que se encuentra en la barra de herramientas.

A.2.6. Opciones de Tablero

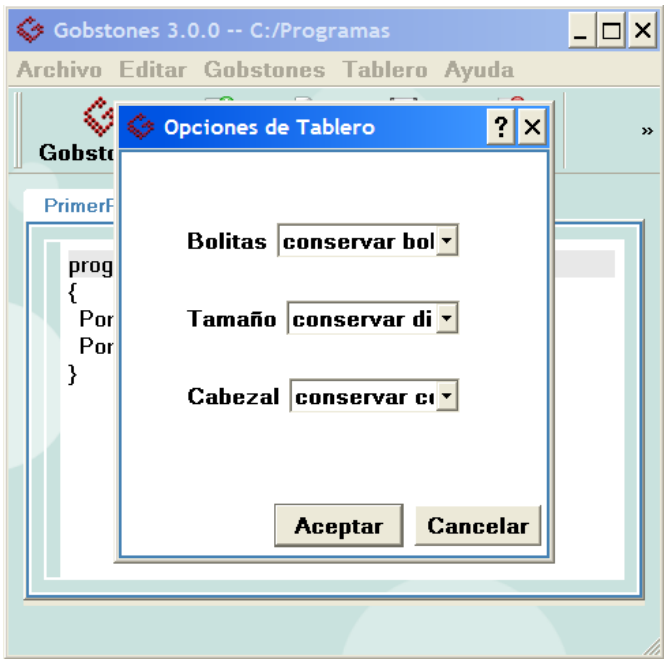
Por defecto, el tablero inicial en el cual se ejecutan los programas posee un tamaño fijo (de 8x8), no contiene bolitas, y el cabezal se encuentra ubicado en la esquina suroeste. Sin



G.A.8. Ventana de Modo de resultados



G.A.9. Visualización del logger



G.A.10. Opciones de tablero

embargo, no es conveniente acostumbrarse a programar con las opciones de ambiente fijas, y para ello PyGobstones provee varias alternativas para la modificación del tablero inicial. Estas alternativas son accesibles desde el menú `Tablero→Opciones de tablero`. Al acceder a esta opción, se abre una ventana subsidiaria con elementos para modificar las diferentes alternativas; la misma se muestra en el gráfico G.A.10.

Las opciones de tablero incluyen alternativas para modificar la aparición de bolitas, el tamaño del tablero y la posición del cabezal:

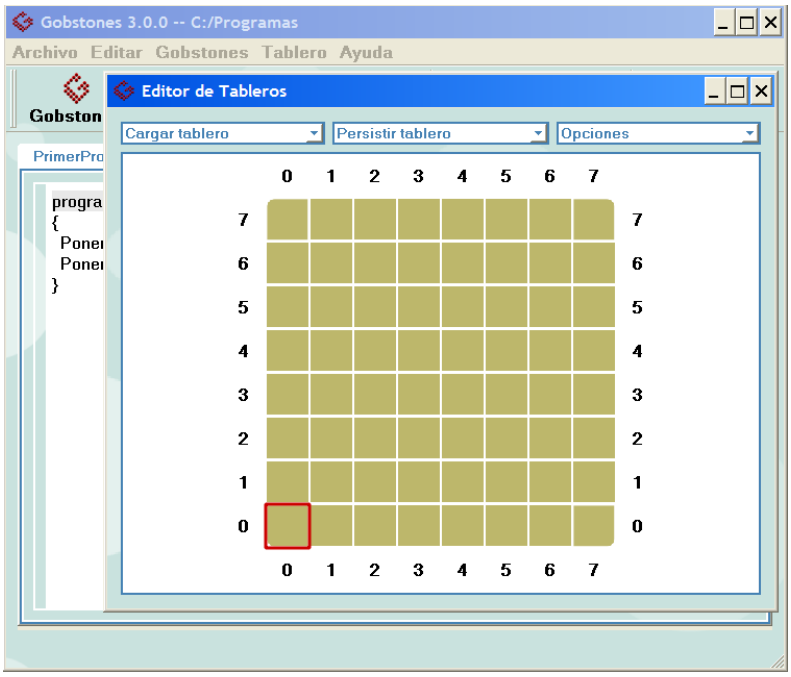
- Con respecto a las bolitas, se puede dejar la configuración de bolitas actual, pedir que se establezca un orden y cantidades aleatorias, o pedir que no haya bolitas en el tablero. La configuración de bolitas actual es útil en conjunción con el editor de tableros, que se discute en la próxima sección.
- Con respecto al tamaño del tablero, se puede conservar el tamaño actual, ingresar un tamaño manualmente, o pedir que se genere un tamaño aleatorio. Esta última opción forzará al cabezal a ser aleatorio, ya que no se conocen las dimensiones finales que va a tener el tablero.
- Con respecto a la posición del cabezal, se puede conservar la posición actual del mismo, ingresar una posición manualmente (limitada por el tamaño del tablero), o bien pedir que se genere una posición aleatoria del cabezal.

A.3. Otras funcionalidades de PyGobstones

La herramienta PyGobstones ofrece algunas funcionalidades avanzadas, destinadas a facilitar el trabajo de prueba de programas, y a mejorar algunos aspectos pedagógicos o de usabilidad. Entre estas opciones se encuentran la manipulación de tableros, la capacidad de incorporar "vestimentas" a los tableros (proveyendo de esa forma una visualización de ciertos procesos de abstracción) y la capacidad de operar interactivamente con un programa Gobstones. Tratamos cada una de estas características en diferentes subsecciones.

A.3.1. Guardar y cargar tableros

Cuando los programas a realizar dejan de ser triviales, muchas veces es necesario contar con tableros iniciales específicos. Para ello, PyGobstones cuenta con opciones de salvar tableros y de cargar tableros salvados y usarlos como tableros iniciales.



G.A.11. Editor de tableros

Para guardar tableros se puede utilizar el botón `Guardar tablero final` en la ventana de resultados, o también utilizar el editor de tableros para crear un tablero y salvarlo. Para cargar un tablero desde el disco se utiliza la opción del menú `Tablero→Cargar Tablero`. Tanto la opción de guardar como la de salvar abren un explorador de archivos para indicar a qué archivo o desde qué archivo debe realizarse la operación. Los tablero se almacenan en archivos de extensión `.gbb` y usualmente residen en la misma carpeta que el programa con el que se asocian.

A.3.2. Editor de Tableros

El editor de tableros es una funcionalidad de PyGobstones que permite editar tableros guardados y crear tableros propios, modificando el tamaño de los mismos, la cantidad de bolitas en las celdas y la posición del cabezal. Para accederlo se utiliza la opción `Tablero→Editor de Tableros`, que cuando se selecciona abre una nueva ventana con el editor, como puede observarse en el gráfico G.A.11.

Para utilizar el editor de tableros se utiliza el mouse y el teclado. Con el mouse se pueden poner y sacar bolitas del tablero; para ello hay que posicionarse sobre una celda, y elegir una de las 4 regiones de la celda (que se colorean al posicionar el cursor sobre ellas), y luego con el botón izquierdo se ponen bolitas y con el derecho se sacan. Con las flechas del teclado se puede mover el cabezal, cambiando la celda actual.

Además de las opciones de edición, sobre el tablero aparecen 3 botones que permiten cargar un tablero, persistir un tablero o modificar las opciones. Al cargar o guardar, se puede indicar que el origen o destino sea el disco o se puede utilizar el tablero inicial. Las opciones abren las opciones de tablero ya mencionadas.

A.3.3. Vestimentas

Una innovación de la versión 1.0de PyGobstones es la posibilidad de "vestir" los tableros. La funcionalidad consiste en asignar una imagen con extensión `.png` o `.jpg` a una configuración de bolitas específica, de manera tal que cada celda que posea esa configuración mostrará la imagen indicada en lugar de las bolitas. Si la configuración de bolitas de una celda no coincide con ninguna de las especificadas, la herramienta la dibujará de la manera tradicional.

Esta asociación entre configuraciones de bolitas e imágenes se especifica mediante un archivo externo, en formato `.xml`, conteniendo los siguientes elementos:

`BoardCoding` : el elemento principal, que contiene a todos los demás. Debe haber solo uno de estos elementos por archivo.

`Cell` : este elemento indica la asociación entre una configuración de bolitas específicas y una imagen (a través de los elementos siguientes). Puede haber cualquier número de ellos dentro del elemento `BoardCoding`; el orden importa, ya que se utilizará el primero que coincida con la configuración de una celda.

`Blue` : indica la cantidad de bolitas azules que debe haber para coincidir con esta configuración. Hay exactamente uno de estos elementos en cada elemento `Cell`. Puede ser un número positivo o un comodín. Los comodines son un asterisco (`*`) o un signo de suma (`+`), e indican cero o más bolitas y una o más bolitas, respectivamente.

`Black` : similar al elemento `Blue`, pero para bolitas negras.

`Red` : similar al elemento `Blue`, pero para bolitas rojas.

`Green` : similar al elemento `Blue`, pero para bolitas verdes.

`Image` : indica el nombre de un archivo (con extensión `.png` o `.jpg`), que debe contener una imagen de 60x60 píxeles que se utilizará para mostrar en la celda en caso de que la configuración de bolitas coincida. Hay exactamente uno por cada elemento `Cell`.

El archivo especificando la vestimenta debe residir en una carpeta de nombre `Vestimentas` en la carpeta de trabajo. Los archivos de imágenes deben ir dentro de una carpeta de nombre `Imagenes`, a su vez dentro de la carpeta `Vestimentas`. Estas dos carpetas se crean automáticamente y pueden contener cualquier número de vestimentas y de imágenes, respectivamente.

Por ejemplo se puede indicar que toda celda donde haya exclusivamente 1 bolita se dibuje como un bloque sólido del color de la bolita y si no hay bolitas, que se dibuje como un bloque sólido gris. Para ello se debe generar un archivo `.xml` con el siguiente contenido:

```
<BoardCoding>
    <Cell>
        <Blue>0</Blue>
        <Black>0</Black>
        <Red>0</Red>
        <Green>0</Green>
        <Image>fondo.png</Image>
    </Cell>
    <Cell>
        <Blue>0</Blue>
        <Black>0</Black>
        <Red>0</Red>
        <Green>1</Green>
        <Image>bloque-verde.png</Image>
    </Cell>
    <Cell>
        <Blue>1</Blue>
        <Black>0</Black>
        <Red>0</Red>
        <Green>0</Green>
        <Image>bloque-azul.png</Image>
    </Cell>
    <Cell>
        <Blue>0</Blue>
        <Black>1</Black>
        <Red>0</Red>
        <Green>0</Green>
        <Image>bloque-negro.png</Image>
    </Cell>
    <Cell>
        <Blue>0</Blue>
        <Black>0</Black>
        <Red>1</Red>
        <Green>0</Green>
        <Image>bloque-rojo.png</Image>
```

```
    </Cell>
</BoardCoding>
```

Si los archivos de imágenes son los correctos, entonces un tablero con la configuración del gráfico G.A.12a se vería como el tablero del gráfico G.A.12b a partir de aplicar esta vestimenta.

Las vestimentas son una herramienta poderosa que se puede utilizar para ejemplificar el principio de abstracción y para mejorar la apreciación abstracta de los programas que se ejecutan. Por ejemplo, en el caso del juego de Zilfost, en lugar de esforzarnos en ver las piezas en ciertos grupos de celdas con bolitas, como deberíamos hacer en el caso del gráfico G.A.13a, podríamos tener una vista como la que se muestra en el gráfico G.A.13b. En esta vestimenta del Zilfost, que denominamos `zilfost-romano`, podemos encontrar una aplicación de los comodines de especificación:

```
...
    <Cell>
        <Blue>0</Blue>
        <Black>*</Black>
        <Red>*</Red>
        <Green>1</Green>
        <Image>bloque-azul.png</Image>
    </Cell>
    <Cell>
        <Blue>0</Blue>
        <Black>*</Black>
        <Red>*</Red>
        <Green>2</Green>
        <Image>bloque-naranja.png</Image>
    </Cell>
...
```

Puesto que las piezas se codifican mediante la cantidad de bolitas verdes, pero las rojas y negras se utilizan con fines de marcación, entonces la pieza de código 1, que tendrá 1 bolita verde y cualquier cantidad de negras o rojas, será la pieza azul, etcétera.

A.3.4. Interactivo

PyGobstones en su versión 1.0 implementa la versión 3.0 del lenguaje Gobstones. Esta versión incluye una base para realizar programas interactivos, por lo que otra innovación en esta versión de la herramienta es la capacidad de interactuar con programas Gobstones.

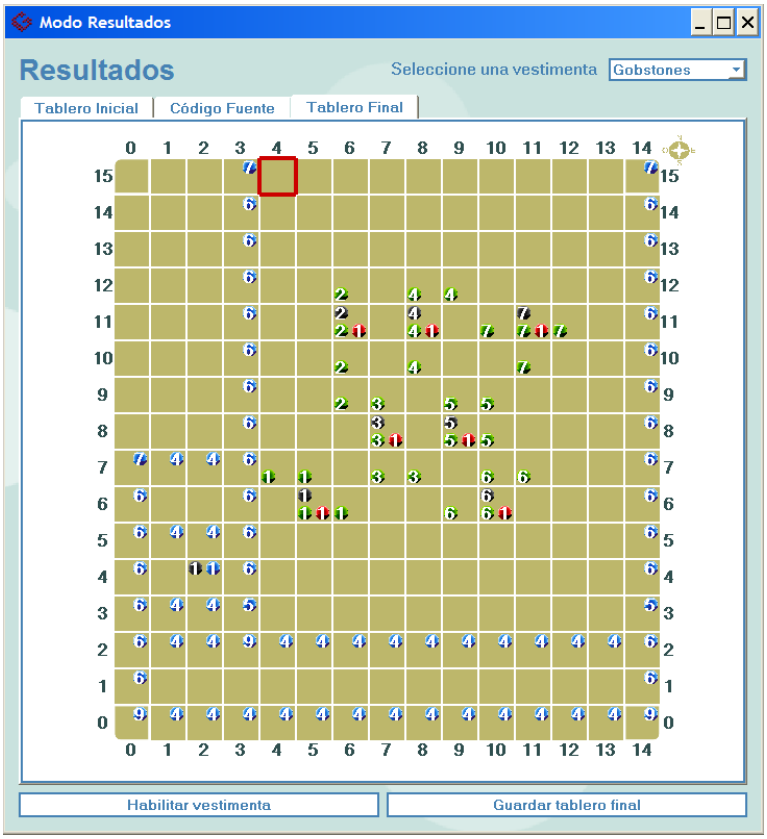
Si el programa del editor de programas es un programa interactivo correcto (comienza con las palabras reservadas `interactive program`, y respeta una sintaxis específica de asociación entre teclas y procedimientos), al ejecutarlo PyGobstones abre una ventana de Modo Interactivo. Esta ventana puede observarse en el gráfico G.A.14. En esta ventana, la luz verde en la esquina inferior derecha indica que la aplicación está esperando que el usuario presione una tecla (la ventana tiene que estar seleccionada para que la tecla presionada sea aceptada por el programa). Al presionar, se ejecuta el procedimiento asignado a dicha tecla, y mientras esto sucede, la luz roja indica que no se pueden ingresar nuevas teclas. Al finalizar la ejecución de dicho procedimiento, la ventana cambia el tablero inicial por el tablero resultante luego de la ejecución, y vuelve al estado de esperar una tecla. Para finalizar la ejecución de un programa interactivo alcanza con cerrar la ventana de resultados o digitar la tecla CTRL-D.

La funcionalidad de programas interactivos sirve para mostrar el poder de un conjunto simple pero fundamental de ideas, como las plasmadas por el lenguaje Gobstones.

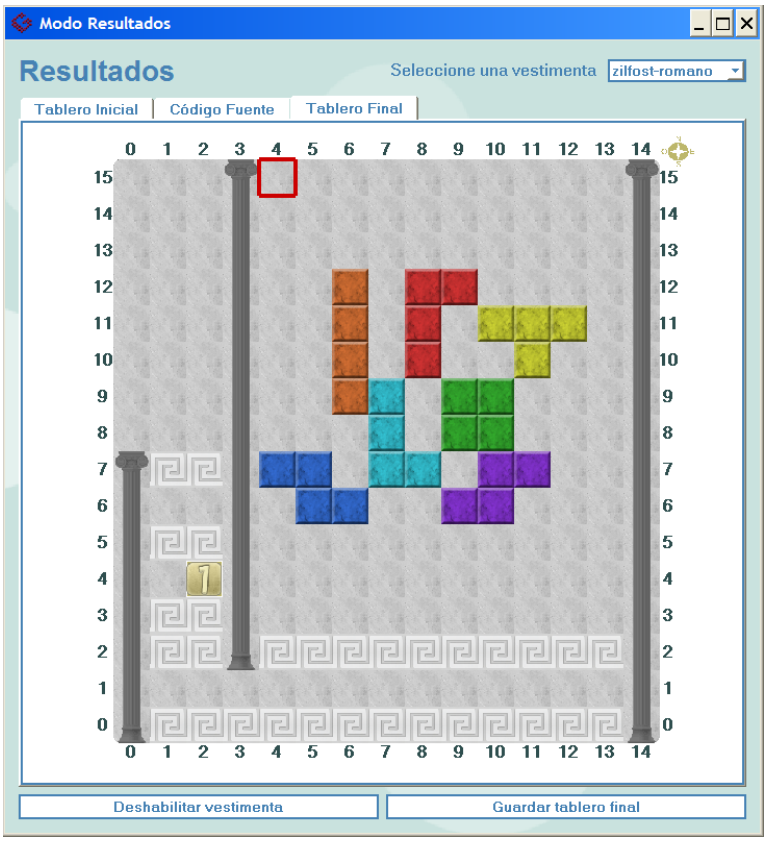
(a). Sin vestir.

(b). Con la vestimenta del ejemplo.

G.A.12. Ejemplos de tableros con y sin vestimentas

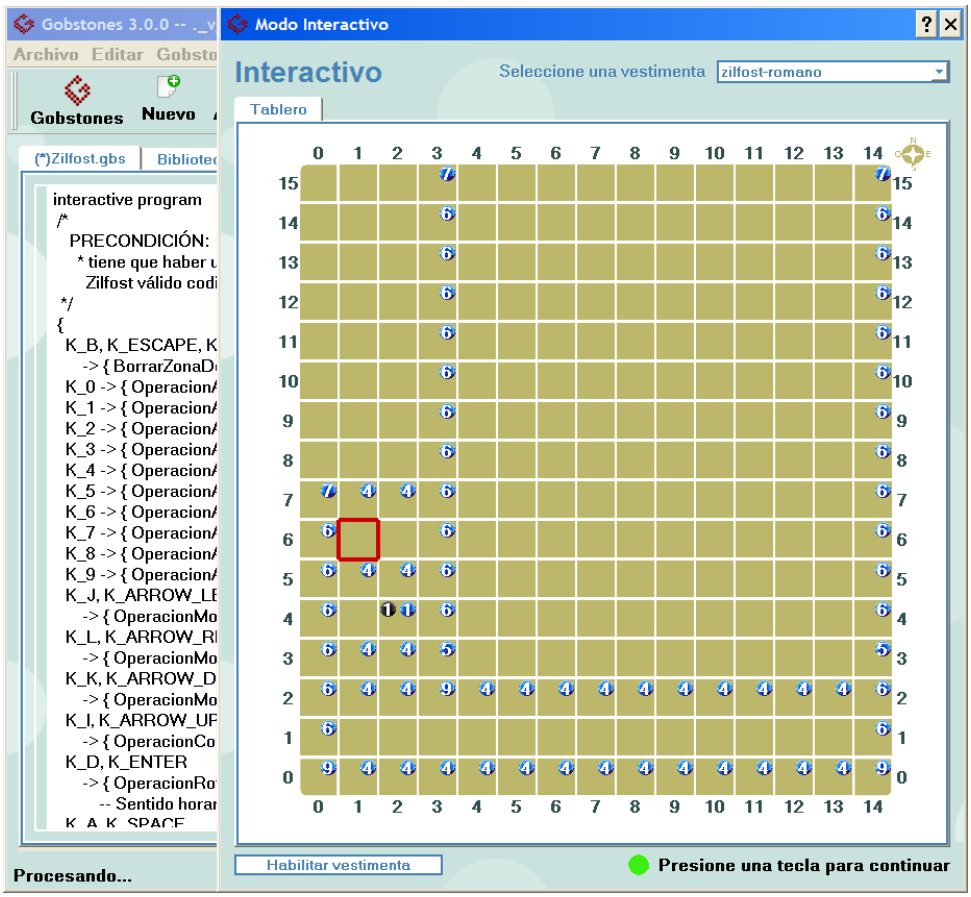


(a). Sin vestir.



(b). Con vestimenta.

G.A.13. Ejemplos de Zilfost con y sin vestimentas



G.A.14. Ventana del Modo Interactivo

B

Código completo del Zilfost

En este apéndice se presenta el código completo del juego ZILFOST, que fuera presentado en el capítulo 5. Se separan las partes en diferentes secciones; para contar con el juego completo deben juntarse todas las secciones (excepto la sección de biblioteca) en un archivo `Zilfost.gbs` y la sección de biblioteca en un archivo `Biblioteca.gbs`. Ambos archivos deben ser colocados en la misma carpeta, y ejecutados con alguna de las herramientas que implementan GOBSTONES en su versión 3.0.

B.1. Código principal

```
/* ------------------------------------------------
   AUTOR: Pablo E. Martínez López
   FECHA: abril-octubre 2013
   OBSERVACIONES:
      * Zilfost es un juego basado en el Tetris, pero donde
        las reglas son levemente diferentes
      * se basa en una idea de un parcial de
        Introducción a la Programación de la
        carrera TPI de la UNQ, con agregados propios
      * el nombre ZILFOST se inspira en la forma de las
        piezas, que recuerdan levemente a distintas letras
      * casi TODAS las operaciones asumen como
        precondición global mínima que hay una
        codificación válida de Zilfost en el
        tablero
      * una codificación válida de Zilfost cumple:
         - se codifican 3 zonas en el tablero: la zona de semilla,
           la zona de juego y la zona de datos
         - sobre la zona de juego:
            . la zona de juego está delimitada en sus extremos
              inferior izquierdo e inferior derecho con
              dos celdas con exactamente 5 bolitas azules,
              de manera que la de la izquierda es la primer
              celda en recorrido NE que tiene exactamente esa
              cantidad de bolitas
            . desde esas dos celdas, hacia arriba hay dos filas
              de bolitas azules con 6 en el cuerpo
              y 7 en el tope; este tope se encuentra en el
              borde Norte
            . estas dos filas delimitan la zona de
              juego
            . la primer celda a la derecha de la celda con 5
              bolitas azules de la izquierda se denomina el
              origen de la zona de juego
         - sobre la zona de datos:
            . a la izquierda de la zona de juego
```

se encuentra la zona de datos, que
tiene la misma altura que la zona de juego.
. la zona de datos contendrá 2 zonas de números
en su parte inferior, la zona de código de la
próxima pieza a ingresar a la zona de juego y
la zona del código de pieza seleccionada
- sobre la zona de semilla:
. abajo de las zonas de juego y de datos se
encuentra la zona de semilla, que tiene el mismo
ancho que ambas zonas juntas
. la zona de semilla es una zona de números
- sobre las zonas de números:
. las zonas de números tienen una fila de
alto y están delimitadas arriba y abajo
por filas cuyas celdas tienen casi todas 4
bolitas azules (algunas pueden variar esta
cantidad en función de otras condiciones)
. los extremos izquierdo y derecho de la fila
de una zona de números estarán delimitados por
celdas con 6 bolitas azules
. las zonas de números utilizan numeración
posicional decimal estándar, de derecha a
izquierda desde las unidades
. cada dígito d en una zona de números
se codifica con 1 bolita
azul para indicar su presencia, y d
bolitas negras para indicar de qué dígito se
trata (0<=d<=9)
. una celda vacía indica que no hay dígito en
ese lugar
- las piezas se codifican en 4 celdas contiguas
utilizando bolitas verdes, rojas y negras.
- el piso se codifica con exactamente 8 bolitas azules
(y puede llevar marcas especiales con bolitas azules
extra)
- una pieza es válida si (ver SECCIÓN 2):
. ocupa 4 celdas contiguas identificadas con
la misma cantidad de bolitas verdes
(el código de la pieza)
. una sola de las celdas es el pivote
(identificado con bolitas negras y
rojas -- las negras indican el
tipo de pieza y las rojas la rotación)
. las celdas de la pieza están dispuestas
según indican el tipo y la rotación
. un tipo válido va entre 1 y 7
(1-Z,2-I,3-L,4-F,5-O,6-S,7-T)
. cada tipo tiene una rotación "natural"
que corresponde a la mnemotecnia
. una rotación válida va entre 1 y 4
y la codificación sigue el sentido de las
agujas del reloj
(1-natural, 2-1/4 de giro horario,
3-1/2 giro, 4-1/4 de giro antihorario)
. el pivote de una pieza puede llevar
una marca de exactamente 7 bolitas rojas
adicionales
- en la zona de juego solo puede haber
piezas válidas o piso, sin marcar
- no hay otra pieza con el mismo código
en la zona de juego
- en las zonas de números solo puede haber
dígitos válidos o celdas vacías

```
 ------------------------------------------------ */

/*=SECCIÓN 1=======================================*
 * Código principal                                *
 *=================================================*/
{-
/*=================================================*/
/*    Programa                                     */
/*=================================================*/
program
 /*
   PRECONDICIÓN:
     * tiene que haber un juego de Zilfost válido
       codificado en el tablero con una zona de
       juego de ancho mínimo 9
 */
 { GenerarLogoZILFOST() }
-}

/*=================================================*/
/* Programa interactivo                            */
/*=================================================*/
interactive program
 /*
    PRECONDICIÓN:
      * tiene que haber un juego de
        Zilfost válido codificado en el tablero
  */
 {
   K_B, K_ESCAPE, K_DELETE, K_BACKSPACE
       -> { BorrarZonaDeSeleccion() }
   K_0 -> { OperacionAgregarDigitoASeleccion(0) }
   K_1 -> { OperacionAgregarDigitoASeleccion(1) }
   K_2 -> { OperacionAgregarDigitoASeleccion(2) }
   K_3 -> { OperacionAgregarDigitoASeleccion(3) }
   K_4 -> { OperacionAgregarDigitoASeleccion(4) }
   K_5 -> { OperacionAgregarDigitoASeleccion(5) }
   K_6 -> { OperacionAgregarDigitoASeleccion(6) }
   K_7 -> { OperacionAgregarDigitoASeleccion(7) }
   K_8 -> { OperacionAgregarDigitoASeleccion(8) }
   K_9 -> { OperacionAgregarDigitoASeleccion(9) }
   K_J, K_ARROW_LEFT
       -> { OperacionMoverPiezaAl(Oeste) }
   K_L, K_ARROW_RIGHT
       -> { OperacionMoverPiezaAl(Este) }
   K_K, K_ARROW_DOWN
       -> { OperacionMoverPiezaAl(Sur) }
   K_I, K_ARROW_UP
       -> { OperacionColocarNuevaPieza() }
   K_D, K_ENTER
       -> { OperacionRotarPieza(True) -- Sentido horario }
   K_A, K_SPACE
       -> { OperacionRotarPieza(False) -- Sentido antihorario }
   _   -> { OperacionBajarPiezas() }
 }
```

B.2. Operaciones sobre zonas

```
/*=SECCIÓN 2=======================================*
 * Operaciones sobre zonas                         *
 *=================================================*
// 2.1 Operaciones de la zona de juego del tablero
```

```
// 2.2 Operaciones en zonas de números
// 2.3 Operaciones de zonas específicas
 *================================================*/
```

B.2.1. Operaciones sobre la zona de juego

```
/*=SECCIÓN 2.1====================================*
 *                                                *
 * Auxiliares - Operaciones de                    *
 *              la zona de juego del tablero      *
 *                                                *
 *   La zona de juego es una zona rectangular,    *
 *   delimitada en sus esquinas inferiores por    *
 *   celdas con 5 bolitas azules, en sus          *
 *   esquinas superiores por celdas con 7         *
 *   bolitas azules y rodeada por celdas con      *
 *   6 bolitas azules a ambos lados.              *
 *                                                *
 *================================================*
// * Geometría de la zona de juego
//   procedure IrAlOrigenDeZonaDeJuego()
//   function desplazamientoXDeZonaDeJuego()
//   function desplazamientoYDeZonaDeJuego()
//   function anchoDeZonaDeJuego()
//   function altoDeZonaDeJuego()
//
// * Movimiento dentro de la zona de juego
//   function puedeMoverEnZonaDeJuego(dir)
//   procedure IrAlBordeDeZonaDeJuego(dir)
//   procedure IrACoordenadaDeZonaDeJuego(x,y)
//   function esFinDelRecorridoNEDeZonaDeJuego()
//   procedure AvanzarEnRecorridoNEDeZonaDeJuego()
 *================================================*/

//-------------------------------------------------
//-------------------------------------------------
procedure IrAlOrigenDeZonaDeJuego()
 /*
    PROPÓSITO: ir al origen de la zona de juego del Zilfost
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * El origen esta al Este de la primer celda
       de 5 bolitas en un recorrido NE de las celdas
 */
 { IrAPrimerCeldaNEConBolitas(Azul,5); Mover(Este) }

//-------------------------------------------------
function desplazamientoXDeZonaDeJuego()
 /*
    PROPÓSITO: retorna la cantidad de celdas al Este
               a moverse desde la esquina suroeste para
               ubicarse en la 1era columna de la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
 */
 {
   IrAlOrigenDeZonaDeJuego()
   return (medirDistanciaAlBorde(Oeste))
 }

//-------------------------------------------------
function desplazamientoYDeZonaDeJuego()
```

```
 /*
    PROPÓSITO: retorna la cantidad de celdas al Norte
               a moverse desde la esquina suroeste para
               ubicarse en la 1era fila de la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
 */
 {
   IrAlOrigenDeZonaDeJuego()
   return (medirDistanciaAlBorde(Sur))
 }

//-----------------------------------------------------
function anchoDeZonaDeJuego()
 /*
    PROPÓSITO: retorna la cantidad de celdas de ancho
               de la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
 */
 {
   IrAlOrigenDeZonaDeJuego()

   // Contar la distancia hasta el otro borde de la zona
   anchoActual := 0
   while (not nroBolitas(Azul)==5)
    {
      anchoActual := anchoActual + 1
      Mover(Este)
    }
   return (anchoActual)
 }

//-----------------------------------------------------
function altoDeZonaDeJuego()
 /*
    PROPÓSITO: retorna la cantidad de celdas de alto
               de la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
 */
 {
   IrAlOrigenDeZonaDeJuego(); Mover(Oeste)

   // Contar la altura hasta las 7 bolitas azules
   altura := 0
   while (nroBolitas(Azul)/=7)
    {
      altura := altura + 1
      Mover(Norte)
    }
   return (altura+1)  // Ajusta contando la ultima
 }

//-----------------------------------------------------
//-----------------------------------------------------
function puedeMoverEnZonaDeJuego(dir)
 /*
    PROPÓSITO: determina si puede moverse en la dirección
               dada sin caerse de la parte de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * la celda actual está dentro de la zona de juego
```

```
    OBSERVACIONES:
      * optimiza la pregunta (no usa cuentas que deban
        recorrer mucho el tablero)
 */
 {
   switch (dir) to
    Norte ->
           // Si dir es Norte, se puede mover dentro de la
           // zona si hay lugar al norte
       { puede := puedeMover(Norte) }
    Este, Oeste ->
          // Si dir es Este u Oeste, se puede mover dentro
          // de la zona si al moverse se topa con el borde
       {
         Mover(dir)
         puede := not (nroBolitas(Azul)==5
                    || nroBolitas(Azul)==6
                    || nroBolitas(Azul)==7
                      )
       }
    _   -> // Solo hay 4 direcciones!!!
           // Si dir es Sur, se puede mover dentro de la
           // zona si al moverse al sur se topa con el borde
       {
         Mover(Sur)
         puede := not (nroBolitas(Azul)==4)
       }

   return (puede)
 }

//-------------------------------------------------------
procedure IrAlBordeDeZonaDeJuego(dir)
 /*
    PROPÓSITO: ir al borde en dirección dir
               dentro de la zona de juego del Zilfost
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
 */
 { while (puedeMoverEnZonaDeJuego(dir)) { Mover(dir) } }

//-------------------------------------------------------
procedure IrACoordenadaDeZonaDeJuego(x,y)
 /*
    PROPÓSITO: ir a la coordenada (x,y) en la zona
               de juego del SeudoTetris
    PRECONDICIONES:
     * (x,y) indica una coordenada válida
             dentro de la zona de juego de Zilfost
    OBSERVACIONES:
     * la zona de juego de Zilfost está desplazada
       al Este por la parte de datos
 */
 {
   IrACoordenada(x + desplazamientoXDeZonaDeJuego()
               , y + desplazamientoYDeZonaDeJuego())
 }

//-------------------------------------------------------
function esFinDelRecorridoNEDeZonaDeJuego()
 /*
    PROPÓSITO: determina si puede moverse a la celda
               siguiente en un recorrido Noreste de
```

```
              las celdas de la zona de juego
  */
 {
  return (not puedeMoverEnZonaDeJuego(Norte)
        && not puedeMoverEnZonaDeJuego(Este))
 }

//-----------------------------------------------------
procedure AvanzarEnRecorridoNEDeZonaDeJuego()
 /*
    PROPÓSITO: avanza a la celda siguiente en un
               recorrido Noreste de las celdas de
               la zona de juego
    PRECONDICIONES: no está en el final del recorrido
  */
 {
   if (puedeMoverEnZonaDeJuego(Este))
     { Mover(Este) }
   else
     { IrAlBordeDeZonaDeJuego(opuesto(Este)); Mover(Norte) }
 }
```

B.2.2. Operaciones sobre zonas de números

```
/*=SECCIÓN 2.2=====================================*
 * Operaciones en zonas de números                 *
 *==================================================*
// Operaciones de movimiento en la zona de números actual
//   function puedeMoverEnZonaDeNumeroAl(dir)
//   procedure IrAlBordeDeZonaDeNumeros(dir)
//
// Operaciones para leer y grabar un número de una
// zona de números
//   function leerZonaDeNumeros()
//     function hayDigito()
//     function leerDigito()
//   procedure GrabarNumeroEnZonaDeNumeros(numero)
//     procedure GrabarDigitoEnCelda(dig)
//
// Operaciones para modificar una zona de números
//   procedure BorrarZonaDeNumeros()
//   procedure AgregarDigitoAZonaDeNumerosPorIzq(dig)
//   procedure IncrementarZonaDeNumeros()
//     procedure IncrementarDigitoDeCelda()
 *==================================================*/

//-----------------------------------------------------
function puedeMoverEnZonaDeNumeroAl(dir)
 /*
    PROPÓSITO:
      devuelve si hay más lugar en dirección dir en
      la zona de números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * la celda actual está en la zona de números a
       determinar si se puede mover
     * dir es Este u Oeste (no tiene sentido que sea
       Norte o Sur, porque las zonas de números tienen
       altura 1)
    OBSERVACIONES:
     * la zona termina con 6 azules al Este y Oeste
 */
 { return(nroBolitasAl(Azul,dir)/=6) }
```

```
//-----------------------------------------------------
procedure IrAlBordeDeZonaDeNumeros(dir)
 /*
    PROPÓSITO:
      ir al borde dir de la zona de números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * se encuentra dentro de una zona de números
     * dir es Este u Oeste (no tiene sentido que sea
       Norte o Sur, porque las zonas de números tienen
       altura 1)
 */
 {
   while(puedeMoverEnZonaDeNumeroAl(dir))
    { Mover(dir) }
 }

//-----------------------------------------------------
//-----------------------------------------------------
function leerZonaDeNumeros()
 /*
    PROPÓSITO:
      devuelve un número codificado en la zona de números
      actual, si tal número existe, o cero si no
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * está en el borde derecho de la zona de números
       a leer
    OBSERVACIONES:
     * se estructura como un recorrido sobre los dígitos
       codificados en la zona de números
     * total guarda el número leído hasta el momento
     * posDig guarda la próxima unidad a leer
 */
 {
   total := 0
   posDig := 1
   while(hayDigito() && puedeMoverEnZonaDeNumeroAl(Oeste))
    {
      // cada digito contribuye según su posición
      total := leerDigito() * posDig + total
      posDig := posDig * 10  // base 10
      Mover(Oeste)
    }
   // si no pudo mover al Oeste y hay dígito, no leyó el
   //  último dígito
   if (hayDigito() && not puedeMoverEnZonaDeNumeroAl(Oeste))
    {
      // cada digito contribuye según su posición
      total := leerDigito() * posDig + total
      posDig := posDig * 10  // base 10
    }
   return(total)
 }

//-----------------------------------------------------
function hayDigito()
 /*
    PROPÓSITO:
      indica si en la celda actual hay un dígito
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
```

```
    OBSERVACIONES:
     * los dígitos se indican con una bolita azul
       y no tienen más de 9 bolitas negras
 */
 { return(nroBolitas(Azul)==1 && nroBolitas(Negro)<=9) }

//-----------------------------------------------------
function leerDigito()
 /*
    PROPÓSITO:
      retorna el dígito codificado en la celda actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * hay un dígito codificado en la celda actual
    OBSERVACIONES:
     * los dígitos se indican con una bolita azul
       y dig negras (0<=dig<=9)
 */
 { return(nroBolitas(Negro)) }

//-----------------------------------------------------
procedure GrabarNumeroEnZonaDeNumeros(numero)
 /*
    PROPÓSITO: guardar el número dado en la zona de
               números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * está en la zona de números donde debe grabarse
 */
 {
   BorrarZonaDeNumeros()
   IrAlBordeDeZonaDeNumeros(Este)
   aGuardar := numero
   while (aGuardar > 0)
     {
       GrabarDigitoEnCelda(aGuardar mod 10)
       aGuardar := aGuardar div 10
       Mover(Oeste)
     }
 }

//-----------------------------------------------------
procedure GrabarDigitoEnCelda(dig)
 /*
    PROPÓSITO:
      agrega el dígito dig codificado en la celda actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * dig es un dígito (entre 0 y 9)
     * está sobre un espacio libre (celda vacía) de
       una zona de números
 */
 { Poner(Azul); PonerN(Negro,dig) }

//-----------------------------------------------------
//-----------------------------------------------------
procedure BorrarZonaDeNumeros()
 /*
    PROPÓSITO:
      borra el contenido de la zona de números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * se encuentra en la zona de números a borrar
```

```
    OBSERVACIONES:
     * se estructura como un recorrido en la zona
       de números
 */
 {
   IrAlBordeDeZonaDeNumeros(Este)
   while(puedeMoverEnZonaDeNumeroAl(Oeste))
    {
      VaciarCelda()
      Mover(Oeste)
    }
   VaciarCelda()
 }

//-----------------------------------------------------
//-----------------------------------------------------
procedure AgregarDigitoAZonaDeNumerosPorIzq(dig)
 /*
    PROPÓSITO:
      agrega el dígito dig codificado en la zona de
      números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * dig es un dígito (entre 0 y 9)
     * se encuentra en la zona de números donde debe
       agregar el dígito
    OBSERVACIONES:
     * los dígitos entran a la zona de izquierda a
       derecha
     * recorre los dígitos de izq a der para
       encontrar dónde poner el dígito
     * los espacios libres solo pueden estar
       a la derecha y si no hay, borra el número
       completo y pone el dígito como el primero
       (alternativamente podría ignorar el dígito)
 */
 {
   IrAlBordeDeZonaDeNumeros(Oeste)
   while(hayDigito()) { Mover(Este) }

   if (nroBolitas(Azul)==0)
     { GrabarDigitoEnCelda(dig) }
   else
     {
       // Si no hay espacio, borra el número anterior
       Mover(Oeste)
       BorrarZonaDeNumeros()
       IrAlBordeDeZonaDeNumeros(Oeste)
       GrabarDigitoEnCelda(dig)
     }
 }

//-----------------------------------------------------
procedure IncrementarZonaDeNumeros()
 /*
    PROPÓSITO: incrementa el número codificado en la
               zona de números actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * el número máximo codificado no excede la cantidad
       de dígitos disponibles
     * se encuentra en la zona de números a incrementar
    OBSERVACIONES:
```

```
     * usa el algoritmo usual de incremento con carry
       ("llevarme uno"), o sea un recorrido sobre los
       dígitos a incrementar
     * puede fallar si excede el máximo representable
 */
 {
   IrAlBordeDeZonaDeNumeros(Este)
   IncrementarDigitoDeCelda()
   carry := (leerDigito() == 0)
   while (carry && puedeMoverEnZonaDeNumeroAl(Oeste))
    {
      Mover(Oeste)
      IncrementarDigitoDeCelda()
      carry := (leerDigito() == 0)
    }
   if (carry) // Se excedió del máximo permitido de piezas!
     {
       BorrarZonaDeNumeros()
       IncrementarDigitoDeCelda()
     }
 }

//-------------------------------------------------
procedure IncrementarDigitoDeCelda()
 /*
    PROPÓSITO: incrementa el dígito actual
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * o bien no hay dígito, o bien es un dígito válido
       (entre 0 y 9)
    OBSERVACIONES:
     * si no hay dígito, lo agrega
     * si se excede, lo vuelve a 0
 */
 {
   // Agrega un dígito si no lo había
   if (not hayDigito()) { Poner(Azul) }
   // Incrementa dicho dígito
   Poner(Negro)
   // Si se excede, vuelve a 0
   if (leerDigito() == 10) { SacarN(Negro, 10) }
 }
```

B.2.3. Operaciones de zonas específicas

```
/*=SECCIÓN 2.3=====================================*
 * Operaciones de zonas específicas                *
 *==================================================*
// procedure IrAlOrigenDeZonaDeProximaPieza()
// procedure IrAlOrigenDeZonaDeSeleccion()
// procedure IrAlOrigenDeZonaDeSemilla()
//
// function leerZonaDeProximaPieza()
// function leerZonaDeSeleccion()
// function leerSemilla()
//
// procedure BorrarZonaDeProximaPieza()
// procedure BorrarZonaDeSeleccion()
//
// procedure AgregarDigitoASeleccion(dig)
// procedure IncrementarZonaDeProximaPieza()
// procedure GrabarSemilla(semilla)
 *================================================*/
```

```
//-----------------------------------------------------
procedure IrAlOrigenDeZonaDeProximaPieza()
 /*
    PROPÓSITO:
      ir al origen de la zona de próxima pieza
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * esta zona está 2 al Oeste del origen de la
       zona de juego y 1 al Norte
 */
 {
   IrAlOrigenDeZonaDeJuego()
   MoverN(Oeste,2); MoverN(Norte, 1)
 }

//-----------------------------------------------------
procedure IrAlOrigenDeZonaDeSeleccion()
 /*
    PROPÓSITO:
      ir al origen de la zona de selección
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * 2 al Norte del origen de la zona de
       próxima pieza
 */
 {
   IrAlOrigenDeZonaDeProximaPieza()
   MoverN(Norte,2)
 }

//-----------------------------------------------------
procedure IrAlOrigenDeZonaDeSemilla()
 /*
    PROPÓSITO: ir al origen de la zona de semilla
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * la semilla se codifica debajo de la zona
        de piezas, todo a lo ancho
 */
 {
   IrAlOrigenDeZonaDeJuego()
   MoverN(Sur, 2)
   IrAlBordeDeZonaDeNumeros(Este)
 }

//-----------------------------------------------------
//-----------------------------------------------------
function leerZonaDeProximaPieza()
 /*
    PROPÓSITO:
      devuelve un número codificado en la zona de
      próxima pieza, si existe, o cero si no
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * va a la zona de próxima pieza y lee el
       número allí codificado
 */
 {
```

```
   IrAlOrigenDeZonaDeProximaPieza()
   return(leerZonaDeNumeros())
 }

//-----------------------------------------------------
function leerZonaDeSeleccion()
 /*
    PROPÓSITO:
      devuelve un número codificado en la zona de
      selección, si existe, o cero si no
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * va a la zona de selección y lee el
       número allí codificado
 */
 {
   IrAlOrigenDeZonaDeSeleccion()
   return(leerZonaDeNumeros())
 }

//-----------------------------------------------------
function leerSemilla()
 /*
    PROPÓSITO: leer el valor de la semilla del tablero
    OBSERVACIONES:
      * la semilla se codifica en la zona de semillas,
        según la codificación de números en zonas
 */
 {
   IrAlOrigenDeZonaDeSemilla()
   return(leerZonaDeNumeros())
 }

//-----------------------------------------------------
//-----------------------------------------------------
procedure BorrarZonaDeProximaPieza()
 /*
    PROPÓSITO:
      borra el contenido de la zona de selección
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * se estructura como un recorrido en la zona
       de selección
 */
 {
   IrAlOrigenDeZonaDeProximaPieza()
   BorrarZonaDeNumeros()
 }

//-----------------------------------------------------
procedure BorrarZonaDeSeleccion()
 /*
    PROPÓSITO:
      borra el contenido de la zona de selección
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
     * se estructura como un recorrido en la zona
       de selección
 */
 {
```

```
   IrAlOrigenDeZonaDeSeleccion()
   BorrarZonaDeNumeros()
 }

//-----------------------------------------------------
//-----------------------------------------------------
procedure AgregarDigitoASeleccion(dig)
 /*
    PROPÓSITO:
      agrega el dígito dig codificado en la zona de
      selección.
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * dig es un dígito (entre 0 y 9)
    OBSERVACIONES:
     * La operatoria varía según lo que ya haya
       en dicha zona (agrega un dígito a los ya
       existentes, o borra los anteriores y deja
       el nuevo como único dígito)
 */
 {
   IrAlOrigenDeZonaDeSeleccion()
   AgregarDigitoAZonaDeNumerosPorIzq(dig)
 }

//-----------------------------------------------------
procedure IncrementarZonaDeProximaPieza()
 /*
    PROPÓSITO: incrementa el número codificado en la
               zona de código de próxima pieza
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
     * el número máximo codificado no excede la cantidad
       de dígitos disponibles
    OBSERVACIONES:
     * puede fallar si excede el máximo representable
 */
 {
   IrAlOrigenDeZonaDeProximaPieza()
   IncrementarZonaDeNumeros()
 }

//-----------------------------------------------------
procedure GrabarSemilla(semilla)
 /*
    PROPÓSITO: guardar la semilla dada como la semilla
               del tablero
    OBSERVACIONES:
      * la semilla se codifica en la zona de semillas,
        según la codificación de números en zonas
 */
 {
   IrAlOrigenDeZonaDeSemilla()
   GrabarNumeroEnZonaDeNumeros(semilla)
 }
```

B.3. Operaciones sobre piezas

```
/*=SECCIÓN 3=====================================*
 * Operaciones para expresar las piezas          *
 *                                               *
 *   Las piezas se representan con bolitas verdes  *
```

```
 *   (tantas como el código de pieza) y su         *
 *   pivote se representa con bolitas negras       *
 *   (tantas como el tipo de pieza) y rojas        *
 *   (tantas como la rotación de la pieza).        *
 *                                                 *
 *   Negras:                                       *
 *     1: pieza Z                                  *
 *     2: pieza I      ZZ   I L  FF OO  SS         *
 *     3: pieza L       XZ  X X  X  XO SX  TXT     *
 *     4: pieza F           I LL F          T      *
 *     5: pieza O           I                      *
 *     6: pieza T        (X marca el pivote        *
 *     7: pieza S           en rotación 1)         *
 *   Rojas:                                        *
 *     1: original, pivote marcado con X           *
 *     2: 1/4 de giro horario                      *
 *     3: 1/2 de giro                              *
 *     4: 1/4 de giro antihorario                  *
 *      (puede estar marcada con 7 extras)         *
 *   Cada pieza tiene un único pivote.             *
 *   Las piezas clase A tienen dos partes          *
 *    directamente adyacentes y otra a en          *
 *    diagonal al pivote. Las piezas clase B       *
 *    tienen las 3 partes directamente adyacentes  *
 *                                                 *
 *=================================================*
// 3.1. Geometría de las piezas
// 3.2. Detección de piezas
 *=================================================*/
```

B.3.1. Geometría de las piezas

```
/*=SECCIÓN 3.1=====================================*
 * Geometría de las piezas                         *
 *=================================================*
// function rotar(rotacion,sentidoHorario)
//
// function esClaseA(tipoPieza)
// function esClaseB(tipoPieza)
// function diresDePiezaClaseA(tipoPieza,rotPieza)
// function diresDePiezaClaseB(tipoPieza,rotPieza)
//   function diresDePiezaZ(rotPieza)
//   function diresDePiezaI(rotPieza)
//   function diresDePiezaL(rotPieza)
//   function diresDePiezaF(rotPieza)
//   function diresDePiezaO(rotPieza)
//   function diresDePiezaS(rotPieza)
//   function diresDePiezaT(rotPieza)
//
//   function id(dA,dB,dC1,dC2)
//   function ajustarDires(dA,dB,dC1,dC2,rotPieza)
 *=================================================*/

//-----------------------------------------------------
//-----------------------------------------------------
function rotar(rotacionBase,sentidoHorario)
 /*
    PROPÓSITO: calcular la rotacion siguiente en
               sentido horario o antihorario, según
               lo indique el booleano sentidoHorario
    PRECONDICIONES:
      * rotacion es una rotación válida
        (de 1 a 4 o con marca de 7 rojas)
```

```
    OBSERVACIONES:
      * el cálculo se realiza de la siguiente forma
        rotacion  1 2 3 4
        . mod 4   1 2 3 0
        . +1      2 3 4 1 // En sentido horario

        rotacion  1 2 3 4
        . + 2     3 4 5 6
        . mod 4   3 0 1 2
        . + 1     4 1 2 3 // En sentido antihorario
 */
 {
   // Recordar si tiene o no marca
   if (rotacionBase>=1 && rotacionBase<=4)
     { marca := 0 }
   else { if (rotacionBase>=8 && rotacionBase<=11)
     { marca := 7 }
   else { }} -- La rotación es inválida

   // Ajusta la rotación si tiene marcas
   rotacion := rotacionBase - marca

   // Calcula la nueva rotación
   if (sentidoHorario)
     { nuevaRotacion := (rotacion mod 4) + 1 }
   else
     { nuevaRotacion := (rotacion+2) mod 4 + 1 }

   // Retorna, restaurando la marca si corresponde
   return (nuevaRotacion+marca)
 }

//-------------------------------------------------------
//-------------------------------------------------------
function esClaseA(tipoPieza)
  /*
     PROPÓSITO: indica si la pieza del tipo dado
                es de clase A
   */
  { return(tipoPieza >= 1 && tipoPieza <= 6) }

//-------------------------------------------------------
function esClaseB(tipoPieza)
  /*
     PROPÓSITO: indica si la pieza del tipo dado
                es de clase B
   */
  { return(tipoPieza >= 7 && tipoPieza <= 7) }

//-------------------------------------------------------
function diresDePiezaClaseA(tipoPieza,rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una pieza
               de clase A, ajustadas según la rotación
    PRECONDICIONES:
     * tipoPieza es un tipo válido
     * rotPieza es una rotación válida
    OBSERVACIONES:
     * realiza una selección alternativa en base al
       tipo de pieza
 */
 {
   switch (tipoPieza) to
```

```
     1 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaZ(rotPieza) }
     2 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaI(rotPieza) }
     3 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaL(rotPieza) }
     4 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaF(rotPieza) }
     5 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaO(rotPieza) }
     6 -> { (dirA,dirB,dirC1,dirC2)
              := diresDePiezaS(rotPieza) }
     _ -> { }

   return (dirA,dirB,dirC1,dirC2)
 }

//-------------------------------------------------------
function diresDePiezaClaseB(tipoPieza,rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una pieza
               de clase B, ajustadas según la rotación
    PRECONDICIONES:
     * tipoPieza es un tipo válido
     * rotPieza es una rotación válida
    OBSERVACIONES:
     * realiza una selección alternativa en base al
       tipo de pieza
 */
 {
   switch (tipoPieza) to
     7 -> { (dirA,dirB,dirC) := diresDePiezaT(rotPieza) }
     _ -> { }

   return (dirA,dirB,dirC)
 }

//-------------------------------------------------------
function diresDePiezaZ(rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una Z
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
      * en rotación 1, Z es
         Norte,Oeste ->  ZZ  <- Norte
                         XZ <- Este
        donde la X representa al pivote y
        las Zs a las demás secciones
 */
 {
   (dA,dB,dC1,dC2) := id(Este,Norte,Norte,Oeste)
   (dA,dB,dC1,dC2) := ajustarDires(dA,dB,dC1,dC2,rotPieza)
   return (dA,dB,dC1,dC2)
 }

//-------------------------------------------------------
function diresDePiezaI(rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una I
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
```

```
      * en rotación 1, I es
               Norte ->  I
                         X
                         I  <- Sur
                         I  <- Sur,Sur
        donde la X representa al pivote y
        las Is a las demás secciones
 */
 {
   (dA,dB,dC1,dC2) := id(Sur,Norte,Sur,Sur)
   (dA,dB,dC1,dC2) := ajustarDires(dA,dB,dC1,dC2,rotPieza)
   return (dA,dB,dC1,dC2)
 }

//-----------------------------------------------------
function diresDePiezaL(rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una L
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
      * en rotación 1, L es
               Norte ->  L
                         X
                 Sur ->  LL  <- Sur,Este
        donde la X representa al pivote y
        las Ls a las demás secciones
 */
 {
   (dA,dB,dC1,dC2) := id(Norte,Sur,Sur,Este)
   (dA,dB,dC1,dC2) := ajustarDires(dA,dB,dC1,dC2,rotPieza)
   return (dA,dB,dC1,dC2)
 }

//-----------------------------------------------------
function diresDePiezaF(rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una F
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
      * en rotación 1, F es
               Norte ->  FF <- Norte,Este
                         X
                         F  <- Sur
        donde la X representa al pivote y
        las Fs a las demás secciones
 */
 {
   (dA,dB,dC1,dC2) := id(Norte,Sur,Norte,Este)
   (dA,dB,dC1,dC2) := ajustarDires(dA,dB,dC1,dC2,rotPieza)
   return (dA,dB,dC1,dC2)
 }

//-----------------------------------------------------
function diresDePiezaO(rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una O
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
      * en rotación 1, O es
               Norte ->  OO  <- Norte,Este
```

```
                        XO  <- Este
       donde la X representa al pivote y
       las Os a las demás secciones
 */
 {
   (dA,dB,dC1,dC2) := id(Norte,Este,Norte,Este)
   (dA,dB,dC1,dC2) := ajustarDires(dA,dB,dC1,dC2,rotPieza)
   return (dA,dB,dC1,dC2)
 }

//-------------------------------------------------------
function diresDePiezaS(rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una S
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
      * en rotación 1, S es
               Norte ->   SS  <- Norte,Este
               Oeste ->  SX
        donde la X representa al pivote y
        las Ss a las demás secciones
 */
 {
   (dA,dB,dC1,dC2) := id(Norte,Oeste,Norte,Este)
   (dA,dB,dC1,dC2) := ajustarDires(dA,dB,dC1,dC2,rotPieza)
   return (dA,dB,dC1,dC2)
 }

//-------------------------------------------------------
function diresDePiezaT(rotPieza)
 /*
    PROPÓSITO: devolver las direcciones de una T
    PRECONDICIONES:
      * la rotación es válida
    OBSERVACIONES:
      * en rotación 1, T es
               Oeste ->  TXT  <- Este
                          T   <- Sur
        donde la X representa al pivote y
        las Ts a las demás secciones
      * se usa una dirección dD como dummy
        para reutilizar ajustarDires
 */
 {
   (dA,dB,dC,dD) := id(Oeste,Este,Sur,Sur)
                                  --^--DUMMY!!
   (dA,dB,dC,dD) := ajustarDires(dA,dB,dC,dD,rotPieza)
   return (dA,dB,dC)
 }

//-------------------------------------------------------
function id(dA,dB,dC1,dC2)
 /*
    PROPÓSITO: retornar varios valores simultáneamente
 */
 { return (dA,dB,dC1,dC2) }

//-------------------------------------------------------
function ajustarDires(dirA,dirB,dirC1,dirC2,rotPieza)
 /*
    PROPÓSITO: ajustar las direcciones en base a la
               rotación
```

```
    PRECONDICIONES:
      * la rotación es válida (y puede llevar una
          marca de 7 bolitas rojas)
 */
 {
   switch (rotPieza) to
    1,8 -> -- La rotación ‘‘natural’’
      {
        ndirA := dirA
        ndirB := dirB
        ndirC1 := dirC1
        ndirC2 := dirC2
      }
    2,9 -> -- 1/4 de giro en sentido horario
      {
        ndirA := siguiente(dirA)
        ndirB := siguiente(dirB)
        ndirC1 := siguiente(dirC1)
        ndirC2 := siguiente(dirC2)
      }
    3,10 -> -- 1/2 giro
      {
        ndirA := opuesto(dirA)
        ndirB := opuesto(dirB)
        ndirC1 := opuesto(dirC1)
        ndirC2 := opuesto(dirC2)
      }
    4,11 -> -- 1/4 de giro en sentido antihorario
      {
        ndirA := previo(dirA)
        ndirB := previo(dirB)
        ndirC1 := previo(dirC1)
        ndirC2 := previo(dirC2)
      }
    _ -> { }

   return (ndirA,ndirB,ndirC1,ndirC2)
 }
```

B.3.2. Detección de piezas

```
/*=SECCIÓN 3.2=====================================*
 * Detección de piezas o su ausencia               *
 *==================================================*
// function esSeccionDeAlgunaPieza()
// function esSeccionPivoteDeAlgunaPieza()
// function esSeccionPivoteDePieza(codPieza)
// function hayPiezaActual()
//
// function leerCodigoDePiezaActual()
// function leerTipoDePiezaActual()
// function leerRotacionDePiezaActual()
//
// function hayLugarParaPiezaTipo(tipoPieza, rotPieza)
//    function hayLgPzClaseAEnDires(dirA,dirB,dirC1,dirC2)
//    function hayLgPzClaseBEnDires(dirA,dirB,dirC)
//    function esCeldaLibre()
//    function hayCeldaLibreAl(dir)
//    function hayCeldaLibreAlY(dir1,dir2)
//--------------------------------------------------
 *==================================================*/

//--------------------------------------------------
```

```
//-----------------------------------------------------
function esSeccionDeAlgunaPieza()
 /*
    PROPÓSITO: determinar si la celda actual es
               sección pivote de alguna pieza
 */
 { return (hayBolitas(Verde)) }

//-----------------------------------------------------
function esSeccionPivoteDeAlgunaPieza()
 /*
    PROPÓSITO: determinar si la celda actual es la
               sección pivote de una pieza
 */
 {
   return (esSeccionDeAlgunaPieza()
        && hayBolitas(Negro)
        && hayBolitas(Rojo))
 }

//-----------------------------------------------------
function esSeccionPivoteDePieza(codPieza)
 /*
    PROPÓSITO: determinar si la celda actual es la
               sección pivote de la pieza codPieza
 */
 { return (esSeccionPivoteDeAlgunaPieza()
          && nroBolitas(Verde)==codPieza) }

//-----------------------------------------------------
function hayPiezaActual()
 /*
    PROPÓSITO: establecer si la celda actual determina
               una pieza seleccionada, lo cual por
               convención quiere decir que está sobre
               la sección pivote de una pieza
 */
 { return (esSeccionPivoteDeAlgunaPieza()) }

//-----------------------------------------------------
//-----------------------------------------------------
function leerCodigoDePiezaActual()
 /*
    PROPÓSITO: determinar el código de la pieza actual
    PRECONDICIONES:
      * la celda actual es el pivote de una pieza
 */
 { return (nroBolitas(Verde)) }

//-----------------------------------------------------
function leerTipoDePiezaActual()
 /*
    PROPÓSITO: determinar el tipo de la pieza actual
    PRECONDICIONES:
      * la celda actual es el pivote de una pieza
 */
 { return (nroBolitas(Negro)) }

//-----------------------------------------------------
function leerRotacionDePiezaActual()
 /*
    PROPÓSITO: determinar la rotación de la pieza actual
    PRECONDICIONES:
```

```
      * la celda actual es el pivote de una pieza
 */
 { return (nroBolitas(Rojo)) }

//-----------------------------------------------------
//-----------------------------------------------------
function hayLugarParaPiezaTipo(tipoPieza, rotPieza)
 /*
     PROPÓSITO: informar si hay lugar en el tablero
                para colocar una pieza de tipo tipoPieza
                y rotación rotPieza con pivote en la
                celda actual
     PRECONDICIONES:
       * tipoPieza es un tipo de pieza válido
       * rotPieza es una rotación válida
     OBSERVACIONES:
       * puede no haber lugar porque se acaba el tablero
         o porque está ocupada
 */
 {
   if (esClaseA(tipoPieza))
     {
       (dirA,dirB,dirC1,dirC2)
            := diresDePiezaClaseA(tipoPieza,rotPieza)
       hayL := hayLgPzClaseAEnDires(dirA,dirB,dirC1,dirC2)
     }
   else // Si no es clase A, es clase B
     {
       (dirA,dirB,dirC)
            := diresDePiezaClaseB(tipoPieza,rotPieza)
       hayL := hayLgPzClaseBEnDires(dirA,dirB,dirC)
     }

   return (hayL)
 }

//-----------------------------------------------------
function hayLgPzClaseAEnDires(dirA,dirB,dirC1,dirC2)
 /*
    PROPÓSITO: completar el trabajo de hayLugarParaPiezaTipo
               para las piezas de clase A
    PRECONDICIONES:
      * está parado sobre el lugar en el que va el pivote
        de la pieza
      * las direcciones dadas codifican la pieza
        correctamente
 */
  {
    return(esCeldaLibre()
           && hayCeldaLibreAl(dirA)
           && hayCeldaLibreAl(dirB)
           && hayCeldaLibreAlY(dirC1,dirC2))
  }

//-----------------------------------------------------
function hayLgPzClaseBEnDires(dirA,dirB,dirC)
 /*
    PROPÓSITO: completar el trabajo de hayLugarParaPiezaTipo
               para las piezas de clase B
    PRECONDICIONES:
      * está parado sobre el lugar en el que va el pivote
        de la pieza
      * las direcciones dadas codifican la pieza
```

```
        correctamente
 */
  {
    return(esCeldaLibre()
           && hayCeldaLibreAl(dirA)
           && hayCeldaLibreAl(dirB)
           && hayCeldaLibreAl(dirC))
  }

//-------------------------------------------------------
function esCeldaLibre()
 /*
    PROPÓSITO: determinar si la celda actual es una
               celda libre
    PRECONDICIONES:
      * la celda actual se encuentra en la zona de
        juego
    OBSERVACIONES:
      * abstrae a la función de Biblioteca que verifica
        si una celda está vacía
 */
 { return (esCeldaVacia()) }

//-------------------------------------------------------
function hayCeldaLibreAl(dir)
 /*
    PROPÓSITO: determinar si hay una celda libre en la
               zona de juego, en la dirección indicada
               por dir
    PRECONDICIONES:
      * la celda actual se encuentra en la zona de
        juego
    OBSERVACIONES:
      * es libre si hay celda y no tiene pieza
 */
 {
   return (puedeMoverEnZonaDeJuego(dir)
           && esCeldaVaciaAl(dir))
 }

//-------------------------------------------------------
function hayCeldaLibreAlY(dir1,dir2)
 /*
    PROPÓSITO: determinar si hay una celda libre en la
               zona de juego, en la dirección indicada
               por las direcciones dir1 y dir2
    PRECONDICIONES:
      * la celda actual se encuentra en la zona de
        juego
      * las direcciones dadas no se "cancelan" mutuamente
        (por ejemplo, no son Norte y Sur o Este y Oeste)
    OBSERVACIONES:
      * es libre si hay celda y no tiene pieza
 */
 {
   if (puedeMoverEnZonaDeJuego(dir1))
     { Mover(dir1)
       if (puedeMoverEnZonaDeJuego(dir2))
         { Mover(dir2)
           celdaLibre := esCeldaLibre() }
       else { celdaLibre := False }} -- No existe la celda2
   else { celdaLibre := False } -- No existe la celda1
```

```
  return (celdaLibre)
 }
```

B.4. Operaciones de procesamiento de piezas

```
/*=SECCIÓN 4=====================================*
 * Procesamiento de piezas                       *
 *================================================*
// 4.1 procedure IrAPiezaSiExiste(codPieza)
// 4.2 procedure ColocarPieza(codPieza, tipoPieza, rotPieza)
//     procedure QuitarPiezaActual()
// 4.3 Operaciones de movimiento de piezas
 *================================================*/
```

B.4.1. Operación de localización de una pieza

```
/*=SECCIÓN 4.1===================================*
 * Procesamiento de piezas (IrAPiezaSiExiste)    *
 *================================================*
// procedure IrAPiezaSiExiste(codPieza)
 *================================================*/

//-------------------------------------------------
procedure IrAPiezaSiExiste(codPieza)
 /*
    PROPÓSITO: va a la celda pivote de la pieza de
               código codPieza, si existeexiste
    PRECONDICIONES:
      * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * se estructura como un recorrido NE sobre las
        celdas de la zona de juego
      * si la pieza no existe, queda en un extremo
        de la zona de juego
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while (not esFinDelRecorridoNEDeZonaDeJuego()
          && not esSeccionPivoteDePieza(codPieza))
    { AvanzarEnRecorridoNEDeZonaDeJuego() }
 }
```

B.4.2. Operaciones para colocar una pieza

```
/*=SECCIÓN 4.2.1=================================*
 *  Procesamiento de piezas (ColocarPieza)       *
 *================================================*
// procedure ColocarPieza(codPieza, tipoPieza, rotPieza)
//   procedure ColocarPzClaseA(codPieza,tipoPieza,rotPieza
//                             ,dirA,dirB,dirC1,dirC2)
//   procedure ColocarPzClaseB(codPieza,tipoPieza,rotPieza
//                             ,dirA,dirB,dirC)
//     procedure ColocarSeccionDePieza(codPieza)
//     procedure ColocarPivote(codPieza,tipoPieza,rotPieza)
//     procedure ColocarSeccionDePiezaEn(codPieza,dir)
//     procedure ColocarSeccionDePiezaEnY(codPieza,dir1,dir2)
 *================================================*/

//-------------------------------------------------
procedure ColocarPieza(codPieza, tipoPieza, rotPieza)
 /*
```

```
    PROPÓSITO: coloca la pieza codPieza en el tablero
    PRECONDICIONES:
      * no hay otra pieza codPieza en el tablero
      * hay lugar para colocar la pieza
      * tipoPieza es un tipo válido
      * rotPieza es una rotación válida
    OBSERVACIONES:
      * la celda actual será el centro de la pieza
        codPieza
      * la pieza codPieza será de tipo tipoPieza
      * la rotación estará dada por rotPieza
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   if (esClaseA(tipoPieza))
     {
       (dirA,dirB,dirC1,dirC2)
             := diresDePiezaClaseA(tipoPieza,rotPieza)
       ColocarPzClaseA(codPieza,tipoPieza,rotPieza
                       ,dirA,dirB,dirC1,dirC2)
     }
   else // Si no es clase A, es clase B
     {
       (dirA,dirB,dirC)
             := diresDePiezaClaseB(tipoPieza,rotPieza)
       ColocarPzClaseB(codPieza,tipoPieza,rotPieza
                       ,dirA,dirB,dirC)
     }
 }

//-----------------------------------------------------
procedure ColocarPzClaseA(codPieza,tipoPieza,rotPieza
                          ,dirA,dirB,dirC1,dirC2)
 /*
    PROPÓSITO: completar el trabajo de ColocarPieza
               para las piezas de clase A
    PRECONDICIONES:
      * hay lugar para colocar la pieza
      * no hay otra pieza de código codPieza en el
        tablero
      * rotPieza es una rotación válida
      * tipoPieza es un tipo válido
      * las direcciones coinciden con el tipoPieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
  {
    ColocarPivote(codPieza,tipoPieza,rotPieza)
    ColocarSeccionDePiezaEn(codPieza,dirA)
    ColocarSeccionDePiezaEn(codPieza,dirB)
    ColocarSeccionDePiezaEnY(codPieza,dirC1,dirC2)
  }

//-----------------------------------------------------
procedure ColocarPzClaseB(codPieza,tipoPieza,rotPieza
                          ,dirA,dirB,dirC)
 /*
    PROPÓSITO: completar el trabajo de ColocarPieza
               para las piezas de clase B
    PRECONDICIONES:
      * hay lugar para colocar la pieza
```

```
      * no hay otra pieza de código codPieza en el
        tablero
      * rotPieza es una rotación válida
      * tipoPieza es un tipo válido
      * las direcciones coinciden con el tipoPieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
  {
    ColocarPivote(codPieza,tipoPieza,rotPieza)
    ColocarSeccionDePiezaEn(codPieza,dirA)
    ColocarSeccionDePiezaEn(codPieza,dirB)
    ColocarSeccionDePiezaEn(codPieza,dirC)
  }

//-------------------------------------------------------
procedure ColocarSeccionDePieza(codPieza)
 /*
    PROPÓSITO: coloca una sección de la pieza codPieza
    PRECONDICIONES:
      * la celda actual está libre
      * no hay otra pieza codPieza en el tablero
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 { PonerN(Verde,codPieza) }

//-------------------------------------------------------
procedure ColocarPivote(codPieza, tipoPieza, rotPieza)
 /*
    PROPÓSITO: coloca el pivote de la pieza codPieza
    PRECONDICIONES:
      * la celda actual está libre
      * no hay otra pieza codPieza en el tablero
      * tipoPieza es un tipo válido
      * rotPieza es una rotación válida
    OBSERVACIONES:
      * la celda actual será el centro de la pieza
        codPieza
      * la pieza codPieza será de tipo tipoPieza
      * la rotación estará dada por rotPieza
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
    ColocarSeccionDePieza(codPieza)
    PonerN(Negro,tipoPieza)
    PonerN(Rojo,rotPieza)
 }

//-------------------------------------------------------
procedure ColocarSeccionDePiezaEn(codPieza,dir)
 /*
    PROPÓSITO: coloca una sección de la pieza codPieza
               en la celda lindante al dir
    PRECONDICIONES:
      * la celda lindante al dir existe y está libre
      * no hay otra pieza codPieza en el tablero
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
```

```
 */
 {
   Mover(dir)
   ColocarSeccionDePieza(codPieza)
   Mover(opuesto(dir))
 }

//-----------------------------------------------------
procedure ColocarSeccionDePiezaEnY(codPieza,dir1,dir2)
 /*
    PROPÓSITO: coloca una sección de la pieza codPieza
               en la celda lindante al dir1 y dir2
    PRECONDICIONES:
      * la celda lindante mencionada existe y está libre
      * no hay otra pieza codPieza en el tablero
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   Mover(dir1);Mover(dir2)
   ColocarSeccionDePieza(codPieza)
   Mover(opuesto(dir1));Mover(opuesto(dir2))
 }
```

B.4.3. Operaciones para quitar una pieza

```
/*=SECCIÓN 4.2.2====================================*
 *  Procesamiento de piezas (QuitarPiezaActual)    *
 *====================================================*
// procedure QuitarSeccionDePiezaActual()
// procedure QuitarPiezaActual()
//   procedure QuitarPzClaseA(codPieza,tipoPieza,rotPieza
//                            ,dirA,dirB,dirC1,dirC2)
//   procedure QuitarPzClaseB(codPieza,tipoPieza,rotPieza
//                            ,dirA,dirB,dirC)
//     procedure QuitarSeccionDePieza(codPieza)
//     procedure QuitarPivote(codPieza,tipoPieza,rotPieza)
//     procedure QuitarSeccionDePiezaDe(codPieza,dir)
//     procedure QuitarSeccionDePiezaDeY(codPieza,dir1,dir2)
 *====================================================*/

//-----------------------------------------------------
//-----------------------------------------------------
procedure QuitarSeccionDePiezaActual()
 /*
    PROPÓSITO: quita una sección de la pieza actual
    PRECONDICIONES:
      * la celda es una sección de pieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   SacarTodasLasDeColor(Verde)
   SacarTodasLasDeColor(Negro)
   SacarTodasLasDeColor(Rojo)
 }

//-----------------------------------------------------
//-----------------------------------------------------
procedure QuitarPiezaActual()
 /*
```

```
    PROPÓSITO: quita la pieza actual del tablero
    PRECONDICIONES:
      * la celda actual es el pivote de una pieza
      * la pieza no está marcada
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   codPieza  := leerCodigoDePiezaActual()
   tipoPieza := leerTipoDePiezaActual()
   rotPieza  := leerRotacionDePiezaActual()
   if (esClaseA(tipoPieza))
     {
       (dirA,dirB,dirC1,dirC2)
             := diresDePiezaClaseA(tipoPieza,rotPieza)
       QuitarPzClaseA(codPieza,tipoPieza,rotPieza
                     ,dirA,dirB,dirC1,dirC2)
     }
   else // Si no es clase A, es clase B
     {
       (dirA,dirB,dirC)
             := diresDePiezaClaseB(tipoPieza,rotPieza)
       QuitarPzClaseB(codPieza,tipoPieza,rotPieza
                     ,dirA,dirB,dirC)
     }
 }

//-----------------------------------------------------
procedure QuitarPzClaseA(codPieza,tipoPieza,rotPieza
                        ,dirA,dirB,dirC1,dirC2)
 /*
    PROPÓSITO: completar el trabajo de QuitarPieza
               para las piezas de clase A
    PRECONDICIONES:
      * hay celdas de la pieza en los lugares correctos
      * rotPieza es una rotación válida
      * tipoPieza es un tipo válido
      * las direcciones coinciden con el tipoPieza
      * la pieza no está marcada
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
  {
    QuitarPivote(codPieza,tipoPieza,rotPieza)
    QuitarSeccionDePiezaDe(codPieza,dirA)
    QuitarSeccionDePiezaDe(codPieza,dirB)
    QuitarSeccionDePiezaDeY(codPieza,dirC1,dirC2)
  }

//-----------------------------------------------------
procedure QuitarPzClaseB(codPieza,tipoPieza,rotPieza
                        ,dirA,dirB,dirC)
 /*
    PROPÓSITO: completar el trabajo de QuitarPieza
               para las piezas de clase B
    PRECONDICIONES:
      * hay celdas de la pieza en los lugares correctos
      * rotPieza es una rotación válida
      * tipoPieza es un tipo válido
      * las direcciones coinciden con el tipoPieza
      * la pieza no está marcada
```

```
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
  {
    QuitarPivote(codPieza,tipoPieza,rotPieza)
    QuitarSeccionDePiezaDe(codPieza,dirA)
    QuitarSeccionDePiezaDe(codPieza,dirB)
    QuitarSeccionDePiezaDe(codPieza,dirC)
  }

//-------------------------------------------------------
procedure QuitarSeccionDePieza(codPieza)
 /*
    PROPÓSITO: quita una sección de la pieza codPieza
    PRECONDICIONES:
      * la celda es una sección de la pieza codPieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 { SacarN(Verde,codPieza) }

//-------------------------------------------------------
procedure QuitarPivote(codPieza, tipoPieza, rotPieza)
 /*
    PROPÓSITO: quita el pivote de la pieza codPieza
    PRECONDICIONES:
      * hay celdas de la pieza en los lugares correctos
      * tipoPieza es un tipo válido
      * rotPieza es una rotación válida
      * la pieza no está marcada
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
    QuitarSeccionDePieza(codPieza)
    SacarN(Negro,tipoPieza)
    SacarN(Rojo,rotPieza)
 }

//-------------------------------------------------------
procedure QuitarSeccionDePiezaDe(codPieza,dir)
 /*
    PROPÓSITO: quita una sección de la pieza codPieza
               en la celda lindante al dir
    PRECONDICIONES:
      * la celda lindante al dir es una sección de
        la pieza codPieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   Mover(dir)
   QuitarSeccionDePieza(codPieza)
   Mover(opuesto(dir))
 }

//-------------------------------------------------------
procedure QuitarSeccionDePiezaDeY(codPieza,dir1,dir2)
 /*
```

```
    PROPÓSITO: quitar una sección de la pieza codPieza
               en la celda lindante al dir1 y dir2
    PRECONDICIONES:
      * la celda lindante al dir1 y dir2 es una sección de
        la pieza codPieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   Mover(dir1);Mover(dir2)
   QuitarSeccionDePieza(codPieza)
   Mover(opuesto(dir1));Mover(opuesto(dir2))
 }
```

B.4.4. Operaciones de movimiento de piezas

```
/*=SECCIÓN 4.3=====================================*
 *  Operaciones de movimiento de piezas             *
 *==================================================*
// function puedeMoverPiezaActual(dir)
// procedure MoverPiezaActual(dir)
//
// function puedeBajarPiezaActual()
// procedure BajarPiezaActual()
//
// procedure RotarPiezaActual(rotacionSentidoHorario)
 *==================================================*/

//-------------------------------------------------------
//-------------------------------------------------------
function puedeMoverPiezaActual(dir)
 /*
    PROPÓSITO: determina si la pieza actual se puede
               mover en la dirección dir
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * para saber si puede mover, se la quita y se
        determina si hay lugar en la celda
        correspondiente (si es que existe)
      * si no hay pieza, no la puede mover...
 */
 {
   if (esSeccionPivoteDeAlgunaPieza()
    && puedeMoverEnZonaDeJuego(dir))
    {
     tipoPieza := leerTipoDePiezaActual()
     rotPieza  := leerRotacionDePiezaActual()
     QuitarPiezaActual() // Se puede pues hay pieza
     Mover(dir)          // Se puede pues hay lugar
     puedeM := hayLugarParaPiezaTipo(tipoPieza,rotPieza)
    }
   else
    { puedeM := False }

   return (puedeM)
 }

//-------------------------------------------------------
procedure MoverPiezaActual(dir)
 /*
    PROPÓSITO: mover la pieza actual en dirección dir,
```

```
              si se puede o nada si no se puede
    PRECONDICIONES:
      * la celda actual está sobre el pivote de una
        pieza
    OBSERVACIONES:
      * para mover una pieza se la quita toda y se la
        pone de nuevo en el nuevo lugar, si hay lugar
      * la celda actual queda en el pivote de la pieza
        ya sea que se movió o no
 */
 {
   codPieza  := leerCodigoDePiezaActual()
   tipoPieza := leerTipoDePiezaActual()
   rotPieza  := leerRotacionDePiezaActual()
   if (puedeMoverEnZonaDeJuego(dir))
     {
       QuitarPiezaActual()
       Mover(dir) // Puede, porque se verificó
       if (hayLugarParaPiezaTipo(tipoPieza,rotPieza))
         {
           // Coloca la pieza en la nueva posición
           ColocarPieza(codPieza, tipoPieza, rotPieza)
         }
       else
         {
           // Coloca la pieza en la celda inicial
           Mover(opuesto(dir))
           ColocarPieza(codPieza, tipoPieza, rotPieza)
         }
     }
 }

//-------------------------------------------------------
//-------------------------------------------------------
function puedeBajarPiezaActual()
 /*
    PROPÓSITO: determina si la pieza actual se puede bajar
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * para saber su puede bajar, se la quita y se determina
        si hay lugar una celda más abajo (si es que hay
        celda más abajo)
      * si no hay pieza, no la puede bajar...
 */
 { return (puedeMoverPiezaActual(Sur)) }

//-------------------------------------------------------
procedure BajarPiezaActual()
 /*
    PROPÓSITO: baja la pieza actual
    PRECONDICIONES:
      * la celda actual es el pivote de una pieza
      * la pieza actual se puede bajar
    OBSERVACIONES:
      * para bajar una pieza se la quita toda y se la
        pone más abajo (se puede porque se pregunta si
        se puede bajar antes, y entonces hay lugar)
      * la celda actual queda en el pivote de la pieza
 */
 { MoverPiezaActual(Sur) }

//-------------------------------------------------------
```

```
procedure RotarPiezaActual(sentidoHorario)
 /*
    PROPÓSITO: rotar la pieza actual, si se puede
               o nada si no se puede
    PRECONDICIONES:
      * la celda actual está sobre el pivote de una
        pieza
    OBSERVACIONES:
      * para rotar una pieza se la quita toda y se la
        pone rotada,si hay lugar
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
   codPieza  := leerCodigoDePiezaActual()
   tipoPieza := leerTipoDePiezaActual()
   rotPieza  := leerRotacionDePiezaActual()
   nuevaRot  := rotar(rotPieza, sentidoHorario)
   QuitarPiezaActual()
   if (hayLugarParaPiezaTipo(tipoPieza,nuevaRot))
     { ColocarPieza(codPieza, tipoPieza, nuevaRot) }
   else
     { ColocarPieza(codPieza, tipoPieza, rotPieza) }
 }
```

B.5. Operaciones de la mecánica del juego

```
/*=SECCIÓN 5=====================================*
 * Mecánica del juego                            *
 *================================================*
// 5.1 procedure ColocarNuevaPieza(codPieza,tipoPieza,ubicacion)
// 5.2 procedure BajarPiezasDeZonaDeJuego()
// 5.3 procedure ExtenderElPiso()
// 5.4 procedure EliminarFilasLlenas()
// 5.5 procedure GenerarLogoZILFOST()
 *================================================*/
```

B.5.1. Operación de colocar nueva pieza

```
/*=SECCIÓN 5.1===================================*
 *  Operaciones del juego (ColocarNuevaPieza)     *
 *================================================*
// procedure ColocarNuevaPieza(codPieza,tipoPieza,ubicacion)
 *================================================*/

//------------------------------------------------
procedure ColocarNuevaPieza(codPieza,tipoPieza,ubicacion)
 /*
    PROPÓSITO: coloca una nueva pieza de código
               codPieza y tipo tipoPieza en la zona de
               juego, en la columna indicada por
               ubicacion
    PRECONDICIONES:
      * la pieza codPieza no existe en la zona de juego
      * ubicación es mayor o igual a 0 y menor que el
        ancho de la zona de juego
    OBSERVACIONES:
      * se coloca la pieza en la 2da fila desde arriba
        en rotación 1 (posición original)
        (ATENCION: para colocarla en otra rotación hay
          que analizar dónde quedaría el pivote, para
```

```
         bajar acorde. En rotación 1 todos los pivotes
         van en fila 2 desde arriba)
     * si no hay lugar, no hace nada
 */
 {
   IrACoordenadaDeZonaDeJuego(ubicacion
                              ,altoDeZonaDeJuego()-1)
   if (tipoPieza /= 7)
     { Mover(Sur) }
         // Ajuste para estar 1 más abajo
         // pues los pivotes van acá siempre en
         // rotación 1, excepto en la pieza 7 (T)
   if (hayLugarParaPiezaTipo(tipoPieza,1))
     { ColocarPieza(codPieza,tipoPieza,1) }
 }
```

B.5.2. Operaciones para bajar piezas

```
/*=SECCIÓN 5.2=====================================*
 *  Procesamiento del juego (BajarPiezas...)       *
 *===================================================*
// procedure BajarPiezasDeZonaDeJuego()
//   procedure UnicamenteBajarPiezas()
//   procedure IrAPiezaBajableNoMarcadaSiExiste()
//     function esSeccionPivoteDePiezaNoMarcadaBajable()
//     function esCeldaConMarcaDePieza()
//   procedure MarcarPiezaActual()
//   procedure QuitarMarcasDePiezas()
//     procedure DesmarcarPiezaActualSiHay()
 *===================================================*/

//-----------------------------------------------------
procedure BajarPiezasDeZonaDeJuego()
 /*
    PROPÓSITO: bajar un lugar todas las piezas que
               pueden bajar
    OBSERVACIONES:
      * al bajar piezas se puede generar nuevo piso
        y por eso se invoca a la operación de extender
        el piso
 */
 {
   UnicamenteBajarPiezas()
   ExtenderElPiso()
 }

//-----------------------------------------------------
procedure UnicamenteBajarPiezas()
 /*
    PROPÓSITO: bajar un lugar todas las piezas que
               pueden bajar
    OBSERVACIONES:
      * luego de bajar una pieza, la misma se marca
        para no volver a procesarla
      * esto es necesario cuando hay varias piezas
        bajables que se enciman y unas bloquean a
        otras para bajar
        Ej: (las piezas son 1,2,3 y 4, todas S)
           11223344
          11223344
        Todas las piezas deberían bajar, pero solo
        la 4 lo hace en una primera vuelta; la
        marca garantiza que solo bajará un lugar
```

```
      * se estructura como un recorrido de piezas
        sin marcar
      * el proceso de pasar al siguiente puede
        excederse si no hay más, terminando siempre
        en el final de un recorrido de celdas (sin
        pieza); de ahí que se llame
        "IrAPiezaBajableNoMarcadaSiExiste"
 */
 {
   IrAPiezaBajableNoMarcadaSiExiste()
   while (puedeBajarPiezaActual())
     {
       BajarPiezaActual()
       MarcarPiezaActual()
       IrAPiezaBajableNoMarcadaSiExiste()
     }
   QuitarMarcasDePiezas()
 }

//-------------------------------------------------------
procedure IrAPiezaBajableNoMarcadaSiExiste()
 /*
    PROPÓSITO: va a un pieza de la zona
               de juego que puede ser bajada
               y no tiene marca
    OBSERVACIONES:
     * se estructura como un recorrido de búsqueda
       sobre las celdas de la zona de juego
     * si no hay pieza, no hay marca y no puede bajar
       con lo cual sigue buscando
     * si para, es porque encontró una pieza no
       marcada que puede bajar o porque se terminaron
       las celdas sin que haya ninguna
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while(not esFinDelRecorridoNEDeZonaDeJuego()
         && not esSeccionPivoteDePiezaNoMarcadaBajable())
     { AvanzarEnRecorridoNEDeZonaDeJuego() }
 }

//-------------------------------------------------------
function esSeccionPivoteDePiezaNoMarcadaBajable()
 /*
    PROPÓSITO: informa si la celda actual es celda
               pivote de una pieza no marcada que
               puede bajar
 */
 {
   return (esSeccionPivoteDeAlgunaPieza()
           && not esCeldaConMarcaDePieza()
           && puedeBajarPiezaActual())
 }

//-------------------------------------------------------
function esCeldaConMarcaDePieza()
 /*
    OBSERVACIONES:
      la marca de 7 se superpone con la
      codificación de giro (de 1 a 4), por eso
      pregunta por mayor
 */
 { return (nroBolitas(Rojo)>7) }
```

```
//-----------------------------------------------------
//-----------------------------------------------------
procedure MarcarPiezaActual()
 /*
    PROPÓSITO: marcar la pieza actual
    PRECONDICIONES:
      * la celda actual es el pivote de una pieza
        y la misma no está marcada
    OBSERVACIONES:
      * se marca con 7 bolitas rojas (verificar que
          otras marcas no usen esta misma codificación)
 */
 { PonerN(Rojo,7) }

//-----------------------------------------------------
procedure QuitarMarcasDePiezas()
 /*
    PROPÓSITO: quita todas las marcas de las piezas
    OBSERVACIONES:
      * se estructura como un recorrido sobre las
        celdas de la zona de juego, quitando todas
        las marcas de pieza
 */
 {
   // Iniciar recorrido NE en la zona de juego
   IrAlOrigenDeZonaDeJuego()
   while (not esFinDelRecorridoNEDeZonaDeJuego())
    {
      // Procesar es sacar la marca de pieza, si existe
      DesmarcarPiezaActualSiHay()
      AvanzarEnRecorridoNEDeZonaDeJuego()
    }
   // Procesar último
   DesmarcarPiezaActualSiHay()
 }

//-----------------------------------------------------
procedure DesmarcarPiezaActualSiHay()
 /*
    PROPÓSITO: desmarcar la pieza actual, si existe
               y está marcada; si no, no hacer nada
    OBSERVACIONES:
      * se marca con 7 bolitas rojas (verificar que
          otras marcas no usen esta misma codificación)
 */
 { if (nroBolitas(Rojo)>7) { SacarN(Rojo,7) } }
```

B.5.3. Operaciones para extender el piso

```
/*=SECCIÓN 5.3=====================================*
 *  Procesamiento del juego (ExtenderElPiso)       *
 *===================================================*
// procedure ExtenderElPiso()
//   procedure MarcarTodasLasCeldasDelPiso()
//   procedure ExtenderElPisoEnLaFilaBase()
//     procedure MarcarLaPosicionDeBase()
//     procedure IrAMarcaDePosicionDeBaseYDesmarcar()
//     function esPiso()
//     function hayPisoAl(dir)
//     procedure PonerPiso()
//     function esCeldaDePisoMarcada()
//     procedure MarcarElPiso()
```

```
//     procedure DesmarcarElPiso()
//     procedure IrACeldaDePisoMarcada()
//
//   procedure TransformarEnPisoSiEsPieza(marca)
//     procedure TransformarEnPisoPiezaActual(marca)
//     procedure TransfPzClaseA(codPieza,tipoPieza,rotPieza
//                              ,dirA,dirB,dirC1,dirC2, marca)
//     procedure TransfPzClaseB(codPieza,tipoPieza,rotPieza
//                              ,dirA,dirB,dirC, marca)
//     procedure TransformarCeldaEnPiso(marca)
//     procedure TransformarCeldaEnPisoAl(dir, marca)
//     procedure TransformarCeldaEnPisoAlY(dir1,dir2, marca)
 *================================================*/

//-------------------------------------------------
procedure ExtenderElPiso()
 /*
    PROPÓSITO: analiza si hay nuevas posiciones que pueden
               estar en el piso, y las convierte en piso
    OBSERVACIONES:
      * no hace falta procesar el último elemento pues
        seguro está en la fila más al Norte y no puede
        tener nada encima
      * se estructura como un recorrido NE sobre celdas
      * para cada celda de piso, se transforma la pieza
        al Norte en piso (este nuevo piso puede producir
        la transformación de más celdas)
 */
 {
   ExtenderElPisoEnLaFilaBase()
   MarcarTodasLasCeldasDelPiso()
   IrACeldaDePisoMarcadaSiExiste()
   while (esCeldaDePisoMarcada())
     {
       DesmarcarElPiso()
       if (puedeMoverEnZonaDeJuego(Norte))
         {
           Mover(Norte)
           TransformarEnPisoSiEsPieza(True)
                                  //-^- Piso marcado
               // La pieza se transforma en piso marcado!
               // Esta operación mueve el cabezal de lugar
         }
       IrACeldaDePisoMarcadaSiExiste()
     }
   EliminarFilasLlenas()
 }

//-------------------------------------------------
procedure MarcarTodasLasCeldasDelPiso()
 /*
    PROPÓSITO: marca todas las celdas del piso en
               la zona de juego
    OBSERVACIONES:
      * se estructura como un recorrido sobre las celdas
        de la zona de juego
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while (not esFinDelRecorridoNEDeZonaDeJuego())
    {
      if (esPiso()) { MarcarElPiso() }
      AvanzarEnRecorridoNEDeZonaDeJuego()
```

```
    }
   if (esPiso()) { MarcarElPiso() }
 }

//-------------------------------------------------------
procedure ExtenderElPisoEnLaFilaBase()
 /*
    PROPÓSITO: analiza si hay piezas en la fila
               base que puedan estar en el piso, y las
               convierte en piso
    OBSERVACIONES:
      * se estructura como un recorrido sobre las
        celdas de la fila base
      * las piezas se transforman a piso sin marcar
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while (puedeMoverEnZonaDeJuego(Este))
     {
       MarcarLaPosicionDeBase()
          // Necesario para volver al mismo lugar

       TransformarEnPisoSiEsPieza(False)
                              //-^- Piso sin marcar
       // Esta operación mueve el cabezal de lugar

       IrAMarcaDePosicionDeBaseYDesmarcar()
          // Vuelve al mismo lugar para continuar
          //  el recorrido
       Mover(Este)
     }
   TransformarEnPisoSiEsPieza(False)
     // Esta operación mueve el cabezal de lugar
 }

//-------------------------------------------------------
procedure MarcarLaPosicionDeBase()
 /*
    PROPÓSITO: marca el piso DEBAJO de la línea de base
               para poder volver.
    OBSERVACIÓN: ¡no marca en la misma posición, porque
                 la misma va a cambiar!
 */
 {
   Mover(Sur)
   MarcarElPiso()
   Mover(Norte)
 }

//-------------------------------------------------------
procedure IrAMarcaDePosicionDeBaseYDesmarcar()
 /*
    PROPÓSITO: ejecuta la acción de volver a una posición
               de piso marcada en la fila base
    OBSERVACIÓN: la marca está en la base y no en la misma
                 posición, porque el piso cambió ahí
 */
 {
  IrAPrimerCeldaNEConBolitas(Azul, 64)
    // Son 64 por las 60 de la marca, y las 4 del borde
    // y es la única marcada de esta forma
  DesmarcarElPiso()
  Mover(Norte)
```

```
 }

//-------------------------------------------------------
function esPiso()
 /*
    PROPÓSITO: informa si la celda actual es piso
               no marcado
 */
 { return(nroBolitas(Azul)==8) }

//-------------------------------------------------------
function hayPisoAl(dir)
 /*
    PROPÓSITO: informa si la celda lindante al dir
               es piso no marcado
    PRECONDICIONES:
      * hay una celda lindante al dir en la zona de
        juego
 */
 {
   Mover(dir)
   return(esPiso())
 }

//-------------------------------------------------------
function esCeldaDePisoMarcada()
 /*
    PROPÓSITO: informa si la celda actual es piso con
               marca de piso (simple)
 */
 { return(nroBolitas(Azul)>60) }

//-------------------------------------------------------
procedure MarcarElPiso()
 /*
    PROPÓSITO: marcar el piso
 */
 { PonerN(Azul,60) }

//-------------------------------------------------------
procedure DesmarcarElPiso()
 /*
    PROPÓSITO: desmarca el piso
    PRECONDICIONES:
     * está sobre una celda de piso marcada
 */
 { SacarN(Azul,60) }

//-------------------------------------------------------
procedure IrACeldaDePisoMarcadaSiExiste()
 /*
    PROPÓSITO: va a una celda con marca de piso
    OBSERVACIONES:
     * si no hay celda con marca de piso, termina
       en la última celda del recorrido NE de la
       zona de juego
 */
 {
   IrAlOrigenDeZonaDeJuego()
   while (not esFinDelRecorridoNEDeZonaDeJuego()
          && not esCeldaDePisoMarcada())
    { AvanzarEnRecorridoNEDeZonaDeJuego() }
 }
```

```
//-----------------------------------------------------
procedure TransformarEnPisoSiEsPieza(marca)
 /*
    PROPÓSITO: transforma en piso a la pieza que
               intersecciona con la celda actual,
               si existe, y agrega la marca si
               corresponde
    OBSERVACIONES:
      * si no hay pieza, entonces no hace nada
 */
 {
   if (esSeccionDeAlgunaPieza())
    {
      IrAPiezaSiExiste(leerCodigoDePiezaActual())
      TransformarEnPisoPiezaActual(marca)
    }
 }

//-----------------------------------------------------
procedure TransformarEnPisoPiezaActual(marca)
 /*
    PROPÓSITO: transforma en piso la pieza actual
               y agrega la marca si corresponde
    PRECONDICIONES:
      * la celda actual es el pivote de una pieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
 {
    codPieza  := leerCodigoDePiezaActual()
    tipoPieza := leerTipoDePiezaActual()
    rotPieza  := leerRotacionDePiezaActual()
    if (esClaseA(tipoPieza))
      {
        (dirA,dirB,dirC1,dirC2)
              := diresDePiezaClaseA(tipoPieza,rotPieza)
        TransfPzClaseA(codPieza,tipoPieza,rotPieza
                       ,dirA,dirB,dirC1,dirC2,marca)
      }
    else // Si no es clase A, es clase B
      {
        (dirA,dirB,dirC)
              := diresDePiezaClaseB(tipoPieza,rotPieza)
        TransfPzClaseB(codPieza,tipoPieza,rotPieza
                       ,dirA,dirB,dirC,marca)
      }
 }

//-----------------------------------------------------
procedure TransfPzClaseA(codPieza,tipoPieza,rotPieza
                         ,dirA,dirB,dirC1,dirC2,marca)
 /*
    PROPÓSITO: completar el trabajo de
               TransformarEnPisoPiezaActual
               para las piezas de clase A
    PRECONDICIONES:
      * hay celdas de la pieza en los lugares correctos
      * rotPieza es una rotación válida
      * tipoPieza es un tipo válido
      * las direcciones coinciden con el tipoPieza
    OBSERVACIONES:
```

```
      * la celda actual queda en el mismo lugar que
        empezó
 */
  {
    TransformarCeldaEnPiso(marca)
    TransformarCeldaEnPisoAl(dirA,marca)
    TransformarCeldaEnPisoAl(dirB,marca)
    TransformarCeldaEnPisoAlY(dirC1,dirC2,marca)
  }

//-----------------------------------------------------
procedure TransfPzClaseB(codPieza,tipoPieza,rotPieza
                        ,dirA,dirB,dirC,marca)
 /*
    PROPÓSITO: completar el trabajo de
               TransformarEnPisoPiezaActual
               para las piezas de clase B
    PRECONDICIONES:
      * hay celdas de la pieza en los lugares correctos
      * rotPieza es una rotación válida
      * tipoPieza es un tipo válido
      * las direcciones coinciden con el tipoPieza
    OBSERVACIONES:
      * la celda actual queda en el mismo lugar que
        empezó
 */
  {
    TransformarCeldaEnPiso(marca)
    TransformarCeldaEnPisoAl(dirA, marca)
    TransformarCeldaEnPisoAl(dirB, marca)
    TransformarCeldaEnPisoAl(dirC, marca)
  }

//-----------------------------------------------------
procedure TransformarCeldaEnPiso(marca)
 /*
    PROPÓSITO: transforma en piso la celda actual
               y agrega la marca si corresponde
    PRECONDICIONES:
      * la celda es sección de una pieza
 */
 {
   QuitarSeccionDePiezaActual()
   PonerPiso()
   if (marca) { MarcarElPiso() }
 }

//-----------------------------------------------------
procedure PonerPiso()
 /*
    PROPÓSITO: pone piso en la celda actual
    PRECONDICIONES:
      * la celda está vacía
    OBSERVACIONES:
      * el piso se indica con 8 bolitas azules
 */
 { PonerN(Azul,8) }

//-----------------------------------------------------
procedure TransformarCeldaEnPisoAl(dir, marca)
 /*
    PROPÓSITO: transforma en piso la celda lindante
               al dir si la misma es sección de pieza
```

```
    PRECONDICIONES:
      * hay una celda lindante al dir y es sección de
        una pieza
    OBSERVACIONES:
      * no cambia la celda actual
 */
 {
   Mover(dir)
   TransformarCeldaEnPiso(marca)
   Mover(opuesto(dir))
 }

//-----------------------------------------------------
procedure TransformarCeldaEnPisoAlY(dir1,dir2, marca)
 /*
    PROPÓSITO: transforma en piso la celda lindante
               al dir1 y dir2 si la misma es sección
               de pieza
    PRECONDICIONES:
      * hay una celda lindante en las dir1 y dir2
        y es sección de una pieza
    OBSERVACIONES:
      * no cambia la celda actual
 */
 {
   Mover(dir1);Mover(dir2)
   TransformarCeldaEnPiso(marca)
   Mover(opuesto(dir1));Mover(opuesto(dir2))
 }
```

B.5.4. Operaciones para eliminar filas llenas

```
/*=SECCIÓN 5.4=====================================*
 *  Procesamiento del juego (EliminarFilasLlenas)  *
 *===================================================*
// procedure EliminarFilasLlenas()
//   procedure EliminarMientrasSigaLlena()
//   function esFilaLlena()
//   procedure BajarFilasSobreEsta()
//   procedure BajarFilaSuperior()
//   procedure VaciarDePisoLaFilaActual()
//   procedure QuitarPiso()
 *===================================================*/

//-----------------------------------------------------
procedure EliminarFilasLlenas()
 /*
    PROPÓSITO:
      eliminar todo el piso de las filas llenas de
      la zona de juego, bajando el piso de las superiores
      a la eliminada
    PRECONDICIONES:
      * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * debe ir de abajo-arriba para no repetir trabajo
      * se estructura como un recorrido sobre filas
  */
 {
   IrAlOrigenDeZonaDeJuego()
   while(puedeMoverEnZonaDeJuego(Norte))
     {
       EliminarMientrasSigaLlena()
       Mover(Norte)
```

```
    }
  // Procesa la fila más al Norte
  EliminarMientrasSigaLlena()
}

//-----------------------------------------------------
procedure EliminarMientrasSigaLlena()
 /*
    PROPÓSITO:
      eliminar las secciones de piso de la fila
      actual bajando las de piso que están sobre ella,
      hasta que la fila actual no esté llena
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * al bajar las filas sobre la actual, se borra
        la que estaba, y la condición puede cambiar
      * al terminar, la fila actual no está llena
  */
 {
   while (esFilaLlena())
    { BajarFilasSobreEsta() }
 }

//-----------------------------------------------------
function esFilaLlena()
 /*
    PROPÓSITO:
      determinar si la fila actual está llena de piso
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * la variable esLlena indica si hasta el momento
        se encontró evidencia que la fila no está
        llena
      * se estructura como un recorrido de búsqueda
        sobre las celdas de la fila actual
  */
 {
   IrAlBordeDeZonaDeJuego(Oeste)
   esLlena := True
   while(esLlena && puedeMoverEnZonaDeJuego(Este))
     {
       esLlena := esLlena && esPiso()
       Mover(Este)
     }
   esLlena := esLlena && esPiso()
   return(esLlena)
 }

//-----------------------------------------------------
procedure BajarFilasSobreEsta()
 /*
    PROPÓSITO:
      bajar todas las filas de piso sobre la actual,
      sobreescribiendo la fila actual,
      solo en la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * al bajar, se quita todo el piso de la fila
        más al Norte
      * el cabezal no se debe mover de la fila
```

```
        actual (por lo que debe devolvérselo a su
        lugar al terminar)
      * la variables desplazamiento guarda
        cuántos lugares debe volverse al Sur
      * se estructura como un recorrido sobre
        las filas arriba de la actual
  */
 {
   desplazamiento := 0
   while (puedeMoverEnZonaDeJuego(Norte))
     {
       BajarFilaSuperior()
       desplazamiento := desplazamiento + 1
       Mover(Norte)
     }
   VaciarDePisoLaFilaActual()
   MoverN(Sur, desplazamiento)
 }

//-----------------------------------------------------
procedure BajarFilaSuperior()
 /*
    PROPÓSITO:
      duplica la fila sobre la actual en la actual
      borrando el contenido de la fila actual,
      solo en la zona de juego
    PRECONDICIONES:
      * no está en la fila más al Norte
    OBSERVACIONES:
      * se estructura como un recorrido sobre
        las celdas de la fila actual
      * solo baja las celdas de piso; las de piezas
        no se bajan
      * Si al bajar el piso puede romper una pieza,
        entonces no baja (y puede resultar "aplastado"
        por otros pisos sobre él)
  */
 {
   IrAlBordeDeZonaDeJuego(Oeste)
   while (puedeMoverEnZonaDeJuego(Este))
     {
       BajarCeldaAlNorte()
       Mover(Este)
     }
   BajarCeldaAlNorte()
 }

//-----------------------------------------------------
procedure BajarCeldaAlNorte()
 /*
    PROPÓSITO:
      baja la celda al norte de la actual,
      si corresponde
    PRECONDICIONES:
      * no está en la fila más al Norte
    OBSERVACIONES:
      * solo baja las celdas de piso; las de piezas
        no se bajan
      * Si al bajar el piso puede romper una pieza,
        entonces es absorvido (no rompe la pieza)
  */
 {
   if (not esSeccionDeAlgunaPieza())
```

```
     // Con este if, las piezas no resultan destruidas
     //  por el piso
     {
       if (hayPisoAl(Norte))
         // con este if, las piezas arriba de este
         //  piso no bajan
         { if (not esPiso()) { PonerPiso() } }
       else
         // si arriba no hay piso (está vacío o hay
         //  pieza), debe vaciarse porque las celdas
         //  de pieza no bajan
         { VaciarCelda() }
     }
 }

//-------------------------------------------------
procedure VaciarDePisoLaFilaActual()
 /*
    PROPÓSITO:
          vacía la fila actual de piso de la zona
          de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * se estructura como un recorrido sobre
        las celdas de la fila actual
  */
 {
   IrAlBordeDeZonaDeJuego(Oeste)
   while (puedeMoverEnZonaDeJuego(Este))
    {
      if (esPiso()) { QuitarPiso() }
      Mover(Este)
    }
   if (esPiso()) { QuitarPiso() }
 }

//-------------------------------------------------
procedure QuitarPiso()
 /*
    PROPÓSITO: quita el piso de la celda actual
    PRECONDICIONES:
      * la celda actual tiene piso
 */
 { SacarN(Azul,8) }
```

B.5.5. Operaciones adicionales

```
/*=SECCIÓN 5.5=======================================*
 *  Operaciones adicionales                           *
 *====================================================*
// procedure GenerarLogoZILFOST()
//   procedure VaciarZonaDeJuego()
 *====================================================*/

//-------------------------------------------------
procedure GenerarLogoZILFOST()
 /*
    PROPÓSITO: Dibuja ZILFOST con piezas
    PRECONDICIONES:
     * la zona de juego tiene un ancho mínimo de 9
 */
 {
```

```
    VaciarZonaDeJuego()
    ColocarNuevaPieza(1,1,1)
    ColocarNuevaPieza(6,6,6)
    BajarPiezasDeZonaDeJuego()
    BajarPiezasDeZonaDeJuego()
    ColocarNuevaPieza(3,3,3)
    ColocarNuevaPieza(5,5,5)
    BajarPiezasDeZonaDeJuego()
    BajarPiezasDeZonaDeJuego()
    ColocarNuevaPieza(7,7,7)
    BajarPiezasDeZonaDeJuego()
    ColocarNuevaPieza(2,2,2)
    ColocarNuevaPieza(4,4,4)
    BajarPiezasDeZonaDeJuego()
    BajarPiezasDeZonaDeJuego()
    BajarPiezasDeZonaDeJuego()
}

//-----------------------------------------------------
procedure VaciarZonaDeJuego()
 /*
    PROPÓSITO: quita todas las piezas y el piso
               de la zona de juego
    PRECONDICIONES:
     * default <hay un tablero de Zilfost codificado>
    OBSERVACIONES:
      * se estructura como un recorrido sobre las
        celdas de la zona de juego, quitando todas
        las piezas
 */
 {
   // Iniciar recorrido NE en la zona de juego
   IrAlOrigenDeZonaDeJuego()
   while (not esFinDelRecorridoNEDeZonaDeJuego())
    {
      VaciarCelda()
      AvanzarEnRecorridoNEDeZonaDeJuego()
    }
   VaciarCelda()
 }
```

B.6. Operaciones de interfaz

```
/*=SECCIÓN 6=======================================*
 * Operaciones para interfaz                       *
 *=================================================*
// 6.1 function determinarNuevaPieza(semilla)
// 6.2 Operaciones de interacción
 *=================================================*/
```

B.6.1. Determinar nueva pieza

```
/*=SECCIÓN 6.1=====================================*
 * Operaciones para interfaz (determinarNuevaPieza) *
 *=================================================*
// function determinarNuevaPieza(semilla)
 *=================================================*/

//-------------------------------------------------
function determinarNuevaPieza(semilla)
 /*
```

```
    PROPÓSITO: devolver código, tipo y ubicación para
               una nueva pieza de manera seudorandómica
               en base a una semilla. También devuelve
               la nueva semilla
    OBSERVACIONES:
      * la semilla de parámetro se usa como semilla inicial
      * el resultado de cada número seudorandómico generado
        se usa como próxima semilla, y el final se retorna
 */
 {
   (ubicacion,nuevaSemilla)
      := randomEntre0YConSemilla(anchoDeZonaDeJuego()-1
                                 ,semilla)
   (tipoPieza,nuevaSemilla)
      := randomEntre0YConSemilla(6
                                 ,nuevaSemilla)
   return(leerZonaDeProximaPieza(), tipoPieza+1
          ,ubicacion               , nuevaSemilla)
 }
```

B.6.2. Operaciones de interacción

```
/*=SECCIÓN 6.2=======================================*
 * Operaciones de interacción                        *
 *====================================================*
// procedure OperacionColocarNuevaPieza()
// procedure OperacionMoverPiezaAl(dir)
// procedure OperacionRotarPieza(sentidoHorario)
// procedure OperacionAgregarDigitoASeleccion(dig)
// procedure OperacionBajarPiezas()
 *====================================================*/

//-----------------------------------------------------
procedure OperacionColocarNuevaPieza()
 /*
    PROPÓSITO:
      combinar las acciones necesarias para la aparición
      de una nueva pieza en la zona de juego (si entra)
  */
 {
   semilla := leerSemilla()
   (codPz,tipoPz,ubicPz,semilla) := determinarNuevaPieza(semilla)
   GrabarSemilla(semilla)
   ColocarNuevaPieza(codPz,tipoPz,ubicPz)

   // Al exceder el máximo, vuelve a 1
   IncrementarZonaDeProximaPieza()
   BorrarZonaDeSeleccion()
 }

//-----------------------------------------------------
procedure OperacionMoverPiezaAl(dir)
 /*
    PROPÓSITO:
      combinar las acciones necesarias para mover la
      pieza indicada en la zona de selección (si existe)
  */
 {
   IrAPiezaSiExiste(leerZonaDeSeleccion())
   if (hayPiezaActual())
     { MoverPiezaActual(dir); ExtenderElPiso() }
   BorrarZonaDeSeleccion()
 }
```

```
//-----------------------------------------------------
procedure OperacionRotarPieza(sentidoHorario)
 /*
    PROPÓSITO:
      combinar las acciones necesarias para rotar la
      pieza indicada en la zona de selección (si existe)
  */
 {
   IrAPiezaSiExiste(leerZonaDeSeleccion())
   if (hayPiezaActual())
     { RotarPiezaActual(sentidoHorario); ExtenderElPiso() }
   BorrarZonaDeSeleccion()
 }

//-----------------------------------------------------
procedure OperacionAgregarDigitoASeleccion(dig)
 /*
    PROPÓSITO:
      agregar un dígito a la selección actual, y bajar
      las piezas al terminar de ingresar un código válido
    OBSERVACIONES:
      * como los dígitos se ingresan de izquierda a derecha
        pero se leen de derecha a izquierda, determinar si
        se completó el ingreso de un código válido se puede
        realizar leyendo el número y viendo que es distinto
        de cero
  */
 {
   AgregarDigitoASeleccion(dig)
   if (leerZonaDeSeleccion()/=0) // Al terminar de seleccionar
                                 //  una pieza, baja todas
     { BajarPiezasDeZonaDeJuego() }
 }

//-----------------------------------------------------
procedure OperacionBajarPiezas()
 /*
    PROPÓSITO:
      baja todas las piezas y resetea la zona de selección
  */
 {
   BajarPiezasDeZonaDeJuego()
   BorrarZonaDeSeleccion()
 }
```

B.7. Operaciones de biblioteca

```
/* ---------------------------------------------- *
 *    Biblioteca.gbs                              *
 * ---------------------------------------------- *
// procedure PonerN(color,n)
// procedure SacarN(color,n)
// procedure MoverN(dir,n)
// procedure DejarN(color,n)
// procedure SacarTodasLasDeColor(color)
// procedure VaciarCelda()
// function esCeldaVacia()
// function hayCeldaVaciaAl(dir)
//
// procedure IrALaEsquina(dir1,dir2)
// procedure IrACoordenada(x,y)
```

```
//
// function nroBolitasAl(col,dir)
// procedure CopiarAcaCeldaAl(dir)
//
// procedure IrAPrimerCeldaNEConBolitas(col,n)
// function medirDistanciaAlBorde(dir)
//
//-----------------------------------------------------
//  De recorrido
//-----------------------------------------------------
// procedure IniciarRecorridoDeCeldas(dirE,dirI)
// function esFinDelRecorridoDeCeldas(dirE,dirI)
// procedure AvanzarASiguienteDelRecorridoDeCeldas(dirE,dirI)
//
//-----------------------------------------------------
//  De generación de números randómicos
//-----------------------------------------------------
// function randomEntre0YConSemilla(maximo,semilla)
//   function min_stand(semilla)
 * ------------------------------------------------ */

//-----------------------------------------------------
procedure PonerN(color, n)
 { repeat(n) { Poner(color) } }

//-----------------------------------------------------
procedure SacarN(color,n)
 /* PRECONDICIÓN: hay al menos n bolitas de color */
 { repeat(n) { Sacar(color) } }

//-----------------------------------------------------
procedure MoverN(dir,n)
 /* PRECONDICIÓN: el cabezal puede moverse n veces
                  en dirección dir
 */
 { repeat(n) { Mover(dir)   } }

//-----------------------------------------------------
procedure DejarN(color,n)
 { SacarTodasLasDeColor(color); PonerN(color,n) }

//-----------------------------------------------------
procedure SacarTodasLasDeColor(color)
 { SacarN(color, nroBolitas(color)) }

//-----------------------------------------------------
procedure VaciarCelda()
 {
   foreach color in [minColor()..maxColor()]
     { SacarTodasLasDeColor(color) }
 }

//-----------------------------------------------------
function esCeldaVacia()
 {
   return (not hayBolitas(Azul) && not hayBolitas(Negro)
        && not hayBolitas(Rojo) && not hayBolitas(Verde))
 }

//-----------------------------------------------------
function esCeldaVaciaAl(dir)
 /* PRECONDICION: el cabezal puede moverse
                  en dirección dir
```

```
 */
 {
   Mover(dir)
   return (esCeldaVacia())
 }

//-----------------------------------------------------
function hayCeldaVaciaAl(dir)
 { return (puedeMover(dir) && esCeldaVaciaAl(dir)) }

//-----------------------------------------------------
//-----------------------------------------------------
procedure IrALaEsquina(dir1,dir2)
 { IrAlBorde(dir1); IrAlBorde(dir2) }

//-----------------------------------------------------
procedure IrACoordenada(x,y)
 /* PRECONDICION: coordenada x,y esta en el tablero */
 { IrALaEsquina(Sur,Oeste); MoverN(Este,x); MoverN(Norte,y) }

//-----------------------------------------------------
//-----------------------------------------------------
function nroBolitasAl(col,dir)
 /* PRECONDICION: el cabezal puede moverse al dir */
 {
   Mover(dir)
   return (nroBolitas(col))
 }

//-----------------------------------------------------
procedure CopiarAcaCeldaAl(dir)
 /* PRECONDICION: el cabezal puede moverse al dir */
 {
   foreach color in [minColor()..maxColor()]
     { DejarN(color, nroBolitasAl(color,dir)) }
 }

//-----------------------------------------------------
//-----------------------------------------------------
procedure IrAPrimerCeldaNEConBolitas(col,n)
 /* PRECONDICION: existe una celda con n bolitas de
                  color col en el tablero
 */
 {
   IniciarRecorridoDeCeldas(Norte,Este)
   while (not (nroBolitas(col)==n))
    { AvanzarASiguienteDelRecorridoDeCeldas(Norte,Este) }
 }

//-----------------------------------------------------
function medirDistanciaAlBorde(dir)
 {
   cont:=0
   while (puedeMover(dir))
    {
      cont := cont + 1
      Mover(dir)
    }
   return(cont)
 }

//-----------------------------------------------------
//  Operaciones de recorridos genericas
```

```
//-----------------------------------------------------
procedure IniciarRecorridoDeCeldas(dirE,dirI)
 {
   IrAlBorde(opuesto(dirE))
   IrAlBorde(opuesto(dirI))
 }

//-----------------------------------------------------
function esFinDelRecorridoDeCeldas(dirE,dirI)
 { return (not puedeMover(dirE)
        && not puedeMover(dirI))
 }

//-----------------------------------------------------
procedure AvanzarASiguienteDelRecorridoDeCeldas(dirE,dirI)
 /* PRECONDICION: no esta en el final del recorrido */
 {
   if (puedeMover(dirI))
     { Mover(dirI) }
   else
     { IrAlBorde(opuesto(dirI)); Mover(dirE) }
 }

//-----------------------------------------------------
//-----------------------------------------------------
function randomEntre0YConSemilla(maximo,semilla)
 /*
    PROPÓSITO: calcula en base a una semilla dada (x_i),
               un número seudoaleatorio entre 0 y el
               máximo dado, y una nueva semilla (x_{i+1})
               a ser usada en invocaciones futuras
    OBSERVACIONES:
      * este código fue copiado del artículo "Functional
        Programming with Overloading and Higher-Order
        Polymorphism" de Mark Jones, publicado en "Advanced
        Functional Programming", J.Jeuring y E.Meijer, editores,
        LNCS 925, Springer Verlag, 1995.
      * para usarlo correctamente, la nuevaSemilla debe ser
        usada como siguiente semilla en posteriores llamados,
        como en
         semilla := 42
         repeat(17)
          {
            (n,semilla) := randomEntre0YConSemilla(10,semilla)
            PonerN(Verde,n)
            Mover(Norte)
          }
 */
 {
   nuevaSemilla := min_stand(semilla)
   return(nuevaSemilla mod maximo, nuevaSemilla)
 }

//-----------------------------------------------------
function min_stand(semilla)
 /*
    PROPÓSITO: calcula un número seudoaleatorio según una semilla dada
    OBSERVACIONES:
      * auxiliar para randomEntre0YConSemilla(maximo, semilla)
      * Mark Jones lo atribuye a "Random Number Generators: Good
        Ones are Hard to Find" de S.K.Park y K.W.Miller, publicado
        en la revista "Communications of the ACM", 31(10):1192-1201,
        en octubre de 1988.
```

```
     * este artículo en realidad lo toma de una propuesta de 1969
       por Lewis, Goodman and Miller y lo propone como estándar
       mínimo de generación de números seudoaleatorios
     * el comentario sobre su funcionamiento fue agregado por mí,
       en base a alguna lectura que encontré alguna vez que lo
       explicaba, y de la que no recuerdo la cita:
         x_{i+1} = a*x_i mod m
       donde
         a = 7^5 = 16807
         m = 2^31 - 1 = 2147483647
         q = m div a = 127773
         r = m mod a = 2836
       y entonces
         x_{i+1} = a*(x_i mod q) - r*(x_i div q) + delta*m
       siendo
         delta = 1 si (a*(x_i mod q) - r*(x_i div q) > 0)
         delta = 0 si no
*/
{
  hi := semilla div 12773  -- semilla div (2^31 mod 7^5)
  lo := semilla mod 12773  -- semilla mod (2^31 mod 7^5)
  preresultado := 16807 * lo - 2836 * hi
              --  7^5  * lo - (2^31 mod 7^5) * hi
  if (preresultado > 0) { delta := 0 }
  else                  { delta := 1 }
  return (preresultado + delta * 2147483647)
                    --  delta * 2^31
}
```

Bibliografía

[Dijkstra and others, 1989] E.W. Dijkstra et al. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.

[Hutton, 1999] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[Jones, 1996] Mark P. Jones. Overloading and higher order polymorphism. In Erik Meijer and John Jeuring, editors, *Advanced Functional Programming*, volume 925 of *Lectures Notes in Computer Science*, pages 97–136. Springer-Verlag, May 1996.

[Kernighan and Pike, 1999] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Professional Computing Series. Addison-Wesley, 1999. ISBN 0-201-61586-X.

[Mannila, 2010] L. Mannila. Invariant based programming in education–an analysis of student difficulties. *Informatics in Education*, 9(1):115–132, 2010.

[Martínez López and Sawady O'Connor, 2013] Pablo E. Martínez López and Federico A. Sawady O'Connor. *Introducción a la programación para la carrera de Licenciatura en Artes y Tecnologías*. Universidad Nacional de Quilmes, marzo 2013. ISBN 978-987-1856-39-8.

[Martínez López *et al.*, 2012] Pablo E. Martínez López, Eduardo A. Bonelli, and Federico A. Sawady O'Connor. El nombre verdadero de la programación. una concepción de la enseñanza de la programación para la sociedad de la información. In *Anales del 10mo Simposio de la Sociedad de la Información (SSI'12), dentro de las 41ras Jornadas Argentinas de Informática (JAIIO '12)*, pages 1–23, setiembre 2012. ISSN 1850-2830.

[Meijer *et al.*, 1991] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[Park and Miller, 1988] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[Scholl and Peyrin, 1988] P.C. Scholl and J.P. Peyrin. *Schémas algorithmiques fondamentaux: séquences et itération*. Université Joseph Fourier Institut Grenoblois d'études informatiques, 1988.

Gobstones